The impact of API evolution on API consumers and how this can be affected by API producers and language designers

Sawant, Anand

DOI
10.4233/uuid:3d7bc400-2447-4a88-8768-3025d7b54b7f

Publication date
2019

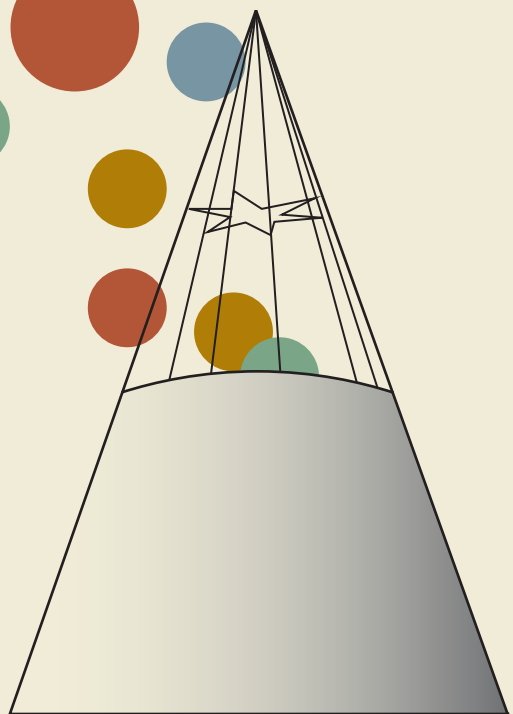
Document Version
Final published version

Citation (APA)
Sawant, A. (2019). *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:3d7bc400-2447-4a88-8768-3025d7b54b7f

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

The impact of API Evolution

on API consumers and how this can be affected by API producers and language designers



Anand Ashok Sawant

The impact of API evolution on API consumers and how this can be affected by API producers and language designers

The impact of API evolution on API consumers and how this can be affected by API producers and language designers

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University ofTechnology
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Thursday 10 October 2019 at 15:00 o'clock

by

Anand Ashok SAWANT

Master of Science in Computer Science,
Delft University of Technology, the Netherlands,
born in Mumbai, India.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof. dr. A. van Deursen,	Delft University of Technology, promotor
Prof. dr. A. Bacchelli,	University of Zurich, promotor
Onafhankelijke leden:	
Prof. dr. A. Bozzon,	Delft University of Technology
Prof. dr. C. Treude,	University of Adelaide, Australia
Prof. dr. D. Shepherd,	Virginia Commonwealth University, United States of America
Prof. dr. ir. A. Iosup,	Vrije Universiteit Amsterdam, the Netherlands
Prof. dr. M.F. Aniche,	Delft University of Technology
Prof. dr. E. Visser,	Delft University of Technology, reserve member

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords: API evolution, deprecation, API usage mining

Printed by: ProefscriptMaken, `www.proefscriptmaken.nl`

Cover: 'Evolution bubbles' by Aditya Parulekar

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

ISBN 978-94-6380-552-0

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

For, each man can do best and excel in only that thing of which he is passionately fond, in which he believes, as I do, that he has the ability to do it, that he is in fact born and destined to do it.

Homi J Bhabha

Contents

Summary

The practice of software engineering involves the combination of existing software components with new functionality to create new software. This is where an Application Programming Interface (API) comes in, an API is a definition of a set of functionality that can be reused by a developer to incorporate certain functionality in their codebase. Using an API can be challenging. For example, adopting a new API and correctly using the functionality can be challenging. One of the biggest issues with using an API, is that the API can evolve, with new features being added or existing features being modified or removed. Dealing with this challenge has led to an entire line of research on API evolution.

In this thesis, we seek to understand to what extent API evolution more specifically API deprecation affects API consumers and how API consumers deal with the changing API. API producers can impact consumer behavior by adopting specific deprecation policies, to uncover the nature of this relationship, we investigate how and why the API producer deprecates the API and how this impacts the consumer. Deprecation is a language feature, *i.e.*, one that language designers implement. Its implementation can vary across languages and thus the information that is conveyed by the deprecation mechanism can vary as well. The specific design decisions taken by the language designers can have a direct impact on consumer behavior when it comes to dealing with deprecation. We investigate the language designer perspective on deprecation and the impact of the design of a deprecation mechanism on the consumer. In this thesis, we investigate the relationship between API consumers, API producers, and language designers to understand how each has a role to play in reducing the burden of dealing with API evolution.

Our findings show that out of the projects that are affected by deprecation of API elements, only a minority react to the deprecation of an API element. Furthermore, out of this minority, an even smaller proportion reacts by replacing the deprecated element with the recommended replacement. A larger proportion of the projects prefer to rollback the version of the API that they use so that they are not affected by deprecation, another faction of projects is more willing to replace the API with the deprecated element with another API. API producers have a direct impact on this behavior with the deprecation policy of the API having a direct impact on the consumer's decision to react to deprecation. If the API producer is more likely to clean up their code *i.e.*, remove the deprecated element, then the consumers are likely to react to the deprecation of the element. This shows us that even for non-web-based APIs, the API producers can impact consumer behavior. We also, observe that the nature and content of the deprecation message can have an impact on consumer behavior. Consumers prefer to know when a deprecated feature is going to go away, what its replacement is and the reason behind the deprecation (informing them of the immediacy of reacting to the deprecation). The design of the deprecation mechanism needs to reflect these needs as the deprecation mechanism is the only direct way in which API producers can communicate with the consumer.

Samenvatting

De praktijk van software engineering omvat het combineren van bestaande softwarecomponenten met nieuwe functionaliteit om nieuwe software te maken. Hierbij wordt de notie van een Application Programming Interface (API) van belang: een interface voor vooraf gedefinieerde set functionaliteit, die door een ontwikkelaar kan worden hergebruikt om bepaalde functionaliteit in de codebase op te nemen. Het gebruik van een API is niet triviaal; het moeten aanpassen aan een nieuwe API, of het daadwerkelijk correct gebruiken van de functionaliteit van een API kunnen bijvoorbeeld uitdagingen zijn. Een van de grootste problemen bij het gebruik van een API is dat de API kan evolueren, waarbij nieuwe functies worden toegevoegd of bestaande functies worden gewijzigd of verwijderd. De uitdaging om hier goed mee om te gaan heeft geleid tot een volledige onderzoekslijn naar API-evolutie.

In dit proefschrift proberen we te begrijpen in hoeverre API-evolutie (meer specifiek, API-deprecatie) van invloed is op API-consumenten, en hoe API-consumenten omgaan met een veranderende API. API-producenten kunnen consumentengedrag beïnvloeden door een specifiek deprecatiebeleid te voeren. Om de aard van deze relatie te ontdekken, onderzoeken wij hoe en waarom de API-producent de API afschrijft en hoe dit de consument beïnvloedt. Afschrijving is een taalkenmerk: een kenmerk dat wordt geïmplementeerd door de ontwerpers van een taal. De implementatie ervan kan per taal verschillen. Als gevolg kan de informatie die door het deprecatiemechanisme wordt overgedragen ook variëren. De specifieke ontwerpbeslissingen van de taalontwerpers kunnen een directe invloed hebben op het consumentengedrag als het gaat om het omgaan met afschrijving. We onderzoeken het perspectief van de taalontwerper op deprecatie en de impact van het ontwerp van een deprecatiemechanisme op de consument. In dit proefschrift onderzoeken we de relatie tussen API-consumenten, API-producenten en taalontwerpers om te begrijpen hoe elk van deze partijen een rol kan spelen bij het verminderen van problemen in het omgaan met API-evolutie.

Onze bevindingen tonen aan dat van de projecten die worden beïnvloed door de deprecatie van API-elementen, slechts een minderheid op de deprecatie van het API-element reageert. Binnen deze minderheid reageert een nog kleiner deel van deze projecten door het verouderde element te vervangen door de aanbevolen vervanging. Een groter deel van de projecten geeft er echter de voorkeur aan om de versie van de gebruikte API terug te draaien, zodat ze niet worden beïnvloed door deprecatie. Een ander deel van de projecten is meer bereid de API te vervangen door een andere API. API-producenten hebben een directe invloed op dit gedrag, waarbij het deprecatiebeleid van de API een directe invloed heeft op de beslissing van de consument om op de deprecatie te reageren. Als de API-producent eerder geneigd is om de code op te schonen door het verouderde element te verwijderen, dan zullen de consumenten met grotere waarschijnlijkheid reageren op de deprecatie van het element. Dit laat ons zien dat zelfs voor niet-webgebaseerde API's, de API-producenten invloed kunnen hebben op consumentengedrag. We merken ook op

dat de aard en inhoud van het deprecatiebericht van invloed kan zijn op het gedrag van de consument. Consumenten willen bij voorkeur weten wanneer een verouderde functie verdwijnt, wat de vervanging ervan is, en de reden van de afschrijving (waarmee de consument geïnformeerd wordt over de onmiddellijke noodzaak om op de afschrijving te reageren). Het ontwerp van het deprecatiemechanisme moet deze behoeften weerspiegelen, aangezien het deprecatiemechanisme de enige rechstreekse manier is waarop API-producenten met de consument kunnen communiceren.

Acknowledgments

It has been ten years since I moved to the Netherlands and never in my life could I have imagined that I would leave India, move to a new country, learn Dutch and, do my undergrad at TU Delft. In all honesty, I thought that I would leave after my undergrad but I was convinced to stay a bit longer at TU to complete my masters. Again, I strongly contemplated leaving after the masters, but thanks to my master thesis advisor (and now PhD advisor) Alberto, I was convinced to stay on for another four years. After this nine-year-long stint (ten years if you count the Dutch course I did at TPM) at TU, I can say that I have achieved everything I could have, and I really have to thank the TU Delft for all the opportunities that it afforded me - a foreigner!

Right at the outset, I want to thank all the co-authors that I have had the pleasure of working with, all the other people in the academic circle that I have had the opportunity to talk to and all the people that I have met over the last ten years of living here in the Netherlands. I tried to name and personally thank as many people as I could. If I did miss out on someone it is not personal but simply an oversight and I thank you for being a part of my life anyway!

Alberto, thank you for offering me this opportunity to do a PhD at TU Delft. We have known each other since 2014 (since the beginning of my Master thesis) and I have learned a lot from you during this period. You pushed me to be the best version of myself and taught me the importance of perfection in every aspect of research such as the actual work, the paper writing and presenting the work. Thank you for always making time to discuss issues other than just my PhD research and being one of the few people that I rely on for advice. You also gave me a lot of freedom to pursue whatever research goal I wanted and never required that I conform to just your ideas. Thank you for never discouraging me from traveling and working with new people. Also, you gave me the opportunity to work with and co-supervise four masters students, something that has taught me a lot about the perspective of an advisor. You have always been the one that has forced me to think more critically about my work, to question why I was doing things and what impact that it could have, all this has helped me a lot over the years and I hope that this learning experience does not end now that the PhD is over. I owe you a lot for this start to my research career!

Arie, thank you for being my promotor and my sounding board for any issues that I had during my PhD. You have always been a tempering presence, which has helped me to learn how to rationally approach difficult situations. One quote (among many) I will always remember is :“Never say never if you are good enough, no door is closed to you”. You have also taught me to always think about the big picture, and not just from a research perspective but also a funding perspective, and that has made me learn how to sell the impact of my research in such a way that someone will want to fund it. Thank you for all the nuggets of wisdom that you have dropped on me over the years (some which took me a long time to understand) and thank you for making me a part of the SERG group!

Mauricio, thank you and Laura (who is like the big sister I never had) for being close friends and a sounding board for every stupid (and occasional good) idea that I may have had. You taught me a lot about the developers perspective, something that stems from your real-world experience. Aside from all the discussions on research, you always pushed me to be better in my personal life and acted as my life coach (read as wingman). Thank you for taking me to Brazil and being our local tour guide! I would like to stress again that no murderer was lurking behind the bushes when we stopped for a break from the driving at 1AM!

Annibale, I have gotten to know you very well since your first stint as a postdoc and must say that I was very happy to see you back at TU. I love all the discussions that we have had over the years on research ethics, authorship, and integrity; long may these discussions continue! Thanks for teaching me appreciate real Italian pizza, now I cannot eat any other kind of pizza without being supercritical. I hope to continue working with you in the future (if only to take our beer sessions to the max)!

Davide, you have been one of my closest confidants and friends at TU! I will always love going to restaurants with you...if only to see if I can out-eat you! Thanks to having a common advisor, we have traveled a lot together (Japan, Argentina, Brazil, USA and assortment of places in Europe), and it has always been fun with you and with Giada! Thank you for bringing Giada with you to the Netherlands, that allowed me to eat her heavenly tiramisu. Thanks for teaching me all the Italian curse words that will probably get me killed in Italy someday. While this friendship has lasted for four years, I hope to hang out with you and Giada all over the world in the future as well!

Luca, thank you for being my go-to guy on traveling to Italy! You and Claudia helped me realize my ten-year-long dream of seeing the Amalfi coast. You are by far the best maker of pizza (known as Pizza Czar in Brooklyn) that I know! Long live your pizza place (whose name I will not disclose here)! You, Davide and I have traveled to a bunch of countries together, shared hotel rooms or AirBnBs, and honestly, all these memories that I will take away from my PhD experience. I know that initially, you were scared of talking to me due to the language barrier (especially with me talking as fast as a freight train going downhill), but today we can talk to each other all the time!

Vladimir, I have always enjoyed the debates with you! The frequent discussions on relevant research in SE and the impact of industry have always been stimulating and have pushed me to do better in my work. You also taught me to not always hold back on my real opinion in the fear of being considered as a too direct a person!

Xavier, it has been nice to get to know you over the last year and a half of my PhD. It has been fun frequently getting beers and dinner together and talking about research and ethical issues! I look forward to continuing this trend every time we meet!

Dave, thanks for the opportunity to conduct research in an industrial setting. I learned a lot from that experience and it taught me about doing more relevant or impactful research! You have been one of the coolest and nicest people that I have had the pleasure of working with during my PhD and hopefully we do so again the future!

Andy, with Alberto having left for Zurich, it has always been nice to have someone physically present to discuss things other than research. You always made time for me, and never really turned me down despite not being my promotor and despite having an incredibly busy schedule. It always helped to get your thoughts on issues ranging from

authorship to my career, thank you!

Romain, thank you for being the first person that I collaborated with! We have worked to together on plenty of papers since, but the first paper together is what helped me start my entire line of work on deprecation and API evolution. Thanks for hosting me in Bolzano for two weeks as well I just wish that I had visited you when you were still in Chile!

Marco, you and I have done our PhDs for almost the same time, and thus have gotten to know each other well, thank you for introducing me to Taralli and always bringing goodies back for me from Italy! Alaaeddin, thanks for being my spirit guide during this final phase of the PhD, I have been able to leach off you and your experience in navigating all the TU Delft bureaucracy. Cynthia, the baby dinosaur of TU Delft, it has been great listening to stories about TU Delft, and the reminiscing about the shared experiences (since you only predate me by a few years) at TU! (since you Pouria, Jean, Joseph, Mehdi, Vivek, Enrique, Ayushi and Luis, while my tenure in this group did not entirely overlap with yours, we have had the opportunity to get to know each other over all the coffee breaks and lunch (in the case of Jean eating an entire cow at the Brazilian place), I hope you guys continue the social nature of this group!

I would also like to take this opportunity to thank Achyudh, Jorden, Fernando, and Dereck for being great Master students that I have had the please to work with and publish with! I hope the best for all your careers and I hope you keep in touch!

To my close friends that have stuck with me through the undergrad, masters and then life: Vincent, Reinier, Kaj, Lisette and Jolanda, I hope that we continue meeting at least once a year to do an escape room and then drink whiskey! Without you guys, I would not have known anyone at TU Delft nine years ago and it probably would have made my time at TU hell, but I got lucky to meet you guys who agreed to speak in English with me. We have done a bunch of courses together, skipped more classes than I can remember and worked together on more projects than I care to count. Also, I will never forget the island with its millions of birds, pitchforks and the look of wonder on islanders faces on seeing me.

To my Indian family here in the Netherlands (Madhavi maushi, Manoj uncle, Amit mama, Sheela mami, Vrinda maushi, and Girish kaka), I want to thank you for allowing me to feel Indian on the inside with all the amazing Indian food that you always make and treat me with. Your presence in this country has made it easier for me to transition from my Bangalore life to Dutch life. Thanks for all the Diwalis, Ganesh Chaturthis and assortment of dinner nights, each made me feel like I was back home and not in a foreign country. To Aditya, Mehul and Mohit, it has always been fun hanging out, hopefully, it does not take us another year to plan a Korean bbq dinner!

To all the Indian friends that I have here, I want to thank you for making the Netherlands feel like home! You are just as close as family to me and it has been a pleasure living in this country for the last ten years because of all of you!

Finally, last but definitely not least I want to thank my mom and dad. Dad, when you took the job with Unilever in the Netherlands ten years ago, I do not think we could have ever imagined that I would end up following in your footsteps and doing a PhD. Thank you for making the decision to move to the Netherlands, it has opened more doors for me than I can imagine and given me more opportunities than I could have thought of. Mom,

you have always been my emotional bedrock, thank you for always being there for me whenever I have needed you. I know you always miss me when I travel (something I have a done a lot in the last four years) and that I do not message or call as often as I should, I am going to try to improve on this so that at the very least you know that I am alive! Thank you to both of you for the life and education that you have provided me with!

Anand
Delft, October 2019

1

Introduction

The discipline of Software Engineering revolves around the reuse of pre-existing functionality combined with the development of new features to produce a piece of software. This is where Application Programming Interfaces (APIs) come in. An API is a definition of functionality that a developer can reuse within their code. APIs aim to make the entire development process smoother and eliminate the need to reinvent the wheel. However, using an API comes with its own set of challenges. Adopting API features can often prove to be tricky and the incorrect usage of APIs can introduce bugs in the API consumer's code. The changing of an API can have an adverse impact on the consumer code, as this would require the consumer to learn a new interface and go through the aforementioned adoption cycle all over again. These circumstances have led to a line of research on API evolution.

In this thesis, we deal with API deprecation, which is a sub-case of API evolution. When an API producer deprecates an API element, it indicates that this element is now obsolete and should no longer be used by the consumer. This is often the precursor to the removal of this element from the API, thus providing the consumer with some time to transition away from the element. We study the impact of API evolution on a consumer and how an API producer and language designers can help to keep the cost of dealing with API evolution at a minimum.

1

Application Programming Interfaces (APIs) are as close to a "silver bullet" as we have found in Software Engineering [1]. Brooks acknowledges as much while revisiting "The Mythical Man-Month" after two decades [2]. APIs provide a contract that defines a set of reusable functions, actions, and communication protocols that can be integrated directly by a developer. APIs are not so dissimilar to traditional software systems, APIs evolve and this evolution can have a large impact on the projects that depend on it [3, 4].

Keeping up with the evolution of software is costly. According to Lehman [5], almost 70% of the developmental cost is focused on software maintenance to keep up with evolving software systems. This figure has been reiterated in 2003 by Grubb and Takang [6]. In actual cost terms, the Dutch government spends 3.5 billion euros on software costs, which implies that roughly 2.45 billion euros are spent on just maintaining software. This large cost highlights the need for a more cost-efficient way of dealing with software evolution.

Software evolution is necessitated by the ever-changing requirements placed on developers. This is similar in the case of APIs where the interface or contract between systems can evolve in three different ways: (1) an existing API element can be removed from the API as it is no longer needed or because its implementation is defective, (2) an API element's signature can be changed because the original signature did not fulfill all demands, (3) an API element's behavior can be changed due to the presence of a bug in the original behavior.

API evolution (changes made to an API) can have a widespread impact. In 2016, the `npm` ecosystem was severely crippled as one developer removed the `leftpad` package which had been downloaded 2,486,696 times. APIs can evolve for serious reasons, for example: in 2014 it was discovered that the OpenSSL library suffered from a serious vulnerability that compromised over 17% of the world's servers. This bug, referred to as the Heartbleed bug, was introduced in 2012 and publicly disclosed in 2014. It was patched immediately after discovery, however, projects that used OpenSSL had to be made aware of the vulnerability and then forced to change how they interacted with the API as the semantics of the API element had changed. If some projects did not react to the API evolution, their code would be vulnerable to attack.

Research has focused on understanding the impact of an APIs' evolution (specifically breaking changes introduced in the API) on its consumers. Wu *et al.* [7] analyzed the Eclipse ecosystem to see how an API change would affect its consumers. They found that 11% of API changes produce a ripple effect; meaning that, apart from the projects that depend on the Eclipse API being affected, projects that depend on these aforementioned affected projects would also be affected, thus causing a ripple in the ecosystem. From this study, we have evidence that the impact of an API change reached beyond just the immediate dependent project. Laerte *et al.* [8, 9] found that for a median library 14.78% of changes are breaking changes, whose frequency only increases over time. However, only a minority of projects are affected by these changes. This could be because APIs introduce breaking changes by taking the client's usage into account. Bogart *et al.* [4] analyzed the introduction of breaking changes in the npm, Eclipse and R/CRAN ecosystems. They found that the policies adopted by the API can vary across ecosystems, in Eclipse breaking changes are avoided, in R/CRAN the consumers are directly contacted so that they can fix the issue and in npm, the major version of the API is simply incremented. Which shows that in certain ecosystems, the API developers care about introducing a breaking change,

but this is not always the case.

As an alternative to directly introducing breaking changes in the API, programming language designers provide a *deprecation* mechanism. Deprecation is a precursor to a breaking change being introduced in the API, whereby API developers can indicate that a feature is obsolete and will be removed in a future release of the API. API developers can indicate in the documentation as to how a project using the API should replace a deprecated API element. Software development tools also provide extensive support for the deprecation mechanism: compilers emit warnings when deprecated code is used [10] and IDEs (e.g. Eclipse [11]) visualize the usages of deprecated methods by putting a strike through the call point of the method.

Since deprecation of an API element occurs before its actual removal *i.e.*, the introduction of a breaking change, we can study from the API consumer perspective as to what the decision-making process is on the consumer side when it comes to reacting to API evolution. In the case of a breaking change, the consumer has no option but to react to the API change, however, with deprecation, consumers can take their time to decide on making a change. This allows us to determine at what point the consumer decides to change, how the consumer makes the change in the codebase (*i.e.*, does the consumer use the recommended replacement) and determine other factors that could influence a consumer when it comes to reacting to API evolution. This can better inform a solution that targets specific pain points associated with keeping up API evolution and over time bring down the cost of API consumer code maintenance.

In this Ph.D. thesis, we investigate the impact of API evolution through the lens of deprecation of API features, from two different perspectives:

- **API consumer.** These are the developers that depend on an API's features. The projects developed/maintained by these developers are directly affected by API evolution. These developers decide whether and how to keep up with the evolution of the API.

- **API producer.** These developers actively develop and maintain the API. They are responsible for evolving the API.

In addition to studying the API producer and consumer perspective, we analyze as to how programming language design can have an impact on the API consumer behavior regarding API evolution. Language designers can introduce ways in which the burden of dealing with API evolution is lessened on the consumer. For example, Java's Project Jigsaw allows API producers to divide large APIs into smaller logically connected segments, thus narrowing the search space for the consumer when it comes to selecting the appropriate API element to use and include in their project [12].

Several existing studies on API deprecation have shown that both consumers and producers may not behave as expected when it comes to the deprecation mechanism. The reaction of the consumers may be overdue or not happen [3, 13, 14]; also, the API producer may not provide clear instructions for replacement or even fail to provide a rationale for the deprecation [15–17]. Producers may eschew from removing deprecated methods from the API to retain backward compatibility or, oppositely, remove API elements without first deprecating them [18]. They may do so between major versions, or, breaking seman-

1

tic versioning practices, do it between minor versions of the APIs [19]. Certain deprecation policies adopted by producers might have an adverse impact on the consumers [14].

We have seen that similar to traditional software systems, APIs evolve as well. API evolution can have a direct impact on the consumers that depend on the API and there is a need to ease the burden of dealing with API evolution. However, there is a lack of a thorough understanding of how consumers choose to deal with API evolution and why they make certain choices. This leaves dealing with API evolution as an open question. Furthermore, the influence of the API producers and language designers on consumer behavior remains unexplored. This brings us to our central thesis:

> By knowing the impact of API evolution on consumers, we can ascertain the role that API producers and language designers can play in keeping the impact of API evolution at a minimum.

In this introductory chapter, we first present background on API evolution, followed by API deprecation in Java and its history. We then present an outline the research goal and questions answered in this thesis and then we outline the various chapters in the thesis. Finally, we describe the methodology utilized in this thesis and round off by describing the various contributions made during this Ph.D.

1.1 API Evolution

In 1969 Meir M. Lehman at IBM [20] found that developers spend the majority of their time performing software maintenance and dealing with software evolution. In a follow-up study, Lehman identified that 70% of the developmental cost was spent on dealing with software evolution. Lehman and Belady [21] proceeded to define the laws of software evolution (known as Lehman's Laws of Software Evolution). The eight laws defined by Lehman apply to E-type (systems that perform some real-world behavior) software systems. APIs, which are software systems that expose reusable functionality, also adhere to the same eight laws.

While software can evolve to meet increasing demands or needs, it can also decay. David Parnas first spoke about software aging in 1994 [22], where he stated that over time the design of the software can become obsolete. Reasons from this obsolescence vary from developers patching every bug in the system using unstructured methods to the incorrect upgrade of certain sections of the system that causes failures. Analog to this, API design can increasingly become obsolete over time.

Being software systems, APIs evolve and this phenomenon is referred to as API evolution. An API can evolve due to a large number of factors, including: (1) there is a better/-faster implementation of a feature, (2) the existing feature has a security or performance issue, (3) the API would like to use a standardized design pattern and (4) new language features have been released.

From the API consumer perspective, it is not ideal (except in certain extenuating circumstances, such as when an API element changes due to the fixing of a functional defect) that an API they are using changes. If a feature changes or is removed, the consumer code can break, thereby necessitating an unforeseen/unplanned maintenance effort on the part of the consumer. When consumers would like to upgrade to a new version of the API, their code may no longer compile due to the API change.

An example of this can be seen in Figure 1.1, Here we see an example from Guava where the return type of an API element in Guava is changed from the superclass (`List`) to its sub-class (`ArrayList`). For the consumers who start using version $n+1$ (seen in Figure 1.2), their code will not compile with older versions of the API. On the other hand, consumers that use version n of the API (seen in Figure 1.3) and then choose to upgrade to version $n+1$, would not be able to do so without making a change first as the return type class is a sub-class and thus needs to be explicitly declared (this is due to a failure of the Java type inference system).

```
// Method in version n of an API
public static List<Integer> newList () {
    …
}

// Method in version n+1 of an API
public static ArrayList<Integer> newList () {
    …
}
```

Figure 1.1: Example breaking change in an API

```
// Consumer call for version n of an API
List<Integer> myList = newList();

// Consumer call for version n+1 of an API
ArrayList<Integer> myList = newList();
```

Figure 1.2: Example call to API element by consumer

```
// Consumer call for version n of an API
Set<List<Integer>> myLists = ImmutableSet.of(newList());

// Consumer call for version n+1 of an API
Set<ArrayList<Integer>> myLists = ImmutableSet.of(newList());
```

Figure 1.3: Example call to API element by consumer when using parameterized data structure

The primary reason that API consumers choose not to upgrade the version of the API being used is the cost of the upgrade [23, 24]. Research has investigated how the burden to upgrade the version of the API can be lessened [25–29]. However, as of now, there is no accepted solution in place just yet.

1

1.2 API Deprecation

As an alternative to directly introducing breaking changes in the API, API producers can use deprecation. This leads to an API→introduce→deprecate→remove cycle where before any removal of an API feature takes place, it is first deprecated to allow consumers time to wean themselves off the feature with greater control of their scheduling [10].

Deprecation is a mechanism provided by most programming languages [17]. This is in stark contrast to breaking API changes where consumers have no decision-making time when encountered with an API element that no longer exists.

In the case of deprecation, the language mechanism enables the producer to communicate additional information to the consumer, which could aid the consumer in making a transition away from a deprecated feature. As researchers, the usage of the deprecation mechanism allows us to observe the decision-making process on the consumer side when it comes to reacting to API evolution, see how such a reaction takes place, and understand the factors that aid the consumer in making a change.

1.2.1 Deprecation in Java

In this thesis, we focus on deprecation in the Java language. We focus primarily on Java as it was the first language to introduce an explicit deprecation mechanism in 1995, and it is the second most popular language in the world (thus ensuring us an abundance of data).

In the original documentation of deprecation for Java 1.1, we read that: 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" [10]. Not all of these reasons may have an equal weight *e.g.*, it is reasonable to assume that when a feature is deprecated as it is insecure, it is pivotal that the consumer reacts to this change, however, in the case that feature is superseded by something newer then the need to react might not be as pressing.

The deprecation mechanism in Java was first introduced in the form of a Javadoc `@deprecated` annotation in Java 1.1. This annotation indicates in the Javadoc that a feature is deprecated. Additionally, in the Sun JDK, the compiler would throw a warning on encountering the usage of a feature that was deprecated using this annotation, this is despite this behavior not being specified in the Java Language Specification (JLS). Once source code annotations were introduced in Java 1.5, Java introduced a `@Deprecated` annotation. The Java language designers intended that both the Javadoc annotations and the source code annotation to be used in tandem to mark an API feature as deprecated both in source code and documentation [30]. The advantage of this source code annotation was that, as standard behavior, the compiler throws a warning every time it encounters the usage of a feature annotated with this annotation.

1.2.2 Deprecation in other languages

There is no standard convention of providing the deprecation mechanism across programming languages. For instance, deprecation messages are conveyed to the API consumer in some cases as a part of the implementation of the deprecation mechanism itself (as in the case of Java or C#) or in other cases in the form of additional documentation (in the case of Python). The functionality of the deprecation mechanism varies too: For example, in C#, the deprecation mechanism is exposed in the form of a class attribute and can have two

equals

```
@Deprecated
public boolean equals(String s)
```

Deprecated. *As inconsistent with `hashCode()` contract, use `isMimeTypeEqual(String)` instead.*

Compares only the `mimeType` against the passed in `String` and `representationClass` is not considered in the comparison. If `representationClass` needs to be compared, then `equals(new DataFlavor(s))` may be used.

Parameters:

`s - the mimeType to compare.`

Returns:

`true if the String (MimeType) is equal; false otherwise or if s is null`

Figure 1.4: Example Javadoc for deprecated API element

levels (in the first level the compiler only throws a warning when a deprecated feature is used and in the second level it throws a compilation error). However, in languages such as Java and Scala only a compiler warning is thrown, thus ensuring that deprecation is not a breaking change.

1.2.3 Documentation of deprecated API elements

When an API element is marked as deprecated, it is considered good practice to provide deprecation messages that act as an aid in the reaction to a deprecated feature [31]. This 'good' practice entails that a deprecation message should recommend a replacement of this deprecated feature, however, this might not always be the case [15]. This is generally done in Javadoc as seen in Figure 1.4; modern-day IDEs also show this warning as in Figure 1.5.

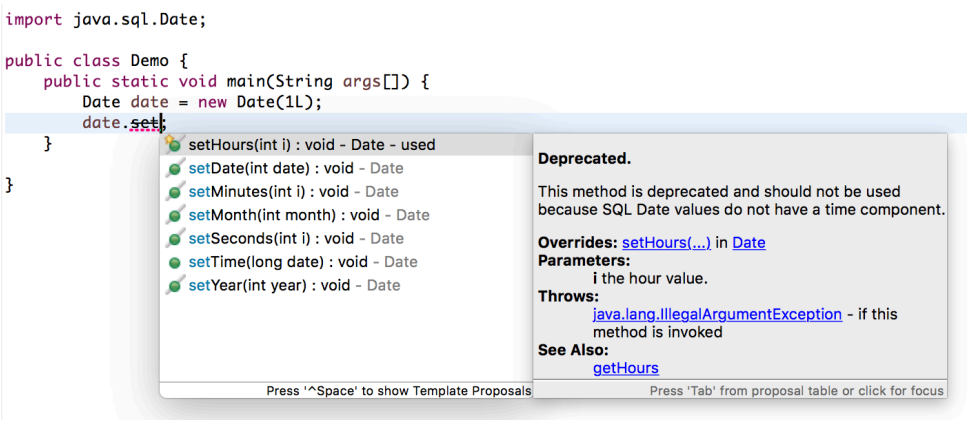


Figure 1.5: Example IDE warning for deprecated API element

1.3 Research goal and questions

To provide evidence towards our thesis, we seek to answer the following four research questions:

> **Research Question 1.** *How are API consumers affected by deprecation of an API element?*

Not all API consumers are affected by API deprecation. We would like to understand the extent to which consumers are affected by deprecation. This will allow us to ascertain how frequently API consumers are affected by this type of API evolution, which will give us an insight into how much we can understand from it. This also helps us in determining various API evolution policies adopted by the API producers and the impact that this can have on the magnitude of consumers affected by API evolution.

> **Research Question 2.** *How and why do API consumers (not) react to deprecation of an API element?*

We would like to understand to what extent API consumers react to the deprecation of an API element. This allows us to ascertain how consumers regard an APIs evolution and whether they keep up with it and to gain a better understanding of how an API's evolution pattern can impact consumers.

For the API consumers that react to deprecation, we would like to understand as to what motivated them to react in the way that they did. We would be able to isolate the factors that impacted their decision to react to deprecation and this can be used as a template for future API evolutions. For those consumers that did not react to deprecation, we strive to ascertain as to what about the deprecated element prevented them from reacting. This gives us insights into the API consumer decision process when it comes to dealing with API evolution. It will allow us to observe the pain points that API consumers face

when reacting to API evolution. This information can aid us in devising strategies to deal with API evolution more effectively.

> **Research Question 3.** *How do API producers support API consumers when reacting to deprecation?*

API consumers require assistance when adopting new features of an API [32]. Deprecation, which indicates that an API element is obsolete, allows the API producers to inform the consumer over how they should transition away from it. API producers could also provide detailed transition guides or tooling support. With this question, we would be able to ascertain as to how producers choose to support consumers and to what extent this has an impact on the consumers.

> **Research Question 4.** *Can language designers improve the deprecation mechanism for both API producers and consumers?*

The deprecation mechanism acts as a means of communication between the API producer and the consumer, where the producer informs the consumer that a feature is obsolete. The behavior of this mechanism can have an impact on an API consumer's choice to react to deprecation. We would like to investigate whether this mechanism fulfills the needs of both API producers and consumers, and how this compares across programming languages.

After answering these four research questions, we will be able to establish how API consumers are affected by API evolution (in this specific case API deprecation), how they choose to react when encountered with a deprecated API element, and what factors affect this behavior. We will be able to understand the role that API producers and language designers can play in reducing the burden of reacting to API evolution. This allows us to answer the central question in this thesis, and allow us to design specific solutions that will aid consumers to keep up with API evolution.

1.4 Research outline

Table 1.1 illustrates the connection between the various chapters in this thesis and the research questions that we have defined.

1.4.1 Background

In the first part of the thesis, we discuss the overall research goal and the fine-grained research questions that this thesis answers. Following this, we provide background information on how API consumer usage is mined from GitHub on a large scale. Finally, we ascertain the degree to which API consumers are affected by API evolution.

Chapter 2 outlines the technique used to mine API usage from GitHub based Java projects. We call this technique fine-GRAPE (fine-Grained APi usage Extractor). To perform any study that seeks to understand how API consumers deal with API evolution, we need to understand how the API is being used by a client. Previous work

1

Table 1.1: Relation between research questions and chapters

Research question	Chapters
How are API consumers affected by deprecation of an API element?	3 and 4
How and why do (not) API consumers react to deprecation of an API element?	5
How can API producers support API consumers when reacting to deprecation?	6
How can language designers improve the deprecation mechanism for both API producers and consumers?	7

has sought to mine API usage from open source repositories such as Sourceforge and the Eclipse repository. The techniques used range from text mining to bytecode mining. All of these approaches suffer from issues that range from inaccuracy (the API usages mined are not type resolved thus resulting in inaccuracies) to scalability (compilation of projects is expensive and impossible to do on a large scale). To overcome these shortfalls, we defined a new approach called fine-GRAPE, which can type resolve the AST of the source code file and extract API usages on a large scale from Java-based consumers on GitHub. We apply this approach to the consumers of APIs to understand what features of an API are used and how they are used. This work forms a basis for the rest of the thesis, where data mined using this technique is used.

Chapter 3 describes two studies performed on large scale data mined from the consumers of five mainstream Java APIs. The first study deals with understanding whether consumers update the version of the API being used. This aids us in understanding what proportion of consumers can be affected by API evolution. We show that consumers typically do not upgrade to the latest version of the API. In our dataset, we see that consumers of frequently releasing APIs tend to not upgrade their dependency version. In the second study, we see what features and how much of the API is being used. This allows us to establish whether the features being used are the same across consumers. Our data shows that only 5 - 10% of the features exposed by an API are used by consumers. This implies that large portions of the API are never adopted by consumers on GitHub. Furthermore, we see that predominantly the original features of the API (core features) have been adopted by consumers, which is startling as it suggests that API producer effort is futile.

1.4.2 The perspective of the consumer

The second part of the thesis deals with the API consumers' perspective. We analyze the scale at which API consumers are affected by deprecation and how these consumers react to deprecation. We investigate the motivation behind the consumers not reacting to deprecation and how API deprecation policies can affect this.

Chapter 4 deals with the scale at which API consumers are affected by deprecation and how this is affected by an APIs deprecation policy. We analyze the consumers of five mainstream APIs - Guava, Guice, Spring, Hibernate and Mockito and the Java

JDK. We focus on consumers that change versions of the API they use, as these consumers are the only ones that can be affected by deprecation or API evolution. Only in the case of Guava are more than 20% of the consumers affected by deprecation. In the case of the Spring API, we find that the scale of affectedness of consumers is very small and insignificant. We see from the API producer perspective that only a maximum of 13% of deprecated elements affect consumers. This implies that 87% of the deprecated features are not used by consumers. We also fast-forwarded consumers to the latest version of the API to see whether they would be affected by deprecation and found that for all APIs over 30% of the consumers would be affected by deprecation. In the case of one consumer from Hibernate, the consumers would have to change over 9,000 invocations in their source code. We also make a lightweight analysis to see whether consumers react and how long it takes them to do so. The median time to react is 200 days and only 2 consumers react by actually replacing the deprecated feature with the recommended replacement. In the case of the consumers of the JDK, we see that consumers are not affected by deprecation at the same scale as that of the consumers of the third-party APIs. Finally, we try to ascertain whether the deprecation policy adhered to by the API influences the scale at which consumers of the API are affected. We determine the APIs policy based on 9 characteristics. Using a k-means clustering algorithm we cluster 50 APIs to based on the 9 characteristics and find 7 distinct policies. We see that the policy of the API has a direct influence on the scale at which consumers are affected, thus, implying that API producers can play a role in how API consumers are affected by API evolution.

Chapter 5 explores how an API consumer reacts to a deprecated feature and how API deprecation policies affect the chosen reaction. This helps us in gaining further insight into the decision process on the consumers side when it comes to dealing with API evolution. We focus on the top 50 Java APIs and their consumers for this study. We start by manually analyzing 380 cases of API consumer code affected by deprecation. This manual analysis yields seven different reaction patterns that can take place. We then proceed to benchmark the frequency of each of the reaction patterns and find that 'no reaction' is by far the most popular way in which consumers choose to react to deprecation. We survey consumer to understand why they choose to not upgrade their dependency and why they do not react to deprecation. Generally, consumers do not upgrade due to the cost of upgrading, typically deprecation does not hinder the upgrade process. To explain the non-reactions, we try to see how the APIs deprecation policy affects the consumers. We base the deprecation policy on the activeness of the API, frequency of removal of deprecated features, the frequency of deprecating features and the frequency with which features are broken. We find nine distinct policies adopted by API producers. In most cases, the deprecation strategies are not strongly linked with the reaction patterns exhibited by the consumers. To further investigate the scale of non-reactions, we ask API consumers as to why they do not react to deprecation. Typically, consumers do not react because they could not find a suitable alternative to the deprecated feature and because the cost of reacting was often too high and not worth it. Also, in the cases where the API releases infrequently, the need to react is not pressing. Overall, we

see that consumers do not react to deprecation due to lack of help that has been provided to them that would reduce the overall cost and effort related to reacting to deprecation.

1.4.3 The perspective of the producer and the language designer

In the third part of the thesis, we would like to ascertain from the API producer perspective whether they would like to improve the entire process of deprecation in any way. We investigate whether API producers provide their consumers with all the help they need to react to the deprecation. Finally, we see what the shortfalls of the Java deprecation mechanism are and how we can address them.

Chapter 6 analyzes to what extent the rationale behind the deprecation of a feature is communicated with the API consumer. This goes to show how API producers support their consumers when it comes to dealing with API evolution. Studies have shown that API producers typically document deprecated features with the replacement feature that the consumer can use. However, the rationale behind the deprecation is rarely found in this documentation. We observe that uncovering the rationale behind deprecation is not as straight forward as it would seem. To find the rationale, an API consumer would have to look at either the commit message of the commit where the feature is deprecated, the issue tracker post which is addressed in the commit which deprecates a feature and the code itself. We manually analyze 380 deprecations from five APIs and find that there are ten different reasons behind deprecating a feature. Some of the reasons to deprecate a feature are off-label usage of the deprecation mechanism. This prompts the question as to how a programming language can prevent this behavior. We then try to see whether there is an automated way to classify the reason behind deprecation by using one or more of the data sources at our disposal. We see that typically for one API, an automated approach can classify the rationale behind deprecation in an accurate manner. However, when trying to cross-project validate this model, the accuracy suffers. We postulate that an automated method can be devised to uncover the rationale behind deprecation and automatically augment existing documentation for deprecated features.

Chapter 7 investigates the deprecation mechanism in Java and whether it fulfills the needs of both API producers and consumers. With this chapter, we gain a thorough understanding of how language designers can play a role in enabling consumers to deal with API evolution. We interview 17 API producers from both open source and industrial contexts to understand how they perceive the deprecation mechanism in Java. In their opinion, the deprecation mechanism is a communication medium between the API producer and the consumer. They explained that they there seven reasons for them to deprecate a feature, with marking a feature as beta being the only standout. Producers do not feel that it is always imperative that their consumers react to deprecation. However, there are certain instances where a reaction is needed, but Java does not allow the producers to indicate this. We surveyed consumers to understand what they felt were the shortfalls of the deprecation mechanism. Consumers indicate that they also needed to know as to when a deprecated feature was

going to be removed so that they could then choose to react. All these findings led us to propose three enhancements to the deprecation mechanism - (1) a way to indicate removal timeline of a deprecated feature, (2) a generic warning mechanism to prevent off-label usage of deprecation and (3) a severity indicator for the deprecation.We validate these enhancements with the Java language designers to see whether Java would benefit from these changes. The language designers support two out of the three proposals, they did not feel that the generic warning mechanism would be an ideal solution. However, there was support for the other two proposals.

1.4.4 Reflection

We conclude the thesis summarizing the findings and proposing future work.

Chapter 8 summarizes the findings in this thesis. We discuss the implications of this thesis and elaborate on future work that can be conducted to take the work done in this thesis forward.

1.5 Research methodology

The work in this thesis belongs to the discipline of Empirical Software Engineering [33]. This field aims to provide analytical insights into the software development process and suggest improvements in the form of tooling or practices by gathering data on open-source software systems and development practices by mining software repositories, interviewing developers, surveying developers and performing controlled experiments.

In this thesis, we employ a *mixed-method approach* [34] to answer our research questions. We explain the various techniques used and their applicability in the following:

1.5.1 Mining software repositories

In the last 15 years, Empirical Software Engineering research has often involved the analysis data from software repositories [33]. This line of research has been boosted by the advent of platforms such as GitHub, which have allowed for the analysis of source code on a large scale, and the interaction between developers during the development process due to the presence of a public issue tracker and pull request system. Which in turn has led to the creation of large scale analytics platforms that provide in-depth insights to developers.

In this thesis, we leverage techniques from the mining software repositories field to develop an understanding of what features of an API are used and how they are used. We can isolate active, non-forked, Maven-based, Java repositories on GitHub to understand how developers deal with API evolution and whether they upgrade the version of the API that they use. We target a broad set of actively developed Java projects for this analysis, thus allowing us to get an accurate impression of developer behavior on a large scale. Another advantage of using data from GitHub is that we have fine-grained commit information, thus allowing us to analyze the entire history of every project. Chapter 2 outlines the exact technique developed to achieve this. In chapters 3, 4 and 5, data collected using this technique has been used for the studies conducted in those chapters.

Aside from GitHub, the Maven central repository is a rich source of data. Over a million Java JAR files are hosted on the central Maven repository. Each JAR file relates to

1

a version of a Java-based API/project that has been released on Maven central. While this data does not possess the fine-grained commit level information as in the case of GitHub, it does contain compiled source code that has been developed and released by professional Java-based developers. Chapter 4 uses the data collected using this methodology.

1.5.2 Interviews with developers

Mining data from open source repositories is one way to understand what and how developers are doing things. Understanding *why* developers make certain choices is another matter altogether. Commit messages, issue tracker discussions and pull request data can all give us an indication to a certain extent as to what the decision-making process of developers is and why certain choices were made. However, these sources can often be unreliable as the rationale behind a change might not be documented [35].

Interviewing developers allows us to infer from a developer as to why a certain change has been made. We can also ask developers for their opinions on certain behavior and understand in depth the decision-making process behind a change. One pitfall for this approach is that our interviewees provide socially desirable responses which are not in sync with reality. We mitigate this bias by interviewing a diverse set of developers who work in different contexts. To explore several possible avenues, we only stop interviews once we hit saturation [36] *i.e.*, when we hear the same responses to our questions without uncovering any new perspectives.

Once the interviews have been conducted, we leverage techniques from grounded theory [37] to analyze the interview transcripts. Specifically, we use an interpretive-description [38] approach which originates from the social sciences. This is an inductive approach to analyze interview transcripts by breaking each part of the transcript into smaller parts and assigning codes to each part based on content. Codes are then clustered based on similarity, allowing us to infer the emergent themes from all the interviews. Based on these themes, we can define a theory which leads us to results found in Chapter 7.

1.5.3 Surveying developers

Data from open source repositories and interviews with developers provide us a view of what is going on in the code, what decisions have been taken by developers and why. However, the themes that emerge from the combination of this data might not be *generalizable*. To challenge and validate our findings we reach out to a larger set of developers and ask them to confirm our findings.

Surveys help with generalizing results as they can reach a broader audience of developers. Developers can either confirm phenomena that we uncover using open source data or qualitative data, however, in the cases that this data is not exhaustive, developers that respond to the survey can augment our existing knowledge with a new perspective. This can allow generalizing to a larger set of developers aside from only our interviewees.

In this thesis, we employ surveys to ask developers to rate their opinion on a variety of themes that emerge from qualitative and quantitative data. We ask developers to indicate on a Likert scale their agreement with certain statements. If a list of statements is not exhaustive, we ask the developer to indicate what else can be added.

We spread our surveys to a diverse set of developers by utilizing personal and professional contacts, mailing lists, and Java code forums. This makes it hard to ascertain an exact response rate, however, we do know exactly how many developers start the survey and what proportion of these developers complete the survey.

1.6 Origins of the chapters

All chapters of this thesis have been published in peer-reviewed journal and conferences. As a result, each chapter is self-contained with its background, related work, and implications section. In the following, the origin of each chapter is explained:

- *Chapter 2* was published in the paper "A dataset for API usage" by Sawant and Bacchelli at Mining Software Repositories (MSR) 2015 Data Track. This chapter is a background section, which was also part of a masters thesis by Sawant *is not an original contribution of this thesis.*

- *Chapter 3* was published in the paper "fine-GRAPE: fine-Grained APi usage Extractor An Approach and Dataset to Investigate API Usage" by Sawant and Bacchelli published in Empirical Software Engineering (EMSE) 2017.

- *Chapter 4* was published in the paper "On the reaction to deprecation of 25,357 clients of 4 + 1 popular Java APIs" by Sawant, Robbes and Bacchelli at International Conference on Software Maintenance and Evolution (ICSME) 2016. This chapter also contains content from the extension of this paper titled "On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK" by Sawant, Robbes and Bacchelli published in Empirical Software Engineering (EMSE) 2018.

- *Chapter 5* was published in the paper "To react, or not to react: Patterns of reaction to API deprecation" by Sawant, Robbes, and Bacchelli in Empirical Software Engineering (EMSE) 2019.

- *Chapter 6* was published in the paper "Why are features deprecated? An investigation into the motivation behind deprecation" by Sawant, Huang, Vilen, Stojkovski, and Bacchelli at International Conference on Software Maintenance and Evolution (ICSME) 2018.

- *Chapter 7* was published in the paper "Understanding developers' needs on deprecation as a language feature" by Sawant, Aniche, van Deursen and Bacchelli at International Conference on Software Engineering (ICSE) 2018.

1.7 Other publications

In addition to the publications that form this thesis, work was performed that does not make it into this thesis:

- The paper "Visualizing code and coverage changes for code review" published at FSE 2016 Tool track by Oosterwaal, van Deursen, Coelho, Sawant, and Bacchelli. This paper presents a tool that publishes a comment on a pull request with the change

in test coverage when the pull request is issued. It aids the contributor in understanding the impact a change will have on the test coverage of the project and also informs the code reviewer what kind of impact the change on the project's testing practices/standard.

- The paper "Mining motivated trends of usage of Haskell libraries" published at WAPI 2017 in the ICSE 2017 companion proceedings by Juchli, Krombeen, Rao, Yu, Sawant, and Bacchelli. In this work, we infer the version of a package that a Haskell client uses. Then we use a combination of manual analysis and automated analysis to infer the reasons behind an API consumer changing the version of the API being used.
- The paper "What makes a code change easier to review: an empirical investigation on code change reviewability" published at FSE 2018 by Ram, Sawant, Castelluccio, and Bacchelli. We analyze what about a change contributes to its reviewablity *i.e.*, the ease with which a reviewer can perform the code review.

1.8 Open Science

Data collected for the various chapters in this thesis has been made publicly available. An overview of the datasets and where to find them can be found in Table 1.2.

Dataset	Chapter	Host
API usage databases	2, 3, 4	Figshare
Large scale API Usage dataset	5	4TU Datacenter
Deprecation annotation interviews	7	4TU Datacenter

Table 1.2: Data storage locations

2

Fine grained API usage mining

An Application Programming Interface (API) is a set of functionalities provided by a third-party component (*e.g.*, library and framework) that is made available to software developers. APIs are extremely popular as they promote reuse of existing software systems [39].

The research community has used API usage data for various purposes such as measuring of popularity trends [40], charting API evolution [26], and API usage recommendation systems [41].

For example, Xie *et al.* have developed a tool called MAPO wherein they have attempted to mine API usage for the purpose of providing developers API usage patterns [42, 43]. Based on a developers' need MAPO recommends various code snippets mined from other open source projects. This is one of the first systems wherein API usage recommendation leveraged open source projects to provide code samples. Another example is the work by Lämmel *et al.* [44] wherein they mined data from Sourceforge and performed an API usage analysis of Java clients. Based on the data that they collected they present statistics on the percentage of an API that is used by clients.

One of the major drawbacks of the current approaches that investigate APIs is that they heavily rely on API usage information (for example to derive popularity, evolution, and usage patterns) that is *approximate*. In fact, one of the modern techniques considers as "usage" information what can be gathered from file imports (*e.g.*, `import` in Java) and the occurrence of method names in files.

This data is an approximation as there is no type checking to verify that a method invocation truly does belong to the API in question and that the imported libraries are used. Furthermore, information related to the version of the API is not taken into account. Finally, previous work was based on small sample sizes in terms of number of projects analyzed. This could result in an inaccurate representation of the real world situation.

With the current work, we try to overcome the aforementioned issues by devising fine-GRAPE (fine-GRained APi usage Extractor), an approach to extract type-checked API method invocation information from Java programs and we use it to collect detailed historical information on five APIs and how their public methods are used over the course of their entire lifetime by 20,263 client projects.

In particular, we collect data from the open source software (OSS) repositories on GitHub. GitHub in recent years has become the most popular platform for OSS developers, as it offers distributed version control, a pull-based development model, and social features [45]. We consider Java projects hosted on GitHub that offer APIs and quantify their popularity among other projects hosted on the same platform. We select 5 representative projects (from now on, we call them only *API*s to avoid confusion with client projects) and analyze their entire history to collect information on their usage. We get fine-grained information about method calls using a custom type resolution that does not require to compile the projects.

The result is an extensive dataset for research on API usage. It is our hope that our data collection approach and dataset not only will trigger further research based on finer-grained and vast information, but also make it easier to replicate studies and share analyses.

For example, with our dataset the following two studies can be conducted:

First, the evolution of the features of the API can be studied. An analysis of the evolution can give an indication as to what has made the API popular. This can be used to design and carry out studies on understanding what precisely makes a certain API more popular than other APIs that offer a similar service. Moreover, API evolution information gives an indication as to exactly at what point of time the API became popular, thus it can be studied in coordination with other events occurring to the project.

Second, a large set of API usage examples is a solid base for recommendation systems. One of the most effective ways to learn about an API is by seeing samples [23] of the code in actual use. By having a set of accurate API usages at ones' disposal, this task can be simplified and useful recommendations can be made to the developer; similarly to what has been done, for example, with Stack Overflow posts [46].

In our previous work titled "A dataset for API Usage" [47], we presented our dataset along with a few details on the methodology used to mine the data. In this chapter, we go into more detail into the methodology of our mining process and conduct two case studies on the collected data which make no use of additional information.

The first case is used to display the wide range of version information that we have at our disposal. This data is used to analyze the amount of time by which a client of an API lags behind the latest version of the API. Also, the version information is used to calculate as to what the most popular version of an API is. This study can help us gain insights into the API upgrading behavior of clients.

The second case showcases the type resolved method invocation data that is present in our database. We use this to measure the popularity of the various features provided by an API and based on this mark the parts of an API that are used and those that are not. With this information an API developer can see what parts of the API to focus on for maintenance and extension.

The first study provided initial evidence of a possible distinction between upgrade behavior of clients of APIs that release frequently compared to those that release infrequently. In the former case, we found that clients tend to hang back and not upgrade immediately; whereas, in the latter case, clients tend to upgrade to the latest version. The results of the second case study highlight that only a small part of an API is used by clients. This finding requires further investigation as there is a case to be made that many

new features that are being added to an API are not really being adopted by the clients themselves.

This chapter is organized as follows: Section 2.1 presents the approach that has been applied to mine this data. For the ease of future users of this dataset an overview of the dataset and some introductory statistics of it can be found in section 2.2.

2.1 Approach

We present the 2-step approach that we use to collect fine-grained type-resolved API usage information. (1) We collect data on project level API usage from projects mining open source code hosting platforms (we target such platforms due to the large number of projects they hosted) and use it to rank APIs according to their popularity to select an interesting sample of APIs to form our dataset; (2) apply our technique, fine-GRAPE, to gather fine-grained type-based information on API usages and collect historical usage data by traversing the history of each file of each API client.

2.1.1 Mining of coarse grained usage

In the construction of this dataset, we limit ourselves to the Java programming language, one of the most popular programming languages currently in use [48]. This reduces the types of programs that we can analyze, but has a number of advantages: (1) Due to the popularity of Java there would be a large source of API client projects available for analysis; (2) Java is a statically typed language, thus making the collection of type-resolved API usages easier; (3) it allows us to have a more defined focus and more thoroughly test and refine fine-GRAPE Future work can be to extend it to other typed-languages, such as C#.

To ease the collection of data regarding project dependencies on APIs, we found it useful to focus on projects that use build automation tools. In particular, we collect data from projects using Maven, one of the most popular Java build tools [49]. Maven employs the use of a Project Object Model (POM) files to describe all the dependencies and targets of a certain project. POM files contain artifact ID and version of each project's dependency, thus allowing us to know exactly which APIs (and version) a project uses. The following is an example of a POM file entry:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
</dependency>
```

In the dependency tag from a sample POM file pictured above, we see that the JUnit dependency is being declared. We find the APIs name in the `artifactId` tag. The `groupId` tag generally contains the name of the organization that has released the API, in this case it matches the artifactId. However, there are other cases such as the JBoss-Logging API for which the `groupID` is `org.jboss.logging` and the `artifactId` is `jboss-logging`. The version of JUnit to be included as a dependency is specified in the version tag and in this case it is version `4.8.2`.

2.1.2 Fine-grained API usage

To ensure that precise API usage information is collected, one has to reliably link each method invocation or annotation usage to class in the API to which it belongs. This can be achieved in five ways:

Text matching: This is one of the most frequently used techniques to mine API usage. For example, it has been used in the investigation into API popularity performed by Mileva *et al.* [40]. The underlying idea is to match explicit imports and corresponding method invocations directly in the text of source code files.

Bytecode analysis: Each Java file produces one or more class files when compiled, which contain Java bytecode that is platform independent. Another technique to mine API usage is to parse byte code in these class files to find all method invocations and annotation usages along with the class to which they belong to. This approach guarantees accuracy as the class files contain all information related to Java program in the Java file in question.

Partial program analysis: Dagenais *et al.* have created an Eclipse plugin called Partial Program Analysis (PPA) [50]. This plugin parses incomplete files and recovers type bindings on method invocations and annotations, thus identifying the API class to which a certain API usage belongs.

Dynamic analysis: Dynamic analysis is a process by which the execution trace of a program is captured as it is being executed. This can be a reliable method of determining the invocation *sequence* in a program as it can even handle the case where type of an object is decided at runtime. Performing dynamic analysis has the potential of being highly accurate as the invocations in the trace are type-resolved being recovered from running of bytecode.

AST analysis: Syntactically correct Java files can be transformed into an Abstract Syntax Tree (AST). An AST is a tree based representation of code where each variable declaration, statement, or invocation forms a node of the tree. This AST can be parsed by using a standard Java AST parser. The Java AST parser can also recover type based information at each step, which aids in ensuring accuracy when it comes to making a connection between an API invocation and the class it belongs to.

All five of the aforementioned approaches can be applied for the purpose of collecting API usage data, but come with different benefits and drawbacks.

The text-matching-based approach proves especially problematic in the case of imported API classes that share method names, because method invocations may not be disambiguated without type-information. Although some analysis tools used in dynamic languages [51] handle these cases through the notion of 'candidate' classes, this approach is sub-optimal for typed languages where more precise information is available.

The bytecode analysis approach is more precise, as bytecode is guaranteed to have the most accurate information, but it has two different issues:

1. Processing class files requires these files to be available, which, in turn, requires being able to compile the Java sources and, typically, the whole project. Even though

all the projects under consideration use Maven for the purpose of building, this does not guarantee that they can be built. If a project is not built, then the class files associated with this project cannot be analyzed, thus resulting in a dropped project.

2. To analyze the history of method invocations it is necessary to checkout each version of every file in a project and analyze it. However, checking out every version of a file and then building the project can be problematic as there would be an ultra-large number of project builds to be performed. In addition to the time costs, there would still be no warranty that data would not be lost due build failures.

The partial program analysis approach has been extensively tested by Dagenais *et al.* [50] to show that method invocations can be type resolved in incomplete Java files. This is a massive advantage as it implies that even without building each API client one can still conduct a thorough analysis of the usage of an API artifact. However, the implementation of this technique relies on Eclipse context, thus all parsing and type resolution of Java files can only be done in the context of an Eclipse plugin. This requires that each and every Java file is imported into an Eclipse workspace before it can be analyzed. This hinders the scalability of this approach to large number of projects.

Dynamic analysis techniques result in an accurate set of type resolved invocations. However, they require the execution of the code to acquire a trace. This is a limitation as not all client code might be runnable. An alternative would be to have a sufficient set of tests that would execute all parts of the program so that traces can be obtained. This too might be unfeasible as many projects may not have a proper test suite[52]. Finally, this technique would also suffer from the same limitations as the bytecode analysis technique; where analyzing every version of every file would require a large effort.

2.1.3 fine-GRAPE

Due to the various issues related to first four techniques, we deem the most suitable technique to be the AST based one. This technique utilizes the JDT Java AST Parser [53], *i.e.*, the parser used in the Eclipse IDE for continuous compilation in background. This parser handles partial compilation: When it receives in input a source code file and a Java Archive (JAR) file with possibly imported libraries, it is capable of resolving the type of methods invocation and annotations of everything defined in the code file or in the provided jar. This will allow us to parse standalone files, and even incomplete files in a quick enough way such that we can collect data from a large number of files and their histories in a time effective manner.

We created fine-GRAPE that, using the aforementioned AST parsing technique, collects the entire history of usage of API artifacts over different versions. In practice, we downloaded all the JAR files corresponding to the releases of the API projects chosen. Although this has been done manually in the study presented here, this process of downloading the JAR files has been automated in the current version for the ease of the user. Then, fine-GRAPE uses Git to obtain the history of each file in the client projects and runs on each file retrieved from the repository and the JAR with the corresponding version of the API that the client project declares in Maven at the time of the commit of the file. The fine-GRAPE leverages the visitor pattern that is provided by the JDT Java AST parser to visit all nodes in the AST of a source code file of the type method invocation or annotation.

These nodes are type resolved and are stored in a temporary data structure while we parse all files associated with one client project. This results in accurate type-resolved method invocation references for the considered client projects through their whole history. Once the parsing is done for all the files and their respective histories in the client, all the data that has been collected is transformed into a relational database model and is written to the database.

An API usage dataset can also contain the information on the method, annotations, and classes that are present in every version of every API for which usage data has been gathered such that any kind of complex analysis can be performed. In the previous steps we have already downloaded the API JAR files for each version of the API that is used by a client. These JAR files are made up of compiled class files, where each class file relates to one Java source code file. fine-GRAPE then analyzes these JAR files with the help of the bytecode analysis tool ASM [54], and for each file the method, class and annotation declarations are extracted.

2.1.4 Scalability of the approach

The approach that we have outlined runs on a large number of API client projects in a short amount of time. In its most recent state, all parts of the process are completely automated, thus needing a minimum of manual intervention. A user of the fine-GRAPE tool has to just specify the API which is to be mined, and this will result in a database that contains type-resolved invocations made to an API.

We benchmarked the amount of time it takes to process a single file. To run our benchmark, we used a server with two Intel Xeon E5-2643 V2 processors. Each processor consists of 6 cores and runs at a clock speed of 3.5 GHz. We ran our benchmark on 2,045 files from 20 client projects. To get an accurate picture, this benchmark was repeated 10 times. Based on this we found that the average amount of time spent on a single file was 165 milliseconds, the median was 31 milliseconds, the maximum was 1,815 ms for a large file.

2.1.5 Comparison to existing techniques

Previous work mined API usage examples, for example in the context of code completion, code recommendation, and bug finding. We see how the most representative of these mining approaches implemented in the past relate to the one we present here.

One of the more popular applications of API usage datasets is in the creation of code recommendation tools. In this field one of the more known tools is MAPO by Xie *et al.* [42, 43]. The goal of MAPO is to recommend relevant code samples to developers. MAPO runs its analyzer on source code files from open source repositories. MAPO uses the JDT compiler to compile a file and recover type-resolved API usages. These fine-grained API usages are then clustered using the frequent itemset mining technique [55]. In more recent developments tools such as UP-Miner [56] have been developed to mine high coverage usage patterns from open source repositories by using multiple clustering steps. Differently from fine-GRAPE, none of the approaches used here take version of the various APIs used into account. Moreover, our approach as opposed to theirs does not require the building of the files and has no need for all dependencies to the resolved to run.

Mining of API usage patterns has also been done to detect bugs by finding erroneous us-

age patterns. To this end, researchers developed tools such as Dynamine [57], JADET [58], Alattin [59] and PR-Miner [60]. All these tools rely on the same mining technique *i.e.*, frequent itemset mining [55]. The idea behind this technique is that statements that occur frequently together can be considered to be a usage pattern. This technique can result in a high number of false positives, due to the lack of type information. fine-GRAPE tackles this problem by taking advantage of type information.

The earliest technique that was employed in mining API usage was used by the tool CodeWeb [61] that was developed by Amir Michail. More recently it has been employed in the tool Sourcerer [62] as well. This technique employs a data mining technique that is called *generalized association rule mining*. An association rule is of the form $(\bigwedge_{x \epsilon X} x) \Rightarrow (\bigwedge_{y \epsilon Y} y)$. This implies that for an event x that takes place, then an event y will also take place with a certain confidence interval. The generalized association rule takes not just this into account but also takes a node's descendants into account as well. These descendants represent specializations of that node. This allows this technique to take class hierarchies into account while mining reuse patterns. However, just like frequent itemset mining this can result in false positives due to the lack of type information.

Recently, Moreno *et al.* [63] presented a technique to mine API usages using type resolved ASTs. Differently from fine-GRAPE, the approach they propose builds the code of each client to retrieve type resolved ASTs. As previously mentioned in the context of bytecode analysis, this could result in the loss of data, as some client projects may not build, and low scalability.

2.2 A Dataset for API Usage

Using fine-GRAPE we build a large dataset of usage of popular APIs. Our dataset is constructed using data obtained from the open source code hosting platform GitHub. GitHub stores more than 10 million repositories [64] written in different languages and using a diverse set of build automation tools and library management systems.

2.2.1 Coarse-grained API usage: The most popular APIs

To determine the popularity of APIs on a coarse-grained level (*i.e.*, project level), we parse POM files for all GitHub based Java projects that use Maven (ca. 42,000). The POM files were found in the master branch of approximately 250,000 active Java projects that are hosted on GitHub.[1] Figure 2.1 shows a partial view of the results with the 20 most popular APIs in terms of how many GitHub projects depend on them.

This is in-line with a previous analysis of this type published by Primat as a blog post [65]. Interestingly, our results show that JUnit is by far the most popular, while Primat's results report that JUnit is just as popular as SLF4J. We speculate that this discrepancy can be caused by the different sampling approach (he sampled 10,000 projects on GitHub, while we sampled about 42,000 on GitHub), further research can be conducted to investigate this aspect more in detail.

[1] As marked by GHTorrent [64] in September 2014

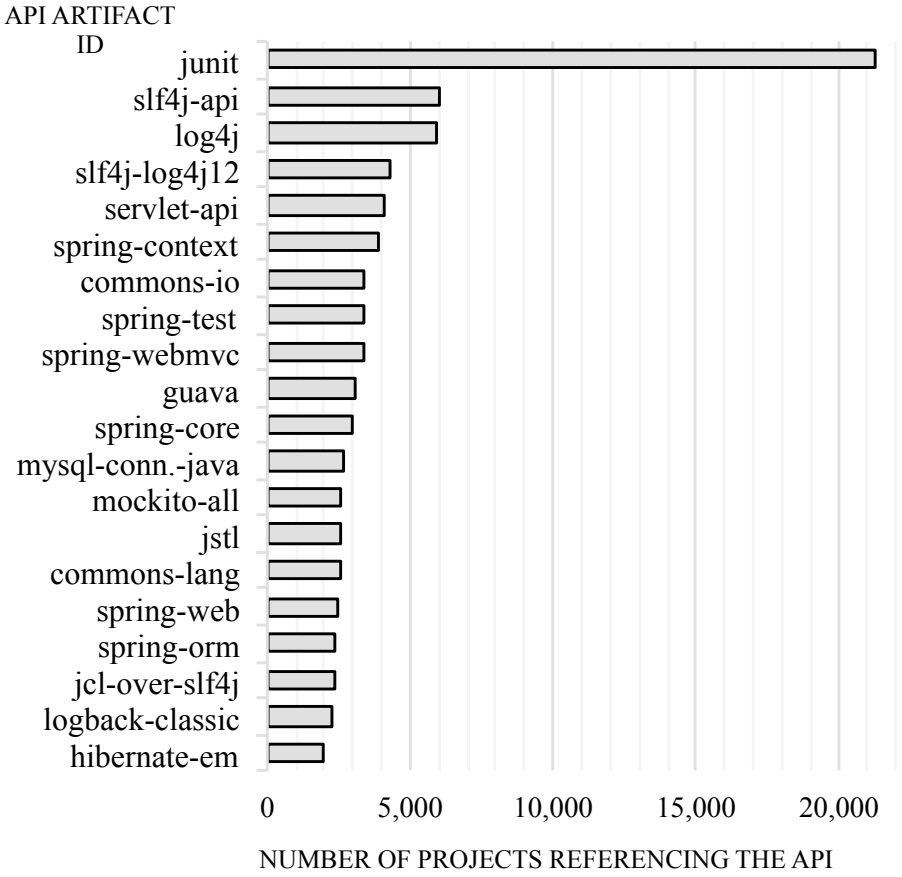


Figure 2.1: Popularity of APIs referenced on Github

2.2.2 Selected APIs

We used our coarse-grained analysis of popularity as a first step to select API projects to populate our database. To ensure that the selected API projects offer rich information on API usage and its evolution, rather than just sporadic use by a small number of projects, we consider projects with the following feature: (1) have a broad popularity for their public APIs (*i.e.*, they are in the top 1% of projects by the number of client projects), (2) have an established and reasonably large code base (*i.e.*, they have at least 150 classes in their history), (3) and are evolved and maintained (*i.e.*, they have at least 10 commits per week in their lifetime). Based on these characteristics, we eventually select the five APIs summarized in Table 2.1, namely Spring, Hibernate, Guava, and Guice and Easymock. We decide to remove JUnit, being an outlier in popularity and having a small code base that does not respect our requirements. We keep Easymock, despite its small number of classes and relatively low amount of activity in it's repository (ca. 4 commits per week) to add variety

to our sample. The chosen APIs are used by clients in different ways: *e.g.*, Guice clients use it through annotations, while Guava clients instantiate an instance of a Guava class and then interact with it through method invocations.

In the following, a brief explanation of the domain of each API:

1. **Guava** is the new name of the original Google collections and Google commons APIs. It provides immutable collections, new collectsion such as multiset and multimaps and finally some new collection utilities that are not provided in the Java SDK. Guava's collections can be accessed by method invocations on instantiated instances of the classes built into Guava.

2. **Guice** is a dependency injection library provided by Google. Dependency injection is a design pattern that separates behavioral specification and dependency resolution. Guice allows developers to inject dependencies in their applications with the usage of annotations.

3. **Spring** is a framework that provides an Inversion of Control(IoC) container. This allows developers to access Java objects with the help of reflection. The Spring framework comprises of a lot of sub projects, however we choose to focus on just the spring-core, spring-context and spring-test modules due to their relatively high popularity. The features provided by Spring are accessed in a mixture of method invocations and annotations.

4. **Hibernate Object Relational Mapping (ORM)** provides a framework for mapping an object oriented domain to a relational database domain. It is made up of a number of components that can be used, however we focus on just two of the more popular one *i.e.*, hibernate-core and hibernate-entity manager. Hibernate exposes its APIs as a set of method invocations that can be made on the classes defined by Hibernate.

5. **Easymock** is a testing framework that allows for the mocking of Java objects during testing. Easymock exposes its API to developers by way of both annotations and method invocations.

2.2.3 Data Organization

We apply the approach outlined in Section 2.1 and store all the data collected from all the client GitHub projects and API projects in a relational database, precisely PostgreSQL [66]. We have chosen a relational database because the usage information that we collect can be naturally expressed in forms of relations among the entities. Also we can leverage SQL functionalities to perform some initial analysis and data pruning.

Figure 2.2 shows the database schema for our dataset. On the one hand we have information for each client project: The Projects table is the starting point and stores a project's name and its unique ID. Connected to this we have ProjectDependency table, which stores information collected from the Maven POM files about the project's dependencies. We use a date_commit attribute to trace when a project starts including a certain dependency in its history. The Classes table contains one row per each uniquely named class in

Table 2.1: Subject APIs

API & GitHub repo	Inception	Releases	Unique Entities	
			Classes	Methods
Guava google/guava	Apr 2010	18	2,310	14,828
Guice google/guice	Jun 2007	8	319	1,999
Spring spring-framework	Feb 2007	40	5,376	41,948
Hibernate hibernate/hibernate-orm	Nov 2008	77	2,037	11,625
EasyMock easymock/easymock	Feb 2006	14	102	623

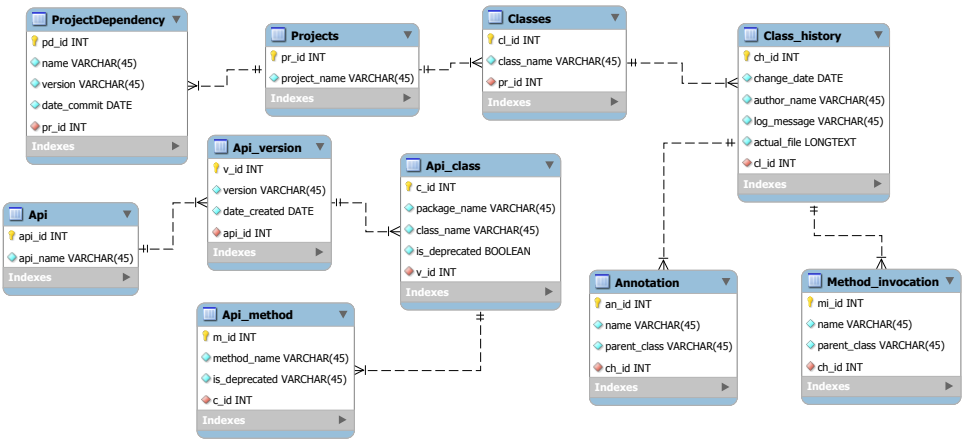


Figure 2.2: Database Schema For The Fine-grained API Usage Dataset

the project; in the table Class_history we store the different versions of a class (including its textual content, actual_file) and connect it to the tables Method_invocation and Annotation where information about API usages are stored. On the other hand, the database stores information about API projects, in the tables prefixed with Api. The starting point is the table Api that stores the project name and it is connected to all its versions (table Api_version, which also stores the date of creation), which are in turn connected classes (Api_class) and their methods (Api_method) that also store information about deprecation. Note that in the case of annotations we do not really collect them in a separate table as annotations are defined as classes in Java.

A coarse-grained connection between a client and an API is done with a SQL query on the tables ProjectDependency, Api and Api_version. The finer-grained connection is obtained by also joining Method_invocation/Annotation and Api_class on parent class names & Method_invocation/Annotation and Api_method on method names.

The full dataset is available as a PostgreSQL data dump on FigShare [67], under the

CC-BY license. For platform limitations on the file size the dump has been split in various tar.gz compressed files, for a total download size of 51.7 GB. The dataset uncompressed requires 62.3 GB of disk space.

2.2.4 Introductory Statistics

Table 2.2 shows an introductory view about the collected usage data. In the case of Guava for example, even though version 18 is the latest (see Table 2.1), version 14.0.1 is the most popular among clients. APIs such as Spring, Hibernate and Guice predominantly expose their APIs as annotations, however we see also a large use of the methods they expose. The earliest usages of Easymock and Guice are outliers as GitHub as a platform was launched in 2008, thus the repositories that refer to these APIs were moved to GitHub as existing projects.

Table 2.2: Introductory usage statistics

API	Most popular release	Usage across history	
		Invocations	Annotations
Guava	14.0.1	1,148,412	—
Guice	3.0	59,097	48,945
Spring	3.1.1	19,894	40,525
Hibernate	3.6	196,169	16,259
EasyMock	3.0	38,523	—

2.2.5 Comparison to existing datasets

The work of Lämmel *et al.* [68] is the closest to the dataset we created with fine-GRAPE. They target open source Java projects hosted on the Sourceforge platform and their API usage mining method relies on type resolved ASTs. To acquire these type resolved ASTs they build the APIs client projects and resolve all of its dependencies. This dataset contains a total of 6,286 client projects that have been analyzed and the invocations for 69 distinct APIs have been identified.

Our dataset as well as that of Lämmel *et al.* target Java based projects, though the clients that have been analyzed during the construction of our dataset were acquired from GitHub as opposed to Sourceforge. Our approach also relies on type resolved Java ASTs, but we do not build the client projects as fine-GRAPE is based on a technique able to resolve parsing of a standalone Java source file. In addition, the dataset by Lämmel *et al.* only analyzes the latest build. In terms of size this dataset is comprised of usage information gleaned from 20,263 projects as opposed to the 6,286 projects that make up the Lämmel *et al.* dataset. However, this dataset contains information on only 5 APIs whereas Lämmel *et al.* identified usages from 69 APIs.

3

Understanding API consumer upgrade behavior

We present two case studies to showcase the value of our dataset and to provide examples for others to use it. We focus on case studies that require minimal processing of the data and are just on basic queries to our dataset.

3.1 Case 1: Do clients of APIs migrate to a new version of the API?

As with other software systems, APIs also evolve over time. A newer version may replace an old functionality with a new one, may introduce a completely new functionality, or may fix a bug in an existing functionality. Some infamous APIs, such as Apache Commons-IO, are stagnating since long time without any major changes taking place, but to build our API dataset we took care of selecting APIs that are under active development, so that we could use it to analyze as to whether clients react to a newer version of an API.

3.1.1 Methodology

We use the *lag time* metric, as previously defined by McDonnell *et al.* [69], to determine the amount of time a client is behind the latest release of an API that it is using. Lag time is defined as the amount of time elapsed between a client's API reference and the release date of the latest version. A client lags if it uses an old version of an API when a newer version has already been released. For example, in Figure 3.1, client uses API version 7 despite version 8 being already released. The time difference between the client committing code using an older version and the release date of a newer version of the API is measured as the lag time.

In practice, we consider the commit date of each method invocation (this is done by performing a join on the METHOD_INVOCATION and CLASS_HISTORY tables), determine the version of the API that was being used by the client at the time of the commit (the PROJECT_DEPENDENCY table contains information on the versions of the API being used by the client and the date at which the usage of a certain version was employed), then consider

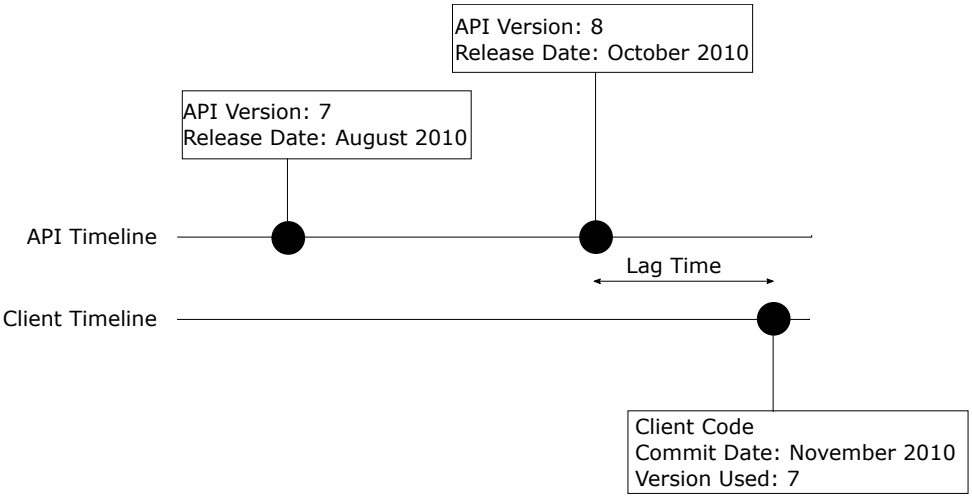


Figure 3.1: An example of the lag time metric inspired by McDonnell *et al.* [69]

the release date of the latest version of the API that existed at the time of the commit (this data can be obtained form the API_VERSION table in the database), and finally combine this information to calculate the lag time for each reference to the API and plot the probability density.

Lag time can indicate how far a project is at the time of usage of an API artifact, but it does not give a complete picture of the most recent state of all the clients using an API. To this end, we complement lag time analysis with the analysis of the most popular versions of each API, based on the latest snapshot of each client of the API (we achieve this by querying the PROJECT_DEPENDENCY table to get the latest information on clients).

3.1.2 Results

Results are summarized in four figures. Figure 3.2 shows the probability density of lag time in days, as measured from API clients, and Figure 3.3 shows the distribution of this lag time. Figure 3.3 shows frequency of adoption of specific releases: the three most popular ones, the latest release (available at the creation of this dataset), and all the other releases. Table 3.1 further specify the dates in which these releases were made public and provides absolute numbers. Finally, Figure 3.5 depicts the frequency and number of releases per API. The data we have ranges from 2004 to 2014, however for space reasons we only depict the range 2009 to 2014. Each year is divided into 3 slots of 4 month periods, and the number of releases in each of these periods is depicted by the size of the black circle.

Guava. In the case of the 3,013 Guava clients on GitHub the lag time varies between 1 day and 206 days. The median lag time for these projects is 67 days. The average amount of time a project lags behind the latest release is 72 days. Figure 3.2 shows the cumulative distribution of lag time across all clients. Since Guava generally releases 5 versions on average per year, it is not entirely implausible that some clients may be one or two versions behind at the time of usage of an API artifact.

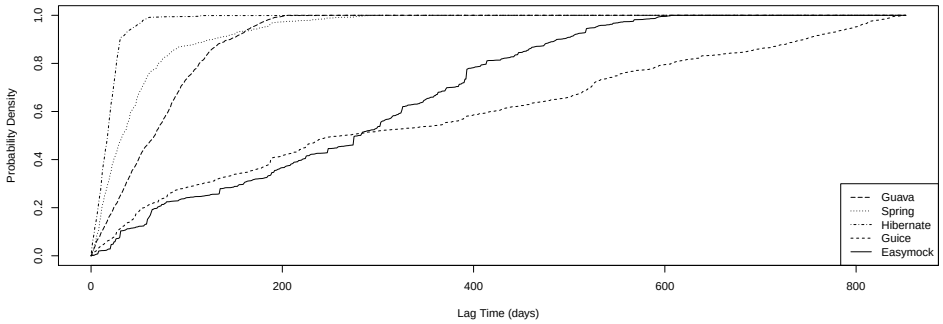


Figure 3.2: Probability density of lag time in days, by API

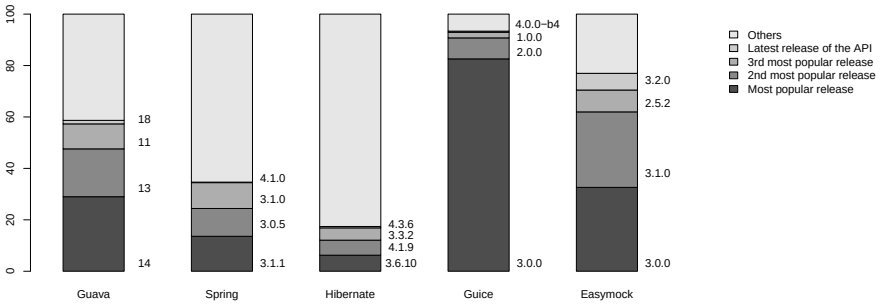


Figure 3.3: Proportion of release adoption, split in the 3 most popular, the latest, and all the other releases, by API

Although the latest (as of September 2014) version of Guava is 18, the most popular one is 14 with almost one third of the clients using this version (as shown in Figure 3.3). Despite 4 versions being released after version 14 none of them figure in the top 5 of most popular versions. Version 18 has been adopted by very few clients (41 out of 3,013). None of the other newer versions (16 and 17) make it in the top 5 either.

Spring. Spring clients lag behind the latest release up to a maximum of 304 days. The median lag time is 33 days and the first quartile is 15 days. The third quartile of the distribution is 60 days. The average amount of lag time for the usages of various API artifacts is 50 days. Spring is a relatively active API and releases an average of 7 versions (including minor versions and revisions) per year (Figure 3.5).

At the time of collection of this data, the Spring API had just released version 4.1.0 and only a small portion (30) of projects have adopted it. The most popular version is 3.1.1 (2,013 projects) as is depicted in Figure 3.3. We see that despite the major

Table 3.1: Publication date, by API, of the 3 most popular and latest releases, sorted by the number of their clients

API	Release	Release Date	Num of clients	(%)
Guava	14	03-2013	868	(29%)
	13	08-2012	557	(19%)
	11	02-2012	291	(10%)
	18	08-2014	41	(1%)
Spring	3.1.1	02-2012	2,013	(14%)
	3.0.5	10-2010	1,602	(11%)
	3.1.0	12-2011	1,489	(10%)
	4.1.0	10-2014	30	(0.2%)
Hibernate	3.6.10	02-2012	376	(6%)
	4.1.9	12-2012	352	(6%)
	3.3.2	06-2009	288	(5%)
	4.3.6	07-2014	32	(0.5%)
Guice	3.0.0	03-2011	536	(83%)
	2.0.0	07-2009	53	(8%)
	1.0.0	05-2009	14	(2%)
	4.0.0-b4	03-2014	3	(0.5%)
Easymock	3.0.0	05-2010	211	(33%)
	3.1.0	11-2011	190	(29%)
	2.5.2	09-2009	55	(9%)
	3.2.0	07-2013	42	(6%)

Figure 3.4: Lag time distribution in days, by API

version 4 of the Spring API being released in December 2013, the most popular major version remains 3. In our dataset, 344 projects still use version 2 of the API and 12

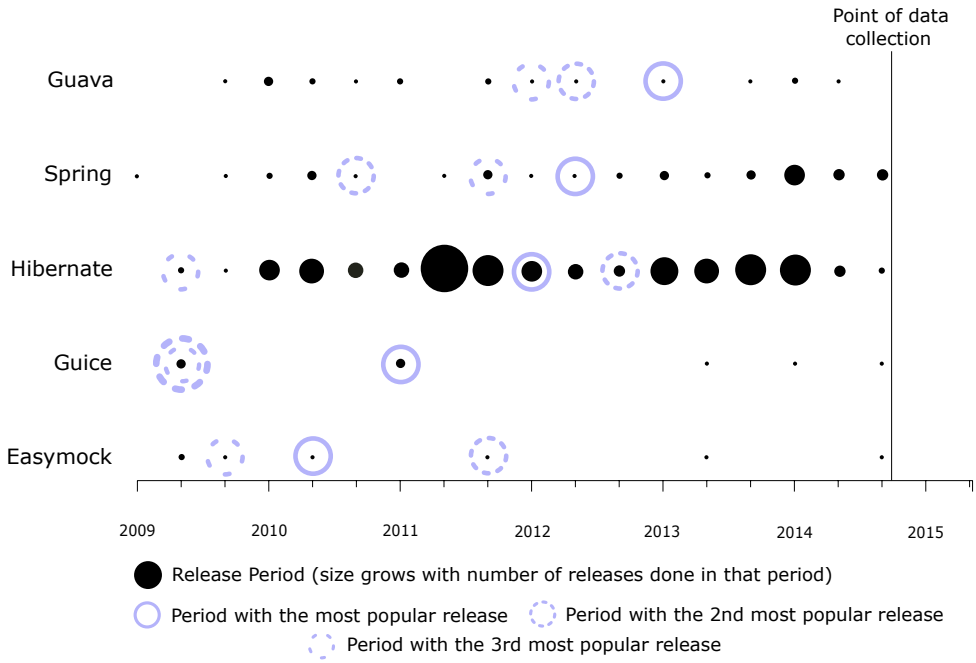


Figure 3.5: Release frequency for each API from 2009 (the dataset covers from 2004)

use version 1.

Hibernate. The maximum lag time observed over all the usages of Hibernate artifacts is 271 days. The median lag time is 18 days, and the first quartile is just 10 days. The third quartile is also just 26 days. The average lag time over all the invocations is 19 days. We see in Figure 3.2 that most invocations to Hibernate API do not lag behind the latest release considerably, especially in relation to the other APIs, although a few outliers exist. Hibernate releases 17 versions (including minor versions and revisions) per year (Figure 3.5).

Version 4.3.6 of Hibernate is the latest release that available on Maven central at the dataset creation time. A very small portion of projects (32) use this version, and the most popular version is version 3.6.10, *i.e.*, the last release with major version 3. We see that a large number of clients have migrated to early versions of major version 4. For instance, version 4.1.9 is almost (352 projects versus 376 projects) as popular as version 3.6.10 (shown in Figure 3.3). Interestingly, in the case of Hibernate, from our data we see that there is not a clearly dominant version as all the other versions of Hibernate make up about three fourths of the current usage statistics.

Guice. Among all usages of the Guice API, the largest lag time is 852 days. The median lag time is 265 days and the first quartile of the distribution is 80 days. The average of all the lag times is 338 days. The third quartile is 551 days, showing that a lot of projects have a very high lag time. Figure 3.2 shows the cumulative distribution of

lag times across all Guice clients. Guice is a young API and, relatively to the other APIs, releases are few and far between (10 releases over 6 years, with no releases on 2010 or 2012, Figure 3.5).

The latest version of Guice that has been released, before the construction of our dataset, is the fourth beta of version 4 (September 2014). Version 3 is unequivocally the most adopted version of Guice, as seen in Figure 3.3. This version was released in March of 2011 and since then there have been betas for version 4 released in 2013 and 2014. We speculate that this release policy may have lead to most of the clients sticking to an older version and preferring not to transition to a beta version.

Easymock. Clients of Easymock display a maximum, median, and average lag time of 607, 280, and 268 days, respectively. The first quartile and third quartile in the distribution are 120 and 393 days, respectively. Figure 3.2 shows the large number of projects that have a large amount of lag, relatively to the analyzed projects. Easymock is a small API, which had 12 releases, after the first, over 10 years (Figure 3.5).

The most recent version of Easymock is 3.3.1, released in January 2015. However, in our dataset we record use of neither that version nor the previous one (3.3.0). The latest used version is 3.2.0, released in July 2013, with 42 clients. Versions 3.0.0 and 3.1.0 are the most popular (211 and 190 clients) in our dataset, as seen in Figure 3.3. Version 2.5.2 and 2.4.0 also figure in the top three in terms of popularity, despite being released in 2009 and 2008.

3.1.3 Discussion

Our analysis lets emerge an interesting relation between the frequency of releases of an API and the behavior of its clients. By considering the data summarized in Figure 3.5, we can clearly distinguish two classes of APIs: 'frequent releaser' APIs (Guava, Hibernate and Spring) and 'non-frequent releaser' APIs (Guice and Easymock).

For all the APIs under consideration we see that there is a tendency for clients to hang back and to not upgrade to the most recent version. This is especially apparent in the case of the 'frequent releaser' APIs Guava and Spring: For these APIs, the older versions are far more popular and are still in use. In the case of Hibernate, we cannot get an accurate picture of the number of clients willing to transition because the version popularity statistics are quite fractured. This is a direct consequence of the large number of releases that take place every year.

For Guice and Easymock ('non-frequent releaser' APIs), we see that the latest version is not popular. However, for Guice the latest version is a beta and not an official release, thus we do not expect it to be high in popularity. In the case of Easymock, we see that the latest version (*i.e.*, 3.3.1) and the one preceding that (*i.e.*, 3.3.0) are not at all be used. In general, we do see that most clients of 'non-frequent releaser' APIs use a more recent version compared to clients of 'frequent releaser' APIs.

By looking at Figures 3.2 and 3.4, we also notice how the lag time of 'frequent releaser' APIs' clients is significantly lower than of 'non-frequent releaser' APIs' clients. This relation may have different causes: For example, 'non-frequent releaser' APIs' clients may be less used to update the libraries they use to more recent versions, they may also be less prone to change the parts of their code that call third-party libraries, or code that calls APIs

that have non-frequent release policy may be more difficult to update. Testing these hypothesis goes beyond the scope of this chapter, but with our dataset researchers can do so to a significant extent. Moreover, using fine-GRAPE, information about more APIs can be collected to verify whether the aforementioned relations hold with statistically significant samples.

3.2 Case 2: How much of an API is broadly used?

Many APIs are under constant development and maintenance. Some API producers do this to evolve features over time and improve the architecture of the API; others try to introduce new features that were previously not present. All in all, many changes take place in APIs over time [27]. Here we analyze which the features (methods and annotations) introduced by API developers are taken on board by the clients of these APIs.

This analysis is particularly important for developers or maintainers to know whether their efforts are useful and to decide to allocate more resources (e.g., testing, refactoring, performance improvement) in more used parts of their API, as resulting returns on investment may be greater. Moreover, API users may have more interest in reusing popular API features, as they are probably better tested through users [70].

3.2.1 Methodology

For each of the APIs, we have a list of features in the API_METHOD and API_CLASS tables [67]. We also have the usage data of all features per API that has been accumulated from the clients in the METHOD_INVOCATION and ANNOTATION tables. Based on this, we can mark features of the API have been used by clients. We can also count how many clients use a specific feature, thus classifying each feature as: (1) *hotspot*, in the top 15% of features in term of usage; (2) *neutral*, features that have been used once or more but not in the top 15% and (3) *coldspot*, if not used by any client. This is the same classification used by Thummalapenta and Xie [70] in a similar study (based on a different approach) on the usage of frameworks' features.

To see which used features were introduced early in an APIs lifetime, we can use the API_VERSION table to augment the date collected above with accurate version information per feature; then, for each of the used features, we see which version is the lowest wherein that feature has been introduced.

3.2.2 Results

The overall results for our analysis are summarized in Figures 3.6, 3.7, and 3.8. The first shows a percentage breakdown of usages of API features (left-hand side) and classes (right-hand side); the second and third report the probability distribution of the logarithm of the number of clients per API features, for 'non-frequent releaser' APIs and 'frequent releaser' APIs, respectively.

Generally, we see that the proportion of used features is never higher than 20% (Figure 3.6) and that the number of clients that use the features has a heavily right skewed distribution, which is slightly flattened by considering the logarithm (Figures 3.7 and 3.8). Moreover, we do not see a special behavior in this context of clients of 'non-frequent releaser' APIs vs. clients of 'frequent releaser' APIs.

In the following, we present the breakdown of the usage based on the definitions above.

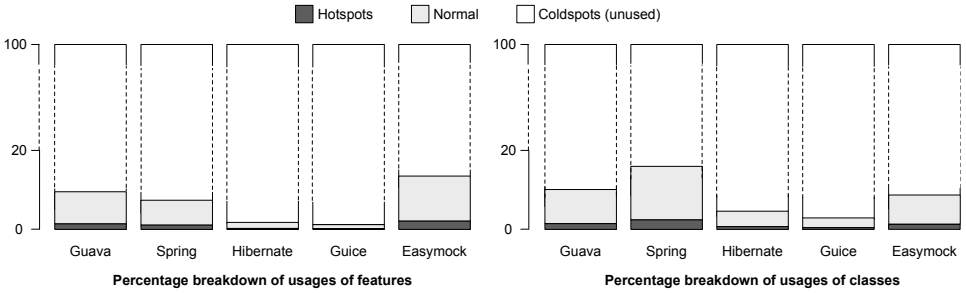


Figure 3.6: Percentage breakdown of usage of features for each of the APIs

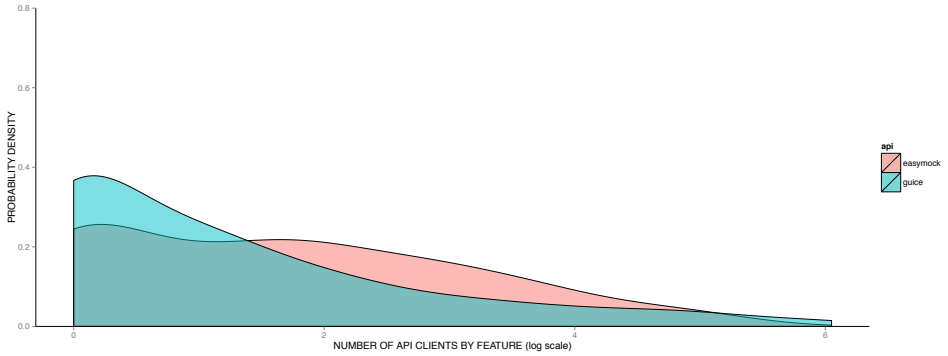


Figure 3.7: Probability distribution of (log) number of clients per API features, by 'non-frequent releaser' APIs

Guava. Only 9.6% of the methods in Guava are ever used; in absolute numbers, out of 14,828 unique public methods over 18 Guava releases, only 1,425 methods are ever used. Looking at the used methods, we find that 214 methods can be classified as hotspots. The rest (1,211) are classified as neutral spots. The most popular method from the Guava API is `newArrayList` from the class `com.google.common.collect.Lists` class and it has 986 clients using it.

Guava provides 2,310 unique classes over 18 versions. We see that only 235 (10%) of these are ever used by at least client. Furthermore, only 35 of these classes can be called hotspots in the API. A further 200 classes are classified as neutral. And we can classify a total of 2,075 classes as coldspots as they are never used. The most popular class is used 1,097 times and it is `com.google.common.collect.Lists`.

With Guava we see that 89.4% of the usages by clients of Guava relate to features that have been introduced in version 3 that was released in April 2010. Following which 7% of the usages relate to features that were introduced in version 10 that was released in October 2011.

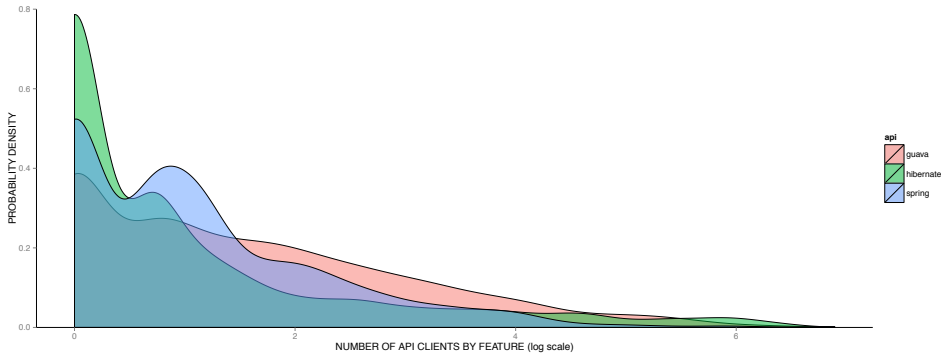


Figure 3.8: Probability distribution of (log) number of clients per API features, by 'frequent releaser' APIs

Spring. Out of the Spring core, context and test projects, we see that 7.4% of the features are used over the 40 releases of the API. A total of 840 features have been used out of the 11,315 features in the system. There are 126 features that can be classified as hotspots. Consequently, there are 714 features classified as neutral. The most popular feature is `addAttribute` from the class `org.springframework.ui.Model` and has been used 968 clients.

The Spring API provides a total of 1,999 unique classes. Out of these there are only 319 classes that are used by any of the clients of the Spring API. We can classify 48 of these classes as hotspot classes and the other 271 can be classified as neutral. We classify 1,680 classes as coldspots as they are never used. The most popular class has 2,417 clients and it is `org.springframework.stereotype.Controller`.

Looking deeper, we see that almost 96% of the features of Spring that are used by clients are those introduced in Spring version 3.0.0 that was released in December 2009.

Hibernate. From the Hibernate core and entitymanager projects we see that only 1.8% of the features are used. 756 out of the 41,948 unique public features provided over 77 versions of Hibernate have been used by clients in GitHub. Of these, 114 features that can be classified as hotspots and a further 642 features can be classified as neutral. The `getCurrentSession` method from the class `org.hibernate.SessionFactory` is the most popular feature, used by 618 clients.

Hibernate is made up of 5,376 unique classes. Out of these only 245 classes are used by clients. We can classify 37 of these classes as hotspots. The rest 208 classes are classified as neutral. We find that Hibernate has 5,131 coldspot classes. The most popular class is `org.hibernate.Session` with 917 clients using it.

In the case of Hibernate over 82% of the features that have been used were introduced in version 3.3.1 released in September 2008 and 17% of the features were introduced in 3.3.0.SP1 released in August 2008.

Guice. Out of the unique 11,625 features presented by Guice, we see that 1.2% (138) of the features are used by the clients of Guice. There are 21 features that are marked

as being hotspots, 117 features marked as being neutral, and 11,487 classified as coldspots. The most popular provided by the Guice API is `createInjector` from class `com.google.inject.Guice` and is used by 424 clients.

The Guice API is made up of 2,037 unique classes that provide various features. Out of these only 61 classes are of any interest to clients of the API. We find that 9 of these classes can be classified as hotspots and the other 52 as neutral spots. This leaves a total of 1,976 classes as coldspots. The most popular class provided by Guice is `com.google.inject.Guice` and there are 424 clients that use it.

Close to 96% of the features of Guice that are popularly used by clients were introduced in its first iteration which was released on Maven central in May 2009.

Easymock. There are unique 623 features provided by Easymock, out of which 13.4% (84) are used by clients. This implies that 539 features provided by the API are never by used by any of the clients and are marked as coldspots. 13 features are marked as hotspots, while 71 features are marked as neutral. the The most popular feature is `getDeclaredMethod` from the class `org.easymock.internal.ReflectionUtils` and is used by 151 clients.

Easymock being a small API consists of only 102 unique classes. Out of these only 9 classes are used by clients. Only 1 can be classified as a hotspot class and the other 8 are classified as neutral spots. This leaves 93 classes as coldspots. The most popular class is `org.easymock.EasyMock` and is used by 205 clients.

We observe that 95% of the features that are used from the Easymock API were provided starting version 2.0 which was released in December 2005.

3.2.3 Discussion

We see that for Guava, Spring and Easymock, the percentage of usage of features hovers around the 10% mark. Easymock has the largest percentage of features that are used among the 5 APIs under consideration. This could be down to the fact that Easymock is also the smallest API among the 5. Previous studies such as that by Thummalapenta and Xie [70] have shown that over 15% of an API is used (hotspot) whereas the rest is not (coldspot). However, the APIs that they analyzed are very different to the ones that are here as they are all smaller APIs comparable to the size of Easymock, however none of them are of the size of the other APIs such as Guava and Spring. Also, their mining technique relied on code search engines and not on type resolved invocations.

In the case of Hibernate and Guice we see a much smaller percentage (1.8% and 1.2% respectively) of utilization of features. This is far lower than that of other APIs in this study. We speculate that due to the fact that the most popular features that are being used are also those that were introduced very early in the APIs life (version 3.3.1 in the case of Hibernate and version 1.0 in the case of Guice). These features could be classified as core features of the API. Despite API developers adding new features, there may be a tendency to not deviate from usage of these core features as these may have been the ones that made the API popular in the first place.

This analysis underlines a possibly unexpected low usage of API features in GitHub clients. Further studies, using our dataset, can be designed and carried out to determine

which characteristics make certain feature more popular and guide developers to give the same characteristics to less popular features. Moreover, this popularity study can be used, for example, as a basis for developers to decide whether to separate more popular features of their APIs from the rest and provide them as a different, more supported package.

4

Scale of affectedness by deprecation

An Application Programming Interface (API) is a definition of functionalities provided by a library or framework made available to other developer, as such. APIs promote the reuse of existing software systems [39]. In his landmark essay "No Silver Bullet" [1], Brooks argued that reuse of existing software was one of the most promising attacks on the essence of the complexity of programming: "*The most radical possible solution for constructing software is not to construct it at all.*"

Revisiting the essay three decades later [71], Brooks found that indeed, reuse remains the most promising attack on essential complexity. APIs enable this: To cite a single example, we found at least 15,000 users of the Spring API [72].

However, reuse comes with the cost of dependency on other components. This is not an issue when said components are stable. But evidence shows that APIs are not always stable: The Java standard API for instance has an extensive *deprecated* API [1]. Deprecation is a mechanism employed by API developers to indicate that certain features are obsolete and that they will be removed in a future release. API developers often deprecate features, and when replace them with new ones, changes can break the client's code. Studies such as Dig and Johnson's [25] found that API changes breaking client code are common.

The usage of a deprecated feature can be potentially harmful. Features may be marked as deprecated because they are not thread safe, there is a security flaw, or are going to be replaced by a superior feature. The inherent danger of using a feature that has been marked as obsolete may be good enough motivation for developers to transition to the replacement feature.

Besides the aforementioned dangers, using deprecated features can also lead to reduced code quality, and therefore to increased maintenance costs. With deprecation being a maintenance issue, we would like to see if API clients actually react to deprecated features of an API.

Robbes *et al.* conducted the largest study of the impact of deprecation on API clients [3], investigating deprecated methods in the Squeak [73] and Pharo [74] software ecosystems. This study mined more than 2,600 Smalltalk projects hosted on the Squeak-

[1] see `http://docs.oracle.com/javase/8/docs/api/deprecated-list.html`

Source platform [75]. They investigated whether the popularity of deprecated methods either increased, decreased or did not change after deprecation.

Robbes *et al.* found that API changes caused by deprecation can have a major impact on the studied ecosystems, and that a small percentage of the projects actually reacts to an API deprecation. Out of the projects that do react, most systematically replace the calls to deprecated features with those recommended by API developers. Surprisingly, this was done despite API developers in Smalltalk not documenting their changes as good as one would expect.

The main limitation of this study is being focused on a niche programming community *i.e.*, Pharo. This resulted in a small dataset with information from only 2,600 projects in the entire ecosystem. Additionally, with Smalltalk being a dynamically typed language, the authors had to rely on heuristics to identify the reaction to deprecated API features.

We conduct a non-exact replication [76] of the previous Smalltalk [3] study, also striving to overcome its limitations. We position this study as an explorative study that has no pre-conceived notion as to what is correct behavior with respect to reaction to deprecation. To that end we study the reactions of more than 25,000 clients of 5 different APIs, using the statically-typed Java language; we also collect accurate API version information. The API clients analyzed in this study are open-source projects we collected on the GitHub social coding platform [77].

Furthermore, we also investigate the special case of the Java Development Kit API (JDK), which may present peculiarities, because of its role popularity and tailored integration with most IDEs. For example, due to these features developers might be more likely to react to deprecation in the API of the language, as opposed to deprecations in an API that they use. To perform this analysis, we collect data from Maven Central (which allows for a more reliable way to collect JDK version data). We collected data from 56,410 projects and their histories, out of which we analyze 60 projects (selected to reduce the size of data to be processed) to see how they dealt with deprecated API elements in Java's standard APIs.

Our results confirm that only a small fraction of clients react to deprecation. In fact, in the case of the JDK clients, only 4 are affected and all 4 of these introduce calls to deprecated entities at the time of usage. Out of those, systematic reactions are rare and most clients prefer to delete the call made to the deprecated entity as opposed to replacing it with the suggested alternative one. This happens despite the carefully crafted documentation accompanying most deprecated entities.

One of the more interesting phenomena that the we observed was that out of the 5 APIs for which we observed the reaction pattern, we see that each of the APIs has its own way in which it deprecated features, which then has an impact on the client. APIs such as Spring appear to deprecate their features in a more conservative manner and thus impact very few clients. On the other hand, Guava appears to constantly making changes to their API, thus forcing their clients to deal with deprecation in the API, at the risk of having clients not upgrading to the latest version of the API. Given these patterns that we observed, we investigate whether we can categorize APIs based on the strategy they use when deprecating features. To this end we look at 50 popular Java APIs, and develop heuristics characterizing how these APIs deprecate features.

4.1 Methodology

We define the research questions and describe our research method contrasting it with the study we expand upon [3].

4.1.1 Research Questions

To better contrast our results with the previous study on Smalltalk, we try to maintain the same research questions as the original work whenever possible.

The aim of these research questions is similar to the original chapter and they aim to determine (1) whether deprecation of an API artifact affects API clients, (2) whether API clients do react to deprecation and (3) , and to understand if immediately actionable information can be derived to alleviate the problem. To do this, we find out how often a deprecated entity impacts API clients and how these clients deal with it.

Given the additional information at our disposal in this chapter, we add two novel research questions (RQ0 and RQ6). RQ0 aims to understand the API version upgrade behavior of API clients and RQ6 looks at the impact of various deprecation policies on the reaction of API clients. Furthermore, we alter the original order and partially change the methodology we use to answer the research questions; this leads to some differences in the formulation. The research questions we investigate are:

- RQ0: What API versions do clients use?
- RQ1: How does API method deprecation affect clients?
- RQ2: What is the scale of reaction in affected clients?
- RQ3: What proportion of deprecations does affect clients?
- RQ4: What is the time-frame of reaction in affected clients?
- RQ5: Do affected clients react similarly?
- RQ6: How are clients impacted by API deprecation policies?

4.1.2 Research Method, Contrasted With the Previous Study

Robbes *et al.* analyzed projects hosted on the SqueakSource platform, which used the Monticello versioning system. The dataset contained 7 years of evolution of more than 2,600 systems, which collectively had over 3,000 contributors. They identified 577 deprecated methods and 186 deprecated classes. The results were informative, but this previous study had several shortcomings that we address. We describe the methodology to collect the data for this study by describing it at increasingly finer granularity: Starting from the selection of the subject systems to detecting the use of versions, methods, and deprecations. In this work, the methodologies for the collection of Third-party API usage and JDK API usage is different, and these differences are reflected in Figure 4.1 and Figure 4.2.

For Third-Party APIs, we select candidate APIs based on their popularity (Figure 4.1, top left); we then build the list of their clients (Figure 4.1, bottom left), keeping only active projects; finally, for each project, we locate the usages of individual API elements in successive version of these projects (Figure 4.1, right).

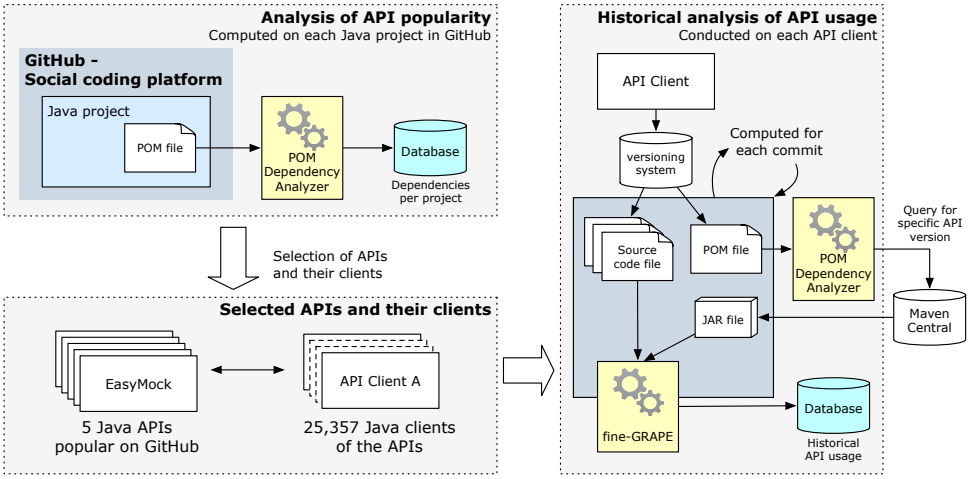


Figure 4.1: Methodology used to mine data from Third-party API clients

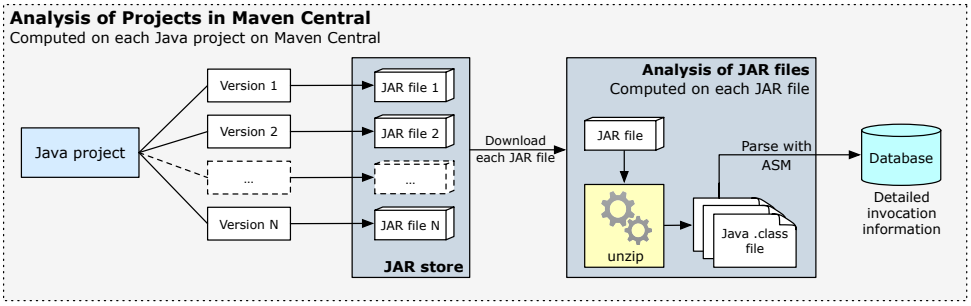


Figure 4.2: Methodology used to mine data from JDK API clients

For the JDK, every Java project is essentially a client. We first select a diverse sample of Java projects from Maven Central to study. We then download successive compiled version of these systems from Maven Central (Figure 4.2, left), before analyzing the bytecode to infer both the JDK version used to compile it, and the use of JDK features (Figure 4.2, right).

System Source

The original study was conducted on the Squeak and Pharo ecosystems found on SqueakSource, thus the set of systems that were investigated was relatively small. To overcome this limitation, we focus on a mainstream ecosystem: Java projects hosted on the social coding platform GitHub and in Maven central. Java is the most popular programming language according to various rankings [48, 78], GitHub is the most popular and largest hosting service [79] and Maven Central is the largest store of JAR files [19].

- **Third-party APIs** (Figure 4.1, top left).
  Our criteria for selection of APIs includes popularity, reliability, and variety: We

measure popularity in terms of number of clients each API has on GitHub and select from the top 20 as identified by the fine-GRAPE dataset [47]. We ensure reliability by picking APIs that are regularly developed and maintained *i.e.*, those that have at least 10 commits in a 6 week period before the data has been collected. We select APIs pertaining to different domains. These criteria ensure that the APIs result in a representative evolution history, do not introduce confounding factors due to poor management, and do not limit the types of clients.

We limit our study to Java projects that use the Maven build system because Maven based projects use Project Object Model (POM) files to specify and manage the API dependencies that the project refers to. We searched for POM files in the master branch of Java projects and found approximately 42,000 Maven based projects on GitHub. By parsing their POM files, we obtained all the APIs they depend on. We then created a ranking of the most popular APIs, which we used to guide our choice of APIs to investigate.

This selection step results in the choice of 5 APIs, namely: Easymock [80], Guava [81], Guice [82], Hibernate [83], and Spring [84]. The first 6 columns of Table 4.1 provide additional information on these APIs.

- **JDK APIs.**
  Clients of the JDK are not necessarily hard to find, GitHub alone contains 879,265 Java based projects. However, we are interested in accurately inferring the version of the JDK being used and whether these clients react to the deprecation of features in various versions of the JDK. This is not a trivial endeavor, given that Java source code files do not specify the version of Java that they are meant for. To overcome this challenge, we use the Java Archive (JAR) files of projects that were released to Maven Central.

  Maven Central is the central repository for all libraries that can be used by Maven based projects. It is one of the largest stores of JAR files, where most small and large organizations release their source code in built form. JAR files consist of class files, which are a result of compiling Java source code. These class files contain metadata on the version of the JDK being used. Thus, making JAR files an appropriate source of data for JDK clients, given that version resolution can be done in it.

Selection of main subjects

We select the *main subjects* of this study:

- **Third-party APIs** (Figure 4.1, bottom left).
  To select the clients of APIs introducing deprecated methods, we use the aforementioned analysis of the POM files. We refine our process using the GHTorrent dataset [85], to select only active projects. We also remove clients that had not been actively maintained in the 6 months preceding our data collection, to eliminate 'dead' or 'stagnating' projects. We totaled 25,357 projects that refer to one or more of 5 aforementioned popular APIs. The seventh column in Table 4.1 provides an overview of the clients selected, by API.

Table 4.1: Summary information on selected clients and APIs

API (GitHub repo)	Description	Inception	Releases	Unique entities		Number of clients	Usage across history	
				Classes	Methods		Invocations	Annotations
EasyMock (easymock/ easymock)	A testing framework that allows for the mocking of Java objects during testing.	Feb 06	14	102	623	649	38,523	-
Guava (google/guava)	A collections API that provides data structures that are an extension to the datastructures already present in the Java SDK. Examples of these new datastructures includes: multimaps, multisets and bitmaps.	Apr 10	18	2,310	14,828	3,013	1,148,412	-
Guice (google/guice)	A dependency injection library created by Google.	Jun 07	8	319	1,999	654	59,097	48,945
Hibernate (hibernate/ hibernate-orm)	A framework for mapping an object oriented domain to a relational database domain. We focus on the core and entitymanager projects under the hibernate banner.	Nov 08	77	2,037	11,625	6,038	196,169	16,259
Spring (spring-projects/ spring-framework)	A framework that provides an Inversion of Control(IoC) container, which allows developers to access Java objects with the help of reflection. We choose to focus on just the spring-core, spring-context and spring-test modules due to their popularity.	Feb 07	40	5,376	41,948	15,003	19,894	40,525

- **JDK APIs** (Figure 4.2, left).
  At the time of data collection, Maven Central included 1,297,604 JAR files pertaining to 150,326 projects that build and release their code on the central repository. Analyzing all these projects would pose serious technical challenges, thus, we produced a selection criterium to reduce the number of projects, while preserving representativeness.

  As a first step to filter our project list, we decide to eliminate those projects that have 4 or fewer releases on Maven Central. Projects that have had so few releases are not likely to have changed the version of the JDK that they use to compile their codebase, thus rendering them uninteresting to our current purpose. Performing this elimination step leaves us with 56,410 projects and 1,144,134 JARs that we can analyze. All these JAR files for each of the projects is downloaded, unzippped and all method invocations is stored in a database.

  Despite having this amount of data, we cannot process all projects for the purpose of this chapter, thus, as a further step of filtering we use the technique outlined by Nagappan *et al.* [86] to sample our set of projects, creating a small and representative dataset of *diverse* projects to analyze. The technique allows for the creation of a diverse and representative sample set of projects that likely cover the entire spectrum of projects. They define coverage as:
  $coverage = \frac{|U_{p \in P}\{q | similar(p,q)\}|}{|U|}$
  Here U is the universe of projects from which a selection is to be made. P is the set of dimensions along which these projects can be classified. A similarity score is computed for all projects based on the defined dimensions of the entire universe. The algorithm keeps adding projects to the subset that increases the similarity score until a coverage of 100% is achieved.

  For our analysis, we define the following dimensions to model the projects:

  – **Number of versions released:** The higher the number of versions released by a project, the more likely the project has made a change in the version of

the JDK (or any other API) that it uses.

- **Median version release time:** Projects with a lower median release time, release new versions more often than those with a high inter-release period. This distinction is important because projects that release often might not actually react to deprecation due to the cost involved, on the other hand, projects with long inter-release time spans might have a lot of time at their disposal to fix their code.
- **Lifespan of the project:** Projects that have lasted a long time (*e.g.*, Spring which has been around for 13 years) have more of a chance of having used multiple versions of Java and being affected by deprecation due to upgrading the version of the JDK being used, as opposed to those projects that are young.
- **Number of classes:** Projects that are larger (those that have more classes) might have a different reaction pattern to those that are smaller, measuring the number of classes allows us to make this distinction.
- **Starting version:** Projects that start with a more recent version of the JDK might not be affected by deprecation as opposed to those that use an old version of Java.

Nagappan *et al.* provide R scripts implementing their project selection technique. We ran these scripts with our dimensions of interest as input and collected a list of 60 diverse projects that cover the entire space of projects (100% coverage). We thus use these 60 projects for our analysis of client's reactions to deprecation in entities of the JDK API.

API version usage.
Explicit library dependencies are rarely mentioned in Smalltalk. There are several ways to specify these dependencies, often programmatically and not declaratively (for instance, Smalltalk does not use import statements as Java does). Thus, detecting and analyzing dependencies between projects requires heuristics [87]. In contrast, Maven projects specify their dependencies explicitly and declaratively: We can thus determine the API version a project depends on. Hence, we can answer more questions, such as if projects freeze or upgrade their dependencies. This is more complicated in the case of JDK clients, as the version definition is not explicit. To overcome this flaw, we use an alternative data source (Maven Central) such that all information at our disposal can be considered accurate.

- **Third-party APIs.**
  We only consider projects that encode specific versions of APIs, or unspecified versions (which are resolved to the latest API version at that date). We do not consider ranges of versions because very few projects use those (84 for all 5 APIs, while we include 25,357 API dependencies to these 5 APIs). In addition, few projects use unspecified API versions (269 of the 25,357, which we do include).

- **JDK APIs.** (Figure 4.2, right)
  Accurate resolution of the version of the Java API is a challenge on its own, since there is no easy way to find out the version of the JDK being used by a Java project

by only inspecting the source code. However, there are three techniques that can be used and these are outlined below:

1. Projects that use maven sometimes use the maven compiler plugin in the POM file. This plugin allows the specification of the version of Java that is being used in the source code and the bytecode version to which this source code is to be compiled. Often these versions can be the same, there is no hard requirement for them to be different. One can download all the POM files and parse them to find the usage of this plugin and see what version of Java is being used.
2. The Java JDK has evolved over time. With every new version, new features have been introduced. These features are incompatible with older versions of the JDK. However, these features are forward compatible, thus ensuring that anyone using that feature must either use the version of the JDK in which the feature was introduced or a later version. Thus, based on the language features being used one can ascertain a range of JDKs that may have been used to compile a certain file. Also, cross-referencing the date of usage of the feature with the date of JDK releases could allow us to narrow the range. Overall, this would give us a small range of versions that might have been used.
3. The previous two approaches both rely on parsing the source code to extract the usage of the Java features and to estimate the version of the JDK being used. However, the most reliable way to infer the version of Java being used is to look at the compiled class files of the source code. These class files contain a two-digit integer header that specifies the version of the JDK that was used to compile it, *e.g.*, a class header could be 50, which implies that the class was compiled using JDK 1.6.

We tried the first method outlined above to resolve the version of the JDK being used. However, when we looked at 135,739 POM files that we managed to download from GitHub, we found that only 7,722 files used the maven compiler plugin. In the larger context, this was a very small number of files that specified the version of the JDK. Thus, we found this option nonviable.

The second method can at times result in an inaccurate resolution of the version being used. This would make it hard to answer the research questions we have with the same accuracy as for the clients of the 5 Java APIs, hence we decided against using it.

Reading the compiled version of a class is the most accurate way to infer the version of the JDK being used by a Java project. However, this would imply that we would have to compile all the Java based project at our disposal. Even with these projects using the Maven build tool, this task is problematic. First, there is a chance that the dependencies specified in the POM file might relate to certain internal dependencies that are hosted by the project's developers and are not publicly available. Second, some of these POM files might use a PGP key to verify the authenticity of the dependencies being used, and this key is not available to us. Third, often the projects on GitHub do not ship with tests that work in all environments and they might need a certain testing environment to work properly, without which the Maven build fails.

Fourth, it is very time-consuming to compile every version of every file to get its class file.

With the limitations of compiling GitHub based Java projects, we found an alternative source of Java based data; Maven central. Maven Central is the standard repository host for Java projects. Here developers release their projects as libraries such that others can use them. These projects are in the form of Java Archive (JAR) files that contain class files. We find this to be an appropriate source of data, given that there are 150,326 distinct projects released on Maven Central with 1,297,604 JAR files associated with them. These JAR files can be unpacked to parse the usage of Java features and at the same time ascertain the version of the JDK that was used to compile the source code.

Fine-grained method/annotation usage

Due to the lack of explicit type information in Smalltalk, there is no way of actually knowing if a specific class is referenced and whether the method invocation found is actually from that referenced class. This does not present an issue when it comes to method invocations on methods that have unique names in the ecosystem. However, in the case of methods that have common names such as `toString` or `name` or `item`, this can lead to some imprecise results. In the previous study, Robbes *et al.* resorted to a manual analysis of the reactions to an API change but had to discard cases which were too noisy.

- **Third-party APIs.** (Figure 4.1, right)
  In this study, Java's static type system addresses this issue without the need for a tedious, and conservative manual analysis. On the other hand, Java APIs can be used in various manners. In Guava, actual method invocations are made on object instances of the Guava API classes, as one would expect. However, in Guice, clients use annotations to invoke API functionality, resulting in a radically different interaction model. These API usage variabilities must be considered.

  While mining for API usage we must ensure that we connect a method invocation or annotation usage to the parent class to which it belongs. There are multiple approaches that can be taken to mining the usage data from source code. The first uses pattern matching to match a method name and the import in a Java file to find what API a certain method invocation belongs to. The second uses the tool PPA [50] which can work on partial programs and find the usage of a certain method of an API. The third builds the code of a client project and then parse the bytecode to find type-resolved invocations. Finally, the fourth uses the Eclipse JDT AST parser to mine type-resolved invocations from a source code file. We created a method, fine-GRAPE, based on the last approach [47, 72] that meets the following requirements:[2] (1) fine-GRAPE handles the large-scale data in GitHub, (2) it does not depend on building the client code, (3) it results in a type-checked API usage dataset, (4) it collects explicit version usage information, and (5) it processes the whole history of each client.

- **JDK API.** (Figure 4.2, right)
  In the case of JDK clients, we look at class files that are retrieved from JAR files.

[2]More details on fine-GRAPE can be found in Chapter 2.

These class files contain accurate API usage information. This information can be parsed using the ASM [88] library, which uses the visitor pattern to visit a class file. For each class, we extract information on the version of the JDK being used, the annotations used in the class, and the method invocations made. For each of the method invocations, we have accurate information on the class that the method belongs to, the parameters and type of parameters being passed to the method, and what the expected return value is. Overall, we ensure that while parsing the JDK clients we obtain an accurate representation of usage of the JDK.

4.1.3 Detect deprecation

In Smalltalk, users insert a call to a deprecation method in the body of the deprecated method. This call often indicates which feature replaces the deprecated call. However, there is no IDE support. The IDE does not indicate to developers that the feature being used is deprecated. Instead, calls to deprecated methods output runtime warnings.

In contrast, Java provides two different mechanisms to mark a feature as deprecated. The first is the `@deprecated` annotation provided in the Javadoc specification. This annotation is generally used to mark an artifact as deprecated in the documentation of the code. This feature is present in Java since JDK version 1.1. Since this annotation is purely for documentation purposes, there is no provision for it to be used in compiler level warnings. This is reflected in the Java Language Specification(JLS). However, the standard Sun JDK compiler does issue a warning to a developer when it encounters the usage of an artifact that has been marked as deprecated using this mechanism. More recently, JDK 1.5 introduced a second mechanism to mark an artifact as deprecated with a source code annotation called `@Deprecated` (The same JDK introduced the use of source code annotations).

This annotation is a compiler directive to define that an artifact is deprecated. This feature is part of the Java Language Specification; as such any Java compiler supports it. It is now common practice to use both annotations when marking a certain feature as deprecated. The first is used so that developers can indicate in the Javadoc the reasons behind the deprecation of the artifact and the suggested replacement. The other is now the standard way in which Java marks features as deprecated.

To identify the deprecated features, we first download the different versions of the APIs used by the clients from the Maven central server. These APIs are in the form of Java Archive (JAR) files, containing the compiled classes of the API source. We use the ASM [88] class file parsing library to parse all the classes and their respective methods. Whenever an instance of the `@Deprecated` annotation is found we mark the entity it refers to as deprecated and stores this in our database. Since our approach only detects compiler annotations, we do not handle the Javadoc tag. See the threats to validity section for a discussion of this. We also do not handle methods that were removed from the API without warning, as these are out of the scope of this study.

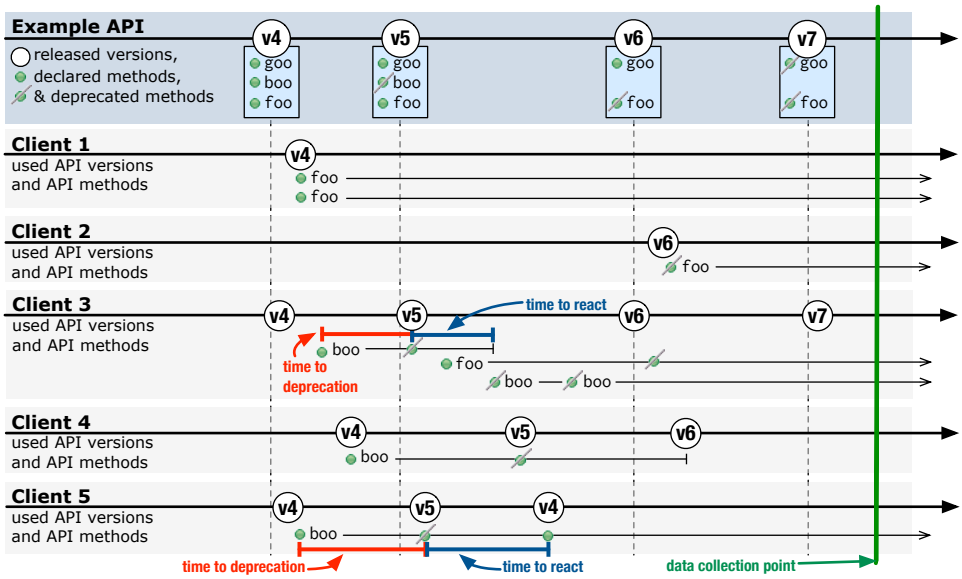


Figure 4.3: Exemplification of the behavior of an API and its clients

4.2 RQ0: What API versions do clients use?

Our first research question seeks to investigate the popularity of API versions and to understand the behavior of the clients towards version change. This sets the ground for the subsequent research questions.

We start considering all the available versions of each API and measure the popularity in terms of how many clients were actually using it at the time of our data collection. In the example in Figure 4.3, we would count popularity as 1 for v7, 2 for v6, and 2 for v4. The column 'number of clients' in Table 4.1 specifies the absolute number of clients per each API and Figure 4.4 reports the version popularity results, by API.

- **Third-party APIs.**

  The general trend shows that a large number of clients use different versions of the APIs and that there is significant fragmentation between the versions (especially in the case of Hibernate, where the top three versions are used by less than 25% of the clients). Further, the general trend is that older versions of the APIs are more popular.

  This initial result may indicate that clients have a delayed upgrading behavior, which could be related with how they deal with maintenance and deprecated methods. For this reason, we analyze whether the clients updated or never updated their dependencies. In the example in Figure 4.3, we count three clients who upgraded version in their history. If projects update we measure how long they took to do so (time between the release of the new version of the API in Maven central and when the project's POM file is updated).

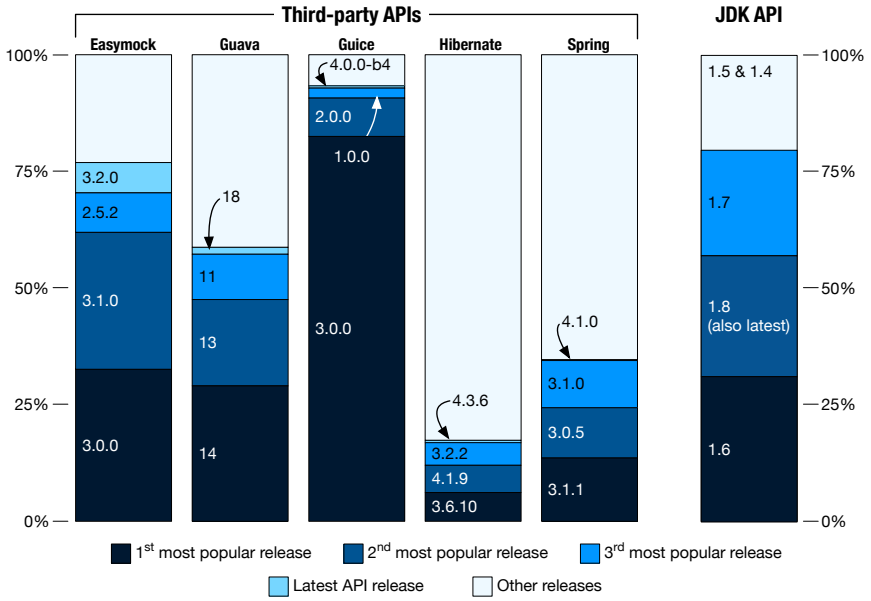


Figure 4.4: Popularity breakdown of versions, by API

Table 4.2: Update behavior of clients, by API

	updated clients		update time (in days)			
			mean	median	Q_1	Q_3
Easymock	63	10%	404	272	103	592
Guava	610	20%	140	72	32	139
Guice	49	8%	783	909	251	1,150
Hibernate	2,454	41%	245	63	33	368
Spring	11,112	74%	195	69	37	186
JDK	19	33%	745	1,507	996	1,738

Table 4.2 summarizes the results. Most clients freeze to one version of the API they use. This holds for all the APIs except for Spring, whose clients have at least one update in 74% of the cases. In terms of time to update, the median is lower for clients of APIs that have more clients updating, such as Hibernate and Spring. In general, update time varies considerably—we will come back to this in RQ3.

- **JDK API.**
  When doing this analysis, we notice one anomaly: two of the projects we analyzed have no bytecode associated with it. This is because the developers who released these projects to Maven central, released them only as Javadoc JAR files, without source code files. During our data collection process, we ensured that anything marked as Javadoc was not downloaded, however, in the case of these projects they were not marked as Javadoc but as source files. This seems to indicate that a non-

negligible amount of Maven Central JAR files are not particularly useful or do not provide source code in some way. From this point forward, we discard the two projects made only of Javadoc and continue focusing on the 58 projects for which we do have the source code.

The clients are evenly distributed among versions 1.6, 1.7, and 1.8. Java 1.5 lags behind these three other versions, but despite being more than 13 years old, 11 clients adopt it. The number of clients using Java 1.6 is 18; despite that, there have been several years for Java clients to update to Java 1.7 (released in 2011) or Java 1.8 (2014).

Only 19 out of the 58 clients ever change the version of Java that is being used. The median time to update is almost 5 years; this is to be expected since Java has few major releases in any given timespan.

4.3 RQ1: How does API method deprecation affect clients?

Answering RQ0, we found that most clients do not adopt new API versions. We now focus on *the clients that use deprecated methods* and on whether and how they react to deprecation.

Affected by deprecation. From the data, we classify clients into 4 categories, which we describe referring to Figure 4.3:

- *Unaffected:* These clients never use a deprecated method. None of the clients in Figure 4.3 belong to this category.
- *Potentially affected:* These clients do not use any deprecated method, but should they upgrade their version, they would be affected. Client 1 in Figure 4.3 belongs to this category.
- *Affected:* These clients use a method which is already in a deprecated state, but do not change the API version throughout their history. It happens in the case of Client 2.
- *Affected and changing version:* These clients use at least one method which gets deprecated *after* updating the API version being used. Clients 3, 4, and 5 belong to this category.

Figure 4.5 reports the breakdown of the clients in the four categories.

- **Third-party APIs.**
  Across all third-party APIs, *most clients never use any deprecated method* throughout their entire history. This is particularly surprising in the case of Hibernate, as it deprecated most of its methods (we discuss this in RQ3). Clients affected by deprecation vary from more than 20% for Easymock and Guava to less than 10% for Hibernate and almost 0% for Spring. Of these clients, less than one third also change their API version, thus highlighting a stationary behavior of clients with respect to API usage, despite our selection of active projects.

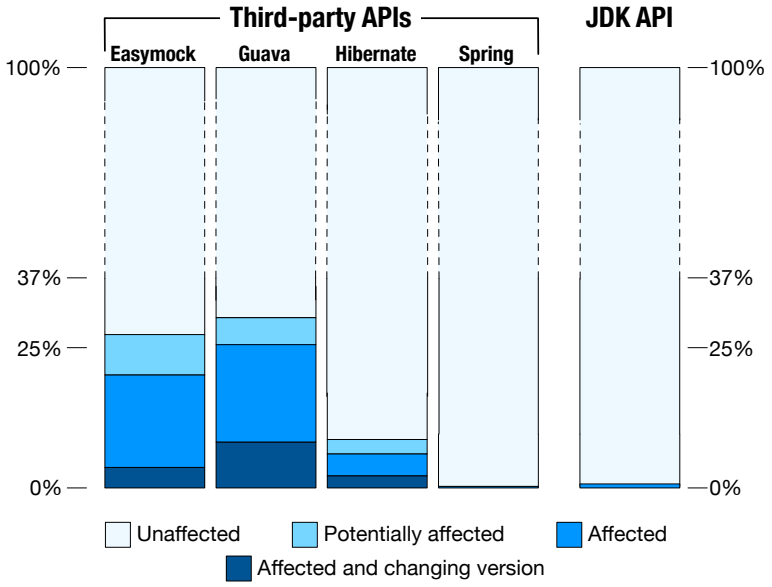


Figure 4.5: Deprecation status of clients of each API

Common reactions to deprecation. We investigate how 'Affected and changing version' clients deal with deprecation. We exclude 'Affected' clients, which do not have strong incentives to fix a deprecation warning if they do not update their API, as the method is still functional in their version.

71% and 65% of 'Affected and changing version' clients of Easymock and Guava react to deprecated entities. For Hibernate and Spring, we see 31% and 32% of clients that react. For all the APIs, the number of clients that fix all calls made to a deprecated entity is between 16% and 22%. Out of the clients that react, we find that at the method level, *the most popular reaction is to delete the reference to the deprecated method* (median of 50% to 67% for Easymock, Guava and Hibernate and 100% for Spring). We define as 'deletion' a reaction in which the deprecated entity is removed and no new invocation to the same API is added.

Some Hibernate and Guava clients roll back to a previous version where the entity is not yet deprecated. Easymock, Guava, and Hibernate clients tend to replace deprecated calls with other calls to the same API, however, this number is small. In contrast to what one would expect as a reaction to deprecation (due to the semantics of the deprecation warning), a vast majority of projects (95 to 100%) add calls to deprecated API elements, despite the deprecation being already in place. This concerns even clients that migrate all their deprecated API elements later on.

The strange case of Guice. We analyzed all the Guice clients and looked for usage of a deprecated annotation or method, however, we find that *none of the projects* have ever used deprecated entities. The reason is that Guice does not have many methods or annotations that have been deprecated since it follows a very aggressive deprecation policy. In Guice, methods are removed from the API without being deprecated previously. We observed this behavior in the Pharo ecosystem as well and studied it separately [89]. In our next research questions (RQ2 - RQ5), thus, *we do not analyze Guice*, as the deprecations are not explicitly marked. However, we do not remove Guice from our study to keep it organic and contrast the deprecation policy it uses with that of other APIs in RQ6.

- **JDK API.**
  We see a surprising trend in the case of the Java clients, as also reported on the rightmost bar in Figure 4.5. Only 4 out of 58 projects are affected by deprecation. Also, none of these projects are among those that change the version of the API that they use; this implies that at the time of usage of the deprecated feature, the client already knew that it was deprecated.

4.4 RQ2: What is the scale of reaction in affected clients?

The work of Robbes *et al.* [3] measures the reactions of individual API changes in terms of commits and developers affected. Having exact API dependency information, we can measure API evolution on a per-API basis, rather than per-API element. Hence, we can measure the magnitude of the changes necessary between two API versions in terms of the number of methods calls that need to be updated between two versions. Another measure of the difficulty of the task is the number of *different* deprecated methods one must react to: It seems reasonable to think that adapting to 10 usages of the same deprecation is easier than reacting to 10 usages of 10 different deprecated methods.

- **Third-party APIs.**
  We consider both 'actual reactions' and 'potential ones'.

  Actual reactions. We measure the scale of the actual reactions to API changes. We count separately reactions to the same deprecated method and the number of single reactions. In Figure 4.3, client 3, after upgrading to v5 and before upgrading to v6, makes two modifications to statements including the deprecated method 'boo'. We count these as two reactions to deprecation but count one unique deprecated method. We consider that client 5 reacts to deprecation when rolling back from v5 to v4: We count one reaction and one unique deprecated method.

  We focus on the upper half of the distribution (median, upper quartile, 95th percentile, and maximum), to assess the critical cases; we expect the effort needed in the bottom half to be low. Table 4.3 reports the results. The first column reports the absolute number of non-frozen affected clients that reacted. The scale of reaction varies: Most of the clients react to less than a dozen of statements with a single unique deprecated method involved. Spring stands out with 31 median number of

statements with reactions and 17 median number of *unique* deprecated methods involved. Outliers invest more heavily in reacting to deprecated methods. As seen next, this may explain the reluctance of some projects to update.

Table 4.3: Scale of actual clients' reaction to method deprecation

	non-frozen affected clients that reacted	statements with reaction (unique deprecated methods involved)			
		median	Q_3	95th perc.	max
Easymock	17	11 (1)	21 (2)	109 (3)	109 (3)
Guava	161	3 (1)	8 (2)	127 (5)	283 (10)
Hibernate	40	5 (1)	20 (16)	41 (27)	59 (40)
Spring	10	31 (17)	54 (21)	104 (27)	131 (27)

Potential reactions. Since a large portion of projects do not react, we investigated how much work was accumulating should they wish to update their dependencies. We thus counted the number of updates that a project would need to perform to render their code base compliant with the latest version of the API (*i.e.*, removing all deprecation warnings). In Figure 4.3, the only client that is potentially affected by deprecation is client 1, which would have two statements needing reaction (*i.e.*, those involving the method 'foo') and one unique deprecated method is involved.

As before, we focus on the upper half of the distribution. Table 4.4 reports the results. In this case, the first column reports the absolute number of clients that would need a reaction. We notice that most of the clients use two or less unique deprecated methods. However, they would generally need to react to a higher number of statements, compared to the clients that reacted reported in Table 4.3, except for those using Spring.

Overall, if the majority of projects would not need to invest a large effort to upgrade to the latest version, a significant minority of projects would need to update a lot of methods. This can explain their reluctance to do so. However, this situation, if left unchecked—as is the case now—can and does grow out of control, especially if these APIs start removing the deprecated features. If there is a silver lining, it is that the number of unique methods to update is generally low, hence the adaptations can be systematic. Outliers would have several unique methods to adapt to.

- **JDK API.**
  We consider only 'potential reactions' in the case of the JDK API, as there can be no 'actual reactions'. We analyzed all the 58 clients, fast-forwarding them through all the JDK APIs version to see whether they would be affected by deprecation, should they upgrade. We found that *none* of these clients would be affected in the event that they would all upgrade to the latest version of the JDK. In other words, none

Table 4.4: Scale of potential clients' reaction to method deprecation

	clients potentially needing reaction	statements potentially needing reaction (unique deprecated methods involved)			
		median	Q_3	95th perc.	max
Easymock	178	55 (1)	254 (1)	1,120 (5)	4,464 (7)
Guava	917	12 (1)	42 (2)	319 (7)	8,568 (44)
Hibernate	521	15 (1)	35 (1)	216 (2)	17,471 (140)
Spring	41	3 (1)	4 (1)	51 (2)	205 (55)

of the analyzed clients use features that have been deprecated in newer versions of Java.

Overall, reflecting on the reasons why deprecation has almost zero impact on these clients (answers to RQ0, RQ1, and RQ2), we can hypothesize that the tight integration of the JDK API in the most popular IDEs and the amount of documentation available to aid a developer in selecting the most appropriate API feature play a role on the facts we encountered. For the purpose of this study, we cannot continue with the other research questions (except for RQ6, in which we cluster the API deprecation behavior of JDK API) given that there is no reaction data to be analyzed; however, we discuss (Section 4.10.2) on why we found such a low number of clients affected by deprecation in the case of the JDK API.

4.5 RQ3: What proportion of deprecations does affect clients?

The previous research question shows that most of the actual and potential reactions of third-party API clients to method deprecations involve a few unique methods. This does not tell us how these methods are distributed across all the deprecated API methods. We compute the proportion of deprecated methods clients use.

In Figure 4.3, there is at least one usage of deprecated methods 'boo' and 'foo', while there is no usage of 'goo'. In this case, we would count 3 unique deprecated methods, of which one is never used by clients.

Table 4.5 summarizes the results, including the proportion of deprecated methods per API over the total count of methods and the count of how many of these deprecated methods are used by clients. APIs such as for Guava, Spring, or Hibernate have more than 1,000 deprecations. For Hibernate, 65% of unique methods get eventually deprecated, indicating that this API makes a heavy usage of this Java feature. The proportion of deprecated methods that affect clients is, around 10% in all 4 of the APIs.

Table 4.5: Deprecated methods affecting clients, by API

	unique deprecated methods			
	defined by API		used by clients	
	count	% over total	count	% over all deprecated
Easymock	124	20%	16	13%
Guava	1,479	10%	104	7%
Hibernate	7,591	65%	487	6%
Spring	1,320	3%	149	11%

4.6 RQ4: What is the time-frame of reaction in affected clients?

We investigate the amount of time it takes for a method to become deprecated ('time to deprecation') and the period of time developers take to react to it ('time to react') to see if developers react as soon as they notice that a feature they are using is deprecated. The former is defined as the interval between the introduction of the call and when it was deprecated, as seen in client 3 (Figure 4.3); the latter is the amount of time between the reaction to a deprecation and when it was deprecated (clients 3 and 5).

Time to deprecation. We analyzed the 'time to deprecation' for each of the instances where we found a deprecated entity. The median time for all API clients is 0 days: Most of the introductions of deprecated method calls happen when clients already know they are deprecated. In other words, when clients introduce a call to a deprecated entity, that they know *a priori* that the entity is already deprecated. This seems to indicate that clients do not mind using deprecated features.

Time to react. Figure 4.6 reports the time it takes clients to react to a method deprecation, once it is visible. We see that, for most clients across all APIs, the median reaction time is 0 days for Guava, Hibernate, and Spring, while for Easymock it is 25 days. A reaction time of 0 days can indicate that most deprecated method invocations are reacted upon on the same day the call was either introduced or marked as deprecated. To confirm this, we looked a little deeper at 20 of these cases to see why the reaction time is 0 days. We see that in all of the cases the developers have upgraded a version of the API they use, this leads to them noticing that a feature they use is now deprecated. They react to this deprecation immediately after the upgrade in version, thus resulting in a reaction time of 0 days.

Barring outliers, reaction times for Hibernate and Spring are in the third quartiles, being at 0 and 2.5 days. Reaction times are longer for clients of Guava and Easymock, with an upper quartile of 47 and 200 days respectively. Outliers have a long reaction time, in the order of hundreds of days. We looked individually at the 9 outliers that have a reaction time in excess of 300 days. Distilling the actual rationale behind a change is non-

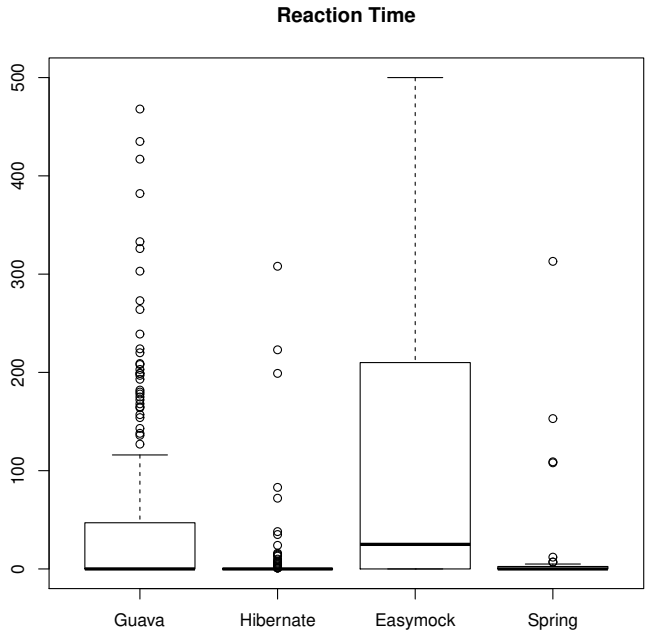


Figure 4.6: Days taken by clients to react to a method deprecation once visible.

trivial. We look at the commit messages and any kind of code comments that might exist. Only one commit message actually references the fact that a deprecated entity was being reacted to. The other 8 commit messages do not add any information. We also do not see any code comments that might explain the rationale. We can at best speculate that the reaction to deprecation takes place as part of a general code cleanup act.

4.7 RQ5: Do affected clients react similarly?

This research question seeks to investigate the behavior of third-party API clients when it comes to replacement reactions.

Such an analysis allows us to ascertain whether an approach inspired by Schäfer *et al.*'s [90] would work on the clients in our sample. Their approach recommends API changes to a client based on common, or systematic patterns in the evolution of other clients of the same API.

4.7.1 Consistency of replacements.

There is no definite way to identify if a new call made to the API is a replacement for the original deprecated call, so we rely on a heuristic: We analyze the co-change relationships in each class file across all the projects; if we find a commit where a client removes a usage of a deprecated method (*e.g.*, `add(String)`) and adds a reference to another method in the same API (*e.g.*, `add(String, Integer)`), this new method invocation is a possible re-

placement for the original deprecated entity. A drawback is that in-house replacements or replacements from other competing APIs cannot be identified. Nonetheless, we compute the frequencies of these co-change relationships to find whether clients react uniformly to a deprecation.

We found that Easymock clients show no systematic transitions: There are only 3 distinct methods for which we see replacements and the highest frequency of the co-change relationships is 34%. For Guava, we find 23 API replacements; in 17% of the cases there is a *systematic* transition *i.e.*, there is only one way in which a deprecated method is replaced by clients. Spring clients only react by deleting deprecated entities instead of replacing them, resulting in no information on replacements of features. In the case of Hibernate clients, we find only 4 distinct methods where replacements were made. There were no systematic replacements and the maximum frequency is 75%.

Since API replacements are rather uncommon in our dataset, with the exception of Guava clients, we find that while an approach such as the one of Schäfer *et al.* could conceptually be quite useful, we would not be able to implement it in our case due to the small amount of replacement data.

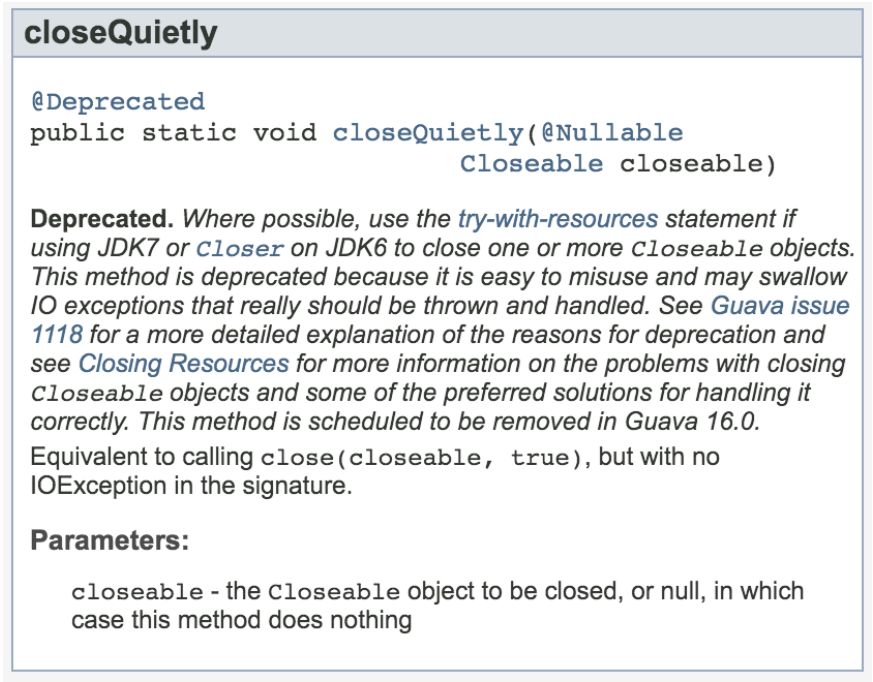
4.7.2 Quality of documentation.

Very few clients react to deprecation by actually replacing the deprecated call with one that is not deprecated. This led us to question the quality of the documentation of these APIs. Ideally one would like to have a clear explanation of the correct replacement for a deprecated method, as in the Javadoc reported in Figure 4.7. However, the results we obtained made us hypothesize otherwise. We systematically inspected the Javadoc to see whether deprecated features had documentation on why the feature was deprecated and whether there was an indication of appropriate replacement (or if a replacement is needed).

We perform a manual analysis to analyze the quality of the API documentations. For Guava, we investigate all 104 deprecated methods that had an impact on clients; for Easymock, we look at all 16 deprecated methods that had an impact on clients; for Spring and Hibernate, we inspected a sample of methods (100 each) that have an impact on the clients.

In Easymock, 15 of the 16 deprecated methods are instance creation methods, whose deprecation message directs the reader to use a Builder pattern instead of these methods. The last deprecation message is the only one with a rationale and is also the most problematic: the method is incompatible with Java version 7 since its more conservative compiler does not accept it; no replacement is given.

In Guava, 61 messages recommend a replacement, 39 of which state that the method is no longer needed and hence can be safely deleted, and 5 deprecated methods do not have a message. Guava is also the API with the most diverse deprecation messages. Most messages that state a method is no longer needed are rather cryptic ("no need to use this"). On the other hand, several messages have more precise rationales, such as stating that functionality is being redistributed to other classes. Others provide several alternative recommendations and detailed instructions and one method provides as many as four alternatives (although this is because the deprecated method does not have exact equivalents), Guava also specifies in the deprecation message when entities will be removed (*e.g.*, "This method is scheduled for removal in Guava 16.0", or "This method is scheduled for deletion in June 2013.").



closeQuietly

```
@Deprecated
public static void closeQuietly(@Nullable
                                Closeable closeable)
```

Deprecated. *Where possible, use the try-with-resources statement if using JDK7 or* `Closer` *on JDK6 to close one or more* `Closeable` *objects. This method is deprecated because it is easy to misuse and may swallow IO exceptions that really should be thrown and handled. See Guava issue 1118 for a more detailed explanation of the reasons for deprecation and see Closing Resources for more information on the problems with closing* `Closeable` *objects and some of the preferred solutions for handling it correctly. This method is scheduled to be removed in Guava 16.0.*

Equivalent to calling `close(closeable, true)`, but with no IOException in the signature.

Parameters:

`closeable` - the `Closeable` object to be closed, or null, in which case this method does nothing

Figure 4.7: Example of Javadoc associated with deprecated API artifact

For Hibernate, all the messages provide a replacement, but most provide no rationale for it. The only exceptions are messages stating the advantages of a recommended database connection compared to the deprecated one.

For Spring, the messages provide a replacement (88) or state that the method is no longer needed (12). Spring is the only API that is consistent in specifying in which version of the API the methods were deprecated. On the other hand, most of the messages do not specify any rationale for the decision, except JDK version testing methods that are no longer needed since Spring does not run in early JDK versions anymore.

Overall, maintainers of popular APIs provide their clients with sufficient support to clients concerning deprecation. We found rationales as to why a method was deprecated, but not systematically. Replacement is the most commonly suggested solution; this is in contrast to the actual behavior of clients who instead prefer removing references to deprecated entities as opposed to replacing them, as reported in 5.3.1.

4.8 RQ6: How are clients impacted by API deprecation policies?

During this study, we noticed that each API has its own way to deprecate features. It seems reasonable to think that this deprecation policy of features may impact a clients' decision to adopt and react to these deprecated features. We thus decided to look at this

particular issue in more detail.

4.8.1 Methodology

To see what different kind of policies of deprecation exist, we first aimed to look at the top 50 APIs that are popularly used by GitHub based Java projects, to get a sufficiently diverse set of APIs, with a sufficient number of clients that may react differently. However, looking at only the top 50 most popular APIs might have one downside: Given that many of the APIs have the same vendor, the deprecation policy adopted by the APIs from the same vendor may be similar. To overcome this limitation, we looked at the top 200 APIs in terms of popularity (ranked based on usage of the API among Java projects on GitHub) and selected the first 50 that had different vendors, this resulted in the APIs listed in Table 6.1.

Once the APIs had been selected we defined certain criteria to categorize their deprecation behavior on:

Number of deprecated: Number of unique features deprecated during the entire history of the API. The larger the number of features that are deprecated by an API, the larger the chance is that a client using that API will be affected by deprecation.

Percentage of deprecated: Percentage of total features deprecated during the entire history of the API. When an API deprecates a large portion of its features, there is a higher chance that it might deprecate features that are being used.

Time to deprecate: Median time, in days, taken to deprecate a feature from the moment it was introduced in the API. A long time to deprecate can indicate that API developers do not change their API at a fast pace. This fact can be reassuring to clients who are ensured the stability of the API and its features.

Time to remove: Median time, in days, taken to remove a deprecated feature after the moment at which it was deprecated. A short removal time gives API clients a very short window within which they can react to the deprecation of the feature, after which the change becomes a breaking change. A longer removal time may indicate that the API does not perform regular cleanup of its code.

Rollbacks: Number of times a deprecated method was marked in a future release as non-deprecated. A rollback may be performed because the API developers changed their mind about deprecating a feature. This would send a confusing signal to the API client as they cannot be sure about the future of the feature that they are using. Ideally, this behavior should be avoided, because it gives no clear indication about the future of a feature.

Percentage of removed: Percentage of deprecated features eventually removed from the API. A high percentage suggests that the API performs a lot of cleanup of its deprecated features. On the other hand, a low percentage indicates that the API is lax about removing deprecated features, thus allowing clients to assume they do not have to react to deprecation.

Number of never-removed: Number of deprecated features that were never removed from the API and that are still present despite being deprecated. An API leaving a

Table 4.6: List of 50 APIs selected for RQ6

API Artifact ID	Domain	Popularity	Rank
junit	Testing	67,954	1
slf4j-api	Logging	18,521	2
log4j	Logging	17,421	3
spring-core	Dependency injection	15,086	4
mysql-connector-java	Database	14,333	6
servlet-api	Server	12,044	8
jstl	Server	12,007	9
commons-io	IO utility	10,821	12
guava	Collections	9,542	14
hibernate-entitymanager	Object relational mapper	8,413	16
logback-classic	Logging	7,597	20
mockito-all	Testing	7,010	22
commons-lang	Utility	6,485	26
jackson-databind	JSON handling	5,905	28
httpclient	Server	5,584	32
commons-dbcp	Database	5,486	34
joda-time	Time utility	5,045	36
commons-logging	Logging	4,947	37
aspectjrt	Aspect oriented	4,685	38
testng	Testing	4,485	40
commons-codec	Codec utility	4,337	41
commons-fileupload	Fileupload utility	4,001	45
h2	Database	3,952	46
postgresql	Database	3,816	47
hsqldb	Database	3,633	49
validation-api	Bean validation	3,509	52
commons-collections	Collections	3,406	54
json	JSON handling	2,952	56
hamcrest-all	Testing	2,867	57
cglib	Bytecode generation	2,816	60
selenium-java	Browser	2,694	61
lombok	Eclipse	2,472	65
javassist	Bytecode generation/manipulation	2,466	66
jsoup	HTML parser	2,333	67
mybatis	Database	2,199	72
standard	Tagging library	2,150	74
commons-beanutils	Java beans	2,147	75
mongo-java-driver	Database	2,077	78
poi	File format manipulation	1,940	83
commons-cli	Command line utility	1,855	84
jersey-client	Server	1,844	85
dom4j	HTML parser	1,798	87
c3p0	Database	1,782	88
commons-httpclient	HTTP Client	1,676	91
primefaces	UI building	1,608	95
commons-pool	Object pooling	1,583	96
guice	Dependency injection	1,556	97
freemarker	Java beans	1,531	98
assertj-core	Testing	1,527	99
easymock	Testing	1,484	100

lot of its deprecated features never-removed may signal that the client should not worry about their code breaking in the near future. Ideally, this number should be quite high given that the traditional pattern of deprecation is to first deprecate a feature and then after an interval remove it from the API.

Average deprecations per version: Average number of features deprecated per version of the API. A high average number of deprecations per version tells us that the API is very volatile, which might factor into a client's decision on upgrading the version of the API being used.

Average removals per version: Average number of deprecated features removed per version of the API. If the removals per version are high, then this adversely impacts a client's decision to change the version being used as making a change ensures that their code would break.

4.8.2 Clustering

Using these dimensions, we can run a clustering algorithm on the APIs to see whether clusters emerge and their nature. The most widespread clustering algorithm which fits our model is *k*-means [91]. The *k*-means clustering algorithm aims to partition a set of elements into *k* clusters where each element in the set belongs to a cluster with the nearest mean. One issue with this clustering technique *which is unsupervised*, is the estimation of the number of clusters to be produced. To do so we choose to use the elbow method [92], that allows for a visual estimation of the value of *k*.

The elbow method looks at the percentage of variance explained as a function of the number of clusters. After a point, adding more clusters in the *k*-means algorithm should not result in a better estimation of the data. Using this technique, we calculated the sum of squares within each cluster for each number of clusters (where the number of clusters varied from 1 to 15) provided to *k*-means algorithm and plotted these sums. Looking at the plot, we determined that the number of clusters that best describes our data is 7. Using this value as our input for the *k*-means algorithm, we could establish what the clusters are, what are the characteristics of these clusters, and which APIs belongs to them.

In Figure 4.8, we see a silhouette plot of the clusters that we have obtained from our data. A silhouette plot essentially provides a succinct graphical representation of how well each object lies within its cluster. The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from -1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate.

Looking at Figure 4.8 we see that our clusters are separate from each other. In most cases, the silhouette values are positive and high. There are only 4 negatives in each cluster, thus indicating that most objects in each cluster are matched with the others. We see that there are two clusters with just one API each (Apache commons-collections and Java), these APIs appear to be outliers in our dataset, in particular, this was expected in the case of the JDK API.

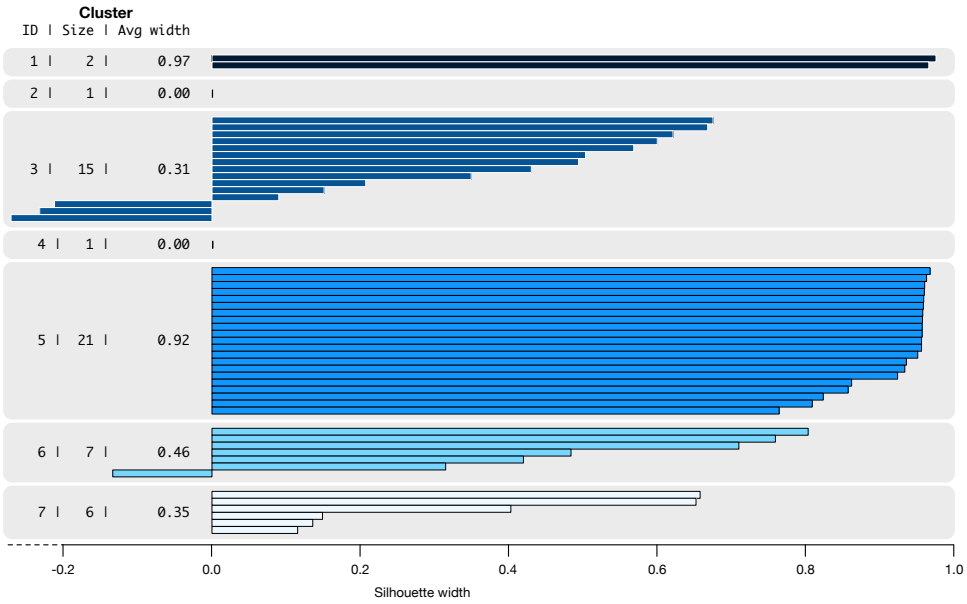


Figure 4.8: Silhouette plot of the 7 clusters

4.8.3 Results

We use the dimensions listed above to characterize an API's policy of deprecating its features. The goal is to create a categorization of the APIs that are under consideration. Analyzing the clusters, we derive the defining characteristics of a project that falls in each cluster, and how the clients of such a project might potentially react. A summary of this can be found in Table 4.7. To not bias ourselves, we choose to not look at the APIs that we have already studied so that we do not allow our previous results to dictate the properties of each cluster. In the following we describe the characteristics of each cluster and discuss their potential implications:

1. **Cluster 1:** In this cluster (2 elements), the deprecation times and the removal times are both more than 10 years, as opposed to cluster 7 where the deprecation and removal times were less than 3 years. This implies that in this case when the APIs in this cluster deprecate or remove a feature, it is often those features that were introduced early in the API that are affected. However, these APIs also deprecate very few features in total and less than 6 features per version. Thus, the scope of a client being affected by deprecation is minimal at best.

2. **Cluster 2:** This cluster has only one element. This API (Apache Commons-collections) is the one out of all 50 APIs that deprecates the largest percentage (30%) of its API in its history. The median times for deprecation and removal are also really low for this API, and in many instances, features are introduced in a deprecated state. This API is also very good at cleaning up its code base by removing 83% of all deprecated

Table 4.7: Summary of clusters

Cluster	Characteristic	Number of APIs
1	Very high deprecation time and removal time	2
2	Large portions of the API are deprecated	1
3	Deprecated features are removed with urgency	15
4	Deprecate very little and take a very long time to do so	1
5	Deprecate a lot at the time of introduction, most likely for experimental features	21
6	Deprecates a lot of features, many of which that are not too old	7
7	Features are not deprecated or removed very easily	6

features at some future release. Given the large usage of deprecation in this API, we can conclude that clients have to be wary about what feature they use.

3. **Cluster 3:** The APIs that fall into this cluster (15 elements) usually have a high median time to deprecate a feature (more than 2 years), but on the other hand have a short period to remove the feature (less than 30 days). This gives a client limited opportunity to react to a deprecation as the deprecated entity will likely be removed in an upcoming release. This might influence a client to not upgrade the version of the API being used.

4. **Cluster 4:** There is just one element in this cluster: The JDK API. This cluster is characterized by the fact that a very small percentage of the API is deprecated (less than 1%), and for the features that have been deprecated, there are 621 rollbacks in deprecation in its lifetime. The median time to deprecate is less than 3 years, and in the event there are removals of a deprecated entity, then the time to remove is very long, more than 6 years. The number of deprecations per version is high in comparison to other clusters, so is the number of features removed per version. Given that a very small percentage of the API is deprecated and the removal time being high, we suppose that clients of this API are not affected by deprecation to a large extent and those that are affected do not react at all.

5. **Cluster 5:** In this cluster (21 elements), we see the most number of APIs. All these APIs in general do not deprecate a large percentage of their APIs. However, when they do, it is generally done immediately after the introduction of the feature. This generally affects only those features that have been introduced later in the API's life. Thus, only those clients that adopt the latest feature are going to be the ones that are affected to a great extent. But given the fact that very few features are deprecated, we assume the number of affected clients to be very low for these APIs.

6. **Cluster 6:** In this cluster (7 elements), the APIs have a low median deprecation time and median removal time both under 40 days. At the same time, the projects in this cluster deprecate quite a lot of methods per version. Many features that are new and introduced in one release are generally deprecated and rendered obsolete in the immediate future. This kind of behavior may discourage a client from adopting new features of the API, due to the fear of being forced to update in the near future. Thus, for the APIs here we should see minimal clients affected by deprecation as most might not even adopt the features that get deprecated.

7. **Cluster 7:** This cluster (6 elements) is characterized by median removal and deprecation times higher than 3 years, thus giving the clients of the APIs in this category the safety of not having to worry about their code breaking in any manner. This policy may not incentivize clients of the API to react to a deprecated feature.

Based on the cluster definitions we make the following hypotheses, which can be tested in future work:

1. **Hypothesis 1:**
   Clients of cluster 1, 4 and 7 do not react to deprecated entities as opposed to clients of APIs that belong to other clusters.

2. **Hypothesis 2:**
   Clients of cluster 3 will react to deprecation with a low reaction time, otherwise, their code will break.

3. **Hypothesis 3:**
   Clients of cluster 6 will not adopt newer features since these features are the ones that are generally deprecated.

4. **Hypothesis 4:**
   Clients of cluster 5 will not be affected by deprecation due to the fact a lot of features are experimental and marked as deprecated due to this.

5. **Hypothesis 5:**
   Clients of cluster 2 will be affected by deprecation regularly due to the fact that the API deprecates a lot.

In Table 4.8 we see in the category into which each of the APIs that we study fits into. In the case of Guava and Hibernate, both fall into cluster 3. We expect to see their clients not upgrading to the latest version of the API in most cases. This is reflected in the results of RQ0 as well, where we see that both for Guava and Hibernate the latest releases has very minimal adoption, whereas older releases have been adopted to a much larger extent.

Spring fits into the seventh cluster, where the clients are not expected to be impacted by deprecation. We see that reflected in the results of RQ1, where Spring has the least number (and percentage) of clients affected by deprecation. This is to the credit of the Spring developers who have adopted a policy that does not have any adverse impact on its clients.

Table 4.8: API deprecation characteristics

API	Spring	Hibernate	Guava	Easymock	Java
Number deprecated	551	110	529	95	1,910
Percentage deprecated	11%	1%	3%	4%	1%
Time to deprecate	1,454	1,043	620	0	927
Time to remove	1,456	698	228	1,557	3,456
Rollbacks	10	0	47	2	629
Percentage removed	45%	100%	60%	90%	59%
Number unremoved	295	0	172	7	152
Average deprecations per version	4	0	11	7	239
Average removals per version	2	0	6	6	141
Cluster	1	2	2	3	7

Easymock falls into cluster 6, and here we expect to see a minimal number of clients to be affected, given that only new features are deprecated, whereas the ones that were originally introduced are not necessarily deprecated by the API. However, we see from RQ1 that there are many clients affected by deprecation. This might indicate that it is not only new features that are being deprecated in Easymock. This result makes Easymock an imperfect fit for the cluster. Looking at Figure 4.8, this does indeed appear to be the case, all elements in cluster 6 do not appear to fit with each other to make it a homogeneous cluster. Given that we did not allow our cluster definition to be defined by the data that we already had in place, it is to be expected that for some of these APIs, the cluster fit would be imperfect.

4.9 Summary of findings

We first investigated how many API clients actively maintain their projects by updating their dependencies. We found that, for all the APIs including the JDK API, only a minority of clients upgrade/change the version of the API they use. As a direct consequence of this, older versions of APIs are more popular than newer ones.

We then looked at the number of projects that are affected by deprecation. We focused on projects that change version and are affected by deprecation as they are the ones that show a full range of reactions. Clients of Guava, Easymock, and Hibernate (to a lesser degree) were the ones that were most affected, whereas clients of Spring were virtually

unaffected by deprecation. For Guice, we could find no data due to Guice's aggressive deprecation policy. In the case of the JDK API, we found very few clients to be affected, but none of them changed versions, thus we could not analyze their reaction data. We also found that for most of the clients that were affected, they introduced a call to a deprecated entity, despite knowing that it was deprecated.

Looking at the reaction behavior of these clients, we saw that 'deletion' was the most popular way to react to a deprecated entity. Replacements were seldom performed, and finding systematic replacements was rarer. This is despite the fact that these APIs provide excellent documentation that should aid in the replacement of a deprecated feature. When a reaction did take place, it was usually almost right after it was first marked as deprecated.

As a final step, we looked at how the different APIs deprecate their features and how such a deprecation policy can impact a client. We clustered 50 APIs based on certain characteristics (such as the number of deprecated API elements, and the time to remove deprecated API elements), and documented the patterns that emerged in seven clusters. For each cluster, we define its primary characteristic and predict the behavior of a client that uses an API that belongs to the clusters, leading to five hypotheses that can be confirmed or infirmed in future work. We see that in the case of Guava, Hibernate and Spring clients our clusters fit perfectly. However, in the case of Easymock, the fit is not as good. This suggests that further investigation is needed in this case.

4.10 Discussion

We now discuss our main findings and contrast them with the findings of the Smalltalk study we expand upon. Based on this, we give recommendations on future research directions.

4.10.1 Comparison with the deprecation study on Smalltalk

Contrasting our results with those of the study we partially replicate, several interesting findings emerge:

Proportion of deprecated methods affecting clients

Both studies found that only a small proportion of deprecated methods affects clients. In the case of Smalltalk, this proportion is below 15%, but in our results we found it to be around 10%. Considering that the two studies investigate two largely different ecosystems, languages, and communities, this similarity is noteworthy. Even though API developers do not know exactly how their clients use the methods they write and would be interested in this information [93], the functionalities API developers deprecate are mostly unused by the clients, thus deprecation causes few problems. Nevertheless, this also suggests that most effort that API developers make in properly deprecating some methods and documenting alternatives is not actually necessary: API developers, in most of the cases, could directly remove the methods they instead diligently deprecate.

Not reacting to deprecation

Despite the differences in the deprecation mechanisms and warnings, most of the clients in both studies do *not* react to deprecation. In this study, we could also quantify the impact

of deprecation should clients decide to upgrade their API versions and find that, in some cases, the impact would be very high.

By not reacting to deprecated calls, we see that the technical debt accrued can grow to large and unmanageable proportions (*e.g.*, one Hibernate client would have to change 17,471 API invocations).

One reason behind the non-reaction to deprecation might be that some of these deprecated entities find themselves in dead-code regions of API client code as opposed to essential parts. This might impact the client's decision to react. However, the impact of this is hard to determine given that the cost of executing thousands of API clients in representative execution scenarios is prohibitive (assuming it is even possible in the first place).

We see that in many cases the preferred way to react to deprecation is by deleting the invocation. This reaction pattern might be due some APIs advising that the deprecated feature need not be used anymore and can be safely deleted with no replacement. The impact of this might be quite high given our findings in Section 4.7.

We also found more counter-reactions (*i.e.*, adding more calls to methods that are known to be deprecated) than for Smalltalk clients. This may be related to the way in which the two platforms raise deprecation warnings: In Java, a deprecation gives a compile-time warning that can be easily ignored, while in Smalltalk, some deprecations lead to run-time errors, which require intervention.

Systematic changes and deprecation messages

The Smalltalk study found that in a large number of cases, most clients conduct systematic replacements to deprecated API elements. In our study, we find that, instead, replacements are not that common. We deem this difference to be extremely surprising. In fact, the clients we consider have access to very precise documentation that should act as an aid in the transition from a deprecated API artifact to one that is not deprecated; while this is not the case for Smalltalk, where only half of the deprecation messages were deemed as useful. This seems to indicate that proper documentation is not a good enough incentive for API clients to adopt a correct behavior, also from a maintenance perspective, when facing deprecated methods. As an indication to developers of language platforms, we have some evidence to suggest more stringent policies on how deprecation impacts clients' run-time behavior.

Clients of deprecated methods

Overall, we see in the behavior of API clients that deprecation mechanisms are not ideal. We thought of two reasons for this:(1) developers of clients do not see the importance of removing references to deprecated artifacts, and (2) current incentives are not working to overcome this situation. Incentives could be both in the behavior of the API introducing deprecated calls and in the restriction posed by the engineers of the language. This situation highlights the need for further research on this topic to understand whether and how deprecation could be revisited to have a more positive impact on keeping low technical debt and improve maintainability of software systems. In section 4.10.4 we detail some of the first steps in this direction, clearly emerging from the findings in our study.

4.10.2 Comparison between Third-party APIs and the JDK API

We observe that deprecations in the JDK do not affect the JDK clients to a large degree. Only 4 out of 58 projects are affected by deprecation and all these 4 introduced a call to the deprecated artifact despite knowing that it was deprecated. Such a low proportion of JDK clients being affected was an unexpected finding, we rationalize it with the following hypotheses:

Early deprecation of popular JDK features

Some of the more popular or used features of the JDK that have been deprecated, were deprecated in JDK 1.1 (*e.g.*, the Java Date API). For these features, replacement features have been readily available for a long time. As a sanity check, we looked for the usages of the `Date` class in our database on API usages that was mined from GitHub based data. There we see that only 47 projects ever use this class out of 65,437 Java based projects. This indicates that almost all clients already use the replacement features instead of the features that have been deprecated a long time ago.

Nature of deprecated features

Manually analyzing the list of features deprecated in the JDK, we found that many of these features belong to the `awt` and `swing` sub-systems. Both these sub-systems provide GUI features for developers. The nature of the projects hosted on Maven Central is such that most of these projects do not provide a graphical interface as they are, in most cases, intended to be used as libraries. Nevertheless, the analysis of the 65,437 GitHub clients also shows the same behavior, thus mitigating the risk of a sample selection bias. Other than just GUI features, the JDK also has internal features and security features that have been deprecated. These are not intended for public use, hence, we do not see these among the projects that we investigate.

Nature of projects

Our dataset contains all the projects from Maven Central. The fact that a project is released in an official site such as Maven Central indicates that high level of adherence to Software engineering practices among its developers. Given that these projects are in the public eye, and free for all to reuse, developers of these projects must have made every effort to ensure high code quality. This might have resulted in us seeing such low usage of deprecated features. Moreover, our dataset contains information on over 56,000 projects, and for each project, we have data on each release. However, we do not have any information at commit level. This might prevent us from detecting real time usage of a deprecated artifact and any reaction that might take place. All usages of deprecated features might have been taken out by the time the release is made to Maven Central. Thus, we might miss some deprecation information. Nevertheless, results from the 65,437 GitHub clients are in line with the findings from Maven Central.

Documentation of the JDK

The JDK API is the best-documented API out of the ones that we have studied in this chapter. They have detailed reasons behind every deprecation, thus allowing a developer to make an informed choice on reacting to the deprecation. This documentation also mentions the replacement feature that should be used in the event that a developer would

like to react to the deprecated feature. Java is also one of the most popular languages in the world [48, 78], thus leading to the generation of a large amount of community-based documentation (*e.g.*, Stackoverflow, blog posts, and books) that provide a developer with every aid imaginable to use the Java API in the right manner. Also, Java is one of Oracle's most important projects and the company ensures that there are plenty of programming guides available on its own website. This amount of developer support could be one of the reasons why we see very few projects who are affected by deprecation.

Deprecation policy in JDK
The Java developers have made a commitment to not removing deprecated features in any current or future release [94]. However, the JDK developers recommend removing all the deprecated features as soon as possible. The main reason they keep deprecated features is to ensure backward source code compatibility with previous versions. This does not act as an incentive for a developer to change the version of the JDK being used, hence, it might result in fewer projects changing the JDK version and being affected by deprecation.

4.10.3 Impact of deprecation policy

We see in RQ6 that different APIs deprecate their features in different manners. They all differ in terms of time taken to deprecate a feature or remove a feature or the number of features that are removed from the API. Based on different characteristics that we define, we find that we can cluster 50 APIs into 7 distinct clusters, each with their own defining characteristic.

We see that in the case of Guava, Spring, and Hibernate, the clients react as predicted by the clusters in which these APIs found themselves. However, in the case of Easymock, we do not see the expected behavior among its clients. This tells us that the clusters are not perfect in every case and may have to be expanded upon by studying more APIs. However, what we do see is that the deprecation policy adopted by an API does indeed have an impact on its clients, thus providing API developers with an insight into how the deprecation policy they adopt affects a client and what policy they should adopt in the event they want to minimize the impact on their client.

4.10.4 Future research directions

Below we enumerate a couple of promising future lines of research worth pursuing:

If it ain't broke, don't fix it
We were surprised that so many projects did not update their API versions. Those that often do it slowly, as we saw in the cases of Easymock or Guice. Developers also routinely leave deprecated method calls in their code base despite the warnings and even often add new calls. This is despite all the APIs providing precise instructions on which replacements to use. As such the effort to upgrade to a new version piles up. Studies can be designed and carried out to determine the reasons for these choices, thus indicating how future implementations of deprecation can give better incentives to clients of deprecated methods.

Further investigating the deprecation polices
We see that different APIs do actually adopt different deprecation strategies and this appears to have an impact on the clients of these APIs. However, we have been only able to discuss this for the APIs in our analysis. As a further step, one could assess the impact of deprecation policies on the clients for all the APIs that were used to make the clustering. This would reinforce the idea of deprecation policies and their impact on a client.

Impact of deprecation messages
We also wonder if the deprecation messages that Guava has, which explicitly state when the method will be removed, could act as a double-edged sword: Part of the clients could be motivated to upgrade quickly, while others may be discouraged and not update the API or roll back. In the case of Easymock, the particular deprecated method that has no documented alternative may be a roadblock to upgrade. Studies can be devised to better understand the role of deprecation messages and their real effectiveness.

Volume of available documentation
In the case of popular APIs such as the JDK API or JUnit, there is a large amount of documentation that is available. This might impact the reaction pattern to deprecation given that there is likely some document artifact that addresses how to react to a deprecated entity. On the other hand, for smaller or less popular APIs which do not have as much community documentation or vendor based documentation, support for the developer might not be available. Overall, the volume of API documentation might impact the reaction pattern to deprecation, a fact that warrants future investigation.

4.11 Related Work

4.11.1 Studies of API Evolution

Several works study or discuss API evolution, the policies that regard it, and their impact on API developers and clients.

When it comes to an API, one of the first decisions is what to leave out. Many projects have so-called *internal APIs*, that, despite being publically accessible, are reserved for internal usage by the project [95]. They are not intended to be used by clients and may change without warning from one release to the next. Businge *et al.* [95] found that 44% of 512 Eclipse plugins that they analyzed used internal Eclipse APIs, a finding echoed in a larger study by Hora *et al.* [96], that found that 23.5% of 9,702 Eclipse client projects on GitHub used internal APIs. Shedding more light on the issue, a survey by Businge [97] found that 70% of 30 respondents used internal APIs because they couldn't find a public API with the equivalent functionality, and re-implementation of the functionality would be too costly. Hora *et al.* [96] observed that some internal APIs were later promoted to public APIs, and presented an approach to recommend internal APIs for promotion. These findings agree with our study, in that they also show that maintainability often takes a back to functionality, a fact reflected in the unwillingness to update APIs, which can lead to considerable technical debt.

Studies closely related to this chapter (*i.e.*, [3] and [89]), that deal with deprecation policies of APIs and their impact on API clients have been performed on the Pharo ecosystem. The first study focused on API deprecations and their impact on the entire Pharo

ecosystem. The second study focused on API changes that were not marked as deprecations beforehand. They look at the APIs policy to change features and the impact these changes have on the client.

Brito *et al.* [98] analyze deprecation messages in more than 600 Java systems, finding that 64% of deprecated methods have replacement messages. This implies that API clients are provided with support when reacting to deprecation.

While neither Brito's study nor ours look extensively into the reasons why API elements are deprecated, other works did. Hou and Yao [99] studied release notes of the JDK, AWT and Swing APIs, looking for rationales for the evolution of the APIs. In the case of deprecated API elements, several reasons were evoked: Conformance to API naming conventions, naming improvements (increasing precision, conciseness, fixing typos), simplification of the API, coupling reduction, improving encapsulation, or replacement of functionality. Of note, a small portion of APIs elements was deleted without replacements. Our manual analysis of deprecation messages found that was also the case in the APIs that we studied. We found relatively few rationales for deletion of API elements, with the most common reason being that the method concerned was no longer needed.

The versioning policy adopted by an API might give an API client an indication as to what kind of changes could be expected in that version. To that end, Raemaekers *et al.* investigated the relation among breaking changes, deprecation, and the semantic versioning policy adopted by an API [100]. They found that API developers introduce deprecated artifacts and breaking changes in equal measure across both minor and major API versions, thus not allowing clients to predict API stability from semantic versioning.

The evolution policy of Android APIs has been extensively studied. McDonnell *et al.* [101] investigate stability and adoption of the Android API on 10 systems; the API changes are derived from Android documentation. They found that the Android API's policy of evolving quickly leads to clients having troubles catching up with the evolution. Linares-Vásquez *et al.* also study the changes in Android, but from the perspective of questions and answers on Stack Overflow [102], not API clients directly. Bavota *et al.* [103] study how changes in the APIs of mobile apps (responsible for defects if not reacted upon) correlate with user ratings: successful applications depended on less change-prone APIs. This is one of the few large-scale studies, with more than 5,000 API applications.

Web based API evolution policies have also been studied. Wang *et al.* [104] study the specific case of the evolution of 11 REST APIs. Instead of analyzing API clients, they also collect questions and answers from Stack Overflow that concern the changing API elements. Among the studies considering clients of web APIs, we find for example the one by Espinha *et al.* [105], who study 43 mobile client applications depending on web APIs and how they respond to web API evolution.

APIs also break. Dig and Johnson studied and classified the API breaking changes in 4 APIs [26]; they did not investigate their impact on clients. They found that 80% of the changes were due to refactorings. Cossette and Walker [106] studied five Java APIs to evaluate how API evolution recommenders would perform in the cases of API breaking changes. They found that all recommenders handle a subset of the cases, but that none of them could handle all the cases they referenced.

In a large-scale study of 317 APIs, Laerte *et al.* [9] found that for the median library, 14.78% of API changes break compatibility with its previous versions and that the fre-

quency of these changes increased over time. However, not that many clients were impacted by these breaking changes (median of 2.54%). On the topic of breaking APIs, Bogart *et al.* [107] conducted interviews with API developers in 3 software ecosystems: Eclipse, npm, and R/CRAN. They found that each ecosystem had a different set of values that influenced their policies and their decisions of whether to break the API or not. In the Eclipse ecosystem, developers were very reluctant to break APIs, strongly favoring backward compatibility; some methods were still present after being deprecated for more than 10 years. The R/CRAN ecosystem places emphasis on the ease of installation of packages by end-users, so packages are extensively tested to see if they work. As a result, API developers notify their clients of the coming API changes, and may also coordinate with them to quickly resolve issues, a finding also echoed by Decan *et al.* [108]. Finally, the npm ecosystem values ease of change for API developers. Following semantic versioning, breaking the API can be done by changing the major version number of the package. Since packages stay in the repository, clients are free to upgrade or not when this happens.

Regarding the clusters of APIs that we found, APIs in clusters 1, 4, and 7 seem wary of imposing too much work on their clients, and as such seem closer to the strategy employed in the Eclipse ecosystem. APIs in Cluster 5 behave somewhat similarly, at least for older API elements. On the other hand, the APIs in clusters 2, 3, and 6 are much less wary of deprecating entities, similarly, to the npm and R/CRAN ecosystems.

Bogart *et al.* [107] also detail some strategies clients use to cope with change, including actively monitoring APIs for changes, doing so reactively, and limiting the number of dependencies to APIs. The latter can go to the extreme of keeping a local copy of the API to avoid migrating to a newer version in the case of R/CRAN. Another behavior is observed by Decan *et al.* [109] in the case of R/CRAN: the rigid policy of forcing all packages to work together can become burdensome. Indeed, since packages have to react to API changes, the coordination and reaction costs can be excessive. As a result, an increasing number of R packages are now primarily found on GitHub, not CRAN, meaning that the developers are not affected by R/CRAN's strict policies.

Decan *et al.* [108] also investigated the evolution of the package dependencies in the npm, CRAN, and RubyGems ecosystems. An interesting finding in the context of this chapter is the increasing tendency in the npm and (to a lesser extent) RubyGems packages to specify maximal version constraints. This means that some package maintainer specifies a maximal version number of the packages they depend on, to shield themselves from future updates that might force them to react to breaking changes. This strategy is complementary to the strategies documented by Bogart *et al.* mentioned above. They note that this behavior was not observed in R/CRAN, where a single version of each package—the latest—is stored at any given time, so specifying a specific version is of limited usefulness; package maintainers have to update anyways. In this study, we found that a large number of API clients did not update their API version (the exception being Spring), which seems to go along the lines of the behavior observed by Decan.

4.11.2 Mining of API Usage

Studies that present approaches to mining API usage from client code are related to our work, especially with respect to the data collection methodology.

One of the earliest works done in this field is the work of Xie and Pei [42], where they

developed a tool called MAPO (Mining API usage Pattern from Open source repositories). MAPO mines code search engines for API usage samples and presents the results to the developer for inspection.

Mileva *et al.* [40] worked in the field of API popularity; they looked at the dependencies of projects hosted on Apache and Sourceforge. Based on this information they ranked the usage of API elements such as methods and classes. This allowed them to predict the popularity trend of APIs and their elements.

Hou *et al.* [110, 111] used a popularity based approach to improve code completion. They developed a tool that gave code completion suggestions based on the frequency with which a certain class or method of an API was used in the APIs ecosystem.

Lämmel *et al.* [44] mine usages of popular Java APIs by crawling SourceForge to create a corpus of usage examples that form a basis for a study on API evolution. The API usages are mined using type resolved Java ASTs, and these usages are stored in a database.

4.11.3 Supporting API evolution

Beyond empirical studies on API evolution, researchers have proposed several approaches to support API evolution and reduce the efforts of client developers. Chow and Notkin [112] present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [113] propose CatchUp!, a tool using an IDE to capture and replay refactorings related to the API evolution. Dig *et al.* [114] propose a refactoring-aware version control system for the same purposes.

Dagenais and Robillard observe the framework's evolution to make API change recommendations [24], while Schäfer *et al.* observe the client's evolution [90]. Wu *et al.* present a hybrid approach [7] that includes textual similarity.

Nguyen *et al.* [115] propose a tool (LibSync) that uses graph-based techniques to help developers migrate from one framework version to another.

Finally, Holmes and Walker notify developers of external changes to focus their attention on these events [116].

4.12 Conclusion

We have presented an empirical study on the effect of deprecation of Java API artifacts on their clients. This work expands upon a similar study done on the Smalltalk ecosystem. The main differences between the two studies is in the type systems of the language targeted (static type vs dynamic type), the scale of the dataset (25,357 vs 2,600 clients) and the nature of the dataset (third- party APIs vs third-party and language APIs).

We found that few API clients update the API version that they use. In addition, the percentage of clients that are affected by deprecated entities is less than 20% for most APIs—except for Spring where the percentage was unusually low. In the case of the JDK API, we saw that only 4 clients were affected, and all of them were affected by deprecation because they introduced a call to the deprecated entity at the time it was already deprecated, thereby limiting the probability of a reaction from these clients.

Most clients that are affected do not typically react to the deprecated entity, but when a reaction does take place it is—surprisingly—preferred to react by deletion of the offending

invocation as opposed to replacing it with recommended functionality. When clients do not upgrade their API versions, they silently accumulate a potentially large amount of technical debt in the form of future API changes when they do finally upgrade; we suspect this can serve as an incentive not to upgrade at all.

The results of this study are in some respects similar to that of the Smalltalk study. This comes as a surprise to us as we expected that the reactions to deprecations by clients would be more prevalent, owing to the fact that Java is a statically typed language. On the other hand, we found that the number of replacements in Smalltalk was higher than in Java, despite Java APIs being better documented. In this study, we also studied how clients of a language API (JDK API) are affected by deprecation, and we see that in contrast to Smalltalk APIs, clients are rarely affected by deprecation. We also went further and looked at the impact of deprecation policies on the reactions of clients, and found that an API's policy on deprecation may have a major role to play in a client's decision to react. This leads us to question as future work what the reasons behind this are and what can be improved in Java to change this.

This study is the first to analyze the client reaction behavior to deprecated entities in a statically-typed and mainstream language like Java. The conclusions drawn in this study are based on a dataset derived from mining type-checked API usages from a large set of clients. From the data we gathered, we conclude that deprecation mechanisms as implemented in Java do not provide the right incentives for most developers to migrate away from the deprecated API elements, even with the downsides that using deprecated entities entail.

5

Scale of reaction to deprecation

The most radical possible solution for constructing software is not to construct it at all.

Frederick P. Brooks Jr., 1975

Application Programming Interfaces (APIs) are as close to a "silver bullet" as we have found in Software Engineering [1]; Brooks acknowledges as much while revisiting his seminal essay "The Mythical Man-Month" after two decades [2]. However, just as like other software systems, APIs have to evolve and this evolution can have a large impact on its consumers if not done carefully [3, 4].

To smoothen the evolution of their API, producers can rely on the mechanism of *deprecation*. Whenever an API element is found to be inadequate, this element can be marked as deprecated to signal the consumers that its use is discouraged. A message can be added to the deprecation, for example, to suggest a replacement, to encourage consumers to migrate their code to the new version of the API, and to provide a rationale for the change. Software development tools support the deprecation mechanism: Compilers emit warnings when deprecated code is used and IDEs (*e.g.*, Eclipse [11]) visualize the usages of deprecated methods struck through. The API evolution is completed when, after a suitable period of time, the deprecated API element is removed from the API. At this point, any consumer that still uses the deprecated element would be unable to compile their code against the latest version of the API without first removing the calls to this element.

Several studies have shown that both consumers and producers may not behave as expected when it comes to the deprecation mechanism. The reaction of the consumers may be overdue or not happen [3, 13, 14]; also, the API producer may not provide clear instructions for replacement or even fail to provide a rationale for the deprecation [15–17]. Producers may eschew from removing deprecated methods from the API to retain backward compatibility or, oppositely, remove API elements without first deprecating them [18]. They may do so between major versions, or, breaking semantic versioning practices, do it between minor versions of the APIs [19]. Certain deprecation policies adopted by producers might have an adverse impact on the consumers [14].

The current implementation of the deprecation mechanism in Java 8 has been changed for Java 9 [117]. The Java language designers who made the call to change the mechanism cite a lack of credibility surrounding the deprecation mechanism as the driver behind the change. According to the Java language designers, API consumers are unaware of whether a deprecated feature they use is going to be ever removed. All this has led API consumers to not taking deprecation seriously, thereby continuing with their use of the deprecated entities.

In this study, we seek to ascertain the scale of reactions or non-reactions to deprecated entities and the diversity of these reaction patterns. There is no current understanding as to how an API consumer can react to a deprecated feature or what the frequency of these reactions and the rationale behind them is. An in-depth understanding of whether consumers react to deprecation would allow us to understand whether consumers take deprecation seriously or whether they allow technical debt to accrue over time by not reacting. Concurrently, we would be able to assess if Java's deprecation mechanism is achieving its stated goal. Additionally, knowing the different kinds of reactions and their frequency allows API producers to understand whether their effort with evolving the API is worthwhile.

- We first conduct a qualitative study (presented in Section 5.3.1) to analyze the *diversity* [118] of how consumers react to API deprecation. We manually track a sample of 380 deprecated API elements in consumers' code across their lifetime and we observe the following patterns (beyond the expected pattern of replacing with the recommended replacement): non-reaction, deletion, replacement by another API, replacement with an in-house solution, and rollback to a previous API version.
- We then gather quantitative information (Section 5.3.2) about the *frequencies* of the reaction patterns we previously observed, by means of mining software repositories. Specifically, we quantify the reaction to API deprecation of 50 popular Java APIs, with a process that analyzed 297,254 Java projects on Github. The prevalent finding is that the most common reaction, which constitutes the 88% of the consumers' reactions, is to *not react*.
- We analyze if and how the reaction patterns vary depending on the considered API (Section 5.3.3). This also allows us to analyze if certain deprecation strategies are associated with specific reaction patterns (Section 5.3.4). We find that 20 of the APIs affect no consumers with deprecation, a further 18 APIs deprecate elements that they know have limited impacts on the consumers, and APIs that release rarely have fewer reactions than ones that release often.
- Since most consumers do not react to deprecation, we report on a survey of the reasons for non-reaction to deprecation (Section 5.3.4). We analyze 79 responses, and find that the top three reasons reported by respondents are: (i) the lack of a suitable alternative, (ii) the too high cost of reacting, and (iii) no perceived incentive to react since the API does not release frequently.

We conclude discussing the implications of our findings (Section 5.4), in particular, that deprecation seems to be viewed not seriously by consumers, who rarely react to it. This is in line with the view of the Java language designers.

5.1 Background: Deprecation in Java

Deprecation as a language feature exists to give API producers a way in which they can indicate that a feature should no longer be used. According to the official Java documentation: "A deprecated class or method is ... no longer important. It is so unimportant, in fact, that you should no longer use it, since it has been superseded and may cease to exist in the future" [10].

The principal idea of having a deprecation mechanism is to allow API consumers to take their time in adapting to API changes [10]. As the API evolves, some features might be replaced by newer features that are better, faster or more secure. However, simply removing the obsolete functionality would break API consumer code and allow them no transition period. During this transition period, the consumer is given ample indication in the IDE that the feature being used is deprecated, as seen on lines 3 and 4 of Example 5.2.

Example 5.1: Example of deprecated usage and reaction to it

```
1 public class DateCalculator {
2     public static void main(String[] args) {
3 -       Date date = new Date();
4 -       int day = date.getDay();
5 +       Calendar calendar = Calendar.getInstance();
6 +       int day = calendar.get(Calendar.DAY_OF_WEEK);
7         System.out.println("Today is the " + day + "th of the
            week.");
8     }
9 }
```

getDay

```
@Deprecated
public int getDay()
```

Deprecated. *As of JDK version 1.1, replaced by* `Calendar.get(Calendar.DAY_OF_WEEK)`.

Returns the day of the week represented by this date. The returned value (`0` = Sunday, `1` = Monday, `2` = Tuesday, `3` = Wednesday, `4` = Thursday, `5` = Friday, `6` = Saturday) represents the day of the week that contains or begins with the instant in time represented by this `Date` object, as interpreted in the local time zone.

Returns:

the day of the week represented by this date.

See Also:

`Calendar`

Figure 5.1: API documentation for deprecated entity

Java first introduced deprecation in Java 1.1 as a `@deprecated` Javadoc annotation. This allowed API producers to indicate in the documentation that a feature is deprecated, give a reason behind the deprecation, and possibly indicate an alternative feature to use as seen

in Figure 5.1. Additionally, some APIs even provided a recommended course of action to deal with the deprecated feature. Subsequently, with the release of Java 5, annotations were added to the Java language, including a source code annotation `@Deprecated`. When a feature is marked with this annotation, the Java compiler throws a warning.

The Java documentation states that deprecation allows API producers to keep obsolete functionality around for a certain period of time to preserve "backward compatibility" [10]. Once this period is passed, the obsolete feature can be removed as it would be likely that API consumers already transitioned away from using this obsolete feature. Hence, it is recommended that API consumers react to a deprecated API feature, unless they want to encounter a breaking change later in the evolution of the API. The consumer can react in a number of ways, one such example can be seen on lines 5 and 6 of Example 5.2.

However, according to the Java JDK developers, API consumers do not appear to be taking deprecation seriously [117]. By not removing deprecated features from an API after a transition period has passed, API producers and the Java JDK developers themselves have cheapened the meaning of deprecation. This behavior has prompted few consumers to react to a deprecated feature. Java would like to change this in the upcoming release of Java 9 by enhancing the deprecation mechanism with information about future removal of a deprecated feature.

5.2 Methodology

In this section, we present the research questions and the research method.

5.2.1 Research Questions

The overall goal of this work is to understand the nature of reaction to a deprecated API artifact. This involves understanding how developers react to deprecation, observe the most popular way to react to deprecation, and how API policies are associated with reaction patterns. We structure our work along the following research questions:

RQ_1: **How do API consumers react to deprecation?** We only know that replacing the deprecated feature with its recommended replacement is something that the Java documentation on deprecation recommends. However, there is currently no empirical knowledge on the *diversity* [118] of how consumers react to API deprecation. To that end, we question as to what the possible reaction patterns are.

RQ_2: **How do API consumers deal with deprecation?** Based on the observed reaction patterns, we seek to uncover their frequency in an open source setting. This helps us understand as to how on a large scale API consumers prefer to react to deprecated features. To gain this understanding, we ask two sub-questions. The first attempts to establish the overall upgrade behavior of the consumers with respect to their dependencies and the second benchmarks the frequency of each reaction pattern.

- RQ_{2a}: **How often do consumers upgrade their dependencies?**
- RQ_{2a}: **How often does each reaction pattern occur?**

RQ_3: **How do reaction patterns vary across APIs?** Once we know the frequency of the observed reaction patterns, we seek to uncover if there is any dominant pattern for the consumers of any of the analyzed APIs. If the majority of the consumers react to deprecation in just one way for an API, we may hypothesize that the behavior of the API producer may influence this. Furthermore, an insight into the distribution of the reaction pattern for an API can help this API's producer understand how its consumers react to deprecations.

RQ_4: **What are the reasons behind API consumers not reacting to deprecation?** Finally, the results to our previous research questions showed that not reacting is the most common reaction pattern across all consumers of all APIs. With this research question, we would like to investigate this lack of reactions to deprecation. By not reacting, consumers are theoretically allowing technical debt to accrue over time; we would like to uncover the reasons behind this.

5.2.2 Subject selection

To understand how API consumers react to the deprecation of features in APIs, we select a set of 50 popular Java APIs and their consumers to study. The popularity of an API is defined by the number of Maven-based Java projects on GitHub that use that API. We restrict ourselves to the Maven ecosystem among Java projects on GitHub because: (1) projects that use Maven can be considered to adhere to the most basic of software engineering principles and (2) Maven-based projects explicitly declare their dependencies in a project object model (POM) file that allows us to establish the API being used and the exact version in use.

We download the POM files of all Maven-based Java projects on GitHub. To ensure that all POM files are unique, we do not include forks of projects in our dataset, relying on the aid of GHTorrent [119]. This results in a total of 135,729 POM files. Subsequently, we parse each one of these POM files to determine the list of APIs being used in the project. With this data, we classify the most popular Java-based APIs among GitHub clients; for example, JUnit is the most popular API, with 67,954 client Java projects.

We select the top 50 APIs—from different vendors—ranked by popularity in GitHub. Concerning the vendors, we see, for example, that the APIs `spring-core`, `spring-context`, and `spring-test` are all in the top 10 in terms of popularity and that they are all released by the same vendor (*i.e.*, org.springframework). By analyzing clients that use APIs from the same vendor, it is harder to isolate factors stemming from API policies on deprecation when it comes to reaction patterns to deprecation. Hence, we consider at most one API from each vendor.[1]

This selection process results in 50 APIs (a complete list can be found in Appendix B) where the most popular API (JUnit) is used by 67,954 Java projects and the least popular API (jetty-server) is used by 1,362 projects. By targeting these 50 APIs, the total number of API consumers that we analyze in this is 297,254.

[1]We have more API producers from Apache because Apache is an ecosystem and not a vendor in the traditional sense.

5.2.3 API usage data collection

To understand what features of an API consumers use, one can select from different proposed approaches that collect API usage data, *e.g.*, MAPO by Xie and Pei [42] and SOURCERER by Bajracharya*et al.* [62]. We lean on the technique FINE-GRAPE developed by Sawant and Bacchelli [120]. This technique gives us three advantages: (1) it uses Maven-based Java projects, (2) it results in a type-checked API usage dataset, (3) it determines the API usages over the entire history of a given project.

fine-GRAPE only focuses on projects that are under active development [2] *i.e.*, those that have been actively committed to in the last 6 months. We download all 297,254 active projects for the 50 APIs under study and then run the FINE-GRAPE analyzer on the source code of each project. This results in a dataset which contains 1,322,612,567 type-checked API usages across the entire history of the selected API consumers. The usage data we have collected spans from 1997 to 2017. The overall size of the dataset on disk is 604GB and it can be found at `http://doi.org/10.4121/`.

5.2.4 Determining the reaction patterns

There is no empirical knowledge on how API consumers react to deprecated features in an API they use. For example, a consumer might react to deprecation by replacing the deprecated feature with the recommended replacement or by rolling back the version of the API being used so that the feature is no longer deprecated.

Our aim is to create a taxonomy of possible reactions to deprecation. For this purpose, we perform a manual analysis of how an API consumer behaves when a deprecated usage is encountered. We select a sample of 380 usages of deprecated features and manually analyze these in depth. A sample size of 380 ensures a 95% confidence interval and 5% margin of error.

For each usage of a deprecated feature from our sample, we isolate the commit in which the method was originally marked as deprecated, and the consumers' file that uses it. We then look at all commits to the file from the point to see what happens to that usage. To see what changes in the entire project, we isolate the git diffs for each commit.

We analyze each usage and how it evolves over time. We try to decipher the reason behind the introduction of the deprecated usage and the nature and purpose of the API feature being used. Then we look at the documentation of the API to understand the API producers' recommendation (if any) as a reaction to the deprecated feature. We look at the entire history of the file to see what happens to that deprecated usage. If there is a change to it, we note down the nature of the change (a reaction pattern), if there is no change till the end of history we mark it as a non-reaction. The result of this analysis is an empirical understanding of what the API consumer does and the reasons behind the change, which we distill into a taxonomy of reactions to deprecated methods in APIs.

5.2.5 Quantifying the reaction patterns

Once we have an understanding of the various types of reaction patterns that API consumers can adopt, we seek to quantify these patterns. For this, we look at the clients of all

[2] As indicated by the GHTorrent dataset [119]

50 APIs and see how those that are affected by deprecation react to deprecation by looking for the reaction patterns found during the manual analysis.

For each API client, we have the method invocations for each file and information on how these invocations evolve over time in each file. This allows us to automatically infer what happens to a deprecated invocation over time. In Section 5.3.2, we detail the method we apply to automatically recognize and count each reaction pattern.

5.2.6 Associating API evolution to reactions

We want to see whether and how the evolution policies of APIs are associated with the way in which clients react to deprecation. An API might have a policy to deprecate very few features, thus impacting very few clients; or an API might remove deprecated features very often and this may persuade clients to react to a deprecation just to keep up with the APIs evolution.

We use four dimensions to benchmark the APIs along:

1. Actively releasing: This determines if the API has released a version of the API in the recent past and if it has a history of releasing frequently. APIs that change regularly are more likely to affect a consumer with deprecation as opposed to those that rarely or never release a new version due to the high volatility of features.

2. Deprecated feature removal: This benchmarks if the API has a tendency to remove a deprecated feature or not. APIs that frequently remove deprecated features are more likely to force consumers to react to deprecation due to the risk of a new version introducing breaking changes.

3. Percentage deprecated: This indicates the percentage of the API that has been deprecated on average over each version of the API. We take the average as opposed to the median as we believe that it provides a balanced figure over the entire lifetime of the API. Furthermore, we expect the number of deprecated features in the major version to remain constant in the minor versions of the API [19]. When a larger proportion of an API is deprecated there is a higher chance that consumers are affected by deprecation as opposed to APIs that deprecate few to none of the features.

4. Breaking changes: This indicates the number of breaking changes the API introduces without first deprecating the feature being removed. If an API has a propensity to introduce breaking changes as opposed to first deprecating a feature and then removing it, fewer consumers are likely to be affected by deprecation as the API does not follow the deprecation protocol.

For each of the dimensions, we define thresholds such that each API can be placed in one bin among the thresholds. Then, we hold a card sort session [121] where we cluster APIs with similar evolution traits. The first two authors of the article perform the card sort.

5.2.7 Understanding developer perceptions regarding deprecation

Our goal is to gain an understanding as to why we observed certain reaction phenomena. To address this goal we designed a survey made up of 6 questions to send to developers.

The questions asked in the survey are based on the observations made during the empirical investigation. We ask developers to rate the frequency (on a five-point Likert scale) with which they have reacted to deprecation in one of the ways identified, and to explain the rationale behind adopting this reaction behavior.

We aimed to reach as many Java developers who work on both industrial projects and open source projects. To achieve this goal, we spread the survey on Java developer forums (*e.g.*, Java code ranch), Reddit communities and Twitter. The survey was in the field for a period of 6 months. We obtained 79 responses and a further 88 developers started the survey but did not see through to completion.

28% of developers in our survey work on open source projects, the rest work on industrial/proprietary projects. Our respondents are primarily developers, with 4 respondents who also work on research. All our respondents are experienced, with the average number of years of experience being 12. The origin of our respondents is not limited to one geographical location, we have responses from Europe, North America, South America, Australia, and Asia.

5

5.3 Results

5.3.1 RQ1: Reaction patterns to deprecation

We manually analyze 380 usages of deprecated API artifacts across consumers of 50 APIs. Based on this analysis, we observed seven reaction patterns (RPs) to deprecation. We describe these patterns, following the same order in which we discovered them in the manual analysis:

1. RP1: **No reaction** – API consumers do not do anything with the reference to the deprecated feature in their code base. The reference remains in the source code till the latest available version of the consumer code.

2. RP2: **Delete invocation** – API consumers react by removing the invocation to the deprecated feature, without replacing it with the replacement recommended by the API producers or any other functionality.

Example 5.2: Example of deprecated usage and reaction to it

```
1 public class DateCalculator {
2     public static void main(String[] args) {
3 -       Date date = new Date();
4 -       int day = date.getDay();
5 +       Calendar calendar = Calendar.getInstance();
6 +       int day = calendar.get(Calendar.DAY_OF_WEEK);
7         System.out.println("Today is the " + day + "th of
            the week.");
8     }
9 }
```

Example 5.3: Deletion of deprecated usage

```
- if (JdkVersion.isAtLeastJava15()) {
-         editor =
  descriptor.createPropertyEditor(this.targetObject);
- } else {
-         Class editorClass =
  descriptor.getPropertyEditorClass();
-         if (editorClass != null) {
-             editor = (PropertyEditor)
  BeanUtils.instantiateClass(editorClass);
-         }
- }
```


3. RP3: **Replace with recommended replacement** – API consumers replace the deprecated API element with the alternative proposed by the API producers.

Example 5.4: Replace with recommended replacement

```
ImmutableList$<$String$>$ list = ImmutableList.of("hello");
- list.add("world");
+ ImmutableList<String> list2 = new
  ImmutableList.Builder<String>()
+                         .addAll(list)
+                         .add("world")
+                         .build();
+ list = list2;
```

4. RP4: **Replace with in-house replacement** – API consumers remove the invocation made to a deprecated feature and replace it with a functionality that they themselves create.

Example 5.5: Replace with in-house replacement

```
- ImmutableList<String>list = ImmutableList.of("hello");
- list.add("world");
+ MyImmutableList<String> list =
  MyImmutableList.of("hello");
+ list.add("world");
```

5. RP5: **Replace with Java replacement** – API consumers replace the deprecated invocation with an equivalent functionality provided by the Java Development Kit (JDK).

Example 5.6: Replace with Java replacement

```
- ImmutableList$<$String$>$list =
  ImmutableList.of("hello");
- list.add("world");
+ List<String> list = new ArrayList<String>();
+ list.add("hello");
+ Collections.unmodifiableList(list);
+ List<String> list2 = new ArrayList<String>();
+ list2.addAll(list);
+ list2.add("world");
+ Collections.unmodifiableList(list2);
```


6. RP6: **Replace with other third-party API** – API consumers choose to switch API and replace the deprecated invocation with a non-deprecated one from the API to which they switch.

Example 5.7: Replace with other third-party API

```
- ImmutableList<String>list = ImmutableList.of("hello");
- list.add("world");
+ List<String> stringList = new ArrayList<String>();
+ stringList.add("hello");
+ UnmodifiableList<String> list = new
  UnmodifiableList<String>(stringList);
+ list.add("world");
```

7. RP7: **Rollback version of the API** – API consumers rollback the version being used such that the used feature is no longer marked as deprecated.

Example 5.8: Rollback version of the API

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
-   <version>14.0</version>
+   <version>13.0</version>
</dependency>
```

> ***RQ*$_1$.** The manual inspection of 380 usages of deprecated API artifacts across consumers of 50 APIs lead to the discovery of six reaction patterns, in addition to 'Replace with recommended replacement'.

5.3.2 RQ2: Dealing with the deprecation of a feature

After having identified the possible reaction patterns (RPs), we investigate their occurrence among all the clients.

RQ2a: Version upgrade behavior

We begin our investigation by looking into how many of the API consumers in our dataset have changed the version of the library that they use.

We compute the percentage of consumers that upgrade the version of the dependency in our dataset. Overall in our dataset we see that not many consumers upgrade the version of the API. None of the APIs has more than 13% of its consumers upgrade their dependency version. For APIs such as 'standard' and 'dom4j' the percentage of consumers upgrading version is less than 5%. For the widely popular APIs such as 'slf4j-api' and 'junit', only 12% or less of the consumers upgrade.

To triangulate this unexpected finding with another source, we asked to our survey respondents (see Section 5.2.7) whether they upgrade the version of the API that they use. Among the respondents, 31% of the API consumers indicated that they always upgrade the version of the API being used, while a majority of 69% indicate that they only do this occasionally or never. We asked this 69% to rank (on a five point likert scale) the frequency with which one or more motivations behind not upgrading the version of the API has applied to them. These motivations are a result from previous work, literature on deprecation and documentation on deprecation. The results of this can be seen in Figure 5.2.

The upgrade cost, in terms of time or money, is the most common reason (42% of consumer rating it as 'almost always' to 'always') for not having upgraded. A 41% of the consumers reported not having upgraded ('almost always' to 'always') when everything in the current version of the API worked just fine. This is in line with what Sawant*et al.* [18] found. Breaking changes in the new version only stopped 32% of consumers from upgrading; in fact, 48% of the API consumers are neutral about this. Conversely, deprecation is seen as an even smaller barrier to upgrading, with only 9.7% consumers indicating that it has stopped them from upgrading the version of the API. A 22% of the responding consumers indicate that they have a policy to freeze the version of the API that they use (52% of consumers actually indicate that they have no such policy).

API consumers also provided us with additional reasons to not upgrade. One consumer indicated that management in the company that he worked in did not allow for dependency upgrades. Additionally, when a project reached a stable point it was no longer needed to upgrade the dependency.

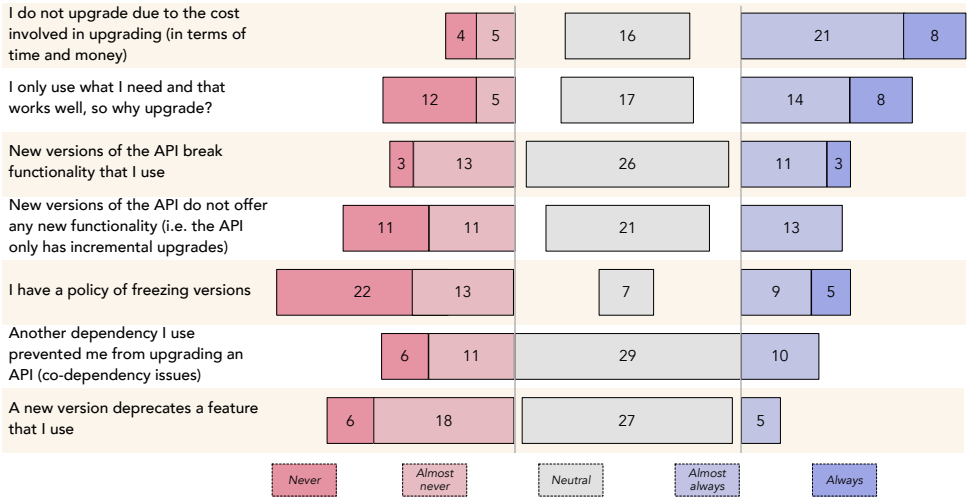


Figure 5.2: Reasons behind not upgrading a dependency

RQ2b: Frequency of reaction patterns

After having investigated the overall upgrading behavior of API consumers, we look into how API consumers react to deprecation by analyzing how frequently they adopt one or more of the reaction patterns that we found in RQ_1.

For each RP, we present: (1) the results gathered during the manual analysis conducted to answer RQ1 (in terms of both RP occurrence and the qualitative description of clients' behavior), (2) the dedicated heuristic we devised to automatically detect whether the RP takes place,[3] and (3) the number of overall occurrences and unique consumers of the RP across the 297,254 API consumers pertaining to the 50 considered Java APIs.

We test the validity and accuracy of our heuristics by running them on the source code files of the 380 samples that were manually analyzed in Section 5.3.1. Our heuristics are able to identify the correct reaction pattern in 100% of the cases. Moreover, this analysis also confirmed the exhaustiveness of the patterns created in RQ_1: None of the analyzed cases let emerge new patterns.

We then analyze the fallibility of our heuristics to see whether they incorrectly classify a pattern (*i.e.*, establish the false positive rate). We manually analyze 100 cases of the automated classification for each of RP1, RP2, RP4, RP6 and RP7. For RP3 and RP5, we analyze all the cases since there is a limited number. For RP1–3, RP5, and RP7 we do not see any false positives. In the case of RP4 we see 7 instances where the replacement of deprecated feature with an in-house replacement did not make sense as the functionality being replaced was not the same. In the case of RP6 we observed 18 instances where the third-party API replacement does something completely different to the original API. However, looking deeper at these 18 cases we found that in 4 of these cases the developers

[3]To automatically infer if a reaction pattern takes place, we start by going through the history of every file that uses a deprecated feature, every time we see that the number of deprecated features being used is decreasing, we attempt to see why the number of deprecated features being used has gone down, using the specific heuristic.

Table 5.1: Breakdown of number of reactions per reaction pattern

Reaction pattern	number of overall occurrences	number of unique consumers
No reaction	146,076	8,910
Delete invocation	1,015	218
Replace with recommended replacement	36	7
Replace with in-house replacement	702	31
Replace with Java replacement	17	3
Replace with other third-party API	15,236	641
Rollback version of the API	2,134	193

had made a conscious choice to change functionality hence a one to one mapping was not needed. For the other 14 cases we could not precisely establish a rationale.

Overall, over 297,254 API consumers, we see a total of 9,317 projects that are affected by deprecation and react in one of the ways we have found. Over these 9,317 projects, we see 165,216 usages of deprecated methods to which reactions take place. The occurrence of each reaction pattern is summarized in Figure 5.3 and Table 5.1.

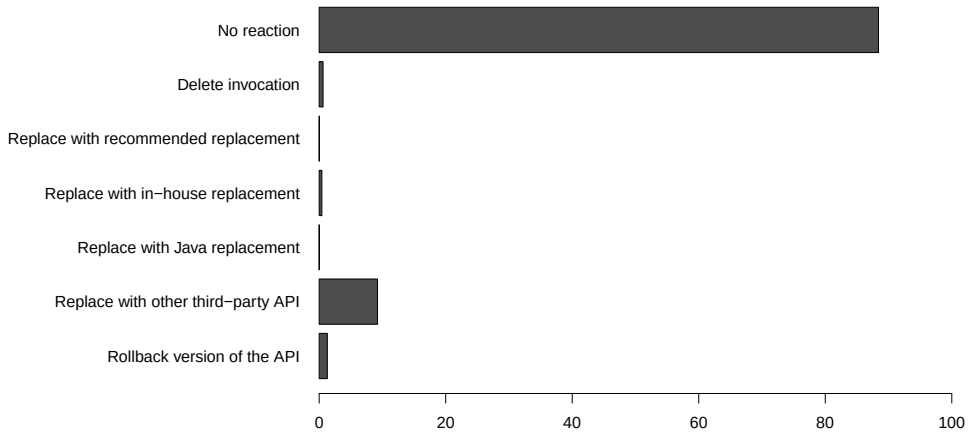


Figure 5.3: Percentage distribution of reaction patterns

1. RP1: **No reaction**

   Qualitative analysis:

   Total occurrences: 290 (76%) *Unique consumers:* 221 (88%)

   In the manually analyzed sample set, not reacting to a deprecated functionality is the most popular reaction pattern. On the one hand, this behavior may be explained by the fact that the cause of deprecation was not severe (we inspected the cause and

there were no security or performance issues); on the other hand, this behavior is also unexpected, since all the deprecated entities that were not reacted to had recommended replacements and were well documented.

Detection heuristic: We look at the first version of a file that contains a deprecated feature and then the last version of the file, we see if the references to deprecated features have been removed in the last version. In the event that these have not been removed, we mark it as a 'non reaction'.

Quantitative analysis:

Total occurrences: 146,076 (88%) *Unique consumers:* 8,910 (95%)

Also in the large scale analysis, we found that the most frequent pattern is to not react to deprecation (88% of the time, for the majority of the projects–8,910 out of 9,317). This is despite the fact that the APIs that we consider are all popular and well documented mainstream Java APIs. Looking deeper at these non-reactions we notice that in 55% of the cases the files containing invocations to deprecated features not being reacted to do not change. This might be due to the fact that the file is already stable and requires no more changes. We cannot ascertain whether this code is being executed currently, however, given the active nature of the projects selected we do expect that the code is still being used in some manner. Due to this behavior, consumers simply might not notice that they are using a deprecated feature.

2. RP2: **Delete invocation**

Qualitative analysis:

Total occurrences: 23 (6%) *Unique consumers:* 4 (1.5%)

Deleting a deprecated invocation occurs frequently among the API consumers, in the manually analyzed dataset. We investigated the deprecated methods that have been removed: In two cases the deprecated feature was supposed to be removed, since its usage was no longer needed; in the rest, the feature has a recommended replacement, however, the consumers delete the reference. In none of the cases do developers give any rationale behind the deletion.

Detection heuristic: When going through the different versions of a file containing a reference to a deprecated feature, if the number of references to deprecated features reduces and no new invocation is added in the same location in its place either from the same API or any third-party API, we mark it as a 'deleted invocation' with no replacement.

Quantitative analysis:

Total occurrences: 1,015 (0.6%) *Unique consumers:* 218 (2%)

Deleting and not replacing the invocation is also seen in over 1,000 cases in the large scale analysis. Some of these deletions with no cause might stem from the fact that the API required the deprecated method to be handled in that manner (as we have

seen in the qualitative analysis), however, it is reasonable to expect that this might not always be true (as we have also seen in the qualitative analysis).

3. RP3: **Replace with recommended replacement**

   Qualitative analysis:

   Total occurrences: 12 (3%) *Unique consumers:* 2 (0.8%)

   Replacing a deprecated invocation with its recommended replacement is unpopular amongst API consumers. In our manually analyzed sample set, all the deprecated methods are documented and provide clear instructions as to how the deprecation should be handled. This makes it all the more surprising that we see very few reactions of this nature. Looking at the commit message for those API consumers that have actually replaced the deprecated invocation with the recommended replacement, they simply state that some changes were made due to the upgrade in the API version being used. In fact, in many cases, the reaction to deprecation was performed at the same time as the upgrade to the version of the library. Some API consumers react to deprecation immediately after noticing the deprecation.

   Detection heuristic: For the API in question, we create a set of package names in the API over the entire history of the API, to verify whether an invocation added to a file belongs to one of these packages. In a version of a file in which a deprecated feature is removed, we check whether a new invocation is added to the same API in its place. If there is a new invocation made to the same API, we mark it as a 'replacement'.

   Quantitative analysis:

   Total occurrences: 36 (0.02%) *Unique consumers:* 7 (0.07%)

   Replacing a feature with its recommended replacement is the second least frequent way in which API consumers reacts. In practice, only 7 API consumers choose to react in this manner. These 7 API consumers have replaced the deprecated feature in 36 (0.02%) cases and they do not react in any other manner to deprecation; furthermore, they replace all invocations being made to deprecated features. Thus, similarly to the qualitative analysis, we observe that consumers that do react to deprecation "as intended" tend to be systematic about it.

4. RP4: **Replace with in-house replacement**

   Qualitative analysis:

   Total occurrences: 2 (0.5%) *Unique consumers:* 1 (0.4%)

   In two cases, the deprecated invocation is replaced by some functionality developed by the API consumer itself. Both cases belong to the Hibernate API. The API consumers have in each case replaced a database mapping invocation with their own wrapper around the database. The reason we found evidence for is that the Hibernate API was changing too much and it was not deemed worth keeping up with the changing API.

Detection heuristic: We start by looking at all the files and packages in a given project and create a set of package names that the project itself has defined, this is the list of packages from which a feature can be imported. When in a version of a file we see that a deprecated feature is removed, we look to see if it has been replaced by a feature pertaining to one of these in-house packages.

Quantitative analysis:

Total occurrences: 702 (0.4%) *Unique consumers:* 31 (0.3%)

Replacing with in-house functionality is done in 702 cases. Thus, in a non-negligible number of cases, API consumers have taken the effort to implement functionality that has been provided, yet deprecated by an external API.

5. RP5: **Replace with Java replacement**

Qualitative analysis:

Total occurrences: 6 (1.5%) *Unique consumers:* 1 (0.4%)

In our sample set, many of the APIs extend the functionalities of the Java API or provide alternatives to existing Java libraries. These are seen in the case of the Guava API's consumers. The consumer replaced references to the Guava Map class with those to the Java Map class. There was no reasoning given by the API consumer in the commit or in any pull request as to why the change was made. We speculate that using functionality from the Java API was deemed to be easier and safer than using deprecated features from the Guava API.

Detection heuristic: Java libraries start with one of three prefixes: `java`, `sun` and `javax`. If we see a deprecated feature is removed and—in its place—we see a new invocation being made to a method which belongs to a class whose package name starts with one of the prefixes, we infer that this reference to a deprecated feature has been replaced by a feature from Java and mark it as such.

Quantitative analysis:

Total occurrences: 17 (0.01%) *Unique consumers:* 3 (0.03%)

The least frequent way (17 cases or 0.01% of the time) for an API consumer to react is to replace the deprecated functionality with an invocation to a Java API feature. This could be explained by the fact that all third-party APIs seek to offer functionalities beyond the Java API building on top of it.

6. RP6: **Replace with other third-party API**

Qualitative analysis:

Total occurrences: 28 (7%) *Unique consumers:* 13 (5%)

More than one API may provide the same functionality. For example, Easymock and Mockito are both libraries that allow developers to mock objects in test cases. We see in 5 cases, all of which pertain to consumers of commons-collections API, deprecated features are replaced by functionality from the Guava API. The primary

reason is that the commons-collections API had become obsolete, while Guava is more modern and provides more updated functionality.

Detection heuristic: There is no way to determine a list of package names for third-party replacement APIs as it is hard to understand which APIs can replace a certain API, thus making a complete list of replacement packages a non-trivial endeavor. Instead, we rely on data we have collected for all the aforementioned replacement based reaction patterns as for each pattern we create lists of packages to which a replacement method could belong to. When a replacement method does not belong to a class from any of those packages, we infer that it belongs to another third-party API.

Quantitative analysis:

Total occurrences: 15,236 (9%) *Unique consumers:* 641 (7%)

The second most frequent way (7%) in which API consumers (641 consumers out of 9,317) react to deprecation is by replacing a reference to a deprecated feature with a third-party API feature. This behavior might be prevalent for the consumers of some APIs as opposed to others as not all APIs have competitors.

7. RP7: **Rollback version of the API**

   Qualitative analysis:

   Total occurrences: 19 (5%) *Unique consumers:* 8 (3%)

   Deprecations in an API are visible to API consumers when they upgrade the version of the API being used. At this point, a consumer can choose to stick with this version or rollback to the previous version. In one case (consumer of JUnit) in our sample set, we see this to be the case. The reasoning was that the new version had deprecated certain features being used. Needless to say, this defeats the purpose of deprecation.

   Detection heuristic: When going through various versions of a file containing a reference to a deprecated feature ordered chronologically, if the file no longer contains deprecated features, we check whether the API consumer has rolled back the version of the API. We also ensure that the method invocation has not been removed from the source code. If both conditions are met, we mark it as a rollback.

   Quantitative analysis:

   Total occurrences: 2,134 (1%) *Unique consumers:* 193 (2%)

   Rolling back the version of the API being used is also seen 1.2% of the time. This may indicate that several API consumers do not wish to take the effort to adapt to a new version of an API, but prefer to stick with an older version.

To challenge our aforementioned findings concerning frequency of reaction patterns, we ask in our survey (which targets API consumers from a different setting) which one of these reaction patterns they have most frequently adopted. Figure 5.4 reports the results.

69% of API consumers in our survey reported to always react by replacing the deprecated invocation with the recommended replacement from the API. This is in direct

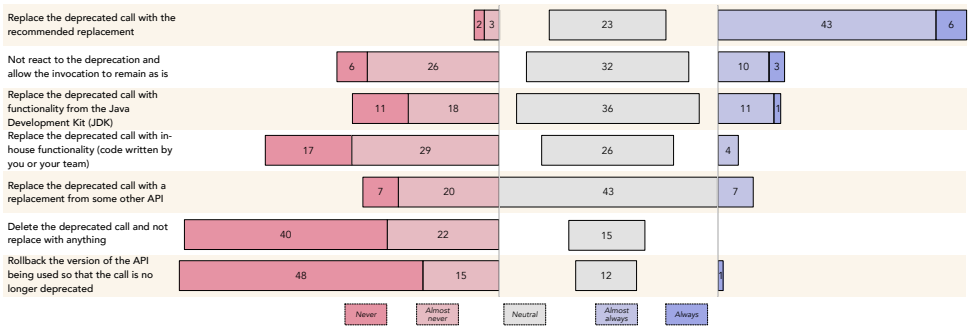


Figure 5.4: API consumers' preferred way to react to deprecation

contradiction of the trends seen in GitHub data. While only 20% of consumers indicate that they do no react to a deprecated entity, in fact, 44% actually indicate that not reacting to deprecation is something that they would not consider as acceptable behavior. This again contradicts the behavior observed on GitHub. However, this could be explained by our survey respondents answer our questions to conform with what they perceive as acceptable behavior (known as social desirability bias [122]).

The other 5 reaction patterns all receive less than 17% of support from consumers. In all the cases, the majority of consumers indicate that they do not react in such a manner. We see that rolling back the version of the API and deleting a deprecated call with no replacement are by far the most negatively viewed reaction patterns with 80% and 75% of consumers indicating that they would never adopt such a pattern.

> ***RQ*$_2$.** A small minority (less than 13%) of API consumers update their API version. Furthermore, the most common (88%) reaction to deprecation in our sample from GitHub is 'No reaction'; this result is not confirmed by the answers collected in our survey, whose respondent say to often (69%) 'Replace with recommended replacement'.

5.3.3 RQ3: Variance of reaction patterns across APIs

By answering RQ2, we have analyzed the distribution of reaction patterns across all the consumer projects considered in our dataset, regardless of the API producer. Now we move our attention to investigating whether the distribution of reaction patterns varies across the consumers of different APIs. We differentiate between those APIs that affect several consumers vs. those that do not.

In Figure 5.5 we see the percentage of consumers affected by deprecation. Based on the logical groupings that we observe, we define the following thresholds:

Unaffecting consumers: 0% of consumers are affected by deprecations done by the API

Minimally affecting consumers: < 2% of consumers are affected by deprecations

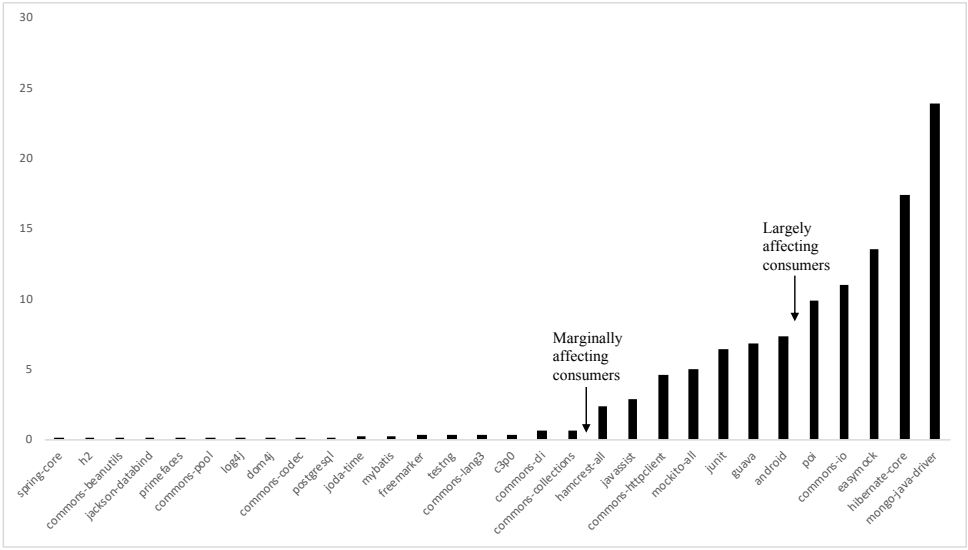


Figure 5.5: Percentage of consumers affected per API

Marginally affecting consumers: between 2% and 7% of consumers are affected by deprecations

Largely affecting consumers: > 7% of consumers are affected by deprecated features

Figure 5.6 shows a breakdown of the percentage of API producers belonging to each of these categories. Predominantly APIs do not affect their consumers, with 20 (40%) APIs never affecting any consumer and 18 (36%) APIs affecting less than 2%. Six (12%) APIs affect between 2% and 7% consumers and a further six (12%) affect more than 7% of the consumers.

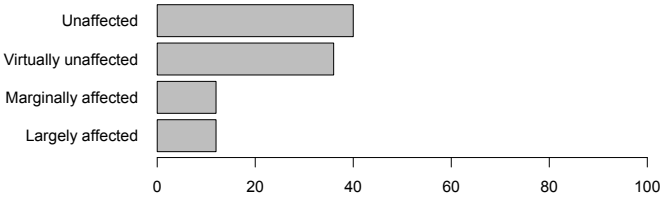


Figure 5.6: Distribution of the sampled APIs based on degree to which their consumers are affected by deprecation

Unaffecting consumers: For 20 APIs (40%), we observe that no consumer is affected by

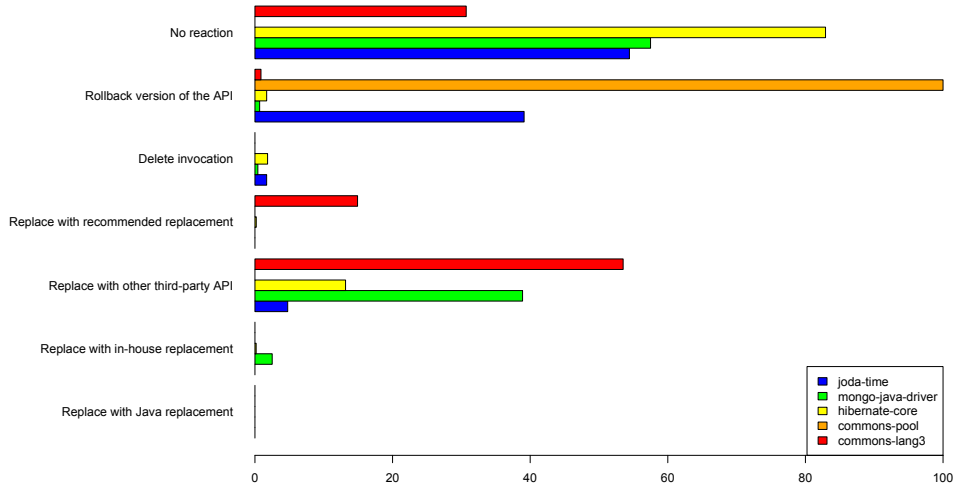


Figure 5.7: Frequency of reaction patterns for the 'joda-time', 'mongo-java-driver', 'hibernate-core', 'commons-pool' and 'commons-lang3' APIs.

deprecation, because our sampled API consumers do not use any of the deprecated features. For the other 30 APIs, consumers are affected to various degrees.

Minimally affecting consumers: Considering the distribution of reaction patterns across the 18 APIs that minimally affect their consumers (< 2%), consumers of 12 of these APIs predominantly do not react to deprecation. For other 3 APIs, replacing references to deprecated entities with references to a third-party API (RP6) is the dominant reaction pattern. In the case of 'commons-lang3' (seen in Figure 5.7), some consumers react by replacing deprecated features with features from the same API (RP3). Despite this fact, replacing with a third-party API is still the most common reaction pattern amongst consumers of this API. For the 'joda-time' API we see four different kinds of reaction patterns: no reaction (RP1), deletion (RP2), rollback and replacing with third-party API (RP6), as seen in Figure 5.7. The consumers of the last API from this set, *i.e.*, 'commons-pool', always rollback the version of the API being used (RP7), as visible in Figure 5.7. This seem to indicate that for the consumers of this API, deprecation acts as a deterrent to upgrading the version of the API being used.

Marginally affecting consumers: Among the APIs that marginally affect their consumers, the predominantly popular way to react to deprecation is by not reacting at all: For all the APIs, in over 60% of the cases consumers do not react to deprecation (RP1). There is a little bit of variance in terms of reaction patterns in the cases where a reaction does actually take place. In the case of 'guava', 'hamcrest-all', and 'junit', in 15-20% of the cases consumers have reacted by replacing a deprecated feature

with a third-party API (RP6). In the case of the other 3 APIs that fall in category ('javassist', 'mockito', and 'commons-httpclient'), in over 25% of the cases features are replaced with third-party API ones (RP7).

Largely affecting consumers: Among those APIs that affect consumers the most, 'hibernate-core' has affected 17% (1,391 out of 7,983) of its consumers and 'mongo-java-driver' affects 24% (496 out of 2,077) of its consumers; deprecations in 'hibernate-core' and 'mongo-java-driver' are more exposed to the consumers than the deprecations in most other APIs. In the final set we see that for 'android', 'poi', 'commons-io', and 'easymock', consumers predominantly (~80% of the cases) do not react (RP1). For these APIs, the alternative reaction patterns is typically to react by replacing the deprecated features with a feature from a third-party API (RP7). Out of this set of APIs, only the consumers of 'hibernate-core' (in Figure 5.7) show cases of replacing a deprecated feature with its recommended replacement (RP3). For this API the consumers appear to display the most varied reaction patterns. This may be explained by the large number of consumers (over 1,000) affected by deprecation. For 'mongo-java-driver' (the API that affects the largest percentage of consumers seen in Figure 5.7), in over 40% of the cases the consumers prefer to start using a new API (RP6).

***RQ*$_3$.** Over 75% of the APIs marginally or don't affect consumers with deprecation. Only 2 APIs (hibernate-core and mongo-java-driver) affect more than 15% of their consumers. For these APIs we see reactions, where consumers abandon the API in favor of another.

5.3.4 RQ4: Explaining the non-reactions

We see that not reacting to deprecation is by far the most popular reaction pattern. This is an unexpected finding and we delve deeper into our data and survey developers to gain a thorough understanding behind this phenomenon.

We do this in two ways: (1) we analyze the impact of the API's evolution strategy in the consumers' reaction pattern and (2) we ask API consumers what motivates them to not react to deprecation.

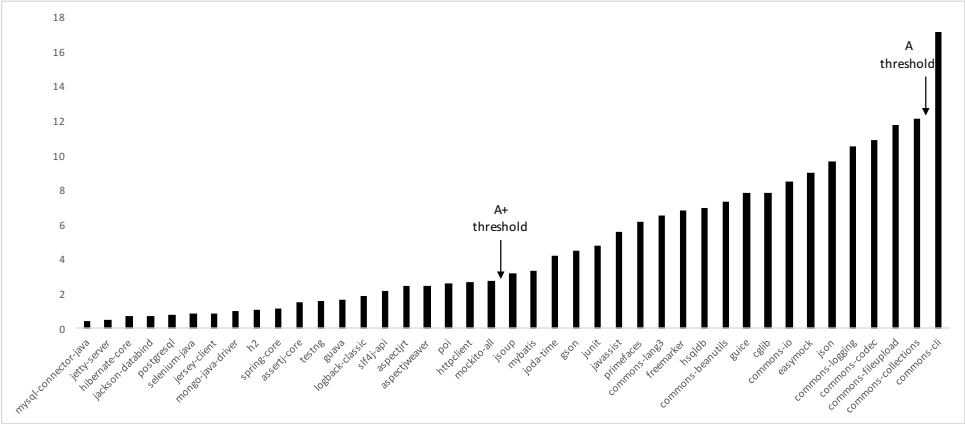
Consumers' reactions and API deprecation policies

In most cases, API consumers do not react to the deprecation of an API artifact; in some instances where a reaction does take place, the nature of these reactions can be diverse. We would like to investigate whether the API's deprecation and evolution strategies are associated with the consumer's behavior toward deprecation.

In Section 5.2.6 we list the four dimensions along which we measure the behavior concerning an API's evolution; for each of the dimensions we define the thresholds for specific categorization by manually analyzing the graphs [4]. We choose such a methodology over using quartiles as it allows us to take into account large changes in values and assign

[4]https://www.xaprb.com/blog/2015/11/07/setting-thresholds-with-quantiles/

appropriate buckets to these values as opposed to lumping all values within a quartile in the same bucket. The thresholds are detailed in Table 5.2.



5

Figure 5.8: Activeness of APIs, excluding inactive APIs.

We start by looking at the release activeness of the APIs. We notice that some APIs have not been active for more than 5 years and denote these as inactive. For the rest, we define thresholds based on the number of releases per months. A breakdown of this metric per API along with the chosen threshold points can be seen in Figure 5.8.

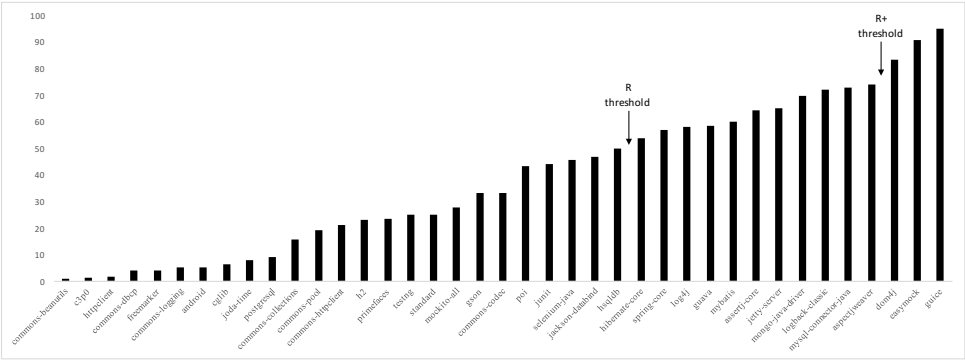


Figure 5.9: Removal of deprecated features, excluding APIs that never remove a deprecated feature.

We look at the percentage of deprecated features that an API removes in its lifetime. For some APIs, no deprecated features are removed. For the rest, we define thresholds based on the graph seen in Figure 5.9.

We also take into account the percentage of the API that is deprecated over its lifetime. For all our APIs there is at least one feature that has been deprecated. We capture the frequency with which features are deprecated in different bins, the thresholds for which can be seen in Figure 5.10.

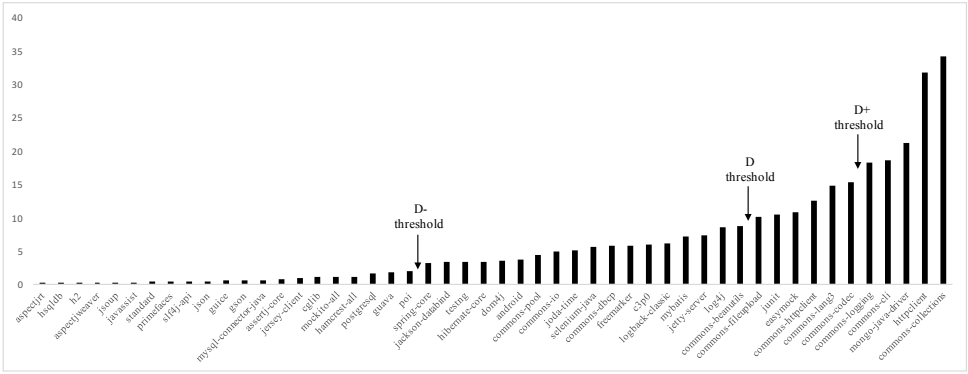


Figure 5.10: Frequency of deprecating features, excluding APIs that do not deprecate at all.

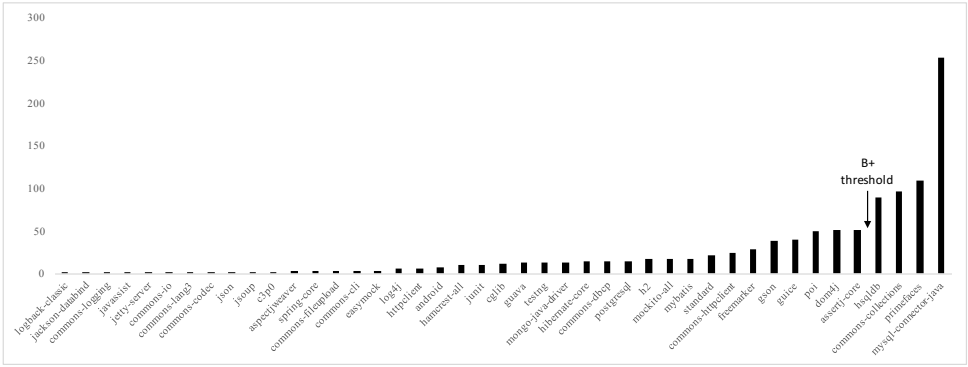


Figure 5.11: Frequency of breaking features, excluding APIs that never directly introducing breaking changes.

Finally, we look at the number of breaking changes that an API introduces. The thresholds defined for this can be seen in Figure 5.11. We scored each of the APIs based on the categories defined in Table 5.2 and conducted a card sort (as described in Section 5.2.6). This card sort resulted in nine API deprecation strategies (DSs), each with its own defining characteristic. In the following, we provide more details about each emerged DS, in terms of its description along with the aforementioned dimensions, which APIs fit in the pattern and an analysis of the behavior of the consumers of the APIs adapting this DS.

1. DS1: **Inactive**

   Description: The API is not actively releasing new versions, thus deprecated features are not being removed.

   APIs: log4j ($\not{A}$, $R+$, D, B), commons-dbcp ($\not{A}$, $R-$, D, B), hamcrest-all ($\not{A}$, $\not{R}$, $D-$, B), standard ($\not{A}$, R, $D-$, B), dom4j ($\not{A}$, $R++$, D, B), c3p0 ($\not{A}$, $R-$, D, B), commons-httpclient ($\not{A}$, R, $D+$, B), android ($\not{A}$, $R-$, D, B), commons-pool ($\not{A}$, R, D, $\not{B}$).

   Expected reaction: APIs in this category are not at all active, *i.e.*, they have not released a new version in the last 5 years, hence there is no danger in using a depre-

Table 5.2: Dimensions of API evolution and thresholds

Dimension A: Actively releasing			
$\not{A}$: last release >5 years ago	A^-: releases every >12 months	A: releases every 4-12 months	A^+: releases every 0-3 months
Dimension R: Removal of deprecated features (percentage)			
$\not{R}$: no deprecated methods removed	R: 0-50% of deprecated methods removed	R^+: 50-75% of deprecated methods removed	R^{++}: >75% of deprecated methods removed
Dimension D: Deprecated features (percentage)			
D^-: 0 - 2% of methods deprecated	D: 3-9% of methods deprecated	D^+: 10-15% of methods deprecated	D^{++}: >16% of methods deprecated
Dimension B: Breaking changes (cardinality)			
$\not{B}$: 0 breaking changes	B: 1-90 changes that break methods	B^+: >90 changes that break methods	

cated feature as there is no immediate danger of it being removed in the new release of the API. Hence, here we expect no reactions to take place.

Analysis of consumers: This is the most common strategy with 9 out of 50 APIs belonging to this category. Two of the APIs in this set never affect their consumers, while for 5 APIs over 80% of the consumers never react, as expected. For the last two APIs, some reactions do take place. Out of these APIs, 'commons-pool' is the only outlier where 100% consumers affected by deprecation react and do so by rolling back the version of the dependency being used.

2. DS2: **Very little deprecation**

   Description: The API has no history of removing features, deprecated or otherwise.

   APIs: slf4j-api ($A+$, $\not{R}$, $D-$, $\not{B}$), aspect-jrt ($A+$, $\not{R}$, $D-$, $\not{B}$), json (A, $\not{R}$, $D-$, B), javassist (A, $\not{R}$, $D-$, B), jsoup ($A+$, $\not{R}$, $D-$, B), jersey-client ($A+$, $\not{R}$, $D-$, $\not{B}$).

   Expected reaction: Here APIs deprecate features, however, they never remove deprecated features from their APIs. Thus they always remain backward compatible. In this case, we expect no reactions.

   Analysis of consumers: Six APIs appear to be backward compatible in all cases, this implies that deprecated features are never removed. For 5 of these APIs, no clients are affected (this is expected, since a defining trait of this strategy is that rare deprecations take place). However, for 'Javassist', in over 35% of the cases that a reaction does take place by replacing a deprecated call with one to a third-party API (RP6). Which implies that consumers of this API are not aware that there is no danger in not reacting as no deprecated features are ever removed.

3. DS3: **Little deprecation**

Description: The API does not deprecate a lot, but when it does, it does not remove the deprecated features.

APIs: joda-time (A, $R-$, D, $\not{B}$), postgresql ($A+$, $R-$, $D-$, B), cglib (A, $R-$, $D-$, B), commons-beanutils (A, $R-$, D, $\not{B}$), freemarker (A, $R-$, D, B).

Expected reaction: These APIs remove very few deprecated features, thus most deprecated features remain in the API and there is very little danger that they will be removed later. Here we expect no reactions.

Analysis of consumers: Five APIs that fall under this category. These APIs are mostly backward compatible, thus consumers do not need to react much as there is very little inherent danger in a deprecated feature being removed. Consumers of two APIs do appear to exhibit this behavior (RP1). However, for 'joda-time', we see rollbacks (RP7, 40% of the cases) and for 'postgresql' we see migrations to another API (RP6, 60% of the cases).

4. DS4: **Never cleans up API**

   Description: The API deprecates a lot but never really removes any deprecated feature.

   APIs: commons-io (A, $\not{R}$, D, B), commons-lang3 (A, $\not{R}$, $D+$, B), httpclient ($A+$, $R-$, $D++$, B), commons-logging (A, $R-$, $D++$, B), commons-fileupload (A, $\not{R}$, $D+$, B), commons-cli ($A-$, $\not{R}$, $D++$, B).

   Expected reaction: The APIs threaten a lot of breaking changes by deprecating a lot of features, but none of these deprecated features are removed. Thus, no reaction has to take place due to the lack of danger of using a deprecated feature. No reactions are expected.

   Analysis of consumers: Six APIs fall in this category. For three of these APIs, no consumer is affected by deprecation. On the other hand for the consumers of the other 3 APIs, we do see quite some reactions. For commons-cli (>50% of the cases), commons-lang3 (>50% of the cases) and commons-io (20% of the cases) we see that deprecated calls are replaced by another third-party API (RP6).

5. DS5: **Rarely cleans up API**

   Description: The API deprecates several features, yet only removes a few of these features.

   APIs: junit (A, R, $D+$, B), commons-codec (A, R, $D+$, B), commons-collections (A, R, $D++$, $B+$).

   Expected reaction: These APIs deprecate a lot of features, however, also remove a few of these deprecated features. Thus there is moderate danger in using a deprecated feature from this API. We expect a few reactions, but not too many.

   Analysis of consumers: Three APIs belong to this category. Just like the previous category we expect to see reactions as a lot of deprecations take place but few removals. Consumers of two APIs out of this set do not react to deprecation in over 80% of the cases. However, consumers of commons-collections do follow the more

expected pattern of reacting, by replacing deprecated invocations with those being made to other third-party APIs (RP6, >60% of cases).

6. DS6: **Directly breaks few methods**

   Description: The API removes the features sometimes, but not frequently.

   APIs: mockito-all (*A+*, *R*, *D–*, *B*), gson (*A*, *R*, *D–*, *B*), h2 (*A+*, *R*, *D–*, *B*), poi (*A+*, *R*, *D–*, *B*), primefaces (*A*, *R*, *D–*, *B+*).

   Expected reaction: Not a lot of features are deprecated, instead, some breaking changes are introduced directly. There is quite some danger to using a (deprecated or otherwise) feature from an API in this set as the API can remove a feature at any time. We expect to see reactions.

   Analysis of consumers: Five APIs exhibit this strategy. We see that in the case of 3 APIs over 98% of the consumers do not react, and one does not affect consumers at all. Only for mockito-all do consumers react by replacing deprecated calls with new ones to a third-party API (RP6, 25% of the cases).

7. DS7: **Directly breaks a lot of methods**

   Description: The API removes deprecated and non-deprecated features frequently.

   APIs: mysql-connector-java (*A+*, *R+*, *D–*, *B+*), guava (*A+*, *R+*, *D–*, *B*), aspectjweaver (*A+*, *R+*, *D–*, *B*), hsqldb (*A*, *R+*, *D–*, *B+*), guice (*A*, *R++*, *D–*, *B*), assertj-core (*A+*, *R+*, *D–*, *B*).

   Expected reaction: Here APIs introduce breaking changes with regularity instead of first deprecating a feature. We expect to see many consumers being affected by deprecation.

   Analysis of consumers: Surprisingly, this strategy is exhibited by 6 APIs. This implies that 6 APIs choose to break their client code as opposed to evolving in a clean manner and first deprecating a feature and only after that removing the feature. Thus, these APIs are not too bothered by the thought of breaking their consumers' code. This is apparent, given that consumers of 5 out of 6 APIs are unaffected by deprecation at all. Only for guava do we see that consumers are affected and they react. These reactions are either by migrating to another API or rolling back the version of the API being used.

8. DS8: **Removes deprecated features**

   Description: The deprecated features are sometimes removed in a future version.

   APIs: jackson-databind (*A+*, *R*, *D*, *B*), testng (*A+*, *R*, *D*, *B*), selenium-java (*A+*, *R*, *D*, ~~*B*~~).

   Expected reaction: These APIs are quite active and they deprecate and remove deprecated features regularly. Given the danger of using these features, we expect reactions.

   Analysis of consumers: 3 APIs exhibit this strategy. For testng, in over 60% of the cases, there is no reaction seen. Whereas for selenium-java no API consumers are

affected by deprecation. However, only for jackson-databind do we see that consumers react in just under 40% of the cases by replacing with another API (RP6).

9. DS9: **Actively cleans up deprecated features**

   Description: The API removes most deprecated features in future versions.

   APIs: spring-core (*A+*, *R+*, *D*, *B*), hibernate-core (*A+*, *R+*, *D*, *B*), logback-classic (*A+*, *R+*, *D*, *B*), mybatis (*A+*, *R+*, *D*, *B*), mongo-java-driver (*A+*, *R+*, *D++*, *B*), easymock (*A*, *R++*, *D+*, *B*), jetty-server (*A+*, *R+*, *D*, *B*).

   Expected reaction: Here APIs remove deprecated features with regularity, thus ensuring that consumers will be confronted with breaking changes when upgrading the version of an API. We expect to see reactions.

   Analysis of consumers: This strategy is exhibited by 7 APIs. We see in the case of spring-core, hibernate-core, mybatis, and easymock that no reactions actually take place in over 80% of the cases. Only in the case of mongo-java-driver do we see that in 35% of the cases do reactions take place, where consumers replace deprecated invocations with those to another API (RP6). This counter-intuitive behavior of the API consumers can be explained by the fact that majority of the consumers do not upgrade the version of the API being used, thus minimizing the chance that they will be affected by a breaking change in the API.

API consumer perspective on non-reaction

We asked API consumers to indicate whether one or more reasons for not reacting to deprecation has applied to them in the past. Results from this survey can be seen in Figure 5.12.

The most common reason (53.4% of respondents) for not reacting is that the specified replacement by the API is either too complicated to use or is not a good enough replacement. This shows that API producers might not be making their replacement features developer friendly. Furthermore, it also calls for API producers to invest in making detailed upgrade guides or improving documentation. 49% of respondents also indicated that they found reacting to deprecation time-consuming and not worthwhile. This might be explained by the fact that our survey respondents work in industry thus not having sufficient time to react to deprecation.

42% of respondents indicate that they do not react to a deprecated feature as they would rather use another API that provides similar functionality. This is in line with the earlier results where consumers found it hard to react to deprecation due to the convoluted and time-consuming nature of the replacement. Consumers also indicate that they are lax when it comes to reacting to deprecation as they do not feel particularly threatened by the deprecation as the immediate danger of a new release of the API that removes the deprecated feature being used is non-existent. This is reflected in the results seen in Section 5.3.4.

Approximately the same percentage of consumers indicate that the fact that deprecation is a non-breaking change has an impact on their decision to react (38%) as those that indicate that this fact has no impact on their decision to react (37%). Thus, consumers have diverging opinions over the effectiveness of the deprecation mechanism. However,

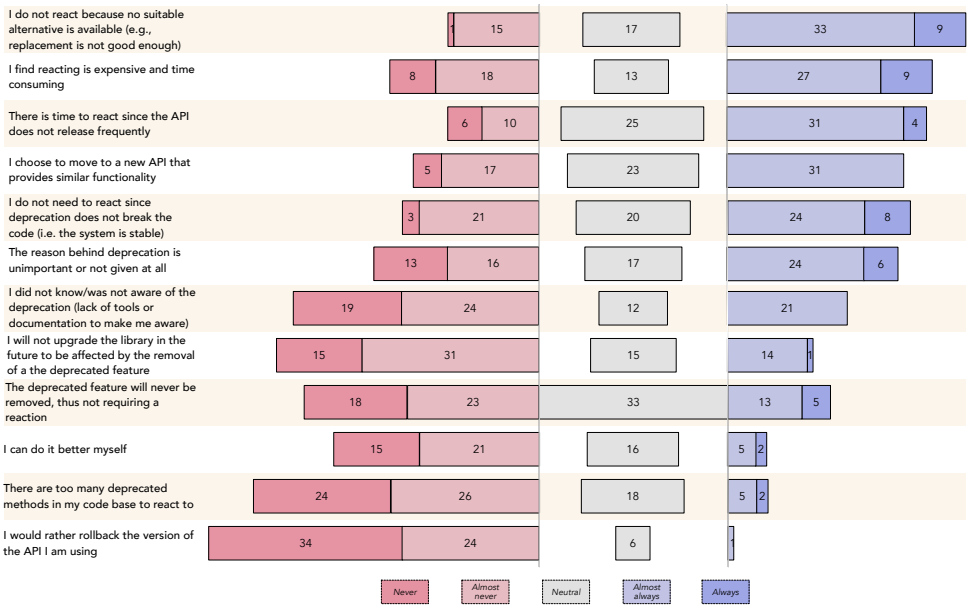


Figure 5.12: API consumers reasons for not reacting to deprecation

the fact that the deprecated feature might never be removed thereby never becoming a breaking change, does not act as motivation for non-reaction either.

The reason behind deprecation or lack thereof does not have a major impact on the non-reaction pattern observed according to 47% of consumers. This indicates that consumers feel that the reason behind deprecation is a driving factor to react to deprecation, which is in line with previous results[18].

Reasons to not react such as the ability to rollback the version being used or developing an in-house alternative to the deprecated API receive very little support from API consumers. Most indicate that such reasons do not motivate non-reactions.

Consumers also mentioned other reasons that have motivated them to not react to deprecation. One consumer mentioned that the fact that the feature had been deprecated for a long time and never been removed, made it apparent that no reaction was needed. In some cases in industry, the management does not want to invest time or money in upgrading a dependency and reacting to deprecated/breaking changes in the API. Another consumer indicated that upgrading the API binary sometimes leads to incompatibilities with other binaries thus preventing reactions to deprecation.

We wanted to further understand the reasons behind reacting to deprecation, to see what motivated consumers to react. Some of these reasons can be seen in Figure 5.13.

We see that the reason behind deprecation, the low cost of reaction, the seriousness of the deprecation and the need to upgrade the library are all considered to be very important (over 60% of consumers in each case) motivations behind reacting to deprecation. This is in line with the responses that we obtained for non-reactions.

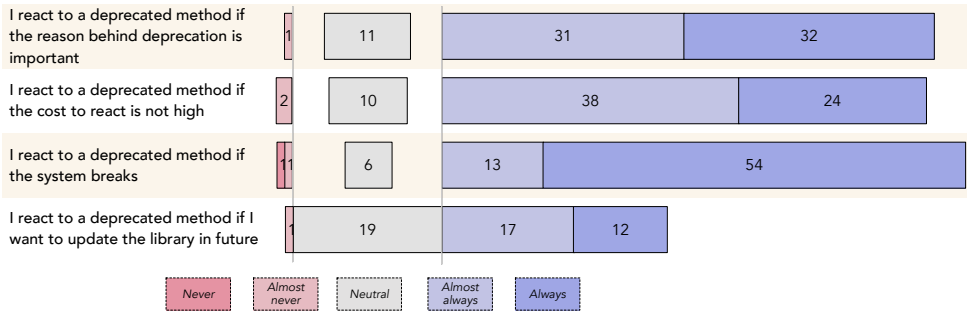


Figure 5.13: API consumers reasons for reacting to deprecation

***RQ*$_4$.** Deprecation strategies adopted by APIs are not strongly linked with the reaction patterns seen. Further diving into the non-reactions, we observe that consumers do not react to deprecation due to the cost and complexity of the required change.

5

5.4 Discussion

5.4.1 Deprecation not considered

The deprecation mechanism is used to indicate that a certain API feature is now obsolete and will be removed in the future. The aim of using such a dedicated mechanism is that it allows API consumers a period of time to react to the deprecation and take the course of action they deem most suitable for their use case.

In our study, we observe that consumers do not heed this deprecation warning. Over 95% of the cases in which a deprecated feature is used by a consumer, the depreaction is never reacted to. In the other 5% of the cases, the nature of the reaction varies and we have observed six different reaction patterns.

In 55% of the cases where no reaction has taken place, the file containing the deprecated invocation is never modified. This could indicate that API consumers may not notice that an API feature being used in their source code is deprecated. If confirmed, this situation would strengthen the case that simple compiler warnings for deprecation might not be sufficient in calling the consumers attention to this issue; this would call for another manner in which the visibility of a deprecated feature can be promoted. Java language developers have attempted to address this by introducing a dedicated tool called jdeprscan[5], which scans Java class files to detect the usage of deprecated features and warns the consumer.

The scale of the non-reactions is surprising and shows us that the deprecation mechanism is not achieving its stated goal. In fact, consumers claim that the reason behind deprecation is what drives their decision to react. This is similar to what Sawant *et al.* [18] found in their qualitative study. They recommend that the deprecation mechanism implementation in Java should make it explicit as to what the reason behind the deprecation is.

[5]https://docs.oracle.com/javase/9/tools/jdeprscan.htm

Seeing the scale of non-reactions, we can speculate that this enhancement in deprecation will entice more consumers to react to deprecation.

Overall, by increasing the overall awareness surrounding the deprecation of a feature, there is a chance that more consumers may react. This would require providing the consumer with more visibility of the deprecated feature and the motivation of the deprecation, in addition to sufficient documentation that acts as a guide to reaction. Further studies can be devised to verify whether and to what extent these three factors combined may reduce the frequency of non-reactions (from the 88% it is at right now).

5.4.2 Lack of affectedness by deprecation

Deprecation is how API producers can indicate obsolete functionality and warn consumers to not use certain features. The principal stated use case for deprecation is when an old/obsolete feature is replaced by a new implementation, but the API producers do not want to directly replace the existing feature as doing so would directly break consumer code, thus making deprecation a compromise solution.

We would expect for the APIs in our dataset that several features would have gone through the deprecation process and deprecation would have affected consumers significantly. In fact, the APIs under consideration are mature, popular, large, and have evolved significantly over time. However, we see that only 9,317 out of 297,254 (3%) Java projects on GitHub are affected by deprecation. This is surprising as we would have expected API evolution to have affected more consumers. Indeed, investigating the API deprecation behavior, we see that it is rare that an API deprecates large portions of the API. Furthermore, we see that for some popular APIs such as Guava, breaking changes are introduced in conjunction with deprecating a feature. Both these facts might explain the lack of affected consumers that we observe.

On the other hand, our results may also be the result of consumers rarely updating the APIs they use. We see that for most APIs only 5% of the consumers upgrade the dependency that they use with none of the APIs having more than 13% of their consumers that upgrade versions. This is in line with previous work by Sawant*et al.* [120] who reported that a small number of consumers upgrade versions. The rare upgrading may contribute to explain why few consumers are affected by deprecation.

5.4.3 API producers' policies are not associated to consumers' reactions

API producers use deprecation to communicate with their consumers about the obsolete nature of an API feature. They can use different evolution strategies when it comes to deprecating features: They might deprecate a lot of features, and never remove these deprecated features, or they could directly introduce breaking changes without first deprecating a method.

We observe that the various deprecation strategies adopted by producers are minimally associated with the consumers' decision to react to deprecation. Even when it would be imperative that a consumer reacted to deprecation due to the danger of that deprecated method being removed by the API, we observe consumers not reacting.

The Java language designers in JEP277 [117] estimate that the variance in deprecation strategies and having no singular convention has led to the confusion surrounding dep-

recation. Consumers are unsure of whether they have to react or not. They are unaware of the future of a deprecated feature. In the opinion of the Java language designers, this confusion has led to consumers doing nothing with a deprecated feature.

5.4.4 Consumers do not keep up with API evolution

Most of the APIs that we study regularly release new versions, which contain a combination of improvements to existing features, addition of new features, and/or removal/deprecation of older features.

We observe that, for any of the studied APIs, only a maximum of 13% of consumers move to the latest version of the API. In the survey, only 31% of developers indicate that they always upgrade the version of the API being used. These numbers are aligned, albeit less extreme, with previous studies, one of which reported that 81.5% of consumers do not change the version [123] and another that less than 4% of library dependencies are upgraded [124].

Considering the consumers that do upgrade, we observe that they tend to not react to deprecation. They choose not to use the new features that replace obsolete features. In fact, replacing with the recommended replacement is one of the least popular ways in which we found consumers reacting to deprecation.

This evidence seems to indicate that consumers are not concerned with keeping up with API evolution. Consumers pick the version that works best for them and stick to it (a fact echoed by 41% of developers in the survey). This leads us to question whether APIs should be evolved with great regularity or if API producers should invest time in making minor improvements and releasing them. Concerning upgrading behavior, Bavota*et al.* [125] found that consumers are more likely to adopt a new version if it includes major improvements.

5.4.5 Need for an automated tool to keep with API evolution

One of the principal reasons behind consumers not keeping up with API evolution, as emerged from the survey, is the cost involved in upgrading dependencies and reacting to deprecated or broken features. In the case of industry, this cost is too high a price to pay to change something that is already working, as explicitly stated by one of our survey respondents.

This hints at the usefulness of tools that can enable developers to deal with API evolution in a cost-effective manner. Some attempts at automatically dealing with deprecation have already been made. Henkel and Diwan [27] proposed to capture refactoring operations made by API producers to their codebase when adapting to their own deprecated features and then replaying these operations on the API consumers. Similarly, Xing and Stroulia [28] 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. These approaches rely on support from the API developers to aid the API consumers in the transition. Attempts have been made to create automated tools that aid developers in dealing with API evolution and not just specifically deprecation. Dagenais and Robillard present a tool called SemDiff [29], which aids developers in dealing with framework changes where a method they use is suddenly no longer provided. Schäfer*et al.* [90] mine framework usage rule changes from already ported usages of the framework

and propose the same to a developer dealing with a breaking change. Kapur *et al.* [126] created a tool call Trident that allowed developers to directly refactor obsolete API calls. Savga and Rudolf created Comeback! [127], a tool that records framework evolution and for each change creates an adapter so that a developer does not have to change his code.

The only tool that has been created specifically to support API consumers in transitioning away from deprecated API features is that created by Perkins *et al.* [128]. This tool 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 this approach is not universal and introduces verbose code to the API consumers codebase.

One assumption that all these aforementioned approaches make is that consumers upgrade the dependency version being used. As we see in this study, developers mostly do not change the version of the API they use. This calls for the development of techniques that actually incentivize the adoption of new versions by reducing the cost and time involved in upgrading. Holmes and Walker [116] have made a start by creating a tool that filters relevant change information from a library so that consumers are made aware of the major changes made to the API. Furthermore, as our study has shown us, consumers are very likely to take deprecation lightly as it is not a breaking change and does not require immediate attention. This calls for a tool that effectively aids an API consumer in the transition away from deprecated API features. Both these tools are still an open research challenge.

While in this context we only talk about the needs of Java developers, such kind of tooling support in other languages would also go a long way in aiding API consumers in dealing with API evolution.

5.4.6 Comparison with other languages

In this chapter, we focus on Java-based APIs and their consumers. We identify seven different ways in which API consumers could react to deprecated API features in their source code. However, we see that there is very little reaction to deprecated API features, in fact, not reacting is the most popular way in which consumers choose to deal with deprecated API features.

Java's deprecation mechanism and deprecation policy have been blamed as the main reason behind the lack of reaction to deprecated features. JEP 277 [117] mentions that more information needs to be conveyed to the consumer, information that can prove pivotal in the decision-making process behind reacting to deprecation. Sawant *et al.* [18] confirm this and suggest that Java's deprecation mechanism be extended to inform the consumer about the severity of the deprecation and the version in which the deprecated feature is to be removed.

Languages such as C#, Kotlin and Visual Basic provide a way in which the severity of the deprecation can be conveyed. Ruby and Dart allow API consumers to indicate the version in which a deprecated feature is going to be removed. We postulate that given the difference in the information that is conveyed by the API producer to the consumer in such languages, the variety and scale of the reaction to deprecation is probably very different to that what we have observed in Java. This hypothesis is strengthened when we observe the results from a previous study by Robbes *et al.* [3] on the Smalltalk (whose deprecation mechanism is similar to Java's in terms of characteristics) ecosystem where

the number of reactions is not that high either.

A study to compare reaction patterns to deprecation across the different programming languages would allow us to confirm whether the implementation of a deprecation mechanism has an impact on consumer behavior. Such a study is out of scope for this paper, however, we propose it as future work.

5.4.7 Semantic versioning impacting deprecation reaction behavior

The practice of deprecation is related to the practice of semantic versioning. In semantic versioning, package version numbers follow a specific scheme consisting of 3 numbers: MAJOR.MINOR.PATCH. When releasing a new version, one of the numbers will be incremented according to the following rules (from https://semver.org):

- The MAJOR number is incremented when a new version makes incompatible API changes
- The MINOR number is incremented when new functionality is added in a backward-compatible manner
- The PATCH number is incremented when a backward-compatible bug fix is issued

On the other hand, the goal of deprecation is to provide an incentive for developers to change their software, but without breaking backward compatibility. Thus, an API deprecating even a large number of API elements does not need to increase its major number to comply with semantic versioning. The actual breaking change would happen when the deprecated API element is removed; only then would an API following semantic versioning need to increase its major version number.

Both mechanisms can be seen as complementary. One could see deprecation as an "advance warning" that an API element is likely to be removed in the future, such as a future major version of the API. Indeed some APIs, such as Guava, explicitly note in their deprecation message when a method is scheduled to be removed (*e.g.*, "This method is scheduled to be removed in Guava 16.0"). This strategy seems to be "the best of both worlds".

While such a strategy seems to be the best way to approach the problem, based on our study and the studies of Raemaekers *et al.* [19, 129] on semantic versioning, our outlook on the chances of such a mechanism leading to a desired behavior in practice (*i.e.*, rapid adaptation to deprecation and API changes) is pessimistic. The work of Raemaekers *et al.* shows that many packages on Maven central do not follow semantic versioning, incurring rework for their consumers. Our work shows that few API consumers actually react to deprecation. Thus, it is unclear whether combining both approaches would be more successful, although a specific study of this would be the best way to obtain concrete evidence.

5.5 Related Work

5.5.1 Studies on API deprecation

Several studies investigate the deprecation of API features, its impact and its need to help developers deal with deprecation.

Robbes *et al.* [3] and Hora *et al.* [89] have studied the impact of deprecation of APIs in the Pharo ecosystem. Robbes*et al.* focused on the deprecation of certain API features and the effect that this deprecation has on the entire Pharo ecosystem. They found that the deprecation of a single feature can have a large impact on the ecosystem, despite this only a small proportion of consumers bother reacting to deprecation. Hora *et al.* looked at changes to the API changes not marked as deprecated beforehand. They find that a larger number of consumers react to API changes marked as deprecated as opposed to those not marked as deprecated, thus showing that in the Pharo ecosystem a larger proportion of consumers consider deprecation to be of importance.

Sawant *et al.* [13] performed a large-scale study on GitHub based consumers of 5 popular Java APIs to see how they are affected by deprecation and whether or not they make a move to react to deprecated features. They found that only a small proportion of consumers update dependency versions. Furthermore, the proportion of consumers affected by deprecation varies per API, however, irrespective of the scale of affected consumers, the number of reactions is minimal. Sawant *et al.* extended this study to look at consumers of the Java JDK API [14]. They found that for the Java API consumers are rarely affected by deprecation, dispelling the notion put forth by the Java developers themselves. Furthermore, they theorized that an APIs deprecation policy would have an impact on the consumers' decision to react to deprecation, but did not show any conclusive evidence in either direction. In this study, we show that the deprecation practices have a minimal impact on the consumers' reaction patterns.

Brito *et al.* [98] analyzed the documentation accompanying deprecated features in 661 Java systems. They found that in 64% of the cases, API producers had taken the effort to recommend a replacement in the documentation. Brito *et al.*'s study does not look at whether these replacement messages also mention the rationale behind the deprecation, a facet of the documentation that the consumers find important as confirmed by our study.

Hou and Yao [130] studied release notes of the JDK, AWT and Swing APIs, looking for rationales for the evolution of the APIs. They found that in the case of deprecated API features, several reasons were evoked: Conformance to API naming conventions, naming improvements (increasing precision, conciseness, fixing typos), simplification of the API, reduction of coupling, improving encapsulation, or replacement of functionality. Many of these rationales mirror those mentioned in the Java documentation on deprecation. They also found that only a small portion of API features were deleted without a replacement specified. Sawant*et al.* [18] asked API producers why they used the deprecation mechanism. They also asked producers if they preferred that consumers would react to deprecation and what they did to support any kind of reaction. Based on their findings, they propose some more changes that have to be made to the deprecation mechanism so that it fulfills the needs of both API producers and consumers.

5.5.2 Studies on API evolution

API producers have to decide what part of an API they have to exclude from public access, these are the so-called internal APIs. These parts of the API are reserved for use by the API itself and not intended for public consumption. Businge *et al.* [95] found that out of 512 Eclipse plugins, 44% use internal Eclipse APIs. This finding is confirmed by Hora *et al.* [96] who found that 23.5% of 9,702 GitHub based Eclipse client projects use an internal

API. Businge *et al.* [97] dived deeper into the reasons behind developers using internal APIs and they found that developers preferred to use an internal API as it provided them with functionality that other public APIs did not, thus sparing them time and development effort. Hora *et al.* [96] found that internal APIs are sometimes promoted to public APIs. To aid API producers in the selection of what API should be promoted, Hora *et al.* presented an approach for such promotion. The consumers' propensity to use features that they are not supposed to use, is reflected in our study as well, where consumers do not want to transition away from deprecated features. This shows that maintainability takes a back seat to functionality, which leads to technical debt.

Raemaekers *et al.* investigated the relationship between breaking changes, deprecation, and the semantic versioning policy adopted by an API [100]. They analyzed a dataset based on 100,000 JAR files on Maven central. They found that API producers introduce deprecated artifacts and breaking changes in equal measure across both minor and major API versions, thus not allowing consumers to predict API stability from semantic versioning. In a follow-up to this study, Raemaekers*et al.* [129] found that these breaking changes induce a lot of rework in consumers. Furthermore, the deprecation tags used by these Maven based projects are often used incorrectly.

APIs often introduce breaking changes that directly affect a consumer. Dig and Johnson studied and classified the API breaking changes in 4 APIs [26], however, they did not investigate the impact of these breaking changes on consumers. They found that 80% of breaking changes were due to refactorings performed by the API producers and released without deprecating the original implementation. Wu *et al.* [131] analyzed the Eclipse ecosystem to see how an API change would affect an API consumer. They found that missing API classes affect consumers more frequently than breaking changes. They also find that 11% of API changes can cause a ripple effect among API consumers. Wu *et al.* [132] propose a tool called ACUA which would give API producers an overview of the impact of the change that they would make to an API.

In a large-scale study of 317 APIs, Laerte *et al.* [9] found that for the median library, 14.78% of API changes break compatibility with its previous versions and that the frequency of these changes increases over time. However, not many clients are impacted by these breaking changes (median of 2.54%). Bogart *et al.* [107] conducted interviews with API developers in 3 software ecosystems: Eclipse, npm, and R/CRAN. They found that each ecosystem had a different set of values that influenced their policies and their decisions of whether to break the API or not. In the case of R/CRAN both Decan*et al.* [109] and Bogart*et al.* found that there is a policy of forcing packages to work with one another, which is perceived to be a problem. Decan *et al.* [108] also investigated the evolution of the package dependencies in the npm, CRAN, and RubyGems ecosystems. They found that there is an increasing tendency in the npm and (to a lesser extent) RubyGems packages to specify maximal version constraints, thus allowing certain package maintainers to protect themselves from package updates.

Bavota *et al.* [125] qualitatively investigated a Java subset of the Apache ecosystem to see how their dependencies change over time. They observed that when an API adds a lot of new features, consumers are more likely to adopt this new version, thus triggering a change in the dependency. However, when the changes are small and insignificant or if a removal of a feature takes place, then consumers prefer not to change versions of the

dependency.

The policy of evolving the Android APIs has been studied as well. McDonnell *et al.* [101] investigate stability and adoption of the Android API on 10 systems. They found that the Android API's policy of evolving frequently leads to the consumers being adversely affected by breaking changes, thereby creating many issues when it comes to dealing with API evolution. Linares-Vásquez *et al.* [102] also focus on the Android ecosystem, however, they look to analyze StackOverflow posts to see how consumers deal with API evolution. They found that there are more StackOverflow conversations when there is a change in the Android API, thus further showcasing issues with dealing with API evolution. Bavota *et al.* [103] analyzed how changes in APIs being used by apps on the Google Play Store affects app ratings. They show that when more breaking changes are introduced, the rating of the app is lowered.

Web-based API evolution policies have also been studied. Wang *et al.* [104] study the case of evolution of 11 APIs by analyzing questions and answers on StackOverflow concerning the evolution of these APIs. study the specific case of the evolution of 11 REST APIs. They identify 21 change types to an API that affects consumers and spark a discussion on StackOverflow. Espinha*et al.* [133] analyze how major web API providers evolve their APIs, and they find that there is no unified way in which web APIs are evolved thus leading to confusion among consumers. Espinha *et al.* [105] also studied 43 mobile consumer applications depending on web APIs and how they respond to mutations in the web API. They show that in over 30% of the cases the mobile app fails when the web API response is changed.

5.5.3 Supporting API evolution

Researchers have proposed many approaches to aid consumers in dealing with the evolution of an API. One of the first approaches was by Chow and Notkin [112], where they require API producers to annotate changed methods with replacement rules that will be used to update consumer code. Henkel and Diwan [27] propose CatchUp!, a tool that captures refactorings in the IDE and replays the, on other unrefactored code in the consumer's code base. Xing and Stroulia [28] propose an approach that analyzes how an API has itself handled changes to its features and then recommends these changes to the API consumer. Dig *et al.* [114] propose MolhadoRef a refactoring-aware version control system that works in a similar manner as CatchUp!.

Dagenais and Robillard present SemDiff [24, 29] a tool that observes the framework's evolution to make API change recommendations. Schäfer *et al.* mine the consumer's reaction to API evolution [90], and then propose these mined changes to other consumers dealing with the same evolution issues. Wu *et al.* present a hybrid approach [7] that uses call dependency and textual similarity to recommend adaptations to the API changes.

Kapur *et al.* [126] created a tool call Trident that allows consumers to directly refactor obsolete API calls in the IDE. Savga and Rudolf created Comeback! [127] which records framework changes and recommends ways to adapt to these changes to the consumer.

Nguyen *et al.* [115] propose a tool called LibSync that uses graph- based techniques to help consumers migrate from one framework version to another. Holmes and Walker notify developers of essential changes made to external dependencies to draw their attention to these events [116].

Cossette and Walker [106] studied five Java APIs to evaluate how API evolution recommenders would perform in the cases of API breaking changes. They found that all recommenders handle only a subset of the cases, but that none of them could handle all the cases.

Finally, Perkins*et al.* [128] created a tool that was specifically targeted to consumers dealing with deprecated features in their codebase. This tool replaces the invocation made to deprecated features with the code from the deprecated method itself, thereby aiding in the removal of the deprecation warning message on the code.

5.6 Implications

The key implications that we distill from this study are:

- The majority of Java-based projects on GitHub do not show substantial reaction to deprecation, this is reflected by the fact that there are very few reactions to deprecated features and API consumers choose to keep references to deprecated features in their source code. This may suggest that a deprecation *per se* is not perceived as a good enough reason to change one's code. This result is in line with previous research [18], which provided evidence that consumers would like to have more information (*e.g.*, severity of the deprecation or version in which the feature will be removed) to take a more informed decision when it comes to reacting to deprecation.

- API consumers need to be made more aware of the API's evolution policy. We notice that in cases where no reaction need take place a reaction does take place and other cases where the need to react is more pressing (due to the APIs propensity to remove deprecated features) we observe no reaction. This seems to indicate that consumers do not know the exact evolution behavior of the API that they are using. This information can either be composed in a new online platform or on the Maven central website where the dependency is hosted, such that consumers have an early warning mechanism at their disposal.

- The scale of non-reactions may also indicate that the effort to manually make every single transition from a deprecated feature to a non-deprecated one may be not trivial. This puts into focus the increasing need for an automated tool that aids in API transition that would reduce developer burden on this front.

5.7 Conclusion

We have presented a large empirical scale study that analyzes how frequently an API consumer reacts to deprecation in an API. This is the first work of its kind that identifies the various possible reaction patterns that can take place. We identify seven reaction patterns by way of manual analysis of API consumer code. We then quantify the frequency of these reaction patterns by mining and analyzing API usages of 50 popular Java based and their consumers. The overall size of the dataset under consideration encompasses 297,254 projects and over 1.3 billion API usages.

In our manual analysis we saw that the bulk of the consumers never react to deprecated features. This fact is reflected in our large scale analysis as well. Surprisingly, replacing

with the recommended replacement only happens in 0.02% of the cases, while non reactions happen in 88% of the cases. This shows that API consumers are either unconcerned with the fact that their code uses deprecated API features or have not noticed this fact. Furthermore, when diving into they upgrade behavior of API consumers, we see that very few consumers even change the version of the API that they use.

We asked developers as to why they would not upgrade the version of the API that they used. And the two major reasons behind this is that the cost involved with the upgrade is often not worth it, and given that the version being used works, there is no pressing need to upgrade. Developers were also asked to rank the reasons behind not reacting to deprecation. Here the developers indicated that the replacement was either too convoluted to use or the cost of reacting was too high. It appears that all these consumers subscribe to the "if it isn't broken then why fix it?" theory.

One of the major contributing factors to this behavior is that deprecation is not viewed as seriously as it should be. A fact that the Java JDK developers accede to as well. Multiple improvements and changes are needed for consumers to take deprecation warning seriously. However, with this study we are able to confirm that issues with the current implementation of deprecation do indeed exist.

6

Why API producers deprecate features

An Application Programming Interface (API) is a set of defined functionalities provided by a programming library or framework.[1] APIs promote the reuse of existing software components [39]. By integrating a third-party API in a code base, a developer can save development time and effort and use a well-tested system.

To remain useful in a mutating environment [21], most APIs evolve by introducing new features, removing older ones, and changing existing features. Some of the changes due to API's evolution can be breaking in nature and can have an adverse impact on the API consumers [135]. One way for API producers to avoid directly introducing a breaking change in their API first to *deprecate* the feature, thus communicating a warning to the consumers. Deprecation is available in most mainstream languages such as C#, PHP, and Java. Regarding a definition of 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" [10].

API deprecations are commonplace [14], however, to what extent is their motivation clear? Recent research has reported that API consumers decide on whether to react to API deprecation, based on the reason behind the deprecation [136], thus hinting at the several possible purposes for deprecating a feature. Indeed, deprecation is a convenient way for API producers to make sure that both popular IDEs and compiler inform API consumers that something is not right—deprecation is a *unique communication mechanism* and, as such, can be used for conveying different messages.

The goal of this study is investigating the reasons behind feature deprecation. The critical motivations for pursuing this goal include: (1) gaining a deeper understanding of a popular language feature, from a new angle, (2) discovering unmet developers' communication needs, by uncovering unorthodox usages of deprecation,[2] which may signal those needs, (3) investigating what deprecation says regarding APIs' evolution. The results can

[1] In this chapter, with the term *API* we refer only to local APIs (*e.g.*, frameworks and libraries), as opposed to web-APIs [134].

[2] In the opinion of the Java language designers, developers often misuse the deprecation mechanism [117].

inform and guide practitioners' practices as well as future academic studies on this and similar mechanisms, and on developers' communication.

To this aim, we conduct an in-depth analysis of 374 deprecated features in four popular Java APIs: Spring [84] (15,086 users), Hibernate [83] (8,143 users), Guava [81] (9,542 users), and Easymock [80] (1,484 users). Our study is exploratory and we answer three research questions: why are API features deprecated, what is the frequency of deprecation rationales, and how well can we automatically classify the reason for a deprecation from software repositories.

To uncover the various rationales for a deprecation we manually analyze over 1,100 documents relating to 374 deprecated features. Three authors conducted this analysis and a fourth validated it. This effort results in the creation of a taxonomy of 12 rationales behind deprecation. We then investigate what rationales have been used most frequently across the considered APIs. Finally, we employ a supervised machine learning approach [137] to create an automated approach to infer the rationale behind a deprecation. We evaluate the performance by using different cross validation [138] techniques.

We found that determining the reason for deprecating a feature is far from trivial: The motivation is rarely mentioned in the accompanying Javadoc. Nevertheless, through the analysis of software repositories (mainly, code versioning and issue tracking systems) we could define a taxonomy of 12 high-level reasons. Of these, two are unorthodox uses of deprecation (for temporary features and for incomplete implementations), thus indicating unmet developers' communication needs. Finally, we found that an automated approach to classify the deprecation reasons based on machine learning reaches promising results, but only if trained on project-specific instances.

6.1 Motivation

Figure 6.1 shows an example deprecation message from the Spring framework. We see that the API producers have deprecated the feature in version 4.2.3 of the API and they recommend that the consumer use an alternative feature. The Javadoc does not explain the rationale behind this change. By recovering the commit (seen in Figure 6.2) in which this feature was deprecated, we find that it contains no rationale behind the deprecation; hence, we have to refer to the JIRA issue ID mentioned in the commit. From the Spring issue tracker post (seen in Figure 6.3), we see that there is a performance slowdown when using specific methods from the Annotation class. To rectify this, Spring introduced a replacement feature to fix the issue and deprecated the original element.

This example shows that the deprecation of a feature itself does not necessarily carry its reason, but uncovering (although complicated, as in this case) and understanding the possible types of deprecation reasons is relevant from many perspectives, including those we describe in the following.

(1) To guide practitioners and research tools. API consumers have indicated that knowing the motivation behind deprecation is critical to decide whether to react [136]. Empirically uncovering the possible reasons for deprecation can suggest to practitioners whether they should respond to a deprecation in principle, as well as whether the motivations are project-specific or can be mostly generalized. Knowing deprecation reasons can inform the design of tools to better support the replacement of a deprecated feature, by exploiting the deprecation reason.

findMergedAnnotation

```
@Deprecated
public static <A extends
Annotation> A findMergedAnnotation(AnnotatedElement element,
String annotationName)
```

Deprecated. *As of Spring Framework 4.2.3, use* `findMergedAnnotation(AnnotatedElement, Class)` *instead.*

Find the first annotation of the specified `annotationName` within the annotation hierarchy *above* the supplied `element`, merge that annotation's attributes with *matching* attributes from annotations in lower levels of the annotation hierarchy, and synthesize the result back into an annotation of the specified `annotationName`.

`@AliasFor` semantics are fully supported, both within a single annotation and within the annotation hierarchy.

This method delegates to `findMergedAnnotationAttributes(AnnotatedElement, String, boolean, boolean)` (supplying `false` for `classValuesAsString` and `nestedAnnotationsAsMap`) and `AnnotationUtils.synthesizeAnnotation(Map, Class, AnnotatedElement)`.

This method follows *find semantics* as described in the class-level javadoc.

Since:

`4.2`

Figure 6.1: Javadoc of a deprecated API feature from the Spring API [139].

(2) To uncover unmet communication needs. When Java was in the process of changing its deprecation mechanism for Java 9, a motivation was that API producers were misusing the deprecation mechanism. This misusage may signal that deprecation is used to fulfill a communication need that is unmet by any other tool. Knowing the various reasons behind deprecation allows us to understand how many different cases of misuse of the deprecation mechanism have taken place and may let us discover what needs future research should address to devise a more appropriate communication tool.

(3) Understand how an API evolves. APIs evolve and replace old features with new ones. The older features are deprecated and not directly removed from the API to minimize the number of breaking changes introduced. Knowing what reasons an API uses most popularly can aid researchers in gaining a deeper understanding as to how and why APIs evolve [4].

(4) Understanding to what extent API documentation is lacking. API documentation is an essential tool that aids API consumers in effectively and accurately using an API's features [32]. API producers must invest in the documentation for their API so that they ease the burden of adoption of API features [142]. In the case of deprecated API features, giving the consumers an indication as to what new feature should be used and how, is essential [15, 143]. In addition to that, explaining the rationale behind the deprecation and providing a timeline for the removal of the deprecated feature have been found to be essential to an API consumer. We see in Figure 6.1 that the rationale is impossible to infer,

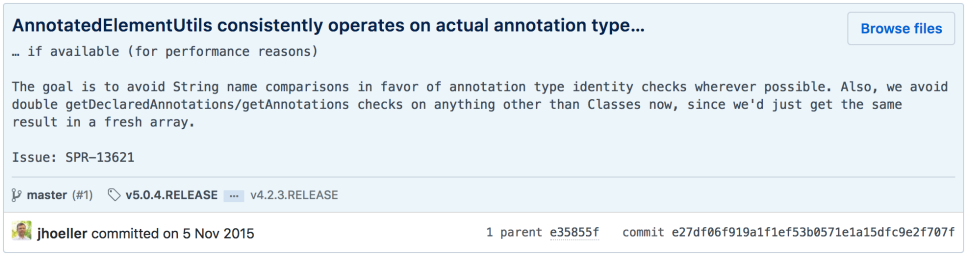


Figure 6.2: The commit deprecating the feature, pointing to the JIRA issue [140].

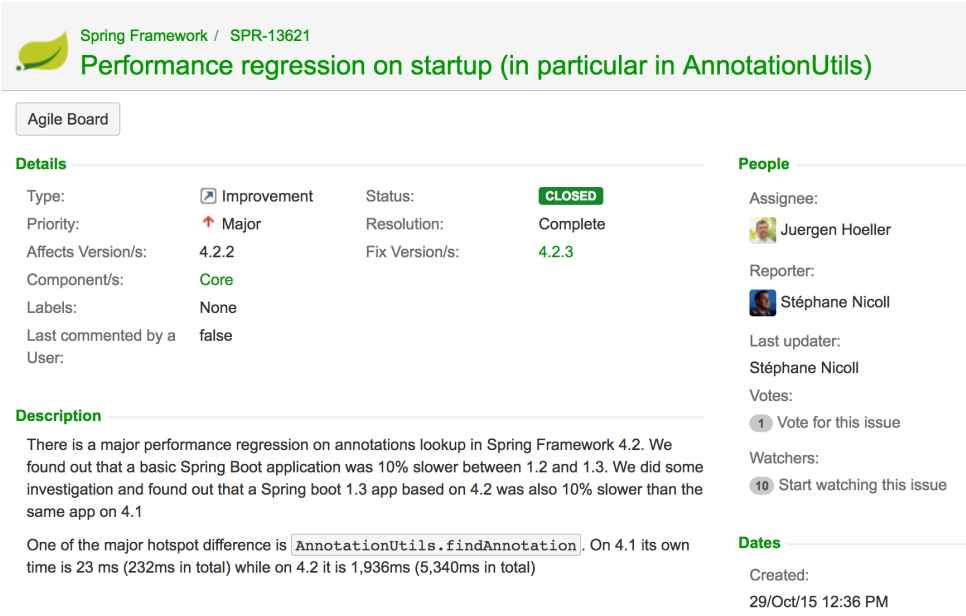


Figure 6.3: Issue detailing the need for changing the deprecated method [141].

a consumer has to read the JIRA issue on the subject (seen in Figure 6.3). In this study, we get an indication to what extent API documentation indicates the reason behind deprecation and conveys the same to the consumer, or where it can be found, thus informing practitioners, as well as researchers investigating tools to support API documentation.

6.2 Methodology

The *goal* of the study is to empirically investigate and classify the reasons that triggered the deprecation of features in popular APIs. The *perspective* is of researchers and practitioners, interested in an empirical understanding of the reasons behind deprecation, to guide practice and future research.

Our study revolves around three research questions:

RQ_1: How can reasons for deprecating features be categorized? With the first research question, we seek to investigate and classify the diversity of reasons that triggered API producers to deprecate a feature in their systems. We do this by manually analyzing the information about these features and their deprecation as they are available in software repositories.

RQ_2: How often does every reason for deprecation occur? After having categorized the reasons triggering deprecation, we analyze their frequency to quantify the different purposes of API producers.

RQ_3: How effective is an automated approach in classifying the reason behind a deprecation? Finally, we exploit the set of manually categorized reasons to investigate how effectively we can automatically classify the rationale via standard machine learning techniques, using the relevant data from the software repositories. Should the results of this automatic classification be promising, future research could investigate tools to automatically augment existing API documentation with the rationale behind the deprecation, thus providing useful information [136] to the API consumers.

6.2.1 Subjects: Systems and Deprecated Features

In this study, we focus primarily on the Java ecosystem, because (1) Java is the most popular programming language [48], (2) this ecosystem has a large number of popular and mature APIs for study, and (3) the deprecation mechanism in Java is very prominent and widely used by API producers [14].

Systems. From the Java ecosystem, we select four third-party open-source software APIs.[3] Our goal and research methods dictate the choice of limiting ourselves to four APIs. On the one hand, we strive to collect as many diverse reasons as possible to increase our empirical understanding of this phenomenon; on the other hand, we can realistically investigate no more than a few hundred deprecated features, because understanding the reason of deprecation requires perusing a possibly large number of documents per feature (as seen in the example in Figures 6.1–6.2). Given these requirements, investigating a large number of systems is suboptimal: Keeping the number of features we can analyze equal, it is reasonable to think that we are more likely to find a smaller diversity of reasons in more systems (*i.e.*, we find only the most occurring reasons per system), than in fewer systems but studied more in-depth. Hence we limit ourselves to four systems.

As criteria for the choice of the four systems, we consider popularity (as defined by the number of Java projects on GitHub that use the system; we use the dataset by Sawant and Bacchelli to benchmark the popularity [47, 120]), size, length of history, number of deprecated features, availability of software repositories, and diversity in producers (*e.g.*, we would not consider two APIs from Google) as well as domain. Table 6.1 describes the APIs we eventually selected (*i.e.*, Guava, Spring, Easymock, and Hibernate).

Deprecated features. We focus on the latest available version of each API. Since these are all Java-based projects, we use the Eclipse JDT AST parser [144] to identify all

[3]We do not consider the Java JDK API for our analysis because uncovering the rationale behind deprecation is not always possible as tracing alternative sources of information (*e.g.*, issue trackers and developers' communication) is hindered by the closed nature of the Oracle JDK.

Table 6.1: Overview of selected APIs

API	Description	Considered Release	Number of Consumers
Easymock	Java object mocking framework	3.5.1	1,484
Guava	Google's collections library	23.0	9,542
Hibernate	Object/relational mapping framework	5.2.12	8,143
Spring	Dependency injection framework	5.0.0	15,086

the deprecated features. The resulting dataset contains almost 2,300 deprecated methods from the four APIs. We randomly select the methods from each of the APIs to create our sample investigation set. Considering that we want to estimate proportions of reasons for RQ2, we choose a sample set size that leads to a 95% confidence level and a margin error of no more than 5% on the computed proportions [145]; this resulted in a sample of 374 deprecated features to manually investigate, together with information from other relevant data sources.

6.2.2 RQ1. Manually determining the reasons for a deprecation

To answer RQ1, we follow a three-step method. Three authors of this chapter conducted the first (S1) and second (S2) steps, while the fourth conducted the third step (S3). S1 regards the determination of the rationale behind the deprecation of an individual feature, S2 regards the grouping of individual deprecation reasons into high-level categories to create a taxonomy of reasons, S3 validates the results of the first two steps. In the following, we detail each step.

S1. Determining the reason of an individual deprecation. This step is conducted by three authors of this chapter together. For each feature, they start by inspecting the documentation that is supposed to contain the rationale and replacement for the deprecation [10]: The Javadoc associated with the feature.

They found that (1) most Javadoc messages include the annotation *@link* that links to the alternative feature that should be used instead of the deprecated feature, but (2) the message seldom includes the rationale behind the deprecation. This lack of rationale made it unfeasible to understand the reason from only the Javadoc. Thus, the investigation is expanded to include data from other software repositories, which are then inspected by the three authors:

1) Commit history. The commit message for a change can contain the rationale behind it and the nature of the change. Thus, we use the JGit project to traverse the history of each file in the master branch of the API. We then isolate the commit wherein one of the deprecated entities was first deprecated. We, thus, inspect the accompanying commit message.

2) Source code. Source code comments (not Javadoc) can often contain the rationale behind changes made to a method. These comments are usually for the benefit of the subsequent contributor to this method or file. For each of the deprecated methods, we isolate the entire source code of the method.

3) Issue tracker. Issue trackers contain discussions among developers and information on issues in the API. The rationale behind a change can be understood from the discussions and issues posted in the issue tracker if they pertain to the method under investigation. We manually isolate the issues (from JIRA or the GitHub issue tracker) mentioned in the commit messages that deprecated a feature.

4) Other sources. We perform a cursory investigation of sources such as StackOverflow, the Google search engine, developer blogs and mailing lists specific to each API. Each of these sources, for example email [146, 147], contains API consumer-/producer-driven content that can contain information on the rationale behind the deprecation of a feature. However, we found that sources are not always consistent and do not contain the information that we require.

Through the analysis of the aforementioned sources, the three authors determine a precise reason for the deprecation of each feature.

S2. A taxonomy of deprecation reasons. In the second step, the same three authors conduct three iterative content analysis sessions [148] to group the individual rationales used into higher level reasons. Iteratively, for each rationale found in the previous step, the involved authors verify whether they have previously identified a reason of this nature to which this rationale can be assigned or whether they need to create a new reason. This iterative process resulted in a taxonomy of 12 reasons for the deprecation of features.

S3. Validation. As the third and last step, another author independently repeats the analysis to verify both (i) the understandability of the category descriptions in the taxonomy from the second step and (ii) the assignment of deprecated features to these categories. The resulting inter-rater agreement between the two classifications was 93%; the authors discussed the 7% that was not agreed upon until they reached a consensus. In Section 6.3 we present the final taxonomy.

6.2.3 RQ2. Frequency of the deprecation reasons

In this research question, we aim at analyzing how frequently each category of our taxonomy appears. To this aim, we compute the frequency with which each high-level category of deprecation reason is assigned to an individual deprecated feature during the iterative content analysis. In Section 6.4 we present and discuss the results, overall and by API.

6.2.4 RQ3. Automatic classification of deprecation reasons

In our third research question, we investigate standard machine learning techniques to automatically classify the reasons of a deprecation into the taxonomy identified in RQ1. While employing a sophisticated method such as deep learning goes beyond the scope of the current work we aim to create an automatic classification technique with a fair level of accuracy.

Machine learning approaches. We employ a supervised machine learning approach [137] to create our automated inference approach. With this approach, a set of features are used to predict the value of a variable (in our case, the *classification* of the reason) using a machine learning classifier (*e.g.*, Naive Bayes [137]). The role of the classifier is to determine the importance and role of each feature in predicting the classification by learning from

already classified examples. In particular, we consider two different kinds of supervised classifiers: (1) probabilistic classifier (specifically, naive Bayes multinomial) and (2) decision tree algorithm (specifically, random forest).These classifiers make different assumptions on the underlying data, as well as have distinct advantages/drawbacks for execution speed and overfitting.

Features. To classify the reason for deprecation, we have the textual data (Javadoc comment, commit message, and issue tracker data) that describes the deprecated feature at our disposal. For this reason, we reduce our task to a text classification problem [137], which we tackle adapting the widespread *Vector Space Model* (VSM) [149]. VSM considers each document (*i.e.*, the deprecated feature and all the relevant text) as a vector of identifiers (*i.e.*, in our case, all the terms that appear in the whole set of available texts in our dataset) whose value is determined by the normalized number of occurrences of each identifier in the document (*e.g.*, 0 if the term never occurs). The identifiers given as output from VSM represent the features for the machine learner and the normalized word counts are the corresponding values.

To determine the terms to consider for VSM, we create a vocabulary by tokenizing each textual resource. We split tokens on whitespace, special characters, and punctuation; moreover, we split variable names that are CamelCased into individual entities. Finally, we do not alter Javadoc tags such as '@deprecated' and '@link'.

Dataset and Evaluation. To train and test the performance of the proposed machine learning approach, we use the dataset produced in RQ1 and RQ2; then we mainly adopt *n-Fold Cross Validation* [138]. This strategy randomly partitions (using stratified sampling to maintain the proportion of classes) the data into *n* folds of equal size, then *n-1* folds are used as training and the last as testing. The process is repeated *n* times, using each time a different fold as a test set. The performance of the experimented models is computed using widespread classification metrics such as precision and recall; in our chapter, for space reasons, we report the *percentage of correctly classified instances*, while the full results are available in the accompanying replication package [150].

6.2.5 Threats to validity

Construct validity. In the manual analysis of the rationale behind deprecating specific features, we may have misclassified or missed out on certain motivations behind deprecation. We ensure the accuracy of our classification by having three authors simultaneously manually analyze all the samples in our dataset and create an initial categorization of the rationale, followed by another author repeating the manual classification process to ensure accuracy. To ensure that we uncover most motivations behind deprecation, we limit ourselves to 4 mainstream Java APIs that pertain to different domains and have different developers and characteristics.

Generalizability. Having focused only on the Java ecosystem, the rationales that we have uncovered may apply only to the Java-based APIs and not to APIs in other languages. We mitigate this by trying to ensure that the rationale we discover is not Java specific, rather as abstract as possible. Furthermore, Java is the most popular language with a deprecation mechanism and other object-oriented languages share similar development practices.

6.3 RQ1 results: Diversity of Reasons

We describe each category in the taxonomy that resulted from our analysis, reporting examples from our dataset.

BC. **Avoid bad coding practices**

Example Javadoc

```
/**
  * Allow injection of the dialect to use.
  * @deprecated The intention is that Dialect should be required to be
  * specified up-front and it would then get ctor injected.
  * @param dialect The dialect
  */
  @Deprecated public void setDialect(Dialect dialect)
```

One of the principal goals of APIs is to provide a set of features to a consumer that can be integrated without introducing issues or bad coding practices in the consumers' code base. There are certain cases where the API does not always achieve this goal. In the example above, it is preferred that the `Dialect` should be specified up front as that would allow the `Dialect` object to be injected directly in the constructor. Using a setter method of a class implicitly means that the dependency is optional, constructor injection instead is used when the class cannot function without the dependency.

DP. **Design pattern**

Example Javadoc

```
/**
  * Creates a mock object that extends the given class, order checking is
  * enabled by default.
  * @param < T > the class that the mock object should extend.
  * @param name the name of the mock object.
  * @param toMock the class that the mock object should extend.
  * @param constructorArgs constructor and parameters used to instantiate the
  * mock.
  * @param mockedMethods methods that will be mocked, other methods will
  * behave normally
  * @return the mock object.
  * @deprecated Use {@link #createMockBuilder(Class)} instead
  */
  @Deprecated public <T>T createStrictMock(final String name,final Class<T>
    toMock,final ConstructorArgs constructorArgs,final Method...mockedMethods)
```

Partial mocking is a very nice feature, but having to use the reflection API directly to get the constructor and methods is less than ideal, so we created a MockBuilder which we've been using for this.

Example commit message

We find cases in which API producers deprecate the old feature and slowly phase them out as consumers are encouraged to use the new version of the functionality that makes use of a design pattern. In the example above, the project moved from accessing a feature through reflection to using the design pattern named Builder.

DU. **Dissuade usage**

Example Javadoc

```
/**
  * Not supported. Use {@link
  * ImmutableSortedMultiset#toImmutableSortedMultiset} instead.
  * This method exists only to hide {@link
  * ImmutableMultiset#toImmutableMultiset} from consumers of {@code
  * ImmutableSortedMultiset}.
  * @throws UnsupportedOperationException always
  * @deprecated Use {@link ImmutableSortedMultiset#toImmutableSortedMultiset}.
  * @since 21.0
  */
 @Deprecated public static <E>Collector<E,?,ImmutableMultiset<E>>
   toImmutableMultiset(){
  throw new UnsupportedOperationException();
 }
```

We find cases where API producers implement an interface in a class, without implementing all of its methods. The un-implemented methods are marked as deprecated so that the consumer is given an indication (as a compiler warning) that this feature should not be used. In the example above, the Javadoc also recommends a replacement.

FD. **Functional defects**

Example Javadoc

```
/**
 * Compare 2 arrays only at the first level
 * @deprecated Use {@link java.util.Arrays#equals(char[],char[])} instead
 */
 @Deprecated public static boolean isEquals(char[] o1, char[] o2)
```

org.hibernate.internal.util.compare.EqualsHelper doesn't consider arrays when comparing objects for equality. Since EqualsHelper is used in many places, such as dirty checking, this problem results in unexpected behavior, such as array type fields always being considered dirty.

Issues related to this problem include:
HHH-4110 CLOSED
HHH-2482 CLOSED
HHH-7810 OPEN
HHH-3009 CLOSED
HHH-7496 CLOSED (probably)

Example issue tracker message

The introduction of flaws in API features is inevitable. We find that, at times, API producers deprecate features with defects instead of removing them. We see an example in the deprecated method above where the implementation of the equals method does not consider arrays when comparing objects for equality, which in turn causes issues in other parts of the API, as seen in the extract from the related issue report.

ME. **Merged to existing method**

Example Javadoc

```
/**
  * Returns an equivalence that delegates to {@link Object#equals} and {@link
  * Object#hashCode}. {@link Equivalence#equivalent} returns {@code true} if
  * both values are null, or if neither value is null and
  * {@link Object#equals} returns {@code true}. {@link Equivalence#hash}
  * returns{@code 0} if passed a null value.
  * @deprecated use {@link Equivalences#equals}, which now has the null-aware
  * behavior
  */
  @Deprecated public static Equivalence<Object> nullAwareEquals()
```

We found instances in which an API provides two features achieving the same end goal, but one has more nuance associated with it and performs extra checks. Over time, the API producer decides to combine these different checks into the same feature, thus resulting in the other being deprecated. In the above example, the `equals` method in the `Equivalences` class now performs a null check, thus rendering the `nullAwareEquals` obsolete.

NF. **New feature introduced**

Example Javadoc

```
/**
  * @deprecated Use {@link ValueGraph#equals(Object)} instead. This method
  * will be removed in late 2017.
  */
  @Deprecated public static boolean equivalent(@Nullable ValueGraph<?,?>
    graphA,@Nullable ValueGraph<?,?> graphB)
```

Now that ValueGraph no longer extends Graph, change all the common.graph interfaces to handle equals()/hashCode() "normally". Deprecate Graphs.equivalent().

Example commit message

Sometimes, when API producers introduce a new feature, the design of the project is also changed. In these cases, the producers deprecate the older, superseded features.

ND. **No dependency support**

Example Javadoc

```
/**
  * Expect any boolean but captures it for later use.
  * @param captured Where the parameter is captured
  * @return 0
  * @deprecated Because of harder erasure enforcement, doesn't compile in
  * Java 7
  */
  @Deprecated public static boolean capture(final Capture<Boolean> captured)
```

Over time APIs upgrade the dependencies on which they depend. With these upgrades, specific features in the API can no longer be supported and need to be removed and replaced with modern functionality. In the cases we analyzed, upgrades in the Java version often cause the incompatibilities. In the example above we see that the `capture` method is not supported in Java 7 and becomes deprecated.

RD. **Redundant methods**

Example Javadoc

```
/**
  * @deprecated since 5.2, to be removed in 6.0 with no replacement.
  */
  @Deprecated public ModificationStore getStore()
```

From a discussion with Adam, there was supposed to be another value, DIFF that would store the diff of String-based value fields. It was never implemented and isn't … needed, so [it's] safe to deprecate and remove.

Example commit message

Redundant is code that is neither required nor essential and need not be executed. The negative consequence of redundant code is that it results in bloated source code and reduced maintainability. We found cases in which the API producers deprecated a feature when it is useless or no longer necessary.

RN. **Renaming of feature**

Example Javadoc

```
/**
  * Old name of {@link #getHost}.
  * @deprecated Use {@link #getHost()} instead. This method is scheduled for
  * removal in Guava 22.0.
  */
  @Deprecated public String getHostText()
```


The considered APIs have been developed over a long time by multiple developers, thus contain inconsistencies in the naming convention. These inconsistencies might have been introduced due to a lack of foresight or a change in the API's nomenclature convention. Just renaming a feature to adhere to new norms would break consumer code and hence would not be backward compatible. The existing name is kept in place. In fact, we notice in the cases that we manually analyze that the original name is intended to be left in the API indefinitely, *i.e.*, there are no plans to remove such features. However, API producers do deprecate the feature with the incorrect name and encourage consumers to adopt the new feature that adheres to the naming convention of the project.

SF. **Security flaws**

Example Javadoc

```
/**
  * Returns a hash function implementing the MD5 hash algorithm (128 hash
  * bits).
  * @deprecated If you must interoperate with a system that requires MD5, then
  * use this method, despite its deprecation. But if you can choose your hash
  * function, avoid MD5, which is neither fast nor secure. As of January 2017,
  * we suggest: For security: {@link Hashing#sha256} or a higher-level
  * API. For speed: {@link Hashing#goodFastHash}, though see its docs for
  * caveats.
  */
  @Deprecated public static HashFunction md5()
```

A security vulnerability might have been inadvertently introduced in a feature of an API at its inception or over time, thus requiring the immediate action of the API producers to address the issue. The producer deprecates the flawed feature and replaced it with one which does not suffer from the same flaw. In this way, the producer warned the consumer in the documentation that usage of such a feature is unsafe.

SC. **Separation of concerns**

Example Javadoc

```
/**
  * @deprecated Use {@link #setImplicitNamingStrategy} or {@link
  * #setPhysicalNamingStrategy} instead
  */
  @Deprecated public void setNamingStrategy(String namingStrategy)
```

In object-oriented programming, each class or module is supposed to have its responsibilities. Sometimes an API feature can do too many things simultaneously, *i.e.*, it has too many responsibilities. To fix this, we found cases in which the API producer decided to split a single feature into multiple ones, also deprecating the original feature and creating a transition guide.

TF. **Temporary feature**

Example Javadoc

```
/**
  * @deprecated (since 5.2.1), while actually added in 5.2.1, this was added
  * to cleanup the audit strategy interface temporarily.
  */
  @Deprecated public EnversService getEnversService()
```

An API producer might introduce a feature only for a temporary purpose to aid consumers in using certain functionality. Once, this is no longer needed, the temporary feature might be deprecated. We see that such temporary features are planned to be removed almost instantly from the API so that no consumer actually has the opportunity to use it in a future version.

The analysis of 374 deprecated features and over 1,100 accompanying documents yielded 12 rationales that API producers have used to deprecate a feature, thus showing a sizeable diversity of purposes.

6.4 RQ2 results: Frequency of reasons

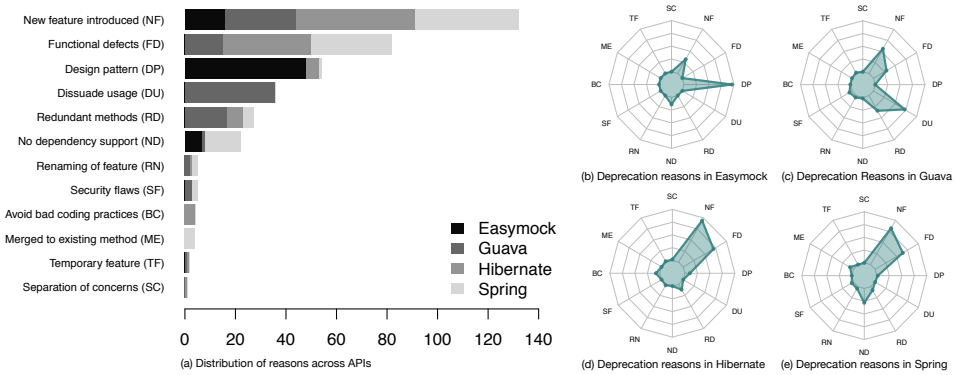


Figure 6.4: Frequencies of reasons for deprecating features, by API

After having categorized and described the *diversity* of reasons that led API producers to deprecate one of their API features, we now focus on determining how each of these reasons is prevalent in our dataset. Figure 6.4 reports the results by reasons in overall decreasing frequency (left-hand side) and by API (right-hand side). Overall, introducing a new feature (NF), the presence of functional defects (FD), the replacement with a design pattern (DP) account for the majority of reasons (268 cases out of 374 or 72%). The most frequent reason (NF) is the only one that appears in all the systems in our dataset, while the others have a more project-specific prevalence. We now describe the results by API.

Easymock. The most frequent reason for deprecation in Easymock is the implementation of a new design pattern that required a change in the interface. There are also 7 cases where the feature in their API was not supported by a newer version of Java. In 16 cases a better implementation of the same feature was superseding the existing one. Overall, we see that in Easymock most deprecations are not of a grave nature: A consumer could safely continue using a deprecated feature from this API.

Guava. Developers in Guava predominantly introduce new features to replace existing ones, hence use the deprecation mechanism. There are also some cases where there were functional defects in the feature and some instances of having security flaws. There is one feature that has been deprecated due to it being introduced as a temporary feature. In 36 cases, Guava deprecates a feature to dissuade usage of it due to an incomplete implementation of an interface; this is a case of misuse of the deprecation mechanism. Here the feature was not deprecated due to it being superseded by a new feature or because the feature had become obsolete. In most cases, it may be safe to continue using a deprecated feature from Guava, but the reasons for deprecation are diverse and one needs to verify them first.

Hibernate. In the case of Hibernate, developers have deprecated almost 50% of the deprecated functions due to the presence of functional issues in the features. This means that when Hibernate deprecates a feature it is usually to fix major issues in the API. In the other cases, the features are deprecated due to new functionality being introduced, because of redundancy, or because it encourages bad coding practices.

Spring. In 41 cases Spring has replaced an existing feature with a better implementation. Although Spring is an old and well-tested API, there have been 32 functional flaws to fix and 2 security issues as well. Spring also has deprecated features due to incompatibilities with newer versions of Java. Overall, there seems to be some danger to using deprecated features from Spring and, in many cases, a consumer needs to replace a deprecated feature with its successor.

Introducing a new feature, the presence of a functional defect, and change of interfaces are the most frequent reasons for deprecating an API. However, only the first is shared across all projects.

6.5 RQ3 results: Automatic reason classification

Our RQ3 investigates to what extent a machine learning approach can automatically classify the reason for deprecation.

6.5.1 Methodological details

Although we always use VSM (Section 6.2.4), we progressively add more information sources to evaluate their effect on the classification. In the first stage, we only use tokens from Javadoc comments to classify the rationale, in the second we add commit messages, in the third, we add issue tracker data.

We evaluate three training/testing conditions: (1) 10-fold cross validation within the same API (*i.e.*, we evaluate each system separately), (2) overall 10-fold cross validation (*i.e.*, we merge all the instances in a single dataset), (3) cross-project validation (*i.e.*, we use the instances from three systems for training and test the resulting model on the last system; we rotate the test system each time).

6.5.2 Results

Since random forest always outperformed the naive Bayes multinomial classifier, we only report results for the former.

Within system validation. The first four groups in Table 6.2 report the results of the classifier when tested within the same system using 10-fold cross-validation. For Guava, the classifier performs well on just Javadoc data. For Easymock, the classifier achieves 100% correct instances with just the Javadoc. This result is probably due to most deprecations being caused by the refactoring of a feature to use design patterns: Since Easymock always uses the same pattern (builder pattern), the terminology is the same. With more data, the accuracy of the classification decreases, probably due to added noise. For both Spring and Hibernate the classification accuracy is below 65% with only Javadoc data. With the addition of commit message data, the accuracy increases. In the case of Spring, when we add issue tracker data, the accuracy increases to 88%. However, in the case of Hibernate, the accuracy suffers slightly with issue tracker data, again probably due to added noise.

Mixed-system validation. We combine the data for all the APIs and treat this combined dataset as our singular vocabulary; Table 6.2 in the group named 'All' shows the results. With only Javadoc data, the classifier correctly classifies almost 76% of the cases. This might be due to all the Easymock instances, which the algorithm can easily classify with only Javadoc data. Adding commit message data improves the classification by almost 10%. Issue tracker data yields a minor improvement.

Cross-project validation. Given that the results are promising at project level, we tried to perform cross-project validation. However, results were consistently lower than 30% in the number of correctly classified instances. For example, in the case of Easymock, the method only reaches 24%. This result seems to indicate that there is project specific terminology that helps the machine learner to discern the different reasons. We also conducted cross-project classification for only one category (binary classification). We choose the addition of a new feature (NF) as our test category since we expect project-specific terms to be minimal. Although the number of correctly classified instances is 54%, this

Table 6.2: Classification results, using 10-fold cross validation within the same system and across all systems.

		% correct instances	K	weighted avg.	
				recall	ROC
Guava	**JD**	0.883	0.843	0.883	0.963
	JD+CM	0.893	0.855	0.893	0.956
	JD+CM+IT	0.903	0.868	0.903	0.963
Easymock	**JD**	1.000	1.000	1.000	1.000
	JD+CM	0.986	0.970	0.986	1.000
	JD+CM+IT	NA	NA	NA	NA
Hibernate	**JD**	0.610	0.334	0.610	0.777
	JD+CM	0.740	0.559	0.740	0.862
	JD+CM+IT	0.720	0.526	0.720	0.866
Spring	**JD**	0.640	0.477	0.640	0.794
	JD+CM	0.850	0.780	0.850	0.956
	JD+CM+IT	0.880	0.824	0.880	0.962
All	**JD**	0.759	0.686	0.759	0.915
	JD+CM	0.853	0.808	0.853	0.959
	JD+CM+IT	0.866	0.825	0.866	0.962

is still poor in comparison to project level classification. This result leads us to conclude that automated classification techniques work best at a project level due to project-specific terminology.

An automatic classification approach can correctly classify more than 85% of deprecation reasons in three systems and 74% in the fourth. However, to achieve these results, data from commit messages and issue reports is often necessary and the classifier must be trained with project-specific instances.

6.6 Discussion

We discuss how our results lead to implications for future research and recommendation for practitioners.

6.6.1 Unmet developers' communication needs

Programming languages provide API producers with deprecation mechanisms to allow them to communicate with the API consumers, in a way that is recognized and rendered distinctively by popular IDEs and compilers. In Java, by marking a feature as deprecated, a compiler warning is thrown and the IDEs render the element as struck-through.

Recently, the Java language designers stated that the deprecation mechanism has been used not only for communicating about obsolete features but also for alternative purposes, which they labeled as "misuses" [117].[4] Previous work indeed found one case where the API producer has used the deprecation mechanism for an unorthodox purpose [136]. In this case, the JUnit API marked beta features as deprecated to warn consumers about these features' beta nature, which may lead to unanticipated future changes.

In our study, we uncover two additional cases in which API producers use deprecation for unorthodox purposes: (1) to indicate a temporary feature that is in place until a permanent solution can be found (as done by Guava and Hibernate), and (2) to indicate that a class only implements part of an interface and the unimplemented methods are essentially just stubs (as done in a widespread manner by Guava).

This finding raises the broader question of why API producers use the deprecation mechanism for purposes besides marking out obsolete features. It is reasonable to think that these are cases of communication needs for an API producer that are not being met by the Java language specification. For example, while the Guava API producers do try also to throw an exception to prevent the usage of specific features, they use the deprecation mechanism to issue compiler warnings. As an additional example, XWiki has introduced a workaround [151] where the usage of an '@Unstable' annotation in combination with an Eclipse plugin issues a warning in the IDE about the nature of the used feature.

This finding leads us to question if Java should invest in introducing a generic warning mechanism as a more flexible communication mechanism that would not lead to misinterpretations. This mechanism would allow API producers to throw a compiler warning for purposes other than deprecation, thus possibly addressing the producers' unmet communication needs. In this vein, languages such as PHP and Ruby have already set a precedent, where deprecation mechanisms and warning mechanisms are simultaneously present. By performing a similar study to the one we present in this chapter in the API ecosystems of those two languages, researchers and the Java language designers can better understand whether there are benefits to having a generic warning mechanism and if indeed the cases of misuse would be minimized while fulfilling developers' communication needs.

6.6.2 Different evolution strategies

API evolution has been studied by researchers to understand *how* APIs evolve [26]. Researchers have also investigated the decision process behind evolving the API and introducing breaking changes in the API [4]. API evolution strategies are generally based on what features they change and how it affects the API consumer [14, 29, 135, 152, 153]. Significant work has also gone into alleviating the burden of dealing with API evolution [27, 28, 126, 127, 154].

[4] We consider these "misuses" from a more constructive perspective, that is, as evidence of developers' communication needs that are unmet by the current mechanisms and can be basis for future research and improvements.

In our study, we investigate the rationale used by API producers to evolve and render specific features as obsolete and introduce new features to replace them. We see that there are 12 reasons behind deprecating a feature. The frequency of usage of these rationales differs per API. For instance, we see in the case of Easymock most deprecations are due to the usage of design patterns, while for Spring, most changes are due to functional defects or newly introduced features.

We found initial evidence that by understanding the rationale behind the deprecation, we also better understand the evolution strategy adopted by an API and how it might affect a consumer. With Spring and Hibernate we see that a large number of deprecations are due to functional defect being present in the API. However, with Easymock and Guava other less important reasons such as redundancy of a method or refactoring to use a design pattern. Based on this, we can deduce that developers in Spring and Hibernate discover issues in features on a regular basis. On the other hand with Easymock and Guava, most of the changes are due to maintaining the API on a regular basis, without introducing several new features.

Further work can be conducted expanding on this line to see whether knowing the rationale behind evolution—thanks to the analysis of deprecation reasons—gives a better approximation of the evolution strategy of an API. This work would help informing tools to support practitioners keeping up with API evolution.

6.6.3 API documentation completeness

API documentation is vital in teaching consumers to adopt the API in a correct manner [32]. Incomplete documentation is a considerable obstacle to API consumers [155]. This is the primary reasons that API producers invest a lot of time in documenting their API in a correct and detailed manner [142].

Much work has gone into augmenting current API documentation to aid an API consumer and reduce the documentation burden on the API producer. Stylos *et al.* [156] have looked at augmenting API documentation by including API usage examples mined from open source repositories. Treude and Robillard [157] seek to improve API documentation with examples from community-driven documentation sources such as StackOverflow.

In the context of documentation of deprecated features, researchers have shown that recommending a replacement feature in the deprecation message is helpful to API consumers [15, 143]. In addition to that, informing the consumer about the rationale behind deprecation and the version in which the deprecated feature will be removed performs a vital role in the consumer's decision to react to deprecation [136].

We found overwhelming evidence that the Javadoc for deprecated features seldom mentions the reasons behind deprecating features. In fact, to uncover the rationale behind deprecation it is necessary to refer to the commit messages and to the issue tracker data. Conducting such a thorough search is error-prone and time consuming, thus impractical in a real-world scenario.

We show the different sources needed to infer the rationale behind deprecation. Research effort can be invested to be able to effectively retrieve the traceability links across all the different sources together, such that the justification for deprecation is evident and existing documentation enhanced.

6.6.4 Automating the classification of rationale

We investigate how accurately the rationale behind deprecation of a feature can be classified based on its Javadoc, the commit message that deprecates it and the issue tracker post that discusses its deprecation (if present). At a project level, we see that having this information allows us to classify the rationale behind deprecation accurately.

The automated classification relies heavily on project-specific terminology as is evidenced by the fact that cross-project classification yields poor results. Not only does project-specific terminology play a role, but also the specificity of technical terms for each rationale play a role too. For example, in the case of Easymock several deprecations took place due to the refactoring of a feature to use the builder pattern. In this case, the word "builder" is a specific case of refactoring to use a design pattern. If the automated classifier learns on other instances of refactoring to design pattern, such as the one from the Spring API, we see that it decreases the accuracy for the cases in Easymock.

Despite the specific circumstances under which an automated approach can work, the classifier performs promisingly at a project level. API producers can run such an approach on their documentation to automatically categorize the rationale of a deprecated feature and use this categorization to augment their existing documentation.

We see that there is a need for more than just the Javadoc to classify the rationale of a deprecated feature. This result puts into focus the need for the creation of a complete information pipeline that stitches together the Javadoc, commit message, and issues regarding a deprecated feature. This approach would go a long way in aiding automating the classification of the rationale of a deprecated feature.

Further research needs to be conducted in the area of automating the classification of the rationale behind the deprecated feature. We show that a machine learning approach can work and provide an initial baseline for future comparison. More research is needed to investigate whether and how the poor performances in cross-project classification can be tackled, for example by considering further features and other classification techniques such as deep learning.

6.7 Related work

We describe work in the related areas of API documentation needs and improving documentation.

Studies on API evolution. Robbes *et al.* [3] analyze the impact of deprecation of an API feature on the SmallTalk ecosystem. They find that while the number of API consumers affected is high, minimal reaction to deprecation takes place. Sawant *et al.* [13, 14] mine 25,357 Java-based API consumers from GitHub and a further 150,326 Maven central based JAR files to see how many consumers are affected by deprecation and their reactions. They observed that over 10% of deprecated methods affect consumers, but consumers do not react. In contrast to this, we look at the reasons behind deprecation of the API from the API producer perspective.

Hou and Yao [130] investigate the intent behind API evolution by studying release notes. They found that API features were deprecated due to conformance to API naming conventions, naming improvements, simplification of the API and replacement of functionality. Sawant *et al.* [136] interview 17 API producers as to why they deprecate features

and catalog seven reasons behind deprecation. In this study, we analyze documentation at a fine-grain (Javadoc, issue tracker and commit messages) level to understand the reason behind deprecation. This analysis leads us to uncover 12 rationales behind deprecation. Moreover, we evaluate how well an automated technique can classify the reason for a deprecation.

Studies on API documentation needs. Robillard and Deline show that API documentation is a vital resource for developers who want to adopt a new API [32]. Myers and Stylos concur with this view and provide evidence that API documentation plays a significant role in making it usable [158]. Maalej and Robillard show that API reference documentation should complement the API by providing information that is not obvious from the API syntax [159].

Uddin and Robillard uncover that consumers find it much harder to understand the API producer's intentions due to inadequate documentation [155]. Monperrus *et al.* analyzed API Javadoc to see what was being talked about and in what cases there was an information shortfall [31]. They state that a deprecation tag in Javadoc with no rationale or conditions is an anti-pattern. Brito *et al.* find that in over 60% of the cases the replacement for a deprecated feature is specified [15, 143]. During Sawant *et al.*'s investigation into the deprecation mechanism, they found that consumers miss is the rationale behind the deprecation itself [136].

In this study, we find that deprecated API documentation often does not document the reason behind the deprecation, despite this being important for consumers.

Studies on improving API documentation. One of the primary challenges with producing high-quality API documentation is the large amount of time and effort that goes into creating the documentation [142].

Dekel and Herbsleb improve API documentation by highlighting specific directives that are present in the documentation so that the consumer is made explicitly aware of particular conditions that he has to be aware of [160]. Researchers have investigated ways to improve existing documentation by augmenting it with examples mined, for instance, from source code repositories [156, 161–163] and StackOverflow [157].

Subramanian *et al.* create a tool to called Baker that links source code examples to API documentation [164]. Baker can do this in a real-time manner thus always keeping the usage examples in the API documentation fresh. Dagenais and Robillard look to recover the traceability links between APIs and their learning resources [165]. This study aims to produce a comprehensive set of documentation that is mined from a variety of sources such as developer blogs, StackOverflow and mailing lists, all in one place.

We found that the rationale behind deprecation can be automatically classified but, more than one source of information is required. This understanding can aid in providing consumers with the rationale behind deprecation.

6.8 Conclusion

We have presented an explorative study we conducted to uncover the rationale behind the deprecation of an API feature. We manually analyzed over 1,100 document artifacts relating to 374 from 4 mainstream Java APIs deprecated features. This analysis led to the creation of a taxonomy comprising 12 reasons for deprecation. We observe that there

are several cases of deprecation being used in an unexpected, unorthodox manner, thus hinting at currently unmet communication needs. Finally, we found that an automated approach to classifying deprecation reasons can reach promising accuracy, but only when it is trained on instances from the same project. We discussed the results and their implications concerning unmet communication needs, API evolution strategies, API documentation completeness, and future work.

7

How to improve the deprecation mechanism

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" [10]. The deprecation mechanism is a commonly used practice [15].

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 [117]. 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 [3, 166] despite API producers taking great care in documenting the deprecation and the changes involved [15].

In this chapter, 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 [167], 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.

7.1 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" [10].

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 7.1).

Recently, there has been a proposal to change the deprecation mechanism in Java (JEP 277 [117]). 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 [168]. 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" [117]. 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" [117]. 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 [117]:

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.

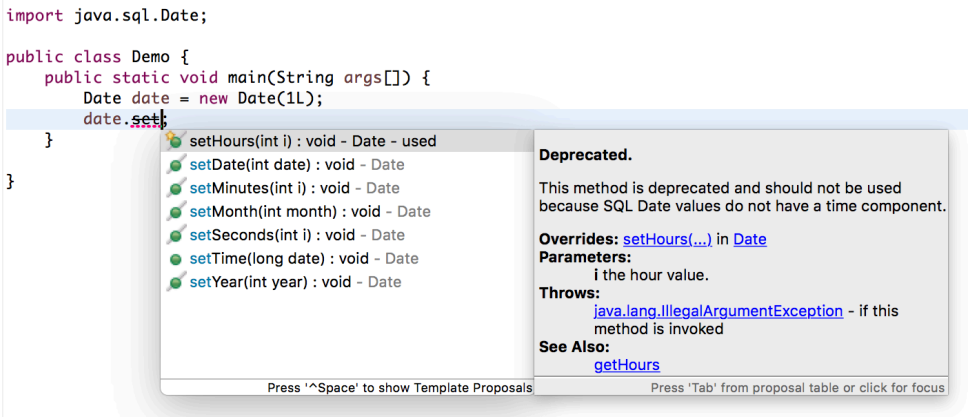


Figure 7.1: Example of deprecation warning in the IDE

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.

7.2 Methodology

The overall goal of this chapter 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 7.5.

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.

7.2.1 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 [169] 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 [170], 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

Table 7.1: 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

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 [171].

7.2.2 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.* [72], 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 chapter). The background of the participants is summarized in Table 7.1.

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.

7.2.3 Limitations

One of the risks of a qualitative study is that determining the validity of our findings is a difficult undertaking [172]. 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 [173]. 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 [174] (a phenomenon where one question could provide context for the next one) may lead our interviewees and survey respondents to specific answers. One approach 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 [175] (*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.

7.3 Results

7.3.1 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

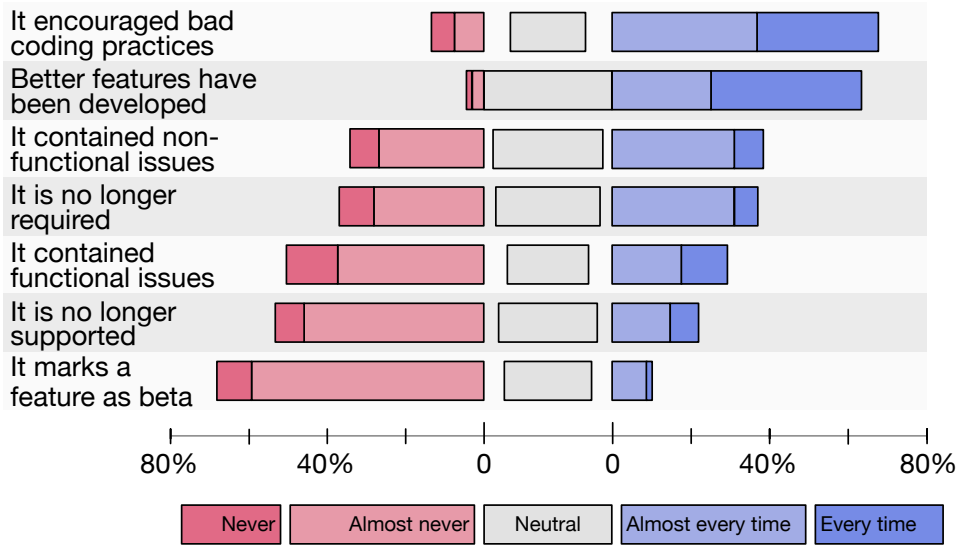


Figure 7.2: Motivations to deprecate a feature according to API producers, by decreasing frequency

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 7.2 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 [3, 166] underlines that messages sent through this channel (regardless of how accurately they are written [15]) 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.

7.3.2 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) [168].

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.

7.3.3 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.

7.3.4 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 [176].

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.* [166] 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 7.3.

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 7.3 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.

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

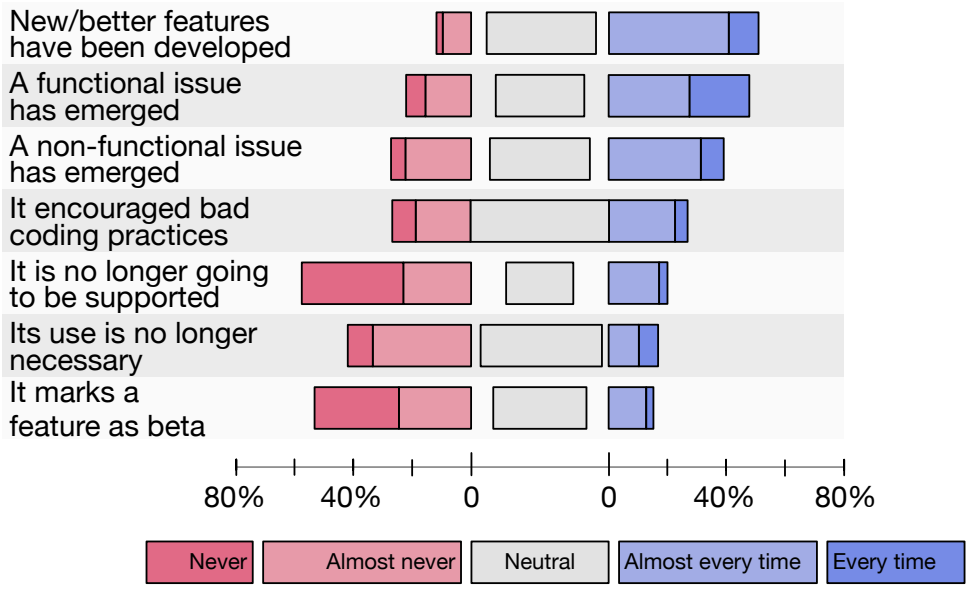


Figure 7.3: Motivations to react to deprecation according to API consumers' experience, by decreasing frequency

7.4 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.

7.4.1 A communication mechanism

Previous work by Sawant *et al.* [166] 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.* [3] 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 7.3.2. 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 [167]. 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 [167] 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 [15].

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" [167]. 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 7.3.3), 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# [177] 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.

7.4.2 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.

7.4.3 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 [27] 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 [28] 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 [128] 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 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.

7.5 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.

7.5.1 Desirability among the Java community

We present the results of our survey in Figure 7.4. 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

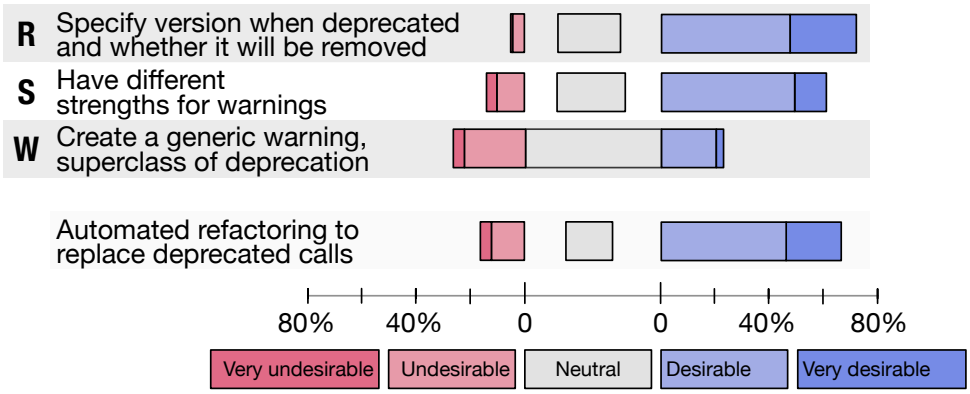


Figure 7.4: Respondent's perspective on the enhancements to the Java deprecation mechanism

reported to use the deprecation mechanism for purposes different than the intended ones (Section 7.3.4). 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" [178]; 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 [178].

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.

7

7.5.2 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 [179] and the specification lead group of JSR-308 [180].

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 con-

sumers 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 [179]. This would then entail creating a generic warning mechanism that could be reused for each specific warning.

7.6 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**.

Table 7.2: 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	✓				

We select the first 15 languages from the Tiobe language popularity index [181] 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 7.2.

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.

7.7 Related work

Previous studies have focused on deprecation from the client perspective. Particularly Robbes *et al.* [3] 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.* [166] 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.* [19, 129] investigated when API developers deprecate features. They found that APIs introduced deprecated features in major and minor releases in equal measure. Brito *et al.* [15] 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 [26] studied and 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 [154] 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.* [135] show that breaking changes in Android APIs have an impact on the rating of an app. Espinha *et al.* [133] looked at the impact of breaking changes introduced in popular web APIs such as Twitter, Facebook etc. Xavier *et al.* [153] 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.* [4] 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 [130] 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.

7.8 Conclusion

Java's deprecation mechanism is planned for change due to issues perceived by the Java language designers [117]: 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 chapter 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.

8

Conclusion

8.1 Research Questions Revisited

In this section revisit the research questions defined in Chapter 1.

Research Question 1. *How are API consumers affected by deprecation of an API element?*

This research question exposes that, for some APIs, a large proportion of consumers is affected by deprecation of an API feature. We determine this in two different studies, where the first used data obtained from mining type-checked API usage from 25,000 Java-based GitHub projects that are consumers of five mainstream Java APIs: Guava, Guice, Spring, Hibernate and Mockito and the second that mined data from 250,000 projects that are consumers of the top 50 most popular Java APIs.

We observe that, in the case of Guava, over 30% of the consumers are affected by deprecation. However, in the case of Spring, where we observe that only 35 out of 15,000 projects analyzed have been affected by deprecation. We also fast-forwarded the consumers that use an API to use the latest version of the API, and we see that in most cases (except Spring) a larger proportion of consumers is affected by deprecation. In the case of Guava there was a consumer that would have to replace 8,568 invocations and in the case of Hibernate one consumer would have to replace 17,471 invocations. This provides evidence that consumers can be adversely affected by deprecation on a large scale and the maintenance effort required to react can be large.

In the case of the 50 APIs that we analyzed, we found that 20 never affect their consumers. A further 18 of these only affect less than 2% of their consumers. The remaining 12 APIs have a significant impact on their consumers. APIs that fall into this category include also very popular APIs.

For those APIs that do not affect their consumers, we try to ascertain as to what deprecation policy adopted by APIs rarely affects their consumers. We find that APIs that are not active with releasing frequently and those that do not deprecate too many features (< 9% of the features) but preferred to directly introduce breaking changes when evolving, rarely affect their consumers with deprecation.

In the other cases where a consumer is affected by deprecation, it can be because the API deprecates more than 9% of its features and rarely removes them, thus actually encouraging the adoption of deprecated features. Some APIs introduce features in an already deprecated state, this usually to indicate that they are beta features or unimplemented stubs.

We see from this research question that API evolution (of the type deprecation) affects API consumers. We see that the API deprecation policy can have a direct impact on the scale at which the consumers are affected by deprecation.

Research Question 2. *How and why do API consumers (not) react to deprecation of an API element?*

With this research question, we sought to understand how and why API consumers react to deprecated API elements. If a consumer chooses not to react we also investigate why. To answer this question we adopt a mixed-method approach that involves: manual analysis, large scale analysis of API usage on GitHub, and surveys with API consumers.

We manually analyzed 380 instances of usage of deprecated features in APIs. This analysis yields seven different ways in which consumers can react to deprecated features. Some examples of reaction patterns are: replacing the deprecated feature with is recommended replacement, replacing the deprecated feature with a replacement from the JDK, rolling back the version of the API being used and not reacting at all. After determining these seven reaction patterns, we mined API usage data from GitHub for 50 APIs to see how frequently each of these reaction patterns took place in this ecosystem.

On analyzing data collected from over 250,000 Java project on GitHub, we found that not reacting to deprecation is the most popular way to deal with deprecation. In 146,000 cases where there is an invocation made to a deprecated entity, the consumer did not react at all. This is a startling finding as the expectation would be that consumer would try to replace the deprecated call with its recommended replacement. In fact, in only 36 cases did consumers use a feature from the same API. The second most popular way to react to deprecation was to replace the library being used with another that provides similar functionality.

In one of our previous studies, we had observed that in the cases that a reaction was to take place, it often took the consumers a median time of 200 days to perform the reaction. While this might explain why we see such a large scale of non-reactions, it does not explain it entirely. Especially because we strive to mine the entire history of the consumer, right from its beginning to determine how it is affected by deprecation and how it reacts. This history spans from the year 2008, thus making it less than likely that by waiting for a further year we would observe more reactions.

We survey consumers to ask them as to what prevents them from updating the version of the dependency that they use and what prevents them from reacting to deprecation. Consumers indicate that the fact that feature that they use is deprecated in a certain version of the API does not stop them from upgrading, it is often the cost of updating a dependency that acts as a barrier. Surprisingly, a large proportion of our survey respondents claim to react to deprecation by replacing the deprecated call with its recommended replacement. This is contradictory to our data, however, to a certain extent, it can be ex-

plained by social desirability bias. Alternatively, the development practices adhered to by our survey respondents are significantly different from those that we observe in the data.

We ask those consumers who indicate that they seldom or never react to deprecation as to why this is the case. The most popular response to this is that they can often not find a suitable replacement. This may hint at a communication issue from the producer's side. The second most popular reason is that the cost associated with reacting to deprecation is so high that the consumer does not want to make this effort.

From this research question, we understand that consumers choose not to react to the deprecation of an API element. The replacement feature provided by the API is often not suitable, indicating that the API producers need to do a better job at documenting the new features that they introduce. Additionally, the cost of reacting is also a major stumbling block and consumers need to be provided with some aid when it comes to dealing with API deprecation or evolution.

Research Question 3. *How do API producers support API consumers when reacting to deprecation?*

We seek to understand what kind of support API producers provide their consumers when it comes to enabling them to react to deprecation. We answer this question by performing a manual analysis of API deprecated features and the various sources that document the deprecation.

When an API deprecates a feature, the API producers typically inform the API consumers as to what feature replaces the one that was just deprecated [31, 98]. While this information can aid a consumer in reacting to deprecation, it need not always be sufficient as found in Chapter 7 of this thesis. Consumers usually need a good reason to react to deprecation *i.e.*, the reason behind the deprecation has to be severe enough for them to react.

Uncovering the rationale behind the deprecation of a feature is not as trivial as it may seem. During our investigation, we found that the reason is seldom mentioned in the Javadoc itself. A much deeper search is needed to uncover the API producer's rationale. We had to look at the commit that deprecated the feature to understand from the commit message as to what reason the producers had given for the change that was committed to the repository. In the cases where the commit message was insufficient, we looked at the issue tracker of the API to understand whether the API producers had discussed the deprecation of the feature. We performed this manual analysis on 380 instances of deprecated features belonging to the Spring, Guava, Hibernate, Guice and Mockito APIs.

Our investigation revealed ten reasons behind the deprecation of a feature. Some of the reasons include: the feature was obsolete and superseded by a newer feature, the feature had a performance issue, the feature had a security flaw or the feature was deprecated as the API wanted to introduce the usage of a design pattern. We also found three additional reasons to deprecate a feature that are not considered to be typical: the feature is temporary, the feature is in beta mode, and the feature is an unimplemented stub feature.

Once we had uncovered all the reasons behind deprecation, we benchmarked them across the APIs to determine the most popular reasons. We found that introducing a new feature was the most popular reason followed by functional defects in the code. This is important, as the presence of functional defects in the code is pertinent information

that the consumers need as in such cases they would deem it necessary to react to the deprecation. The absence of this information in the Javadoc is a barrier to the consumers from making an informed decision.

This shows us that API producers have a diverse set of reasons behind deprecating a feature. However, they do not communicate this reason with consumers. This points to a need for API producers to better document their deprecations in the API.

Research Question 4. *Can language designers improve the deprecation mechanism for both API producers and consumers?*

The deprecation mechanism is a language feature and its implementation is down to the language designers that maintain it. Deprecation acts as a communication medium, it is what allows API producers to directly inform API consumers that the API is going to change.

We started by interviewing 17 API producers from both open-source (projects such as JUnit, Spring, Mockito, and XWiki) and industry (companies such as HP, ING Bank, and ABN Amro) to understand how they perceive the deprecation mechanism. They indicated that in its current form in Java, the deprecation mechanism met most of their needs, in fact, they all perceive it as *the only way in which they can communicate with the API consumer through code.* However, they would have preferred if they could indicate more explicitly that in certain cases it was imperative that the consumer would change the call made to the deprecated entity.

Our interviews with the producers uncovered that they were at times forced to use the deprecation mechanism in a way that it was not designed *i.e.*, they would mark features that are not deprecated as it was the only way in which a compiler warning could be thrown in Java. This meant that they added to the confusion surrounding the deprecation mechanism as mentioned by Stuart Marks [117] (the lead Java language designer on deprecation). This finding is similar to what we found in Chapter 6 when manually analyzing the documentation of deprecated API elements.

We surveyed 170 API consumers to understand how they perceive the deprecation mechanism as well. They supported the changes that the Java language has implemented in Java 9, however, they felt that these changes did not go far enough. Just like the producers, they too demanded that the severity of a deprecation should be indicated by the deprecation mechanism. Furthermore, they would like the API producer to provide a concrete timeline for the removal of the deprecated element. In the current state (even with the Java 9 enhancements) the consumers have no idea as to when exactly a deprecated element will be removed.

From the producer and consumer perspectives, we conclude that certain changes are needed to the Java deprecation mechanism. These changes are: (1) introducing a way to indicate the severity of a deprecation, (2) indicating a timeline for removal of the deprecated feature and, (3) creating a generic warning mechanism that would reduce the chance that the deprecation mechanism would be misused. We took these proposed enhancements to two prominent (including the lead language designer on deprecation) Java language designers to understand whether they would support such changes. They were behind the first two changes, however, felt that the warning mechanism was too extreme a change in Java and other alternatives should be explored first. We also asked API consumers and

producers alike in a survey as to whether they would support these changes, they too supported the first two changes and not the last one.

Our findings and proposed enhancements are not limited to deprecation in Java only. Other languages such as Kotlin, Smalltalk, and Ruby could all benefit from making these changes. For API consumers to be given a more complete message through the deprecation mechanism the communication medium must be as unambiguous as possible.

8.2 Implications

We discuss the implications we can derive from the work done in this thesis.

8.2.1 Better support for API consumers

By studying deprecation and its impact, we show that API evolution affects a significant proportion of consumers. We observe that the majority of these consumers choose not to react to the deprecation of the API element. Our investigation shows that the primary reason for this is the cost involved in actually reacting.

Previous research has attempted to reduce the cost involved with consumers reacting to API evolution. Henkel and Diwan created a tool called CatchUp! [27] that replays the refactorings made by API producers to adapt to an API change, directly in the consumer code. Xing and Stroulia [28] created a tool called DiffCatchUp that recommends alternative features from the API that a consumer can use based on how the API adapts to a change. Dagenais and Robillard created a tool called SemDiff that recommends changes to the consumer code for methods deleted by the API. Schrafer *et al.* [90] mine framework change rules from consumer code that has already adapted to the API change and propose the same to the consumer affected adversely by an API change. Kapur *et al.* [126] created a tool called Trident that allows API consumers to automatically refactor their code. Savga and Rudolf [127] developed a tool called Comeback! that aids consumers in upgrading from one version of the API to a new one. Perkins *et al.* [128] created a tool that specifically targets the case of deprecated features in the API. Their approach involved replacing a deprecated call with the actual method body of the deprecated feature, thus eliminating the deprecated call from the consumer codebase.

All these aforementioned approached suffer from one or more of the following pitfalls: (1) as shown in this thesis, consumers typically do not upgrade the version of the dependency that they use, which implies that there is very little transition data available, (2) the available data is generally in a limited scope and, (3) the suggestions made by such tools are typically not semantic aware, thus their suggestions are not always applicable to all consumers affected by the API evolution.

These facts point to a need for a semantic-aware tool able to efficiently port consumer code from an older version of the API to a newer version with minimal effort. While in this thesis we talk specifically about the deprecation context, this can be generalized to most API evolution cases. More modern approaches to mine API usage semantics such as that of Gu *et al.* [182] have the potential to solve the aforementioned issues.

As a direct consequence of such a tool, the cost of dealing with API evolution would be reduced, thus allowing a larger number of API consumers to update the version of the API that they use. This, in turn, would lead to the overall lessening of the technical debt that is

prevalent in most software systems [183] and also reduces the overall cost associated with the development of software. These reasons motivate the need for further research work and developmental work to be invested in making such a tool viable in the near future.

8.2.2 Language features impact communication

The deprecation mechanism is provided by many languages. It acts as a one-directional medium of communication between the API producers and the consumers. A deprecated element conveys that the element is obsolete and going to be removed in a future version of the API simultaneously the consumer is made aware of the replacement feature that has been introduced and that should ideally be adopted. It is pivotal that the deprecation mechanism allows for unambiguous communication between the API producer and consumer.

In this thesis, we study the Java language deprecation mechanism to see whether all needs are fulfilled. While the thesis is primarily Java-based, the needs that a useful deprecation mechanism should fulfill are language agnostic. We observe that the deprecation feature in Java would benefit from giving producers the ability to indicate the severity of a deprecated feature and the timeline within which it would be removed. Aside from changing the deprecation mechanism, Java could also provide the API producers with alternative ways in which compiler warnings could be thrown, thus providing producers with a more fine-grained ability to communicate with the consumers.

We studied other languages and their deprecation mechanisms to see how they conform to our suggested improvements. We observed that none of the languages conform to all of the features we proposed in this thesis: There is a scope for the improvement of the deprecation mechanism in most languages.

Aside from deprecation, a language feature can have a profound impact on how producers can communicate with consumers. One such example is the introduction of the `@Nullable` annotation that allowed the API producer to indicate that a return value or parameter can be null, thus communicating clearly with the consumer as to what to expect from the API element.

Such fine-grained annotations can help make an API more usable as it allows the producer to convey more information to the consumer. It remains an open challenge to uncover what other annotations would serve a purpose in this context. Currently, API producers have introduced ad-hoc annotations such as `@Beta` (introduced by both Guava, XWiki, and JUnit) to convey more information to the consumer.

Recently, the Java language designers have realized that changes are necessary to improve the communication between the producer and the consumer. They have changed the deprecation mechanism in Java 9, introducing a way for API producers to indicate that certain deprecated features are scheduled for removal in a future release. They also allow API producers to mention in which version the feature was deprecated, thus giving the consumer an indication of the period that the feature that they want to use are using is deprecated. These changes are an improvement over the original deprecation mechanism that has been present since Java 5. However, these changes do not go far enough.

In the future, we hope to implement the changes that have been proposed in this thesis and evaluate the utility of our proposal in the real world. It is also our hope that other languages adopt these changes if their utility can be definitively proven.

8.2.3 Laws of API evolution

Revisiting the introduction, we see that the phenomenon of software evolution has been studied in detail over the years. The seminal work in this field is that of Lehman [5] who defined the eight laws of software evolution. These laws describe the balance between development effort of new features in a system and factors that slow down the progress of the evolution of the system.

In this thesis, we have investigated the reasons behind APIs evolving. In chapter five, we uncover seven different evolution policies, which range from frequently introducing new API elements and removing existing ones to never changing the API. Both extremes are contradictory to the software evolution laws. For instance, the case where the API continually evolves would contradict the law *"Conservation of Familiarity"* as the API consumers would be less familiar with the new API elements and how to use them, however, in some cases the API has to change completely (*e.g.*, JUnit completely changed its interface from major version 4 to major version 5 to adapt to changes in the Java language). In the case that the API never evolves, it would contradict the laws *"Continuing Change"* and *"Continuing Growth"* as it would not be evolving to keep up with consumer needs, however, the lack of change of the API can actually be perceived as a positive by consumers as it would imply lower cost of maintenance of their code.

The original eight laws were written to apply to E-type systems (systems written to perform a real-world activity). However, in the case of APIs, these laws in the current form might not be entirely applicable. The balance between software development effort and evolution of the API should not take into account only the case of the API producers, but also the API consumers. Any change made to an API directly impacts its consumers, and they should be considered to be the primary stakeholders of the system.

Laws for API evolution should take into account the delicate balance between the API and the consumers that depend on it. Essentially, such laws need to multi-faceted and be designed to maintain the balance between development effort of new features, the evolution of the API and the impact of changes on the consumer. This can lead to the creation of a new set of laws: *"The Laws of API evolution"*.

8.3 Concluding remarks

In this thesis, we focus primarily on API evolution. We study this from the perspective of the API consumer to understand as to how they deal with API evolution and what issues they face when dealing with new versions of the API. From the producer perspective, we seek to understand how they support the consumers and what behavior they expect from the consumers when it comes to dealing with API evolution. Finally, we analyze how language designers can have an impact on consumer behavior regarding API evolution. We see that API consumers need to be provided a lot more support when it comes to dealing with deprecation. We hope that in the future more research effort will be focused on this problem.

Bibliography

URLs in this thesis have been archived on Archive.org. Their link target in digital editions refers to this timestamped version.

References

[1] Frederick P Brooks. No silver bullet. *Software state-of-the-art*, pages 14–29, 1975.

[2] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

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

[4] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2016.

[5] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[6] Penny Grubb and Armstrong Takang. *Software Maintenance: Concepts and Practice*. World Scientific Pub Co Inc, 2003.

[7] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 325–334. ACM, 2010.

[8] Laerte Xavier, André C. Hora, and Marco Tulio Valente. Why do we break apis? first answers from developers. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 392–396, 2017.

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

[10] John R. Rose. 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.

[11] Eclipse foundation. Eclipse ide. http://www.eclipse.org/, 2018.

[12] Alex Buckley. Project jigsaw: Under the hood. *Java Platform Group, Oracle*, 2015.

[13] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 400–410. IEEE, 2016.

[14] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, pages 1–40, 2017.

[15] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 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*, volume 1, pages 360–369. IEEE, 2016.

[16] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. On the use of replacement messages in {API} deprecation: An empirical study. *Journal of Systems and Software*, 137:306 – 321, 2018.

[17] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. Why are features deprecated? an investigation into the motivation behind deprecation. In *Proceedings of the 34th IEEE International Conference on Software Maintainance and Evolution*, page in print. IEEE, 2018.

[18] Anand Ashok Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering*, ICSE '18, pages 561–571, 2018.

[19] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 215–224. IEEE, 2014.

[20] Meir M Lehman. The programming process. *internal IBM report*, 2014:313–324, 1969.

[21] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[22] David Lorge Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287. IEEE, 1994.

[23] Martin P Robillard and Robert DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[24] Barthelemy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of 30th International Conference on Software engineering (ICSE)*, pages 481–490, 2008.

[25] Danny Dig and Ralph E. Johnson. The role of refactorings in api evolution. In *ICSM 2005: Proceedings of the 21st International Conference on Software Maintenance*, pages 389–398, 2005.

[26] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

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

[28] Zhenchang Xing and Eleni Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[29] Barthelemy Dagenais and Martin P Robillard. Semdiff: Analysis and recommendation support for api evolution. In *Proceedings of the 31st International Conference on Software Engineering*, pages 599–602. IEEE Computer Society, 2009.

[30] Mark Reinhold. Java 5 release notes. `https://www.oracle.com/technetwork/java/javase/releasenotes-142123.html`, 2004.

[31] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of API documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[32] Martin P Robillard. What makes APIs hard to learn? answers from developers. *IEEE software*, 26(6), 2009.

[33] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[34] R Burke Johnson and Anthony J Onwuegbuzie. Mixed methods research: A research paradigm whose time has come. *Educational researcher*, 33(7):14–26, 2004.

[35] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

[36] Mark Mason. Sample size and saturation in phd studies using qualitative interviews. In *Forum qualitative Sozialforschung/Forum: qualitative social research*, volume 11, 2010.

[37] Kathy Charmaz and Linda Liska Belgrave. Grounded theory. *The Blackwell encyclopedia of sociology*, 2007.

[38] Sally Thorne. *Interpretive description*. Routledge, 2016.

[39] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[40] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining api popularity. In *Testing–Practice and Research Techniques*, pages 173–180. Springer, 2010.

[41] David Mandelin and Doug Kimelman. Jungloid mining : Helping to navigate the api jungle. *Synthesis*, pages 48–61, 2006.

[42] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International workshop on Mining Software Repositories (MSR)*, pages 54–57. ACM, 2006.

[43] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.

[44] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, page 1317, 2011.

[45] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. Cohesive and isolated development with branches. In *Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2012.

[46] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the IDE. In *Proceedings of the 35th International Conference on Software Engineering, Tool Demo Track*, ICSE 2013, pages 1295–1298. IEEE CS Press, 2013.

[47] Anand Ashok Sawant and Alberto Bacchelli. A dataset for api usage. In *Proceedings of 12th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 506–509. IEEE, 2015.

[48] Tiobe index. http://www.tiobe.com/tiobe_index. accessed on 19 FEB 2017.

[49] Vaibhav Saini, Hitesh Sajnani, Joel Ossher, and Cristina V Lopes. A dataset for maven artifacts and bug patterns found in them. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 416–419. ACM, 2014.

[50] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. *ACM Sigplan Notices*, 43(10):313–328, 2008.

[51] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the 2nd International Symposium Symposium on Constructing Software Engineering Tools*, CoSET 2000, 2000.

[52] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. On the effectiveness of manual and automatic unit test generation. In *The Third International Conference on Software Engineering Advances*, ICSEA 2008, pages 252–257. IEEE, 2008.

[53] Lars Vogel. Eclipse JDT-Abstract Syntax Tree (AST) and the java model-tutorial. http://www.vogella.com/tutorials/EclipseJDT/article.html.

[54] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[55] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proceedings of the 20th international conference on Very Large Data bases*, volume 1215 of *VLDB 1994*, pages 487–499, 1994.

[56] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR 2013, pages 319–328. IEEE Press, 2013.

[57] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.

[58] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 07 of *ESEC/FSE 2007*, pages 35–44, 2007.

[59] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE 2009, pages 283–294. IEEE Computer Society, 2009.

[60] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005.

[61] Amir Michail. Data mining library reuse patterns in user-selected applications. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering, 1999.*, ASE 1999, pages 24–33. IEEE, 1999.

[62] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[63] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *Proceedings of the 37th International conference on Software engineering*, MSR 2015, page in press, 2015.

[64] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387. ACM.

[65] Ophir Primat. Github's 10,000 most popular java projects – here are the top libraries they use. http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/, nov 2013.

[66] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[67] Anand Sawant and Alberto Bacchelli. Api usage databases. `http://dx.doi.org/10.6084/m9.figshare.1320591`, 2015.

[68] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 26th ACM Symposium on Applied Computing*, SAC 2011, pages 1317–1324, 2011.

[69] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 70–79. IEEE, 2013.

[70] Suresh Thummalapenta and Tao Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2008, pages 327–336, 2008.

[71] Steven D Fraser, Frederick P Brooks Jr, Martin Fowler, Ricardo Lopez, Aki Namioka, Linda Northrop, David Lorge Parnas, and David Thomas. No silver bullet reloaded: retrospective on essence and accidents of software engineering. In *Proceedings of 22nd ACM SIGPLAN Conference on Object-oriented programming systems and applications (OOPSLA)*, pages 1026–1030. ACM, 2007.

[72] Anand Ashok Sawant and Alberto Bacchelli. fine-grape: fine-grained api usage extractor – an approach and dataset to investigate api usage. *Empirical Software Engineering*, pages 1–24, 2016.

[73] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming (OOPSLA) 1997*, pages 318–326, October 1997.

[74] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[75] Adrian Lienhard and Lukas Renggli. Squeaksource–smart monticello repository esug innovation technology awards 2005. 2005.

[76] Natalia Juristo Juzgado and Sira Vegas. The role of non-exact replications in software engineering experiments. *Empirical Software Engineering*, 16(3):295–324, 2011.

[77] http://www.github.com. last accessed February 2017.

[78] PYPL popularity of programming language. http://pypl.github.io. accessed on 19 FEB 2017.

[79] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387, 2014.

[80] Easymock api repository. https://github.com/easymock/easymock. accessed on 7 April 2016.

[81] Guava api repository. https://github.com/google/guava. accessed on 7 April 2016.

[82] Guice api repository. https://github.com/google/guice. accessed on 7 April 2016.

[83] Hibernate api repository. https://github.com/hibernate/hibernate-orm. accessed on 7 April 2016.

[84] Spring api repository. https://github.com/spring-projects/spring-framework. accessed on 7 April 2016.

[85] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR 2013, pages 233–236, 2013.

[86] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 466–476. ACM, 2013.

[87] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 309–312, 2010.

[88] Asm bytecode manipulator. http://asm.ow2.org/. accessed on 7 April 2016.

[89] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 251–260. IEEE, 2015.

[90] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, pages 471–480, 2008.

[91] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[92] Erik Mooi and Marko Sarstedt. *Cluster analysis*. Springer, 2010.

[93] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering*, ICSE '14, pages 12–23. ACM, 2014.

[94] http://www.oracle.com/technetwork/java/javase/compatibility-417013.html#incompatibilities. last accessed February 2017.

[95] John Businge, Alexander Serebrenik, and Mark GJ van den Brand. Eclipse api usage: the good and the bad. *Software Quality Journal*, 23(1):107–141, 2013.

[96] André C. Hora, Marco Tulio Valente, Romain Robbes, and Nicolas Anquetil. When should internal interfaces be promoted to public? In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 278–289, 2016.

[97] John Businge, Alexander Serebrenik, and Mark van den Brand. Analyzing the eclipse API usage: Putting the developer in the loop. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering, CSMR*, pages 37–46, 2013.

[98] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *23nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016*, page to appear, 2016.

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

[100] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference onSoftware Maintenance (ICSM)*, pages 378–387. IEEE, 2012.

[101] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the android ecosystem. In *Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 70–79. IEEE, 2013.

[102] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proceedings of 22nd International Conference on Program Comprehension (ICPC)*, pages 83–94. ACM, 2014.

[103] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering (TSE)*, 41(4):384–407, 2015.

[104] Shaohua Wang, Iman Keivanloo, and Ying Zou. How do developers react to restful api evolution? *Service-Oriented Computing*, pages 245–259, 2014.

[105] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web api fragility: How robust is your mobile application? In *Proceedings of the 2nd International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 12–21. IEEE, 2015.

[106] Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of 20th International Symposium on the Foundations of Software Engineering (FSE)*, page 55. ACM, 2012.

[107] Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, 2016.

[108] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12, 2017.

[109] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. When github meets CRAN: an analysis of inter-repository package dependency problems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 493–504, 2016.

[110] Daqing Hou and David M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 233–242, 2011.

[111] Daqing Hou and David M. Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 26–30, New York, NY, USA, 2010. ACM.

[112] Kingsum Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 359–368, 1996.

[113] Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support API evolution. In *Proceedings of 27th International Conference on Software Engineering (ICSE)*, pages 274–283, 2005.

[114] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. Refactoring-aware configuration management for object-oriented programs. In *29th International Conference on Software Engineering*, pages 427–436, 2007.

[115] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010.

[116] Reid Holmes and Robert J Walker. Customized awareness: recommending relevant external change events. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 465–474. ACM, 2010.

[117] Stuart Marks. JEP 277: Enhanced Deprecation. `http://openjdk.java.net/jeps/277`, 2014–2017. last accessed Aug 2017.

[118] Harrie Jansen. The logic of qualitative survey research and its position in the field of social research methods. In *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, volume 11, 2010.

[119] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *Proceedings of 9th Working Conference on Mining Software Repositories*, pages 12–21. IEEE, 2012.

[120] Anand Ashok Sawant and Alberto Bacchelli. fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage. *Empirical Software Engineering*, pages 1–24, 2017.

[121] Donna Spencer and Todd Warfel. Card sorting: a definitive guide. *Boxes and Arrows*, page 2, 2004.

[122] Robert J Fisher. Social desirability bias and the validity of indirect questioning. *Journal of consumer research*, 20(2):303–315, 1993.

[123] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[124] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, 2014.

[125] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[126] Puneet Kapur, Brad Cossette, and Robert J Walker. *Refactoring references for library migration*, volume 45. ACM, 2010.

[127] Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 175–184. ACM, 2007.

[128] Jeff H Perkins. Automatically generating refactorings to support api evolution. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 111–114. ACM, 2005.

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

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

[131] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering*, 21(6):2366–2412, 2016.

[132] Wei Wu, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Acua: Api change and usage auditor. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 89–94. IEEE, 2014.

[133] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 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*, pages 84–93. IEEE, 2014.

[134] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[135] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[136] Anand Ashok Sawant, Maurício Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th International Conference On Software Engineering*, ICSE 2018, pages 181–190. IEEE/ACM, 2018.

[137] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4):049901, 2007.

[138] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)*, pages 111–147, 1974.

[139] Javadoc for spring deprecation. `https://www.javadoc.io/doc/org.springframework/spring-core/4.2.3.RELEASE`, 2018. last accessed April 2018.

[140] Commit that deprecates feature in spring. `https://github.com/spring-projects/spring-framework/commit/e27df06f919a1f1ef53b0571e1a15dfc9e2f707f`, 2018. last accessed April 2018.

[141] Spring issue tracker post. https://jira.spring.io/browse/SPR-13621, 2018. last accessed April 2018.

[142] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 127–136. ACM, 2010.

[143] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321, 2018.

[144] Eclipse jdt ast parser. `http://help.eclipse.org/kepler/ntopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html`, 2018. last accessed April 2018.

[145] Mario Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[146] Alberto Bacchelli, Michele Lanza, and Vitezslav Humpa. RTFM (Read The Factual Mails) –augmenting program comprehension with REmail. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR 2011, pages 15–24. IEEE, 2011.

[147] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. Communication in open source software development mailing lists. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR 2013, pages 277–286. IEEE, 2013.

[148] William Lidwell, Kritina Holden, and Jill Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[149] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[150] Replication package. `https://www.dropbox.com/s/veef9j517okmf2d/replication-package.zip?dl=0`, 2018. last accessed April 2018.

[151] Xwiki unstable annotation. `http://dev.xwiki.org/xwiki/bin/view/Community/DevelopmentPractices#H40UnstableAnnotation`, 2018. last accessed April 2018.

[152] Barthélémy Dagenais and Martin P Robillard. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):19, 2011.

[153] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 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*, pages 138–147. IEEE, 2017.

[154] Bradley E Cossette and Robert J Walker. 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*, page 55. ACM, 2012.

[155] Gias Uddin and Martin P Robillard. How API documentation fails. *IEEE Software*, 32(4):68–75, 2015.

[156] Jeffrey Stylos, Brad A Myers, and Zizhuang Yang. Jadeite: improving API documentation using usage information. In *CHI'09 Extended Abstracts on Human Factors in Computing Systems*, pages 4429–4434. ACM, 2009.

[157] Christoph Treude and Martin P Robillard. Augmenting API documentation with insights from stack overflow. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 392–403. IEEE, 2016.

[158] Brad A Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6):62–69, 2016.

[159] Walid Maalej and Martin P Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[160] Uri Dekel and James D Herbsleb. Improving API documentation usability with knowledge pushing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 320–330. IEEE, 2009.

[161] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *20th Working Conference on Reverse Engineering*, WCRE 2013, pages 401–408. IEEE, 2013.

[162] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. Mining multi-level API usage patterns. In *22nd International Conference on Software Analysis, Evolution and Reengineering*, SANER 2015, pages 23–32. IEEE, 2015.

[163] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *37th International Conference on Software Engineering*, volume 1 of *ICSE 2015*, pages 880–890. ACM/IEEE, 2015.

[164] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[165] Barthélémy Dagenais and Martin P Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pages 47–57. IEEE, 2012.

[166] A. A. Sawant and A. Bacchelli. 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)*, page in press. IEEE press, 2016.

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

[168] Stuart Marks. 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.

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

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

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

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

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

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

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

[176] Jens Dietrich, Kamil Jezek, and Premek Brada. 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*, pages 64–73. IEEE, 2014.

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

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

[179] William Pugh. Jsr 305: Annotations for software defect detection. `https://jcp.org/en/jsr/detail?id=305`, 2006. last accessed May 2017.

[180] Michael D Ernst. Jsr 308: Type annotations specification. `https://jcp.org/en/jsr/detail?id=308`, 2008. last accessed May 2017.

[181] TIOBE Index. Tiobe.–2017.[electronic resource]. *Mode of access http://www.tiobe.com/tiobe_index*.

[182] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[183] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

Glossary

API Application Programming Interface, interface of a pre-defined set of functionality that can be resued by a developer in their code.

AST Abstract Syntax Tree, an AST is a tree-like representation of source code tokens.

CI Continuous Integration.

CRAN Comprehensive R Archive Network, repository with R packages.

Deprecation The marking of an API element as defunct or obsolete and is indicated as something that will be removed in a future version of the API.

fine-GRAPE fine-grained APi usage extractor.

GUI Graphical User Interface.

IDE Integrated Development Environment, dedicated tool designed to aid developer during their development process, for example: Eclipse or IntelliJ.

JAR Java ARchive, a file type that packages Java class files.

JEP Java Enhancement Proposal, is a process to collect proposals for enhancements to Java.

JLS Java Language Specification, is a set of rules that specifies the behavior of the Java language.

JSR Java Specification Request, a request that can be made to enhance or change Java.

ORM Object Relation Mapping, a mapping between Java objects and a database schema.

OSS Open Source Software.

POM Project Object Model, a configuration file used by Maven to build a Java project.

PR Pull Request.

Reaction Reaction, defined as the action taken by an API consumer when an deprecated API element is encountered.

RSW Removal Severity Warning, is a proposal to enhance the deprecation mechanism in Java, more details on which can be found in chapter 7.

SDK Software Development Kit.

SQL Structured Query Language.

Curriculum Vitæ

Anand Ashok Sawant

1991/08/05 Date of birth in Mumbai, India

Education

11/2015-10/2019 Ph.D. Student, Software Engineering Group,
Delft University of Technology, The Netherlands,
The impact of API evolution on API consumers and how this can be affected by API producers and language designers,
Promotors: Prof. Dr. Arie van Deursen and Prof. Dr. Alberto Bacchelli

08/2013-11/2015 M.Sc. Computer Science,
Delft University of Technology, The Netherlands,
Thesis: *fine-GRAPE: fine-Grained APi usage Extractor An Approach and Dataset to Investigate API Usage* 9.0

08/2012-12/2012 Minor in Electrical Engineering,
University of Illinois Urbana Champaign, USA

08/2010-07/2013 B.Sc. Computer Science,
Delft University of Technology, The Netherlands,
Thesis: *Project Jackal* 9.0

07/2007-03/2009 Pre University College,
St. Josephs Pre University College, India

03/1999-03/2007 Indian Certificate of Secondary Education,
Sishu Griha English School, India

Experience

11/2018-12/2018 Visting researcher,
ABB Corporate Research, USA

05/2018-08/2015	Research Intern, ABB Corporate Research, USA
07/2017-08/2017	Visting researcher, University of Bolzano, Italy
06/2016-08/2016	Visting researcher, McGill University, Canada
01/2015-06/2015	Visting researcher, Georgia Institute of Technology, USA

Work Experience

10/2013-10/2014	Senior App developer, Milvum B.V., The Netherlands
03/2013-07/2013	Intern, COAS B.V., The Netherlands

Financial awards

2018	ACM SIGSOFT CAPS Grant
2017	ACM SIGSOFT Turing award travel grant
2017	NII SHONAN best student award

Titles in the IPA Dissertation Series since 2016

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities.* Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too.* Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code.* Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems.* Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming.* Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi.* Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems.* Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces.* Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network.* Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems.* Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations.* Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries.* Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages.* Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing.* Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12