



HAL
open science

Analyzing the Impact of Pull Requests to Guide Library Evolution

Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri

► **To cite this version:**

Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri. Analyzing the Impact of Pull Requests to Guide Library Evolution. 2021. hal-03420593v1

HAL Id: hal-03420593

<https://hal.science/hal-03420593v1>

Preprint submitted on 9 Nov 2021 (v1), last revised 22 Mar 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Analyzing the Impact of Pull Requests to Guide Library Evolution

Lina Ochoa

l.m.ochoa.venegas@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

Thomas Degueule

thomas.degueule@labri.fr
Univ. Bordeaux, Bordeaux INP, CNRS,
LaBRI, UMR5800
Talence, France

Jean-Rémy Falleri

jean-remy.falleri@labri.fr
Univ. Bordeaux, Bordeaux INP, CNRS,
LaBRI, UMR5800
Institut Universitaire de France
Talence, France

ABSTRACT

“If we make this change to our code, how will it impact our clients?” It is difficult for library maintainers to answer this simple—yet essential!—question when evolving their libraries. Library maintainers are constantly balancing between two opposing positions: make changes at the risk of breaking some of their clients, or avoid changes and maintain compatibility at the cost of immobility and growing technical debt. We argue that the lack of objective usage data and tool support leaves maintainers with their own subjective perception of their community to make these decisions.

We introduce BREAKBOT, a bot that analyses the pull requests of Java libraries on GitHub to identify the breaking changes they introduce and their impact on client projects. Through static analysis of libraries and clients, it extracts and summarizes objective data that enrich the code review process by providing maintainers with the appropriate information to decide whether—and how—changes should be accepted, directly in the pull requests.

ACM Reference Format:

Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. 2022. Analyzing the Impact of Pull Requests to Guide Library Evolution. In *Proceedings of NIER’22: NIER’22 (NIER’22)*. ACM, New York, NY, USA, 5 pages. <https://doi.org/--->

1 INTRODUCTION

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails the use of external *software libraries*. Rather than implementing new systems from scratch, developers incorporate libraries that provide functionalities of interest into their projects. Libraries expose their features through *Application Programming Interfaces* (APIs), which govern the interactions between client projects and libraries.

Libraries constantly evolve to incorporate new features, bug fixes, security patches, refactorings, and extra-functional improvements [1, 3]. It is critical for clients to stay up-to-date with the libraries they use to benefit from these improvements and to avoid security issues, technical lag, and technical debt [5, 8]. When a library evolves, however, it may break the contract previously established with its clients by introducing *Breaking Changes* (BCs) in its public API, resulting in compilation-time, link-time, or behavioral errors. Seemingly innocuous changes such as altering the visibility or abstractness modifier of a type declaration or inserting a new abstract method can, under certain conditions, break client code. Errors triggered by these changes burden client developers, given the sudden urgency to fix issues out of their control without intrinsic motivation. As a result, clients sometimes hesitate to upgrade their dependencies, raising security

concerns and making future upgrades even more difficult. Thus, it does not come as a surprise that the problem of helping clients respond to library evolution has garnered considerable interest in recent years [16, 19, 21, 26, 27].

Surprisingly, however, the problem of helping library maintainers anticipate the impact of their changes and plan accordingly—the other side of the coin—has received little attention. Libraries are in general accountable to their clients for providing stability [2]. We claim that library maintainers currently lack the necessary information and tool support to live up to this responsibility. Indeed, library maintainers have limited means to foresee the consequences of their actions on their clients [12, 23] and would benefit from knowing precisely how their APIs are used in client projects.

The consequences of breaking an API declaration used by many clients, by a popular library, or by commercial clients might be dramatic: clients might lose trust and migrate to another one. Consequently, some very cautious maintainers refuse to change any existing declaration, thinking it might be too impactful, leading to an accumulation of technical debt and an aging design that will scare people away [2]. Some more adventurous maintainers will push any change, disregarding all clients using the affected declaration, at the risk of breaking too many clients and having to revert the change ultimately [24]. We believe that the sweet spot is at the crossroad of innovation and clients’ stability. Unfortunately, there are only limited ways for library maintainers to acquire evidence-based knowledge on the usage of their API, and they often must rely on their own judgment and the subjective knowledge of their community to figure out how to evolve it [28].

In order to assist maintainers in the evolution of their libraries while avoiding breaking their clients unexpectedly, we propose the following vision: *empowering library maintainers to make evidence-backed decisions regarding the evolution of their libraries by accurately and efficiently informing them about breaking changes and their impact on relevant clients*. Concretely, this will enable maintainers to ask *what-if* questions and explore what consequences evolving their library would have on their clients: “*What if we push this refactoring?*”, “*What if we alter this method’s signature?*”, etc.

In this paper, we introduce a proof-of-concept implementation of our vision: BREAKBOT, a GitHub bot that analyzes the content of Pull Requests (PRs) in Java repositories and employs static analysis to identify BCs and their impact on relevant clients. The resulting reports are fed back into the PRs, enabling maintainers to quickly review changes and their impact and decide whether they should be accepted. Because our tool employs static analysis, we focus on source- and binary-incompatible changes, which are an important class of BCs as they force clients to modify their code at upgrade time, and leave behavioral changes aside. We expect our vision and its implementation in BREAKBOT to fight immobility and stagnation

by pushing maintainers to introduce BCs when it is safe and avoid pushing these changes when they are too impactful. Making the co-evolution of library and clients more harmonious should benefit library maintainers and client developers alike.

2 BACKGROUND & MOTIVATING EXAMPLE

Deciding whether and when to introduce BCs is no trifling matter. The decision depends on values and constraints established at the ecosystem and library levels. However, the desire to preserve *stability* and to prioritize the sense of *community* are generalized concerns among all projects in most ecosystems [2]. To enact stability, libraries tend to assume that BCs are always harmful [13]. This reasoning has led to the misconception that stability is equivalent to change stagnation [2, 4, 18]. However, we claim that stability is the capability to not break clients even under the presence of change. What truly affects stability are *changes that break clients*, not BCs themselves. This distinction is essential: it has been shown that, for instance in Maven Central, most BCs do not impact the clients whatsoever [22]. Another problem strikes when evidence about how changes are impacting clients is not communicated, increasing the mistrust in the community. Thus, some evolving libraries opt for communicating change via their API, versioning conventions such as semantic versioning, or official documentation such as release notes and changelogs. However, these artifacts might be seen as promises that, when broken, result in further mistrust. Furthermore, official documentation tends to be incomplete given that only notable changes are requested to be included [15].

To illustrate the issues that library maintainers run into when introducing BCs, we take an example brought to our attention by a core maintainer of Spoon—a source code analysis and transformation library for Java developed on GitHub. Spoon developers are very concerned about stability and therefore avoid BCs. When they do want to introduce a BC, they use deprecation to warn clients before proceeding with the change. In pull request PR#2683,¹ a new feature to pretty-print Java import statements was rolled in, making a previous implementation—the `ImportScanner` interface and corresponding `ImportScannerImpl` implementation—obsolete. Therefore, the author of the PR labeled both types as deprecated. Two months later, in PR#3184, Spoon developers removed a set of deprecated types and methods from the code, notably the two types mentioned above. The PR was merged, making its way into Spoon’s main branch.² However, three months later, Spoon’s developers noticed that these changes broke two important clients (Astor [17] and DSpot [6]) and realized that the `ImportScanner` interface and implementation were in fact used by clients and thus reconsidered the deprecation. They finally reintroduced them in PR#3266.³

It is evident that having to revert changes after introducing them is not ideal. If other changes had been made based on the original changes, they would have had to be reverted too, triggering a nasty ripple effect within the library’s code. How could this have been avoided? To the best of our knowledge, there are three approaches enabling library maintainers to evaluate the BCs they introduce.

Regression testing. The traditional approach is to use a regression test suite. However, a regression test suite cannot cover every possible

Listing 1: Extract of Astor’s build log file (704 lines in total).

```

1 Started by timer
...
211 [ERROR] The goal you specified requires a project to execute but
      there is no POM in this directory.
...
226 [ERROR] Entities.java:[63,29] unmappable character for UTF-8
...
295 [ERROR] TypeUtilsTest.java:[524,47] incompatible types: inferred
      type does not conform to upper bound(s)
...
654 [ERROR] LibParser.java:[23,29] cannot find symbol
655 [ERROR] symbol: class ImportScanner

```

use of the library, and therefore unanticipated BCs may leak to the clients. In addition, when the test suite detects a regression, it is not able to evaluate its impact on the library’s clients. Maintainers must resort to their subjective opinion to decide whether or not to include the change. In our example case, Spoon’s test suite noticed the BCs introduced in PR#3184, and the `ImportScannerTest` class was deleted as part of the same PR. However, since the deleted types were deprecated, the maintainers did not reconsider the change.

Static analysis. Various tools can scan two versions of a library using static analysis to output the list of BCs between them [14]. This approach is more complete than regression testing (although false positives or negatives can arise), therefore it is unlikely that BCs will leak to clients. For instance, Guava relies on JDiff [9] while the Apache foundation uses JApiCmp [10] to systematically search for BCs. Nevertheless, these tools only list the introduced BCs without showing the actual impact on client projects. If Spoon’s developers were using this type of tool, it would have noticed the changes the same way their test suite did.

Reverse dependency compatibility testing. The third approach—the only one able to evaluate the impact on clients—is Reverse Dependency Compatibility Testing (RDCT) [29]. The idea of RDCT is to identify a set of relevant clients, retrieve their source code, inject the new version of the library in their dependencies, and finally build them and run their test suite. If any error is detected during the process, maintainers are notified and can analyze the issue by browsing through the build log files. Projects like Scala, Rust, or Spoon perform periodical RDCT on their clients. The Coq project goes a step further by directly integrating RDCT as part of its Continuous Integration (CI) process, so that any BC impacting a client is detected right away [29].

However, RDCT suffers from several drawbacks. First, selected clients must always be buildable and have a test suite that passes on the previous version of the library. Second, RDCT takes a considerable amount of time and computational resources to build and test every client. For instance, the Rust project reports requiring up to a week to check 74,234 clients, as of September 2019. Third, when BCs are introduced, the resulting log files are verbose, forcing the maintainers to filter irrelevant information to understand the root cause. Last, whenever a BC is merged, the impacted clients can no longer be used for subsequent RDCT until they are repaired. For this reason, Coq’s maintainers resort to fixing the clients themselves to keep their CI process running, which requires considerable effort.

In our example case, Spoon’s RDCT runs every night on 13 projects. Following PR#3184, it ran into errors for the Astor and DSpot clients. An analysis of the log file of Astor (Listing 1), containing 266 errors among 704 log statements, revealed that the deleted types raised

¹<https://github.com/INRIA/spoon/pull/2683>

²<https://github.com/INRIA/spoon/pull/3184>

³<https://github.com/INRIA/spoon/pull/3266>

missing symbol errors at compile-time. Thanks to RDCT, Spoon developers discovered the impact of the BC. However, Astor’s RDCT build was failing during the three months before the revert and was thus not usable to check for other BCs in this period.

In summary, while RDCT helps analyze the impact of BCs on clients, its drawbacks severely hinder its usability and adoption. In the next section, we present a novel and lightweight approach based on static analysis to evaluate the impact of source- and binary-incompatible BCs on clients and integrate it within the maintainers’ workflow. Naturally, RDCT still goes beyond our approach regarding behavioral-incompatible changes, at the expense of building and running the client test suites.

3 BREAKBOT

BREAKBOT is a proof-of-concept implementation of our vision. It can analyze libraries and clients written in Java and integrates seamlessly with GitHub’s pull-based development workflow. It is implemented as a GitHub bot that provides library maintainers with the right information whenever and wherever needed. Implementing BREAKBOT as a bot allows us to fully automate the process and be as close as possible to the maintainers’ usual development workflow. Because BREAKBOT feeds helpful information directly in GitHub’s PRs, we expect it to enhance the code review process, as was shown for other bots in previous work [25].

3.1 Overview

Once maintainers install BREAKBOT on a library’s repository and configure a list of clients of interest, BREAKBOT registers itself via the GitHub’s Checks API [7] to analyze each new or updated PR. Then, it reports: i) the list of BCs they introduce, and ii) the impact of those BCs on the configured list of clients. Under the hood, BREAKBOT leverages a novel static analysis tool, MARACAS, to analyze libraries and clients and to infer the list and impact of BCs. Note that BREAKBOT’s design enables it to be reused with little effort for other analysis frameworks similar to MARACAS, targeting other programming languages. Figure 1 gives a high-level view of BREAKBOT. We detail below the main steps of the process.

❶ *Breaking changes analysis.* In a first step, BREAKBOT communicates to MARACAS the GitHub PR to analyze. In every PR, the head pointer points to the branch containing the PR’s code, while the base pointer points to the branch where the changes should be pulled into. The diff between head and base represents the changes introduced by the PR. MARACAS first clones and builds both these versions to obtain binary JARs, uses `japicmp` to extract the list of BCs, and stores it in a dedicated Δ -model [22]. Each BC in a Δ -model specifies the affected declaration (a particular type, method, or field) and the kind of BC (e.g., method removed, field now final, class less accessible). A configuration file hosted on the library’s repository allows maintainers to specify how to build their library and which parts of their API should be excluded from the analysis (e.g., `@Experimental` declarations and `*test*` packages) because they are exempt from compatibility guarantees.

❷ *Impact analysis.* In a second step, for each configured client, MARACAS builds an impact model that pinpoints how the changes in the Δ -model impact the project. While Ochoa et al. originally designed MARACAS to analyze the impact of BCs on bytecode [22],

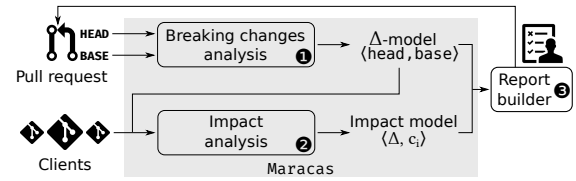


Figure 1: BREAKBOT handles the communication between GitHub and MARACAS to produce meaningful reports.

we redesign it to statically analyze their impact on source code using the Spoon framework. Because this new implementation does not require clients to be built, it frees BREAKBOT from the limitations of state-of-the-practice RDCT regarding build times and the need to repair clients constantly. Each location in client code impacted by a BC is associated with: i) the impacted AST node in client code (e.g., a variable, a method invocation, an import); ii) the library declaration it uses that is impacted by a BC, and; iii) the way the client is using the declaration (e.g., through class instantiation or extension, as a type argument, through field access).

❸ *Report builder.* In a third step, the Δ -model and impact models are sent back to BREAKBOT, which builds a final report incorporated directly within the PR. This report displays the result of the analysis in a way that library maintainers can easily interpret. We design the report to highlight the essential information first: the list of BCs, how each of them individually impacts client code, and how many of the clients are impacted. Then, a more detailed report describes, for each client, every location in its source code impacted by the changes. To make it easier to understand the causes and consequences of BCs, every code reference in the report directly links to the corresponding source code on GitHub.

3.2 Revisiting the Spoon Example

We used BREAKBOT and MARACAS to analyze PR#3184 and showcase how it alleviates the drawbacks of RDCT identified in Section 2. Figure 2 depicts the most important parts of the report.

First, because MARACAS employs static analysis to infer the impact of BCs, it only needs to parse the clients to build their ASTs. It does not require clients to be built and thus avoids the need for a complex CI system that may suffer from additional failures: dependency resolution, flaky tests, *etc.* Besides, it takes considerably less time and resources to analyze the clients: while Spoon’s RDCT took 6 minutes to analyze Astor only, BREAKBOT could analyze all 13 clients in 2 minutes.

Second, RDCT suffers from the need to analyze verbose and cryptic log files for each client, separated from the PR’s code, riddled with unrelated outputs and errors, and messages that do not explicitly map client errors to their root cause at the library level (Listing 1). In contrast, the report built by BREAKBOT centralizes and highlights the essential information: the PR introduces 13 BCs, causing 32 broken locations in client code, and 5 out of 13 clients are impacted by the changes (Figure 2c). The remainder of the report details, for each client, which particular locations are impacted to let the maintainers understand *why* the clients break and how they might improve the PR (Figure 2b for Astor).

In this particular case, BREAKBOT accurately reports that the class removals impact Astor and DSpot, but also that the type `CompilationUnit`,

Declaration	Kind	Status	Impacted clients	Broken locations
<code>spoon.reflect.cu.CompilationUnit</code>	ANNOTATION_DEPRECATED_ADDED	✗	4 (spooet, jPerturb, nopol, astor)	14
<code>spoon.reflect.visitor.ImportScanner.getAllImports()</code>	METHOD_REMOVED	✗	2 (dspot, astor)	2
<code>spoon.reflect.visitor.ImportScanner</code>	CLASS_REMOVED	✗	2 (dspot, astor)	12
<code>spoon.reflect.visitor.MinimalImportScanner</code>	CLASS_REMOVED	✓	None	None

(a) Excerpt of the breaking changes and their impact on clients

Location	Breaking declaration	Kind	Use type
<code>mutSupporter.getFactory().CompilationUnit()</code>	<code>spoon.reflect.cu.CompilationUnit</code>	ANNOTATION_DEPRECATED_ADDED	TYPE_DEPENDENCY
<code>import spoon.reflect.cu.CompilationUnit;</code>	<code>spoon.reflect.cu.CompilationUnit</code>	ANNOTATION_DEPRECATED_ADDED	IMPORT
<code>importContext.getAllImports()</code>	<code>spoon.reflect.visitor.ImportScanner.getAllImports()</code>	METHOD_REMOVED	METHOD_INVOCATION
<code>importContext</code>	<code>spoon.reflect.visitor.ImportScanner</code>	CLASS_REMOVED	TYPE_DEPENDENCY

(b) Excerpt of the impacted locations in Astor

Client	Status	Broken locations
SpoonLabs/CoCoSpoon	✓	0
STAMP-project/dspot	✗	9
Spirals-Team/casper	✓	0
SpoonLabs/metamutator	✓	0
SpoonLabs/spooet	✗	2
Spirals-Team/jPerturb	✗	3
SpoonLabs/gumtree-spoon-ast-diff	✓	0
STAMP-project/AssertFixer	✓	0
SpoonLabs/nopol	✗	4
SpoonLabs/nepfix	✓	0
SpoonLabs/astor	✗	14
SpoonLabs/nepfix	✓	0
SpoonLabs/coming	✓	0

(c) Overview of clients impact

Figure 2: Excerpts of the report generated by BREAKBOT for Spoon’s PR#3184 in GitHub

deprecated in the same PR, is also used by clients (Figure 2a). This helps maintainers reflect on whether deprecating this type is appropriate to avoid making the same mistake again. The removal of the `MinimalImportScanner` type, however, does not impact any client and is thus innocuous. The final report effectively answers the *what-if* questions stemming from the PR.

4 FUTURE PLANS

Our prototype implementation of the vision, BREAKBOT, is a mere stepping stone. In order for our vision and for BREAKBOT to have a real impact on library maintainers, we intend to implement the following research roadmap.

Configurability. The policies spectrum to deal with backward compatibility ranges from permissive (e.g., npm, favoring *innovation*) to restrictive (e.g., Eclipse, favoring *compatibility*) [2]. Regardless of their posture, projects should be given the necessary information to decide whether to introduce a change. Libraries maintainers need, then, highly configurable impact analysis tools to anticipate the impact of their changes. However, *what are the configuration properties and their variability space that impact analysis tools should capture?* For instance, defining the type and number of allowed BCs, the parts of the API subject to analysis (e.g., non-experimental interfaces), and the key set of relevant clients (e.g., commercially-related projects), is of foremost importance.

Discovery of clients. Knowing how BCs impact relevant clients is key when deciding to apply or revert changes. Some projects, such as Coq and Spoon, have a precise list of relevant clients. However, this is not the case for all libraries and even if they have it, the list might not be complete hiding important evidence. Automatically discovering a *diverse and representative sample* [20] of clients should give a clearer picture of the library’s usage. To do so, one must first *pinpoint which versions of the library are affected by a given BC*, and *which versions of the clients might be affected*. As it is likely that clients’ usage of a library will overlap [11], it is essential to *avoid analyzing similar clients multiple times*, e.g., by clustering clients around their usage of the library.

Reporting actionable insights. BREAKBOT identifies the list of library declarations introducing BCs, and the list of impacted locations in client code. Although it improves the state of practice, it may still be hard for developers to interpret this information. Regarding BCs, *should they be represented as atomic modifications to the code, or should more meaningful operations that show the developer’s intentions (e.g., refactorings) be listed instead?* Then, *what is the right level of granularity to represent impacted client code (e.g., at the declaration, line, or AST node level)?* Finally, to decide whether to introduce a BC, *how to measure its criticality? How to define proper thresholds for metrics to enact project-specific policies?*

Approach evaluation. To evaluate BREAKBOT, we plan to use both quantitative and qualitative evaluation methods. We will first investigate the accuracy of MARACAS during the BCs and impact analysis phases using a proper benchmark such as the one introduced by Jezek and Dietrich [14]. Second, we aim at better understanding the current state of practice when dealing with change and the effects of BREAKBOT on these practices. Therefore, we plan to deploy BREAKBOT on the GitHub repositories of a few libraries (e.g., Spoon, ASM, JavaParser, Vallang, GumTree) and interview their maintainers. In particular, we plan to inquire whether BREAKBOT improves the analysis of code changes during code reviews. *Does BREAKBOT “scare” developers more, or does it encourage them to make changes they were afraid of because they could not foresee their impact?*

Future research directions. With BREAKBOT, we have focused on informing about stability issues at the *library* level. As part of our future research directions, we would like to investigate how BCs affect the stability of an ecosystem as a whole, shading light on the impact of changes and ripple effects at the scale of an entire *community*. Second, we intend to leverage BREAKBOT capabilities to develop an improved version of the Dependabot that informs client projects about library BCs and their impact on their own code. Finally, we would like to extend our research beyond static analysis to cover RDCT’s weaknesses regarding behavioral changes: *How to accurately identify the introduction of behavioral BCs? How to efficiently detect the impact of behavioral BCs on client projects?*

REFERENCES

- [1] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *2013 IEEE International Conference on Software Maintenance*. 280–289. <https://doi.org/10.1109/ICSM.2013.39>
- [2] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 42 (2021), 56 pages. <https://doi.org/10.1145/3447245>
- [3] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 255–265. <https://doi.org/10.1109/SANER.2018.8330214>
- [4] John Businge, Simon Kawuma, Moses Openja, Engineer Bainomugisha, and Alexander Serebrenik. 2019. How Stable Are Eclipse Application Framework Internal Interfaces?. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 117–127. <https://doi.org/10.1109/SANER.2019.8668018>
- [5] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.
- [6] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2019. Automatic Test Improvement with DSpot: A Study with Ten Mature Open-Source Projects. *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 2603–2635. <https://doi.org/10.1007/s10664-019-09692-y>
- [7] GitHub. 2021. *GitHub Checks API*. <https://docs.github.com/en/rest/reference/checks>
- [8] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*, Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli (Eds.). Springer International Publishing, Cham, 182–192.
- [9] Google. 2021. *API Differences between Guava 30.0-jre and Guava 30.1.1-jre*. <https://guava.dev/releases/30.1.1-jre/api/diffs/>
- [10] Gary Gregory. 2020. *How we handle binary compatibility at Apache Commons*. <https://garygregory.wordpress.com/2020/06/14/how-we-handle-binary-compatibility-at-apache-commons/>
- [11] Nicolas Harrand, Amine Benelallam, César Soto-Valero, Olivier Barais, and Benoit Baudry. 2019. Analyzing 2.3 Million Maven Dependencies to Reveal an Essential Core in APIs. *arXiv e-prints* (2019), arXiv–1908.
- [12] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. 2018. How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal* 26, 1 (2018), 161–191.
- [13] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* 50, 8 (2021), 1–17. <https://doi.org/10.1109/TSE.2021.3106247>
- [14] Kamil Jezek and Jens Dietrich. 2017. API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *J. Object Technol.* 16, 4 (2017), 2–1.
- [15] Olivier Lacan. 2019. *Keep a ChangeLog*. <https://keepachangelog.com>
- [16] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 601–614. <https://www.usenix.org/conference/atc18/presentation/lawall>
- [17] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany). Association for Computing Machinery, New York, 441–444. <https://doi.org/10.1145/2931037.2948705>
- [18] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *IEEE International Conference on Software Maintenance*. IEEE, USA, 70–79. <https://doi.org/10.1109/ICSM.2013.18>
- [19] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*. 353–363. <https://doi.org/10.1109/ICSE.2012.6227179>
- [20] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in software engineering research. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 466–476.
- [21] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-Based Approach to API Usage Adaptation. *SIGPLAN Not.* 45, 10 (Oct. 2010), 302–321. <https://doi.org/10.1145/1932682.1869486>
- [22] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2021. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central. (2021). <https://hal.archives-ouvertes.fr/hal-03378089>
- [23] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How Do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (Cary, North Carolina) (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 56, 11 pages. <https://doi.org/10.1145/2393596.2393662>
- [24] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, David Pursehouse, and Naoyasu Ubayashi. 2016. Why are Commits Being Reverted?: A Comparative Study of Industrial and Open Source Projects. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–311. <https://doi.org/10.1109/ICSME.2016.83>
- [25] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. 2020. Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–11. <https://doi.org/10.1109/ICSME46990.2020.00011>
- [26] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 325–334. <https://doi.org/10.1145/1806799.1806848>
- [27] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836.
- [28] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L. Glassman. 2020. Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '20). Association for Computing Machinery, New York, NY, USA, 1–13. <https://doi.org/10.1145/3313831.3376382>
- [29] Théo Zimmermann. 2019. *Challenges in the collaborative evolution of a proof language and its ecosystem. (Défis dans l'évolution collaborative d'un langage de preuve et de son écosystème)*. Ph.D. Dissertation. Paris Diderot University, France. <https://tel.archives-ouvertes.fr/tel-02451322>