Understanding Developers' Needs on Deprecation as a Language Feature

Anand Ashok Sawant
Delft University of Technology
Delft, The Netherlands
A.A.Sawant@tudelft.nl

Maurício Aniche
Delft University of Technology
Delft, The Netherlands
m.f.aniche@tudelft.nl

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

ABSTRACT

Deprecation is a language feature that allows API producers to mark a feature as obsolete. We aim to gain a deep understanding of the needs of API producers and consumers alike regarding deprecation. To that end, we investigate why API producers deprecate features, whether they remove deprecated features, how they expect consumers to react, and what prompts an API consumer to react to deprecation. To achieve this goal we conduct semi-structured interviews with 17 third-party Java API producers and survey 170 Java developers. We observe that the current deprecation mechanism in Java and the proposal to enhance it does not address all the needs of a developer. This leads us to propose and evaluate three further enhancements to the deprecation mechanism.

KEYWORDS

API, deprecation, Java

ACM Reference Format:
Anand Ashok Sawant, Maurício Aniche, Arie van Deursen, and Alberto Bacchelli. 2018. Understanding Developers' Needs on Deprecation as a Language Feature. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering (ICSE '18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180170

1 INTRODUCTION

Concerning *deprecation*, the official Java documentation states: "A program element annotated @Deprecated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists" [26]. The deprecation mechanism is a commonly used practice [2].

Deprecation as a language feature has been adopted in many languages, such as PHP and Java. However, there is no uniform implementation or support of deprecation across languages. For example, Java exposes its deprecation mechanism as an annotation captured by the compiler to throw a warning when a program element marked as deprecated is invoked; in PHP deprecation is added as a property to a function and throws a runtime warning.

Furthermore, languages have been changing the deprecation mechanism in the search for improvements, as evidenced by the Java language designers' proposal to revamp the Java deprecation mechanism for the third time [16]. According to the Java language designers, the deprecation mechanism has been open to misuse and the inconsistent removal of deprecated features creates confusion surrounding the fate of deprecated features. In their words, this led to a situation where "everybody was confused about what deprecation actually meant, and nobody took it seriously."

This variability in deprecation mechanisms across languages and volatility of implementations shows that deprecation as a whole is an unsolved problem. There is no current understanding as to what constitutes an effective deprecation mechanism. Currently, API consumers do not appear to react to deprecation [24, 29] despite API producers taking great care in documenting the deprecation and the changes involved [2].

In this paper, our goal is to determine the characteristics that a deprecation mechanism should possess and whether these are desirable amongst developers and feasible to implement, particularly in a mainstream language such as Java. We do this conducting a study set up in two phases: An exploratory investigation, followed by the evaluation of the desirability and feasibility of enhancements we propose to the deprecation mechanism.

In the first part of our study, we investigate why API producers deprecate features, how they expect the API consumers to react, whether they remove deprecated features from their APIs, and what the associated challenges are with using the deprecation mechanism. To that aim, we use an *interpretive descriptive* technique to conduct and analyze interviews with 17 developers who work on APIs both in industry and open source. We challenge our findings by conducting a survey with 170 Java professionals.

With the insights gained from API producers and consumers coupled with interface usability guidelines [19], in the second part of this study, we propose enhancements to the deprecation mechanism. We evaluate the feasibility of this proposal by discussing them with two Java language designers (one of whom is the promoter of the current revamp of the existing Java deprecation mechanism) and its desirability among the aforementioned 170 Java professionals.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180170

2 THE DEPRECATION MECHANISM IN JAVA

Deprecation is provided by most programming languages, as a way for developers of APIs and libraries to avoid introducing breaking changes when classes, fields, or methods are to be removed.

In the original documentation of deprecation, we read: "Deprecation is a reasonable choice in all these cases [where the API is buggy, insecure, disappearing in a future release, or encouraging bad coding] because it preserves backward compatibility while encouraging developers to change to the new API" [26].

Java first introduced the deprecation mechanism in the form of a Javadoc `@deprecated` annotation, which provides information on why a feature was deprecated and what replacement feature should be used. Once source code annotations were introduced in Java 1.5, Java introduced a `@Deprecated` annotation. According to the Java language specification, this annotation generates a compiler warning when a deprecated feature is used in source code. Modern IDEs pick up this warning and display the warning along with the accompanying Javadoc (Figure 1).

Recently, there has been a proposal to change the deprecation mechanism in Java (JEP 277 [16]). Stuart Marks (lead Java and OpenJDK language designer and promoter of the changes in the deprecation mechanism) stated that deprecation warnings are largely ignored by API consumers [17]. Marks attributed this behavior to two main reasons (which he captured by observing the behavior of consumers who use the Java SE API):

(1) **Potential misuse:** The current implementation of the deprecation mechanism is open to potential misuse: "the @Deprecated annotation ended up being used for several different purposes" [16]. This led to API consumers not taking deprecation warnings seriously.
(2) **Inconsistent removal:** There is no consistent removal protocol of deprecated features, leading to: "an unclear message [regarding the future of a deprecated feature] being delivered to developers about the meaning of @Deprecated" [16]. This led clients to leave references to a deprecated features in the source code, given that there is no danger of the code breaking when updating to a newer version of the API.

Given the aforementioned issues, the Java language developers put forward a set of enhancements in the JEP [16]:

(1) **forRemoval():** A method named 'forRemoval()' in the deprecation class, which sets a boolean flag to either true or false, where true signifies that the feature is going to be removed in the future and false signifies that there are no plans to remove the deprecated feature.
(2) **since():** A method named 'since()' in the deprecation class, to set a string during the deprecation of a feature to indicate the version of the API in which this feature has been deprecated.

The involved Java language developers expect that these enhancements would remove some of the confusion surrounding deprecation. In addition to enhancing the deprecation mechanism, the Java language designers are going to remove deprecated features that are currently present in the Java SE API. Their hope is that these initiatives will serve as an inspiration to other API producers to remove their deprecated features and to API consumers to take deprecation seriously and consider reacting to it. Overall, they aim to change the culture surrounding deprecation.

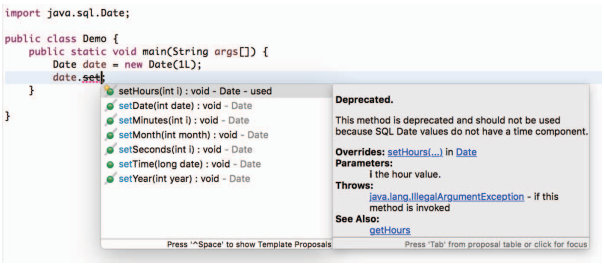

Figure 1: Example of deprecation warning in the IDE

3 METHODOLOGY

The overall goal of this paper is to determine the characteristics that a deprecation mechanism should possess and whether these are desirable among developers and feasible to implement in a mainstream language such as Java. This study has two parts: In the first part, we start by deeply understanding how the deprecation mechanism is used and perceived by both API producers and consumers; in the second part, we propose extensions to Java's deprecation mechanism and determine the feasibility of the same. This section describes the methodology for the first part, the methodology for the second part can be found in Section 6.

In the first part of our study, we propose four research questions:

RQ_1: Why do API producers use the deprecation mechanism? In this RQ, we identify why API producers use the deprecation mechanism. Such an understanding will help us obtain a catalog of reasons adopted by API producers to deprecate features.

RQ_2: When and why do API producers remove deprecated features? The Java language designers claim that inconsistent removal policies related to deprecated features have led to confusion surrounding the implications of deprecation. Having no clear policy of removal sends an unclear message to API consumers. In this RQ, we investigate as to what the different removal protocols are and why API producers adopt them.

RQ_3: How do API producers expect their consumers to react to deprecation? It is an unverified claim that API producers always require their consumers to react to deprecation. In this RQ, we seek to understand and analyze when API producers feel that a reaction should take place.

RQ_4: Why do API consumers react to deprecated features? From our first research question we obtain a catalog of reasons why API producers may deprecate a feature. In this RQ, we investigate the consumers' perspective on deprecation.

3.1 Research Method

Interviews with API producers. To gain an in-depth understanding of how API producers perceive the current implementation of the deprecation mechanism in Java, we conduct a series of semi-structured interviews [15] with industrial and open source software (OSS) API producers.

Before the interviews, (1) we analyze the official Java documentation and study the deprecation mechanism that elucidates when the mechanism should be used, (2) we look at the improvements made to the deprecation mechanism in Java 5 and finally (3) we look

at the proposed enhancement for Java 9. This helps us understand the scenarios in which deprecation is used; this understanding was key in developing and conducting the interviews.

The questions asked during the interviews are based on a guideline derived from our research questions. We ask interviewees questions such as "When do you decide to change an API?" and "Are there any steps that would lessen the burden to upgrade?". We iteratively refine this guideline before every interview, based on the responses. Interviews are conducted in English and transcribed.

We follow an interpretive descriptive approach [31], after an explorative research method, originating from the social sciences, that is an inductive approach to analyzing interviews and deriving theories. As part of the interpretive descriptive technique, each interview transcript was analyzed and broken into smaller parts, where each part was assigned a code based on its content. We clustered these codes based on similarity, to let common themes emerge from the interviews. When we encounter the same code repeatedly across multiple interviews, *i.e.*, saturation, we adjust our interview guideline to explore other topics. Each research question has its own set of codes, which we then present as our results.

Survey with API producers and consumers. To challenge the findings from the interviews, we send out an anonymized survey made up of 29 questions to developers. Our survey consists of questions for both API producers and API consumers, based on the role that the developer plays. This is so that we get both perspectives on the Java deprecation mechanism. It followed the structure of the theory developed as a result of the interviews. The survey respondents were asked to rank the degree (on a five-point Likert scale) to which they agreed with a theme that emerged from the interview process. When a respondent completely disagrees with one of the statements, we ask the respondent to provide us with their perspective. The survey is in our replication package [27].

3.2 Participant Selection

Interviews. We contacted API producers who work for large companies in two different countries (The Netherlands and Brazil) and those that actively work on well-known open source projects. We contacted the industrial developers by mailing the CTOs of certain companies asking to be put in contact with producers of APIs. In the case of open source developers, we mailed the internal developers of JUnit, Spring and Mockito asking for an interview. We chose these three projects due to (1) the popularity of their API (According to Sawant *et al.* [28], JUnit is the first, Spring the third, and Mockito the 10th most used APIs on GitHub), and (2) convenience to access developers working on these projects. Overall this resulted in 17 interview participants (identified as $P1$ - $P17$ in this paper). The background of the participants is summarized in Table 1.

Survey. We aimed to reach as many Java developers from diverse backgrounds. To that end, the survey was spread via Twitter, Java mailing lists, country-specific developer mailing lists, and companies. The survey ran for a period of 3 months. Overall, we obtained 170 responses from which we could derive valid results. The survey respondents were primarily developers (142 out of 170), the rest was composed of architects (10 out of 170), researchers (9 out of 170), analysts (1 out of 170), manager (3 out of 170), consultants (1 out of 170) and testers (4 out of 170). 138 of our respondents work in industry and the rest on open source projects. Our respondents originate primarily from countries such as USA, Italy, Brazil, India and The Netherlands. On average our respondents have 11 years of experience of working with Java.

Table 1: Profiles of the interviewed API producers

ID	Domain	Company/ Project	Experience (in years)
P1	Industry	Large consultancy	6
P2	Industry	Large consultancy	7
P3	Industry	Large consultancy	6
P4	Industry	Large bank	6
P5	Industry	Large bank	5
P6	Industry	Large consultancy	4
P7	Industry	Large SW company	18
P8	Industry	Large SW company	16
P9	Industry	Large SW company	21
P10	Industry	Startup	6
P11	Industry	Startup	9
P12	Industry	Public sector	15
P13	Industry	Small SW company	9
P14	Industry	XWiki	16
P15	OSS	Spring Framework	8
P16	OSS	JUnit	17
P17	OSS	Mockito	4

3.3 Limitations

One of the risks of a qualitative study is that determining the validity of our findings is a difficult undertaking [9]. We followed interpretive descriptive interview guidelines closely. Despite our best efforts, some limitations exist, in the following, we explain how we try to minimize them.

Generalizability. Our selection of developers to interview might not be representative of the Java API producer community. To mitigate this limitation, we questioned three different sets of developers for their opinions from both industry and open source based projects. Furthermore, we surveyed 170 developers who act as API producers and consumers, to challenge our findings. If the study is repeated using a different set of developers, the results may be different. However, we found a large agreement concerning the view on deprecation of interviewees and survey respondents.

Interviewer bias. Our own biases may have played a role when interviewing developers, *e.g.*, by leading interviewees to provide more desirable answers [11]. To mitigate this issue, we challenged and triangulated our findings by conducting a survey with API producers and consumers. We sent our survey out via different media such as Twitter and developer forums. Given this, we do not know the exact context of the population that has accessed our survey. Due to this, we cannot know the exact response rate. However we can report that a total of 535 developers started our survey and 170 (32%) of them filled the entire survey.

Credibility. Question-order effect [30] (a phenomenon where one question could provide context for the next one) may lead our interviewees and survey respondents to specific answers. One approach

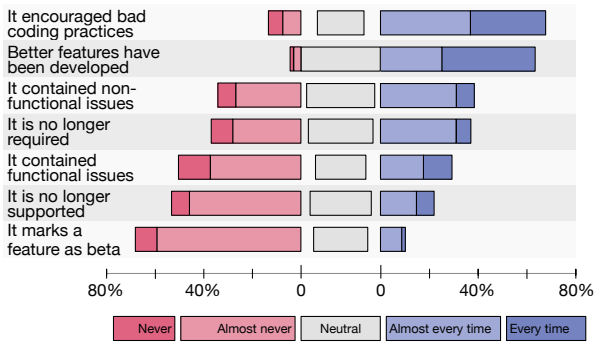

Figure 2: Motivations to deprecate a feature according to API producers, by decreasing frequency

to address this bias could have been to randomize the order of questions. In this study, we decided to order the questions based on the order in which decisions are taken when deprecating a feature. Social desirability bias [8] (*i.e.*, an interviewee's tendency to give a socially acceptable answer to appear in a positive light) may have influenced answers in our interview and survey. To mitigate this issue, we informed participants that the responses would be anonymous and evaluated in a statistical form.

4 RESULTS

4.1 RQ1: Why do API producers use the deprecation mechanism?

We ask API producers whether and why they find the deprecation mechanism relevant and what motivates them to use it. From the interviews and the Java documentation on deprecation, seven main reasons as to why developers deprecate part of their API emerge:

(1) Old interface encourages bad practices
(2) New/updated feature supersedes existing one
(3) Usage of the feature is unnecessary
(4) Functional issue in current implementation
(5) Non-functional issue in current implementation
(6) No longer provide a feature
(7) Mark as beta

The first three reasons emerge from documentation. Reasons 2 - 6 are mentioned by our interviewees, with reason 1 the only unmentioned one.We included these seven main reasons behind deprecation in our survey that was sent to developers. In our survey, API producers were asked how frequently they had used one or more of the reasons; Figure 2 reports the results. We also asked both API producers and consumers if they had encountered any other reasons behind deprecating a feature. Within 170 responses we obtained no more reasons behind deprecation.

Reason 1 is the most popular reason behind deprecation as reported by our survey's respondents. Reasons 2-6 are the ones that all our interviewees agreed upon as being those that they have used in their own APIs. API producers in our survey also agreed that these are common/frequent reasons that they have used to deprecate a feature.

Reason 7 was mentioned by interviewee P16, whose team unexpectedly uses the deprecation mechanism in Java to mark a feature as beta or experimental. Interviewee P16 did acknowledge this as is a misuse of the deprecation mechanism, but they found it effective. P14 mentioned that in their API they faced a similar dilemma, where they wanted to explicitly indicate to an API consumer that a feature being used was new/experimental. However, instead of perverting the meaning of deprecation, they introduced their own annotation `@Unstable` that marked in the documentation that caution should be exercised when using a new feature.

Orthogonal to all the motivations behind deprecation, it emerged that deprecation is an effective, yet imperfect communication channel between API producers and the API consumers. Interviewee P17 put it as: "[deprecation is] still the best communication method as of today". He went on to further state that the language feature of deprecation makes it easier to tell a developer that uses a deprecated feature: "Hey! Be aware! We will remove this feature later, just so you know". This advantage of deprecation is not restricted to the open source software world but is found to be quite important in industry as well. Interviewee P2 said that they find it important "when stopping an existing service, this communication should happen at the time when we are thinking of breaking an existing service". The industrial API producers find it useful to let their customers know that a certain service that they might be using is going to disappear. Nevertheless, the low reaction rates of API consumers reported in literature [24, 29] underlines that messages sent through this channel (regardless of how accurately they are written [2]) are often not acted upon, thus raising concerns on its actual effectiveness, also on our interviewees.

One stated advantage of deprecation is that a developer is given instant feedback in the IDE when using a deprecated feature. Despite this, some of our interviewees (P5, P6, P7, P10) expressed the view that there might be a better way to communicate this change to the consumer. In the opinion of these participants, this warning might be a bit of a later point, as it would require developers to go through each file in the IDE to see if there are any deprecated features that are being used. This happens only after an upgrade is made to a newer version, but the notification only reaches the developers at the last moment, thus necessitating an alternative channel of communication.

> The deprecation mechanism is viewed as an interface for communication with the API consumer. However, it is open to misuse and has shortcomings.

4.2 RQ2: When and why do API producers remove deprecated features?

Most API producers (48%) among our survey respondents indicated that they usually remove a feature two or more releases after its deprecation. Despite most of the positive votes for this policy, only 12% of developers indicate that they always remove a feature after two releases, whereas 22% indicate that they do this almost every time and 29% say they occasionally do this.

The second most popular removal pattern among our survey respondents was to *never remove a deprecated feature*. This may be

at the basis of the lack of reaction by clients to deprecation. 16% of developers say that they always choose to not remove a deprecated feature, 15% say that they do this almost every time and 27% say that they do this occasionally. Only 25% of the respondents say that they have never adhered to this behavior.

Our survey respondents also indicated that they remove deprecated features in an upcoming release or in one release after the next immediate one. However, none of these responses had too much support. Developers do not appear to be very keen on cleaning up their code base after deprecating a feature.

Regarding *why* developers decide (not) to remove a deprecated feature, from the survey results we get the sense that there is a large variance in removal policies of deprecated features, a fact echoed by Stuart Marks (proponent of JEP 277) [17].

Interviewee P8 mentioned that they never remove a deprecated feature because consumers do not appreciate it when functionalities are removed. The other developers of this company (P7 and P9) agree with this perspective. In their opinion, the introduction of breaking changes would be detrimental to customer satisfaction; this is despite this company providing detailed documentation on how to transition to the replacement feature along with customer support to aid the transition.

Interviewee P12 echoed the previous sentiment. Their company too prefers to never remove a deprecated feature. In fact, they are willing to maintain two versions of the same feature in their code base. This company has only ever introduced a breaking change when there was a severe flaw in the feature that was being used and in such a case the feature was not deprecated first.

P14 mentioned that they also do not remove a feature from their API. After a feature is deprecated, they first remove all references to it from their own code base. After this, they move this feature to a legacy package. This does not involve changing the namespace of the feature, they simply build their APIs JAR in such a way that they obtain one version with no legacy features (*i.e.*, no deprecated features) and another which includes them. This gives the consumers of this API the choice of continuing the use of a legacy feature.

Interviewee P17 mentioned that in their API they often remove a deprecated feature, however, they do not have a regular schedule or policy. What can trigger the removal may be major changes such as a modification in the underlying architecture of the API.

Only interviewee P15 mentioned that they have a protocol to remove deprecated features. When deprecating a feature they indicate the release in which this feature is going to be removed. They generally remove deprecated features in the following major release. On being asked about this policy resulting in breaking changes, the interviewee responded: "We deprecate a feature when we have a point where we see it's not useful … and then we remove it in the next major release … because we have a new [,better] implementation."

> API producers are wary about removing deprecated features from their API and mostly have no preset protocol for removal.

4.3 RQ3: How do API producers expect their consumers to react to deprecation?

We asked our interviewees whether they perceived that (1) deprecation on its own was enough to send a message across that a feature should no longer be used and (2) whether it would act as an incentive for their clients to react to API evolution. Predominantly, most of our interviewees said that it was the choice of the consumer to react, but that the deprecation mechanism would have no impact on reaction behavior. In the words of P17: "I think it's an easy way out for developers of an API because it is really easy to edit and notify your users, but you do not actually remove the whole feature. So you are stuck with the sense that a user can be willing to keep using it and not be incentivized to actually stop using it".

The only outlier we had was interviewee P5 who disagreed with the popular sentiment and went on to say that deprecation is a beautiful concept as it gives a consumer the time to consider reacting: "just marking something deprecated will give you an opportunity to think of alternative ways of doing things but at the same time keeping control over when you want to move on … to new features".

As a follow-up, we asked our interviewees if deprecation of a feature could act as an incentive for their clients to change the version of the API that they are using. Interviewee P10 mentioned that the reason behind the deprecation would be key: "Yes, … the reason for deprecation has to be concrete enough for me to switch to the new versions".

In the view of interviewee P2, the decision to switch versions is often based on the cost of an upgrade: given that deprecation is not a breaking change, it does not act as a stumbling block; however, if the reason behind deprecation is serious enough, then there is an incentive to change. Overall, API producers assume that their clients will react to deprecation only if the reason is serious.

We also wanted to establish if deprecation of a feature dissuades the usage of that feature to such an extent that it can be safely removed. Some of our industrial interviewees said that when they deprecate a feature, they can often remove it safely later on as they know that their clients have received the message. Concerning OSS, P16 said: "For JUnit, it hasn't really worked out to deprecate things and then get rid of them … [but it] might work for some other libraries". He recounted a case where JUnit had deprecated a single field in the codebase, which had caused IDE's such as Eclipse to post issues on JUnit as they were opposed to that field being deprecated.

One point of agreement across all the API producers was that an automated tool that helps API consumers to react to deprecation with minimal effort would probably be most beneficial. Such a tool would ensure that clients react, and keep concerns regarding cost to change at a minimum.

> API producers acknowledge the costs for consumers associated to reacting to deprecation. For this reason, they assume a prompt reaction by consumers only if the reason behind deprecation is serious.

4.4 RQ4: Why do API consumers react to deprecated features?

We start by ascertaining whether API consumers react to deprecated features. In the event that they do react, we investigate why.

Do API consumers react to deprecated features? In addition to investigating whether consumers react to deprecated features, we also explore whether deprecation acts as a barrier to upgrading the version of the API being used.

Deprecation is regarded by consumers as the least important reason that prevents them from changing versions of the API. Over 40% of the consumers are neutral about its impact, while only 1% indicate that deprecation has prevented them from upgrading API versions. We see that the largest barrier to API consumers when upgrading a version of an API is the new version breaking some existing functionality. This is in line with previous findings [4].

Most consumers claim that they react to deprecation, we asked these API consumers as to how they react. 67% of consumers indicate that they react by replacing a deprecated feature with its recommended replacement. 66% of our respondents claim to read the documentation and then base the reaction on the motivation behind deprecation. "doing nothing" is the second least popular way to react to deprecation, 26% of the survey respondents indicate that they have never reacted in this manner. The least popular way to react to deprecation is to replace the deprecated feature with an in-house feature. These findings contradict earlier results *et al.* [29] showing that majority of projects on GitHub do not react to API deprecations. However, one explanation for this might be that these responses might be an indicator of the social desirability bias, thus prompting the consumers to claim that they always react.

We ask the respondents who have never reacted to deprecation to explain the reason behind their behavior. Most responses indicated that since deprecation is not a breaking change, reacting to deprecation is not pivotal. This is summarized by one respondent saying: "It is the safest bet to keep things as they are. Deprecation as such does not change the behavior of the solution, so it doesn't need to be acted upon". Other responses include API consumers saying that the cost of a reaction was not justified hence they preferred to wait till the deprecated feature is removed. Poor documentation was also cited as a reason not to react.

Why do API consumers react to deprecated features? We ask our survey respondents to indicate what motivated them to react to deprecation. These results can be found in Figure 3.

We asked the API consumers if knowing the removal policy of an API regarding deprecated features had any impact on their decision to react. This was a point of contention, with 29% of respondents saying that this had indeed motivated them to react. However, a majority of 36% claimed that this had never motivated them and that the reason behind deprecation had far greater significance.

We see in Figure 3 that the motivation behind deprecation plays a large role in eliciting a reaction from the API consumer. When a feature is deprecated due to functional issues, non-functional issues, or because a new feature is an improvement over the old one, there is a large number of API consumers (over 40%) in each case that says they have been motivated to react in that case. 27% or less of the API consumers indicate that other reasons have motivated them to react to deprecation.

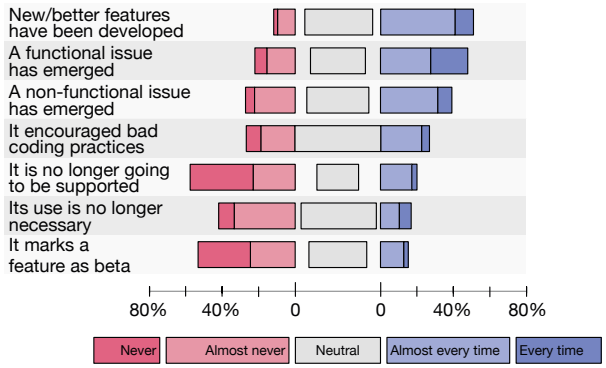

Figure 3: Motivations to react to deprecation according to API consumers' experience, by decreasing frequency

> API consumers predominantly claim that they react to deprecation, the driving factor behind this behavior is the reason behind deprecation.

5 ANALYSIS AND REFLECTION

We now discuss the main findings of our study. The knowledge gained from this study helps us understand the gaps (and how to address them) in the current implementation of the deprecation mechanism and deprecation in Java on the whole.

5.1 A communication mechanism

Previous work by Sawant *et al.* [29] has shown that API consumers in the Java ecosystem do not necessarily replace references to deprecated features in their source code. This is similar to the behavior observed in the SmallTalk sphere by Robbes *et al.* [24] as well.

One of the contributing factors to this phenomenon is the fact that third-party API producers and Java SE API producers do not have consistency when it comes to the removal of a feature from the API as seen in Section 4.2. This points to the fact that the deprecation mechanism in its current form might not be fulfilling its goal in effectively communicating when it is imperative that an API consumer cease to use a deprecated feature.

From the interviews, we observed that API producers agree with this view as they too feel that certain improvements can be made to streamline the communication between producers and clients. To address this, we leverage the interface usability guidelines outlined by Jakob Nielsen [19]. By considering deprecation as a communication interface between API producers and API consumers and by using interface usability guidelines, we hope to be able to understand the shortfall in the effectiveness of the deprecation mechanism as a communication mechanism.

In the following, we discuss two enhancements to the current deprecation mechanism that would better facilitate the communication between API producers and consumers, namely understanding when a feature will be removed from the API and the severity level of the deprecation.

The future of deprecated features. The current implementation of the deprecation mechanism is in direct contravention of Nielsen's [19] guideline on "Visibility of system status," which states: "The system should always keep users informed about what is going on." Indeed, with the current deprecation mechanism provided by the Java language, API consumers have no indication on the future of a deprecated feature.

The enhancements in Java 9's deprecation mechanism attempt to address this shortcoming by allowing API producers to indicate whether a deprecated feature is going to be removed or not. By doing so, API consumers will be given a clear indication about the future of a deprecated feature. This will help the consumer take a decision on the reaction to deprecation.

However, the enhancements do not go far enough in addressing the issues present in the usability of the deprecation interface. Although the future of a deprecated feature has been made explicit, there is still no definite timeline that an API producer can provide to the consumer. By just marking a feature as one that will be removed, no indication is given as to how long the removal could take, which still leaves the future of a deprecated feature in an ambiguous state as the deprecated feature could be removed immediately or after many years. Currently, the onus is on APIs to provide this timeline in the Javadoc (*e.g.*, Spring framework), however, this is not standard practice [2].

The severity of a deprecation. Nielsen's usability guideline on "Consistency and standards" dictates that "Users should not have to wonder whether different words, situations, or actions mean the same thing" [19]. In the current implementation of the deprecation mechanism, there is no way to discern the difference between features deprecated due to serious issues, those that have been deprecated due to small improvements, or even those that have been deprecated because there was no better alternative to communicate with the customers (as in the case of beta features). The proposed enhancements to the deprecation mechanism to be implemented in Java 9 do not address this issue.

We see from API producers (Section 4.3), that there are different suggested reaction patterns for different deprecations. Only in certain cases where there is a serious issue with a feature, do they feel it is imperative for the consumer to react. From the API consumers that answered our survey, we understand that the reason behind deprecation is important when it comes to reacting to a deprecated feature. Functional issues, non-functional issues, and bad coding practices are all major motivations when it comes to reacting to deprecated features, whereas they are less likely to react to a deprecation of a feature due to the fact that usage of it is no longer required. In the current state, the deprecation mechanism does not allow for API producers to inform the API consumers on the severity of a deprecation.

An indication of the severity of deprecation would not be a novel extension to Java's deprecation mechanism. Currently, C# [18] allows API producers to indicate if a deprecation is severe or not with the help of a boolean. In the event that a deprecation is serious, the compiler throws an error when the functionality is invoked. Although C#'s approach can be considered extreme, it shows that indication of severity of a deprecation is a viable feature in a deprecation mechanism. We highlight that this extension to the mechanism is of utmost importance to API producers and consumers alike and will aid the deprecation mechanism in functioning as a more effective communication interface.

5.2 Misuse of deprecation

Currently, in Java, if an API producer wants to issue a compiler warning to communicate with the consumer, the deprecation mechanism is the only straightforward option. API producers have attempted to overcome this limitation in the Java language specification by implementing workarounds (*e.g.*, in the case of XWiki, beta features are marked using a special annotation which requires special IDE support so that it is highlighted). However, none of these workarounds are natively supported by IDEs or Continuous Integration (CI) environments and, thus, not portable.

This has led to the misuse of the deprecation mechanism, where certain features that are not intended for removal in the future are marked as deprecated (*e.g.*, in the case of JUnit where beta features are marked as deprecated).

This leads us to conclude that there is a need for an alternative way for API producers to communicate with the API consumers, especially in the case where they would like to indicate that a feature is beta or experimental. A generic warning mechanism that gives API producers the ability to generate compiler warnings on usages of certain features of the API for reasons other than deprecation could solve this issue.

Such mechanisms already exist in other languages (*e.g.*, Python has a warning system that allows for the specifications of different levels of compiler warnings) and would not be a revolutionary introduction. However, by introducing such a system and making deprecation a sub-case of the warning mechanism the Java language designers would allow API producers increased flexibility. There is currently no proposal to introduce such a mechanism; we postulate that it would be fruitful for the Java community to discuss this.

5.3 API consumer aid with deprecation

Most of our interviewees suggested that refactoring tools to automatically replace references to deprecated features with their recommended replacements could incentivize API consumers to react to deprecation, as it would reduce the overall cost to react to deprecation and reduce the chances of errors.

There is some existing work on providing refactoring support to API consumers to reduce the burden of reacting to deprecation. Henkel and Diwan [10] propose to capture refactorings made by API producers to their codebase when adapting to their own deprecated features and then replaying these refactorings on the API consumers' code. Xing and Stroulia [33] developed an approach that recommends alternative features from an API to replace an obsolete feature by looking at how the API's own codebase has adapted to change. The tool created by Perkins [20] replaces deprecated method invocations with the source code of the deprecated method from the API itself. This has been shown to be effective in 75% of cases.

Although exploring promising avenues, all of the aforementioned tools require a non-trivial amount of effort from the API producers, thus do not scale. Additionally, these tools do not handle more complex cases where the replacement for a deprecated method is

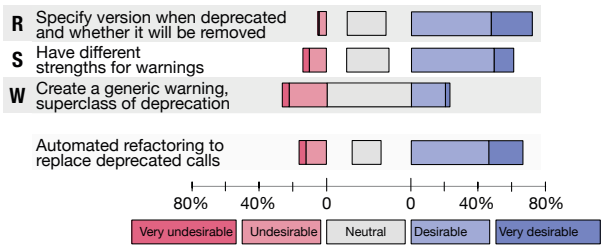

Figure 4: Respondent's perspective on the enhancements to the Java deprecation mechanism

not a one-to-one replacement. This shows that this problem of automatically replacing deprecated features is non-trivial and remains unsolved. The persistent need for such a tool calls for increased research in programming languages and practices supporting automated API restructuring.

6 PROPOSED ENHANCEMENTS TO THE DEPRECATION MECHANISM

Based on our results and analysis, we have uncovered certain aspects surrounding the deprecation mechanism that are especially important for the Java language designers. In the second part of this study, we propose and investigate desirability and feasibility of three enhancements. The first two are related to the deprecation mechanism: they go beyond JEP 277, aiming at defining a more complete deprecation mechanism. The third proposal address an issue at a higher level - the language level.

(1) **R: Removal dates should be marked**: Deprecation should allow developers to indicate the version or date when a deprecated feature will be removed.
(2) **S: Severity should be marked**: Deprecation should allow developers to indicate the severity of a deprecation and raise warnings of according strength.
(3) **W: Warning mechanisms should be generic**: Java should introduce a warning mechanism to allow for other types of warnings to be raised, as well as managed by IDEs, thus minimizing the misuse of deprecation.

We refer to our proposal as **RSW** from this point on. We validate the desirability and feasibility of **RSW** by performing a two-step validation. First, we obtain feedback from the larger community of Java developers (survey with the same 170 professionals) to understand to what extent there would be support for **RSW**. Second, we interview two Java language design experts (one of them being the promoter of JEP 277) to determine whether **RSW** could be implemented in Java and if-so then how this could be done and what the associated difficulties would be.

6.1 Desirability among the Java community

We present the results of our survey in Figure 4. 78% of developers find **R** to be (very) desirable and only 7% do not want such a change. In fact, in our optional write-in survey option, 2 developers expressed even more support for this feature.

We see that **S** is the third most desired change, thus implying that Java developers would like API producers to be allowed to signify to their clients that in certain cases it is pivotal that the client reacts. This supports that API consumers do not get a clear indication as to how severe the deprecation of a feature is.

W aims to give API producers more flexibility when it comes to communicating with their clients. We see that there is no strong trend among Java developers concerning the desirability of this proposal: The 22% of the developers who find it desirable are balanced by 27% who find it undesirable. Moreover, 51% of the developers sit on the fence in this case and have neutral opinion on the warning mechanism. This result may indicate that the Java community does not currently perceive it as necessary to have different warnings other than deprecation; this would be in line with the low number of respondents who reported to use the deprecation mechanism for purposes different than the intended ones (Section 4.4). Nevertheless, this proposal is the one that diverges the most from what developers are already used to, thus, it would be reasonable to consider the results in the light of the theory on the "diffusion of innovations" [25]; in this case, the percentage of the respondents that found this enhancement desirable would be slightly higher than the expected percentage of *early adopters* of innovations [25].

We also asked our survey respondents whether they would welcome an automated refactoring tool to deal with deprecation. The survey respondents were primarily positive, confirming the opinion of the interviewed API producers.

6.2 From theory to practice: RSW's Feasibility

We assessed the feasibility of implementing **RSW** in the Java language by means of an interview with the promoter of JEP 277 (Stuart Marks referred to as J1) and a Java language design expert (referred to as J2) who has been part of the expert group for JSR-305 [21] and the specification lead group of JSR-308 [6].

R: Indication on a removal timeline for deprecated features. Regarding **R**, J1 mentions that the implementation would be possible but not trivial. The principal challenge is how such a feature could be implemented. The first issue with indicating in which version a deprecated feature is going to be removed is that version numbering schemes constantly change. The second issue is that having a concrete date would pose a challenge as well since release schedules constantly change (*e.g.*, Java 9 release has been pushed back twice in the last year). However, J1 did concede that giving such an indication is definitely useful to API consumers, but only if a uniform version numbering convention is adopted by third-party APIs and the Java SE.

J2 confirmed as well that **R** would send a more concrete message about removal to the API consumers. However, he felt that there would be no need to over complicate the annotation to achieve this aim: "[there is no] need to make the annotation that much more complicated". He felt that if an API established and adhered to a uniform policy to deal with deprecated features, that might achieve the same goal.

S: Indication on the severity of deprecation. With **S**, we hope to provide API consumers a clear indication as to how severe a certain deprecation is. J1 said that for JEP 277 something similar was considered: "[the idea was] to include an *enumerator* that specifies the reason behind deprecation such that tools could do filtering based on the reason, and different reasons would have different severities for different users". According to him, the issue was that

compiling a comprehensive list of distinct reasons behind deprecation of a feature is a non-trivial endeavor. In fact, we observed in our interviews that, in many cases, deprecations could fall into more than one category. However, our proposal differs from this as it allows API producers to indicate the level of severity *i.e.*, the seriousness with which API consumers should take this deprecation into account. This is similar to the severity field in the deprecation mechanism in C#. J1 felt that it could be implemented in Java but only after further deliberation on this takes place.

In J2's opinion having two different levels of deprecation: "error versus warning" would be a good idea, as it would give API producers enhanced control over the usage of the features their API provides. However, there are two principal challenges associated with this: First, the deprecation mechanism would have to provide a boolean flag that would allow an API producer to indicate if it is severe or not, which would also indicate to the compiler if a warning or error should be thrown; Second, a better understanding would be needed to demarcate deprecated features that are severe and those that are not.

W: A high-level warning mechanism. W does not impact the deprecation mechanism directly, but seeks to address the misuse of the deprecation as elucidated in JEP 277. In this work, we found just one case of misuse of the deprecation mechanism where a beta feature was marked as deprecated. J1 agreed that marking beta features as deprecated is "off label" usage of the deprecation mechanism. During the discussions on JEP 277, it was considered to have "experimental" as a reason to deprecate a feature. Despite this extension never being implemented, no further steps were taken to minimize this misuse of the deprecation mechanism. J1 felt that a warning mechanism would be extreme. In fact, he suggested that one viable alternative that will be present in Java 9 is "incubator modules, which are sort-of related to beta or experimental". However, these are coarse-grained as they apply to entire modules and not individual classes or methods.

J2 was was not entirely supportive of adding a generic warning mechanism to Java. He felt that "[by attempting] to eliminate every type of misuse, we're only going to open the opportunity for more types of misuse, and we're going to make it harder for people who are going to use it in a sensible way or in an imaginative way". He also felt that introducing an explicit warning mechanism and making deprecation a subclass of that system would be too massive a change to introduce into the Java language. However, he was supportive of first trying out more specific warnings as in the case of JSR-305 [21]. This would then entail creating a generic warning mechanism that could be reused for each specific warning.

7 COMPARISON OF DEPRECATION MECHANISMS IN OTHER LANGUAGES

Besides Java, there are many other languages that provide a mechanism that allows API producers to deprecate features in their APIs. We investigate to what degree these languages implement **RSW**. We focus on languages that have the deprecation mechanism as a built-in feature in the language, unlike languages such as Python or Go that rely on documentation and coding conventions to mark functionality as deprecated. Additionally, we evaluate whether they implement any feature in line with **RSW**.

We select the first 15 languages from the Tiobe language popularity index [13] that have a built-in mechanism, and to analyze if newer languages do a better job with deprecation we look at languages that have been released since 2010. We also add SmallTalk to this comparison as it has been studied in previous work. This results in a comparison between 21 languages seen in Table 2.

We first of all note that no language implements all of **RSW**. Ruby and Dart are the only languages that allow API producers to indicate when a deprecated feature is going to be removed (**R**). Scala, on the other hand, allows API producers to indicate if a deprecated feature is going to be removed. Visual Basic, Kotlin, and C# are the only languages that implement **S**. Only 6 languages allow developers to issue a custom compiler warning (**W**). The warnings thrown in PHP and hack are runtime warnings, whereas for the other languages they are all compiler warnings.

Julia, Swift, Scala, and Rust allow developers to indicate the version in which a feature was deprecated. For the other languages, this fact is typically communicated with the aid of documentation. However, unlike Java's Javadoc system that supports deprecation, most languages do not provide a dedicated mechanism to document deprecation.

In terms of variance between newer and older languages, we see that Kotlin, Rust and Swift are the most advanced as they implement one of the facets of **RSW**. Among the more established languages, C# and Ruby standout as well.

We see that there is no uniform manner in which languages implement their deprecation mechanism. In fact, newer languages who have a clean slate to start with also do not implement all the features that would constitute a more complete deprecation mechanism.

> Languages are not consistent in implementing a deprecation mechanism and none implement **RSW** fully.

8 RELATED WORK

Previous studies have focused on deprecation from the client perspective. Particularly Robbes *et al.* [24] analyzed the reaction to deprecated features in the SmallTalk ecosystem. They found that in most cases clients prefer not to react to an obsolete API feature. This study was extended by Sawant *et al.* [29] who analyzed the reaction to deprecated features of 5 popular Java APIs. This study confirmed the results of Robbes *et al.* with one exception where clients of Java APIs were less inclined to replace an obsolete feature with a replacement feature from the same API.

There have been a few studies focused on an APIs deprecation policy itself. Raemaekers *et al.* [22, 23] investigated when API developers deprecate features. They found that APIs introduced deprecated features in major and minor releases in equal measure. Brito *et al.* [2] investigate whether API developers document deprecated features with a link to the replacement feature. They found that in two-thirds of the cases deprecated features were appropriately documented by the API developers, however, they found that the quality of these deprecation messages did not improve over time.

The introduction of breaking changes in APIs and their impact has been a major topic of study. Dig and Johnson [5] studied and

Table 2: Deprecation mechanisms across languages

Language	Version of feature deprecation	Warning: feature to be removed	R	S	W
Most popular languages with a deprecation mechanism					
Java 8	✓				
C#				✓	
VBasic				✓	
PHP	✓				✓
Ruby		✓	✓		✓
Delphi					✓
R	✓				✓
Obj.-C					✓
Dart		✓	✓		
D					
Scala	✓	✓			
Clojure					
Haskell					
Groovy	✓				
Languages developed since 2010					
Swift	✓				
Rust	✓				
Kotlin	✓			✓	
Ceylon					
Julia					
Hack	✓				✓
Language with deprecation investigated in previous work					
Smalltalk	✓				

classified the API breaking changes in 4 APIs. They found that 80% of the breaking changes introduced in an API were due to refactorings. Cossette and Walker [3] studied 5 Java based APIs to investigate how API evolution recommenders would handle certain cases. Their study showed that none of the recommenders could handle all of the breaking cases.

The impact of breaking changes in APIs can be wide ranging. For example, Linares-Vasquez *et al.* [14] show that breaking changes in Android APIs have an impact on the rating of an app. Espinha *et al.* [7] looked at the impact of breaking changes introduced in popular web APIs such as Twitter, Facebook etc. Xavier *et al.* [32] looked at 317 real-world Java libraries and showed that 14% of API changes are breaking in nature and 2.5% of the clients of these APIs are affected by these changes.

Studies on why APIs evolve over time and what decisions go into evolving an API provide a unique insight into the inner workings of APIs. Bogart *et al.* [1] studied how developers decide to introduce breaking changes in APIs in the Eclipse, R, and NodeJS ecosystems, how these changes are communicated to the clients of the APIs and what tooling and practices are adopted to ensure that the impact of the change is minimal. They show that each ecosystem has its own policy to evolve and this policy is tightly coupled with the nature of the developers that work in that ecosystem. Hou and Yao [12] explore the breaking changes in Java's AWT/Swing framework and find that these changes are limited due to the quality of the pre-existing architecture of the framework.

9 CONCLUSION

Java's deprecation mechanism is planned for change due to issues perceived by the Java language designers [16]: For the third time, the deprecation mechanism in Java is being revamped. Most mainstream languages offer a deprecation mechanism, but there is no uniform, universal support across languages. These facts witness that deprecation as a whole is an unsolved problem and that there is no clear understanding as to what constitutes an effective deprecation mechanism.

With this work, we aimed at empirically determining the developers' needs concerning deprecation. We did this by conducting a two-step study, involving an exploratory investigation and a validation with a large number of Java developers.

Our results show that API producers do not have any kind of preset protocol to remove deprecated features from their codebase, thus making the future of a deprecated feature ambiguous. API consumers, are more concerned with the reason behind deprecation as that proves to be the ultimate motivation to react. Based on these findings, we proposed two enhancements for the deprecation mechanism, namely, the indication of the version in which the deprecated feature will be removed, and different severity levels for different types of deprecation. Furthermore, to counteract the misuse of the deprecation mechanism, we also proposed to extend Java with a warning mechanism. These changes go beyond the proposed revamp of the Java deprecation mechanism, and we showed that the larger Java community would find these extensions valuable and at the same time, Java language designers found these changes to be challenging, yet feasible to implement. The knowledge that we accumulated in this study is not applicable to only Java, but also to other languages which may choose to alter the way in which they implement their deprecation mechanism.

With this paper we make the following *main contributions*:

- An understanding of why and how API producers use the deprecation mechanism and a catalog of reasons that motivate API consumers to react to deprecation, thus providing researchers and language designers with an in-depth understanding of required features of a deprecation mechanism.
- A proposal that enhances Java's deprecation mechanism whose feasibility and desirability is evaluated with the aid of two Java language designers and a survey with 170 respondents, showing that certain aspects of our proposal would be well received by the Java community.
- An analytical comparison between the deprecation mechanisms of 23 popular and new languages, that shows both practitioners and researchers the state of deprecation mechanisms and how they deviate from a mechanism that addresses additional developers' needs.

ACKNOWLEDGMENTS

The authors would like to thank all participants of the interviews and the survey, as well as the anonymous reviewers for their thorough feedback. A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In *24th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 109–120.

[2] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 2016. Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 360–369.

[3] 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*. ACM, 55.

[4] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 64–73.

[5] 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.

[6] Michael D Ernst. 2008. JSR 308: Type annotations specification. https://jcp.org/en/jsr/detail?id=308. (2008). last accessed May 2017.

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

[8] Adrian Furnham. 1986. Response bias, social desirability and dissimulation. *Personality and individual differences* 7, 3 (1986), 385–400.

[9] Nahid Golafshani. 2003. Understanding reliability and validity in qualitative research. *The qualitative report* 8, 4 (2003), 597–606.

[10] Johannes Henkel and Amer Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th international conference on Software engineering*. ACM, 274–283.

[11] Donald C Hildum and Roger W Brown. 1956. Verbal reinforcement and interviewer bias. *The Journal of Abnormal and Social Psychology* 53, 1 (1956), 108.

[12] Daqing Hou and Xiaojia Yao. 2011. Exploring the intent behind api evolution: A case study. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 131–140.

[13] TIOBE Index. [n. d.]. TIOBE.–2017.[Electronic resource]. *Mode of access http://www.tiobe.com/tiobe_index* ([n. d.]).

[14] 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. In *9th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 477–487.

[15] Thomas R Lindlof and Bryan C Taylor. 2011. *Qualitative communication research methods*. Sage.

[16] Stuart Marks. 2014–2017. JEP 277: Enhanced Deprecation. http://openjdk.java.net/jeps/277. (2014–2017). last accessed Aug 2017.

[17] Stuart Marks. 2016. Java One presentation on JEP 277. https://oracle.rainfocus.com/scripts/catalog/oow16.jsp?event=javaone&search=CON3297&search.event=javaone. (2016). last accessed May 2017.

[18] Microsoft. 2012. C# ObsoleteAttribute Class. https://msdn.microsoft.com/en-us/library/system.obsoleteattribute(v=vs.110).aspx. (2012). last accessed May 2017.

[19] Jakob Nielsen. 1995. 10 usability heuristics for user interface design. *Nielsen Norman Group* 1, 1 (1995).

[20] Jeff H Perkins. 2005. Automatically generating refactorings to support API evolution. In *ACM SIGSOFT Software Engineering Notes*, Vol. 31. ACM, 111–114.

[21] William Pugh. 2006. JSR 305: Annotations for software defect detection. https://jcp.org/en/jsr/detail?id=305. (2006). last accessed May 2017.

[22] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *Proceedings of hte IEEE 14th International Working Conference on Source Code Analysis and Manipulation*.

[23] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.

[24] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 56.

[25] Everett M Rogers. 2010. *Diffusion of innovations*. Simon and Schuster.

[26] John R. Rose. 1996. How and When To Deprecate APIs. http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase11-419415.html#7122-jdk-1.1-doc-oth-JPR. (1996). last accessed May 2017.

[27] A.A Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. 2017. Replication Package. https://www.dropbox.com/s/cwchbspdeek6iuh/replication_package.zip?dl=0. (2017). last accessed May 2017.

[28] 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* (2016), 1–24. https://doi.org/10.1007/s10664-016-9444-6

[29] A. A. Sawant and A. Bacchelli. 2016. On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs.. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE press, in press.

[30] LEE SIGELAMAN. 1981. Question-order effects on presidential popularity. *Public Opinion Quarterly* 45, 2 (1981), 199–207.

[31] Sally Thorne, S Reimer Kirkham, Janet MacDonald-Emes, et al. 1997. Focus on qualitative methods. Interpretive description: a noncategorical qualitative alternative for developing nursing knowledge. *Research in nursing & health* 20, 2 (1997), 169–177.

[32] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 138–147.

[33] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836.