Noname manuscript No.
(will be inserted by the editor)

You Broke My Code: Understanding the Motivations for Breaking Changes in APIs

Aline Brito · Marco Tulio Valente · Laerte Xavier · Andre Hora

Received: date / Accepted: date

Abstract As most software systems, libraries and frameworks also evolve, which may break existing clients. However, the main reasons to introduce breaking changes in APIs are unclear. Therefore, in this paper, we first report the results of an almost 4-month long field study with popular Java libraries and frameworks. We configured an infrastructure to observe all changes in these libraries and to detect breaking changes shortly after their introduction in the code. We detected possible breaking changes in 61 projects. After identifying breaking changes, we asked the developers to explain the reasons behind their decision to change the APIs. By analyzing the developers' answers, we report that breaking changes are mostly motivated by the need to implement new features, by the desire to make the APIs simpler and with fewer elements, and to improve maintainability. To complement this first study, we describe a second study, including the analysis of 110 Stack Overflow posts related to breaking changes. We reveal that breaking changes have an important impact on clients, since 45% of the questions are from clients asking how to overcome specific breaking changes; they are also common in other ecosystems—JavaScript, .NET, etc. We conclude by providing suggestions to language designers, tool builders, software engineering researchers, and API developers.

Keywords API Evolution · Breaking Changes · Firehouse Interviews · Stack Overflow

Aline Brito
Department of Computer Science, UFMG, Brazil, E-mail: alinebrito@dcc.ufmg.br

Marco Tulio Valente
Department of Computer Science, UFMG, Brazil, E-mail: mtov@dcc.ufmg.br

Laerte Xavier
Department of Computer Science, UFMG, Brazil, E-mail: laertexavier@dcc.ufmg.br

Andre Hora
Department of Computer Science, UFMG, Brazil, E-mail: andrehora@dcc.ufmg.br

1 Introduction

Software libraries are commonly used nowadays to support development, providing code reuse, improving productivity, and, consequently, decreasing costs [26, 35, 44]. For example, there are more than 200K libraries registered on Maven's central repository,[1] a popular package management for Java. They cover distinct scenarios, from mobile and web programming to scientific and statistical analysis. These functionalities are provided to client systems via *Application Programming Interfaces* (APIs), which are contracts that clients rely on [45]. In principle, APIs should be stable and backward-compatible when evolving, so that clients can confidently rely on them. However, community values also influence the stability of APIs. For example, Bogart et al. [7] show that long-term stability is the core value considered by developers when maintaining Eclipse APIs. By contrast, in other ecosystems developers may value fast change and evolution, in order to continuously improve the services provided by their APIs. This is usually the case of Node.js-based APIs. Indeed, recent studies show that APIs are often unstable and backward-incompatible (e.g., [7, 22, 30, 46, 54]).

API breaking changes comprise from simple modifications, such as the change of a method signature or return type, to more critical and impactful ones, such as the removal of a public element. In this context, one important question is not completely answered in the literature: *despite being recognized as a programming practice that may harm client applications, why do developers break APIs?* Better understanding these reasons may support the development of new language features and software engineering approaches and tools to improve library maintenance practices.

Firehouse Interview Study: In a previous conference paper [11], we first study the motivations driving API breaking changes from the perspective of library developers. By mining daily commits of relevant libraries, we looked for API breaking changes, and, when detected, we sent emails to developers to better understand the reasons behind the changes, the real impact on client applications, and the practices adopted to alleviate the breaking changes. We also characterized the most common program transformations that lead to breaking changes. Specifically, we investigated five research questions:

1. *How often do changes impact clients?* 39% of the changes investigated in the study may have an impact on clients.
2. *Why do developers break APIs?* We identified three major motivations to break APIs, including changes to support new features, to simplify the APIs, and to improve maintainability.
3. *What is the e ort on clients to migrate?* We found that a minor migration effort is required in most cases, according to the surveyed developers.

[1] https://search.maven.org/stats

4. *Why don't developers deprecate broken APIs?* Most developers mentioned the increase on maintainability effort as the reason for not deprecating broken APIs.
5. *How do developers plan to document breaking changes?* Most developers plan to document the detected breaking changes, mainly using release notes and changelogs.

To answer these questions, we followed a firehouse interview method [47]. Basically, we monitored 400 real world Java libraries and frameworks hosted on GitHub during 116 days. During this period, we detected 282 possible breaking changes, sent 102 emails, and received 56 responses, which represents a response rate of 55%. Our approach has an important characteristic: *the developers explained the breaking changes few hours after they had performed them*. As pointed by previous studies [29, 51], this is likely to produce more credible answers (since the change is fresher in the developers' minds) and grab more attention.

Stack Over ow Study: In the first study, we aimed to reveal the motivations and program transformations that cause breaking changes in a single ecosystem: Java libraries and frameworks. Furthermore, the study was based on firehouse interviews with API owners. In this paper, we extend this study in two main dimensions: (1) by considering the views and impact of breaking changes in API clients; (2) by considering breaking changes in other ecosystems, besides Java. To this purpose, instead of firehouse interviews, we based this second study on the analysis of Stack Overflow posts about breaking changes. We analyze a sample of 110 posts, and reveal that breaking changes indeed have an important impact on clients; particularly, 49 of the posts (45%) are clients asking how to overcome specific breaking changes that are causing problems in their code. We also reveal that questions about breaking changes are distributed over different ecosystems, including .NET, JavaScript, and Java.

Contributions: This paper makes five contributions.

1. To our knowledge, we are the first to reveal the reasons of *concrete* breaking changes introduced by practitioners in the source code of popular Java APIs.
2. We show how breaking changes are introduced in the source code, including the most common program transformations used to break APIs.
3. We confirm that breaking changes have an important impact on clients; they also happen in other ecosystems.
4. We provide an extensive list of implications of our study, including implications to language designers, tool builders, researchers, and practitioners.

Structure of the paper. Section 2 introduces the tool and approach used to detect breaking changes. Section 3 details the design of the firehouse interview study, while Section 4 presents its results. Section 5 describes the second study, based on the analysis of 110 Stack Overflow posts. We discuss the implications of both studies in Section 6. Section 7 lists threats to validity and Section 8 presents related work. Finally, we conclude the paper in Section 9.

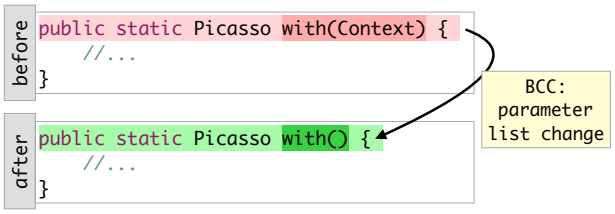


Fig. 1 Example of BCC detected by apidiff at method level

2 APIDiff Tool

To detect breaking changes, we use a tool named apidiff [10], which was implemented and used by Xavier et al. [55] in a study about the frequency and impact of breaking changes. Essentially, apidiff compares two versions of a library and lists all syntactic changes in public elements. In other words, the tool focus in changes in public elements which cause compilation errors in client's source code.[2] It is also important to mention that most changes currently detected by APIDiff are inspired in a catalog of breaking changes proposed by Dig and Johnson [19]. In this paper, the results produced by apidiff are named *Breaking Change Candidates* (BCC). The reason is that changes in public elements—as identified by apidiff—do not necessarily have an impact on API clients. For example, the changed elements may denote internal or low-level services, which are designed only for local usage. To clarify this question, we conducted a survey with API developers, to confirm whether the BCCs detected by apidiff are indeed *breaking changes* (see Section 3).

De nition: Changes detected by apidiff in public API elements are named Breaking Change Candidates (BCC).

Table 1 lists the BCCs detected by apidiff. These changes refer to the following API elements: types, methods, or fields. BCCs on types include, for example, drastic changes, like the removal of a type from the code. But subtle changes in public types are also detected, including changing a type visibility from public to another modifier, changing the supertype of a type, adding a *nal* modifier to a type (to disable inheritance), or removing the *static* modifier of an inner class. Besides the changes detected to types, BCCs in methods include changes in return types or parameter lists. Changes in fields include, for example, changing the default value of a field. Figure 1 shows an example of BCC detected by apidiff in a method of square/picasso (an image downloading library). According to the developer who performed this change, he removed the parameter Context from method with to simplify the API, since this parameter can be retrieved in other ways. apidiff also detects breaking changes involving fields. For example, the tool classified as a BCC the

[2] Currently, apidiff does not detect breaking changes in annotations. Instead, these breaking changes are reported in the fields of the interfaces defining an annotation.

Table 1 BCCs detected by apidiff

Element	BCC
Type	remove type, change in access modifiers, change in supertype, add final modifier, remove static modifier
Method	remove method, change in access modifiers, change in return type, change in parameter list, change in exception list, add final modifier, remove static modifier
Field	remove field, change in access modifiers, change in field type, change in field default value, add final modifier

removal of five public fields of a class called `glprofiler` from `libgdx/libgdx` (a game development framework).[3]

As implemented by the current `apidiff` version, changes in deprecated API elements (i.e., elements annotated with `@Deprecated`) are not BCCs. The rationale is that clients of these elements were previously warned that they are no longer supported, and, therefore, subjected to changes or even to removal. For instance, suppose a method *m* implemented in the first release of a library. In the second release, the owners deprecated this method; and removed it in a third release. Therefore, since the method was deprecated, this last change—in the third release—is not reported as a BCC by `apidiff`. Finally, `apidiff` warns if a BCC is performed in an experimental or internal API [14, 28]. For this purpose, the tool checks if the qualified name of the changed API element includes a package named `internal`, as in this example: `io:reactivex:internal:util:ExceptionHelper`. With this warning, the goal is to alert users that the identified BCC is probably a false breaking change.

As presented in Table 1, `apidiff` does not use the term refactoring to name BCCs. For example, the `rename` of an API element `A` to `B` is identified as the removal of the element `A` from the code. Similarly, a `move class/method/field` from location `C` to a new location `D` is identified as the removal of the element from its original location `C`. In order to use the most appropriate names to identify these operations, we manually inspected the BCCs detected by `apidiff`. For each commit with a BCC, we analyzed its textual diff, as generated by GitHub. The detection of refactorings performed on classes (`rename/move Class`) was facilitated because these operations are automatically indicated in the textual diff computed by GitHub. For example, Figure 2 shows a screenshot of a diff in `facebook/fresco` that includes a `move class`.[4] At the top of the figure, there is an indication that class `DrawableFactory` was moved from package `com:facebook:drawee:backends:pipeline` to package `com:facebook:imagepipeline:drawable`. By contrast, to detect `rename/move method/field` we needed to perform a detailed inspection on the diffs results.

[3] https://github.com/libgdx/libgdx/commit/3eede16

[4] https://github.com/facebook/fresco/commit/f6fe6c3

Fig. 2 Screenshot of a textual diff produced by GitHub in `facebook/fresco`. A `move class` is indicated in the header line.

3 Study Design

3.1 Selection of the Java Libraries

First, we selected the top-2,000 most popular Java projects on GitHub, ordered by number of stars and that not are forks (on March, 2017). We used this criteria because stars is a common and easily accessible proxy for the popularity of GitHub projects [8]. Next, we discarded projects that do not have the following keywords in their short description: *library(ies)*, *API(s)*, *framework(s)*. We also manually removed *deprecated* projects from this list, i.e., projects that have deprecated in their short description, to focus the study on active repositories. These steps resulted in a list of 449 projects. Then, we manually inspected the documentation, wiki, and web pages of these projects to guarantee they are libraries or similar software. As a result, we removed 49 projects. For example, `googlesamples/android-vision` has the following short description: *Sample code for the Android Mobile Vision API*. Despite having the keyword API in the description, this repository is neither a library nor a framework, but just a tutorial about a specific Android API. Thus, the final list consists of 400 GitHub projects, including well-known systems such as `junit-team/junit4` (a testing framework), `square/picasso` (an image downloading and caching framework), and `google/guice` (a dependency injection library).

3.2 Detecting BCCs

During 116 days, from May 8th to August 31th, 2017, we monitored the commits of the selected projects to detect BCCs. To start the study, on May 8th, 2017 we cloned the selected 400 libraries and frameworks to a local repository. Next, on each work day, we ran scripts that use the *git fetch* operation to retrieve the new commits of each repository. We discarded a new commit when it did not modify Java files. Furthermore, on Git, developers can work locally in a change and just submit the new revision (via a *git push*) after a while. Therefore, we also discarded commits with more than seven days, to focus

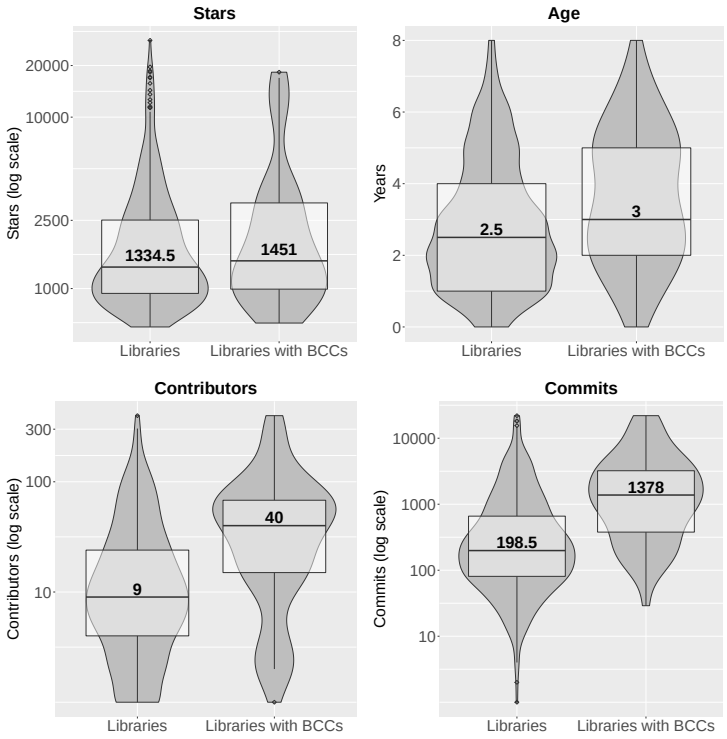


Fig. 3 Distribution of number of stars, age, number of contributors, and number of commits of the initial 400 *Libraries* and of the 61 *Libraries with BCCs*

the study on recent changes, which is important to increase the chances of receiving feedback from developers (see Section 3.3). We also discarded commits representing merges because these commits usually do not include new features; moreover, merges have two or more parent commits, which leads to a duplication of the BCCs identified by apidiff [55, 56]. Finally, we manually discarded commits in branches that only contain test code.

apidiff identified 282 BCCs in 110 commits, distributed over 61 projects (47% of the 130 libraries and frameworks with commits changing public elements, i.e., both commits with BCCs, as listed in Table 2; and with non-BCCs, e.g., adding a new public method). Figure 3 presents the distribution of number of stars, age (in years), number of contributors, and number of commits of the initial selection of 400 libraries and frameworks (labeled as *Libraries*) and of the 61 projects with BCCs (labeled as *Libraries with BCCs*). The distributions of *Libraries with BCCs* are statistically different from the initial selection of 400 libraries in age, number of contributors, and number of commits, but not regarding the number of stars (according to Mann-Whitney U Test, p-value $\leq 5\%$). To show the effect size of this difference, we computed Cliff's delta (or d) [16]. The effect is medium for age, and large for number of contributors and commits. In other words, libraries with BCCs are moderately

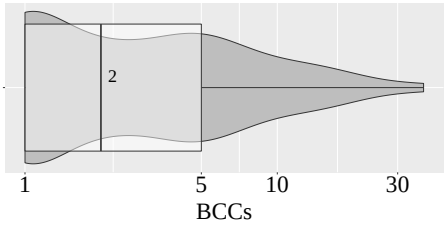


Fig. 4 BCCs per project

Dear [developer name],

I am a researcher working with API usability and evolution. In my research, I am studying the API of [repository/project].

I found that you performed the following changes in this project:

[BCCs list] and [commit links]

Could you please answer the following questions:

1. Why did you perform these changes?

2. Do you agree these changes can break clients? If yes, could you quantify the amount of work to use the new implementation?

3. Why didn't you deprecate the old implementation?

4. Do you plan to document the changes? If yes, how?

Fig. 5 Mail to the authors of commits with BCCs detected by apidiff

older (3 vs 2.5 years, median measures), but have more contributors (40 vs 9) and more commits (1,378 vs 198.5) than the original list of libraries selected for the study. Finally, Figure 4 shows the distribution of BCCs per project, considering only *Libraries with BCCs*. The median is two BCCs per project and the system with the highest number of BCCs is robolectric/robolectric, with 38 BCCs (including 35 BCCs where public API elements were changed to protected visibility).[5]

3.3 Contacting the Developers

Among the 282 BCCs considered in the study, 268 (95%) were detected in commits that contain a public email. Therefore, on each day of the study, after detecting such BCCs, we contacted the respective developers. In the emails sent to them (see a template in Figure 5), we added a link to the GitHub commit and a description of the BCC. Then, we asked four questions. With the

[5] Most cases refer to internal or low-level API (34 occurrences) and one is an accidental change.

Table 2 Numbers about the study design

Days	116
Projects	400
Projects with commits changing public elements	130
Projects with commits and BCCs	61
BCCs detected by apidiff	282
BCCs in commits with public emails	268
Commits confirming/describing BCCs motivations	4
Emails sent to authors of commits with BCCs	102
Received answers	56
Response ratio	55%

first question, we intended to shed light on the real motivation behind the detected changes. With the second question, we intended to confirm whether the BCC detected by apidiff can break existing clients. With the third question, our interest was to understand why the developers have not deprecated the API element where the BCC was detected. Finally, with the last question, our interest was to investigate how often developers document BCCs.

We sent only one email to each developer. Specifically, whenever we detected BCCs by the same developer, but in different commits, we only sent one email to him, about the BCC detected in the first commit. In this way, we reduced the chances that developers perceived our emails as spam. It is also important to mention that before sending each email we inspected the respective commit description to guarantee it did not include an answer to the proposed questions. In the case of six commits, we found answers to the first question (*why did you perform these changes?*). As an example, we have the following commit description:

Lock down assorted APIs that aren't meant to be used publicly subtyped. (D23, Add Final Modi er)

In this message, the developer mentions he is adding a `final` modifier to classes that must not be extended by API clients. We also sent a brief email to the authors of these six commits, just asking them to confirm that the detected BCCs can break existing clients; we received two positive answers. Finally, in two commits we found a message describing the motivation for the change and confirming that it is a breaking change. As an example, we have this answer:

Now, [Class Name] can be con gured to apply to di erent use cases . . . Breaking changes: Remove [Class Name] (D22)

During the 116 days of the study, we sent 102 emails and received 56 responses, which represents a response ratio of 55%. Table 2 summarizes the numbers and statistics about the study design phase, as previously described in this section. After receiving all emails, we analyzed the answers using thematic analysis [17], a technique for identifying and recording *themes* (i.e., patterns) in textual documents. Thematic analysis involves the following steps: (1) initial reading of the answers, (2) generating a first code for each answer, (3) searching for themes among the proposed codes, (4) reviewing the themes to

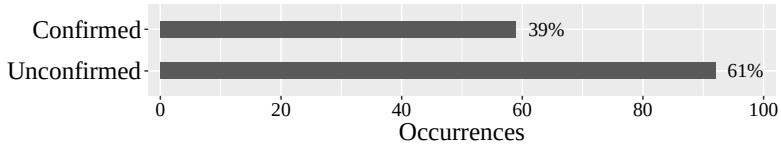


Fig. 6 Confirmed and unconfirmed BCCs; confirmed BCCs are called BCs

find opportunities for merging, and (5) defining and naming the final themes. Steps 1 to 4 were performed independently by two authors of this paper. After this, a sequence of meetings was held to resolve conflicts and to assign the final themes (step 5). When quoting the answers, we use labels D1 to D60 to indicate the respondents (including four developers with answers coming from commits).

4 Results

4.1 How Often do Changes Impact Clients?

To answer this question, we first define breaking changes:

De nition: BCCs confirmed by the surveyed developers are named Breaking Changes (BC).

As presented in Figure 6, regarding the 151 BCCs with developers' answers, only 59 (39%) are classified as BCs. The remaining BCCs—which have not been confirmed by the respective developers—are called unconfirmed BCCs. Next, we characterize the BCs investigated in this study; we also reveal the reasons for the high percentage of unconfirmed BCs.

Breaking Changes (BC): The 59 BCs detected in the study are distributed over 19 projects and 24 commits, including 20 commits with BCs confirmed by email and 4 commits with BCs declared in the commit description. Figure 7 shows the most common BCs. Among the Top-5, three are refactorings, including `move method` (11 occurrences), `rename method` (8 occurrences), and `move class` (8 occurrences). The second most common BCs are the removal of an entire class (10 occurrences), which can be viewed as a drastic API change. The third most popular BCs are `changes in method parameters` (9 occurrences). Considering the 17 types of BCs detected by `apidiff` (see Table 1), only 8 appeared in our study. Regarding the elements affected by the changes, Figure 8 shows the BCs grouped by API element: 35 BCs (59%) are performed on methods, followed by BCs on types (21 instances, 36%) and fields (3 instances, 5%). These three instances refer to changes in the types of fields of an interface used to define an annotation.[6]

[6] In the case of fields, breaking changes occur mainly when changing the type or default value of a public field (or when removing it).

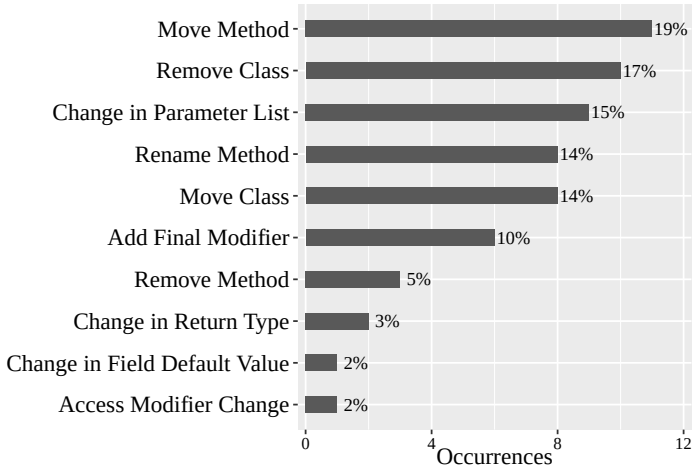


Fig. 7 Most common breaking changes

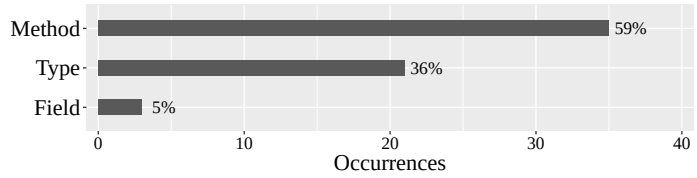


Fig. 8 Most common breaking changes per API element

Summary: The most common BCs are due to refactorings (47%); most BCs are performed on methods (59%).

Unconfirmed BCCs: By contrast, in the case of 92 changes (61%), the surveyed developers did not agree they have an impact on clients. We organized the reasons mentioned by these developers on two major themes: internal APIs and experimental branches/new releases. Regarding the first theme, `apidiff` gives a warning about APIs that are likely to be internal; specifically, the ones implemented in packages containing the string `internal`, as recommended in the related literature [14, 28]. Nonetheless, 21 developers mentioned that the BCCs occurred at internal (or low-level) APIs that do not include `internal` in their names, as in the following answers:

This method is used internally, though it was public. We don't expect people using this method in their applications. (D30)

This could potentially break but this class is used internally as utility and not intended to be used by library users. (D32)

The second cause of unconfirmed BCCs are due to experimental branches. As described in Section 3.2, we monitored all branches of the analyzed reposi-

Table 3 Why do we break APIs?

Motivation	Description	Occur.
new feature	BCs to implement new features	19
api simplification	BCs to simplify and reduce the API complexity and number of elements	17
Maintainability	BCs to improve the maintainability and the structure of the code	14
bug fixing	BCs to fix bugs in the code	3
other	BCs not fitting the previous cases	6

tories to contact the developers just after the changes. Consequently, in some cases, we considered BCCs in branches that do not represent major developments, e.g., branches dedicated to experiments, etc. Ten developers mentioned that the BCC occurred in such branches, as in the following answer:

This is a early extension of [Project Name] to support Java 9 modules. Thus, the code is neither stable nor complete. (D42)

Summary: Most unconfirmed BCCs are related to changes in internal or low-level APIs or in testing branches.

4.2 Why do Developers Break APIs?

As reported in Table 3, we found four distinct reasons for breaking APIs: New Feature, API Simplification, Improve Maintainability, and Bug Fixing. In the following paragraphs, we describe and give examples of each of these motivations.

New Feature. With 19 instances (32%), the implementation of a new feature is the most common motivation to break APIs. As examples, we have the following answers:

The changes in this commit were just a setup before implementing a new feature: chart data retrieval. (D01)

The changes were adding new functionality, which were requested on GitHub by the users, but to avoid unnecessary duplications I had to change the method name to better re ect what the method would be doing after the changes. (D13)

In the first answer, D01 moved some classes from packages, before starting the implementation of a new feature. Therefore, clients should update their `import` statements, to refer to the new class locations. In his answer to the second survey question, D13 mentioned the clients requested a new functionality to configure buttons, which are components instantiated by a class from

the library. The methods were renamed to better reflect the purpose of this new feature.

API Simplification. With 17 instances (29%), these BCs include the removal of API elements, to make the API simpler to use. As examples, we have these answers:

We can access the argument without it being provided using another technique. (D03, Change in Parameter List)

This method should not accept any parameters, because they are ignored by server. (D08, Change in Parameter List)

We are preparing for a new major release and cleaning up the code aggressively. (D09, Remove Class)

In the first two answers, D03 and D08 removed one parameter from public API methods. In the third answer, D09 removed a whole class from the API, before moving to a new major release. In these three examples, the API became simpler and easier to use or understand. However, existing clients must adapt their code to benefit from these changes.

Improve Maintainability. With 14 instances (24%), BCs performed to improve maintainability, i.e., internal software quality aspects, by means of refactoring operations (rename, move, extract, etc) are the third most frequent ones. As examples, we have the following answers:

Because the old method name contained a typo. (D15, Rename Method)

Make support class lighter, by moving methods to Class and Method info. (D24, Move Method)

In the first answer, D15 renamed a method to fix a spelling error, while in the second answer, D24 moved some methods to a utility class to make the master class lighter.

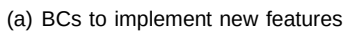
Bug Fixing. In the case of 3 BCs (5%), the motivation is related with fixing a bug, as in the following answers:

The iterator() method makes no sense for the cache. We can not be sure that what we are iterating is the right collection of elements. (D05, Remove Method)

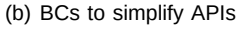
The API element could cause serious memory leaks. (D12, Change in Parameter List)

In the first answer, D05 removed a method with an unpredicted behavior in some cases. In the second answer, D12 removed a flag parameter related to memory leaks.

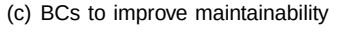
Other Motivations. This category includes six BCs whose motivations do not fit the previous cases. For example, BCs performed to remove deprecated dependencies (2 instances), BCs to adapt to changes in requirements and speci-



(a) BCs to implement new features



(b) BCs to simplify APIs



(c) BCs to improve maintainability

Fig. 9 Top-3 most common BCs, grouped by motivation

cation (2 instances), BCs to eliminate trademark con icts (1 instance), and one BC with an unclear motivation, i.e., we could not understand the speci c answer provided by the developer.

Summary: BCs are mainly motivated by the need to implement new features (32%), to simplify the API (29%), and to improve maintainability (24%).

Figure 9 shows the top-3 most common BCs due to Feature Addition, API Simpli cation, and to Improve Maintainability. move class is the most common BC when implementing a new feature, with 7 occurrences. Speci cally, when working on a new major release, developers tend to start by performing structural changes in the code, which include moving classes between packages. To simplify APIs, developers usually remove classes (5 instances) and also add a final modi er to methods (4 instances). The latter is considered a simpli cation because it restricts the usage of API methods; after the change, the API methods cannot be rede ned in subclasses, but only invoked by clients. Finally, it is not a surprise that BCs performed to improve maintainability are refactorings. In this case, the three most popular BCs are due to move method (11 instances), rename method (2 instances), and move class (1 instance). Interestingly, move class is also used when implementing a new feature.

Fig. 10 E ort required on clients to migrate

Summary: BCs due to refactorings are performed both to improve maintainability and to enable and facilitate the implementation of new features.

4.3 What Is the E ort on Clients to Migrate?

We also added a question in the survey about the e ort required on clients to adapt to the studied BCs. The goal is to collect their perceptions on this e ort; further interviews can be conducted to check whether these perceptions match the client's perceptions or the real e ort required on clients to migrate to the new versions of the studied APIs. We organized the answers of this survey question in three levels: minor , moderate, or major e ort . Seven developers answered the question. As presented in Figure 10, six developers estimated that the e ort to use the new version is minor, while one answered with a moderate e ort; none of them considered the update e ort as a major one.

For example, developer D04|who moved a class between packages with the purpose of improving maintainability |estimates a minor e ort on clients to use the new version:

Work required should be minor, since it is just a change of a package. (D04, Move Class)

A single developer (D09) answered that a class removal may require a moderate e ort on clients. Indeed, the removed class is very simple, having a single method with a single statement that instantiates an object:

The complexity will depend largely on the size of the project and how they use the library. (D09, Remove Class)

Summary: According to the surveyed developers, the e ort on clients to migrate to the new API versions is minor.

4.4 Why didn't you Deprecate the Old Implementation?

17 developers answered this survey question. As presented in Figure 11, they presented three reasons for not deprecating the elements impacted by the BCs.

Fig. 11 Reasons for not deprecating the old versions

Increase Maintenance E ort. 8 developers mentioned that deprecated elements increase the e ort to maintain the project, as in the following answer:

In such a small library, deprecation will only add complexity and maintenance issues in the long run. (D16)

Minor Change/Impact. Four developers argued that the performed BCs require trivial changes on clients or that the library has few clients, as in the following answers:

Because the x is so easy. (D15)

The main reason is that [the number of] users is small. (D14)

Other motivations include the following ones: library is still in beta (1 developers), incompatible dependencies with the old version (1 answer), and trademark con icts (1 answer). Finally, one developer forgot to add deprecated annotations.

Summary: Developers do not deprecate elements a ected by BCs mostly due to the extra e ort to maintain them.

4.5 How do Developers Document Breaking Changes?

This question was answered by 18 developers. Among the received answers, 14 developers stated they intend to document the BCs. We analyzed these answers and extracted seven di erent documents they plan to use to this purpose (see Figure 12). Release Notes and Changelogs are the most common documents, mentioned by four developers each, followed by JavaDoc (3 developers). Finally, four developers do not plan to document the BCs. For example, two of them considered the changes trivial and self-explained.

Summary: BCs are usually documented using release notes or changelogs.

Fig. 12 How do you plan to document the detected BCs?

5 Stack Over ow Study

In this second study, we mine questions on Stack Over ow to better understand the most common problems faced by clients due to breaking changes. Speci - cally, this Stack Over ow analysis provides a complementary view: while the Firehouse Study focused on API developers, this study has its focus on ques- tions stated by API clients on Stack Over ow regarding breaking changes. Essentially, in the rst study, we revealed the reasons for breaking changes performed by owners in their APIs; in this second study, we aim to assess another perspective: whether API clients are passive to breaking changes or active in the sense they look for solutions to deal with these changes. In addi- tion, this second study expands the analysis to other ecosystems to investigate whether client reactions are restricted to Java or also happen in other software ecosystems. Therefore, we propose four new research questions:

1. Who are the authors of questions about breaking changes? With this ques- tion, our goal is to con rm that most questions about breaking changes posted on Stack Over ow come from developers who have been impacted by broken APIs (these developers are the main target of this second study, as mentioned before).
2. What are the most common questions about breaking changes? With this question, we have two goals. First, we intend to con rm that client de- velopers usually resort to Stack Over ow to nd strategies to overcome breaking changes. Second, we also intend to reveal other questions related to breaking changes posted on Stack Over ow.
3. Which ecosystems have more questions about breaking changes? Since our rst study focused on Java, our goal with this question is to investigate whether (and how often) breaking changes happen in other ecosystems.
4. Which documentation types are mentioned in posts about breaking changes? With this question, we aim to check whether developers refer to documenta- tion artifacts when posting questions or answers on Stack Over ow related to breaking changes.

Fig. 13 Example of Stack Over ow question about breaking changes

Section 5.1 presents our methodology to select Stack Over ow questions related to breaking changes. Section 5.3 to Section 5.5 detail the study results, by presenting answers for the proposed research questions.

5.1 Selecting Stack Over ow Questions

Stack Over ow is the de facto question & answer platform for software development [2, 40]. As in October 2018, it hosts over 14M questions and 19M answers covering a broad set of topics, helping more than 50 million developers to learn and share their knowledge[7]. Each question on Stack Over ow can receive several answers, and the community is responsible for evaluating the quality of the proposed answers, giving a positive or a negative score [40]. Similarly, questions also have a reputation, so users can indicate a question as favorite and give positive or negative score as well. When a user marks a question as favorite, he/she starts to receive noti cations about updates in it; therefore, this is a way to show interest on the topic. In addition, Stack Over- ow managers can close bad or incomplete questions[8]. Figure 13 presents an example of a Stack Over ow question about a class removed from AutoMapper, a library to map objects in .NET.[9] It has 11 positive votes and three

[7] https://stackoverflow.com/company

[8] https://meta.stackoverflow.com/questions/tagged/closed-questions

[9] https://stackoverflow.com/questions/35233989

favorite votes. The best answer has 14 positive votes. In his answer, an user recommends to use a new interface, as illustrated by a provided source code example.

We use the dataset provided by Stack Exchange[10] to mine Stack Over ow questions related to breaking changes. Besides weekly updated data, this infrastructure includes an online tool to run T-SQL queries. As a rst step, we run a script to select all questions with the term \breaking change(s)" on the Title or Body elds (on August, 2018); this process resulted in 1,574 questions. From this rst selection, we removed (i) questions without favorite votes and questions with score less than or equal to zero, (ii) questions without at least one answer with score greater than zero, and (iii) that have been frozen by SO curators. In summary, our goal was to clean the initial set of questions to keep only the most important and also the ones that follow the Stack Over- ow guidelines. This process resulted in 352 questions and 881 answers. We use labels Q001 to Q352 to indicate the questions, and labels A001 to A881 to refer to the developers answers.

Next, the rst author of this paper manually inspected the 352 questions to remove false positives, i.e., questions that do not focus on breaking contracts. For instance, we removed question Q181[11], which despite of using the term \breaking changes", the developer is asking for instructions to list les changed under Mercurial revision control system:

So there was a new branch created where we made some breaking changes to the codebase. Now we are going to merge, but before that I want to get a list of all the les that were changed in the branch. How can I get a list of les? (Q181)

As a second example of false positive, we removed questions where the user classi ed his problem as caused by breaking changes (in the question), but it was a misleading classi cation. For example, in question Q121[12], the developer asks about a possible breaking change in the .NET platform:

I recently upgraded my website from ASP.NET MVC3 (Razor) to MVC4 (Razor2), and in doing so found what seemed like a breaking change in the Razor view engine... Is this a known and documented change in the Razor view engine?(Q121)

Another user answered that the reported behavior is not indeed due to a breaking change but to a bug:

This is indeed a bug we decided not to x, note that the syntax is incorrect as there is no transition between C# and markup in this case. (A423)

Finally, we also removed questions referring to breaking changes in programming languages and compilers, since they are outside the scope of this

[10] https://data.stackexchange.com/help
[11] https://stackoverflow.com/questions/8929130
[12] https://stackoverflow.com/questions/15543841

Fig. 14 Numbers of favorite votes, score, and views of the 110 Stack Over ow questions

paper. Speci cally, we removed 13 questions. Interestingly, most cases involve C/C++ (6 occurrences, 46%). As an example, we removed the following question, which includes a discussion about the GCC compiler.[13]

The above code works ne in GCC 4.4.7 and 7.1 and later. It gives an error in GCC 4.5.4 and later releases. So my question is why was this breaking change introduced in GCC? (Q322)

As a result, we manually removed 242 questions, resulting in 110 questions covering breaking changes in di erent programming languages and ecosystems. Figure 14 presents the distribution of number of favorite votes, scores, and views of the 110 selected questions. The median is two favorite votes per question, with a minimum of 1 and a maximum of 179 votes. The score ranges from 1 to 192, and the median is 6. Q017[14] is the question with the highest number of favorites and scores. In this question, a developer asks about breaking changes in the .NET platform. Finally, the number of views ranges from 35 to 48,030, and the median is 1,590. Question Q004[15] is the most viewed one, and it includes a discussion about backward compatibility in Internet Explorer (IE). Finally, the rst author and second authors of this paper analyzed and classi ed each selected question, using thematic analysis (the same technique described in Section 3.3) to provide answers to the four proposed research questions.

5.2 Who are the authors of questions about breaking changes?

Figure 15 shows the distribution of the studied questions, considering two types of authors: clients and owners of the broken interfaces. To identify whether an author is API an owner, we carefully considered the question's context. In cases we were not able to infer the author category, we stated it as unclear. For example, we classi ed the author of Q0621[16] as an owner because he explicitly

[13] https://stackoverflow.com/questions/48854954
[14] https://stackoverflow.com/questions/1456785
[15] https://stackoverflow.com/questions/19638981
[16] https://stackoverflow.com/questions/2678744

Fig. 15 Who are the authors of questions about breaking changes?

Table 4 What are the most common questions about breaking changes?

Question	Occur.	(%)	Clients	Owners	
how to overcome breaking changes	49		45	49	0
how to organize and manage dependencies	17		15	5	12
do new version have breaking changes	14		13	11	3
how to avoid breaking changes	9		8	0	9
how to deal with deprecated elements	6		5	4	2
general Discussions	15		14	1	6

mentions he isdesigning a REST API:

I am designing a REST API for a web application. I want to clearly version the API, so that the interface can be changed in the future without breaking existing services. (...) . (Q062)

As expected, most questions (64%) come from clients, i.e., from developers impacted by the breaking changes. However, almost 29% of the questions are posted by the owners of the broken code, i.e., by developers directly or indirectly responsible for breaking the interfaces. Although our goal is to study clients impacted by breaking changes, we also decided to include in our analysis the questions and answers posted by owners (but they are usually discussed separately). Finally, it is also important to highlight that we were not able to infer the author category in the case of eight questions (7%).

Most Stack Over ow questions about breaking changes come from developers impacted by these changes (64%). However, we also found questions posted by the owners of the broken code (29%).

5.3 What are the most common questions about breaking changes?

As reported in Table 4, we found six main groups of questions about breaking changes in Stack Over ow. We discuss these questions and give examples in the following paragraphs. We also point the number of questions related by

the di erent pro les (owners and clients). [17]

How to overcome breaking changes? In 49 questions (45%), developers essentially ask how to overcome a breaking change found in an API or other system they are using. In some questions, they mention the documentation about the breaking change is incomplete or missing. Therefore, they ask how to overcome the detected breaking change. As an example, in question Q126[18] the developer exposes how hard is it to migrate from Hibernate version 5.0 to 5.1:

I've recently updated Hibernate from 5.0 to 5.1 and the SchemaExport API has changed. The migration docs mention this change, but do not explain how to use the newer API. Moreover, I have not been able to nd any other supporting sample to x the breaking change. (Q126)

As a second example, in question Q080[19], the developer asks about a problem when updating from Angular 1.2 to 1.3:

I am trying to set up a decorator for my controllers... I have it con gured to work in Angular 1.2.x, but there are some breaking changes from 1.3.x onwards that is breaking the code. (Q080)

Out of 49 questions in this category, 38 questions (78%) have at least one accepted answer (i.e., an answer that actually helped the developers to deal with the issue). We manually analyzed these answers and classi ed them into three main themes: Design Changes , Idiom Changes , and Others . First, Design Changes (21 occurrences, 55%) relates to cases where a major design change in the API is propagated to clients. For example, the accepted answer A657[20] mentions a fundamental design change in the way that permissions are checked in Android version 6.0:

As of Android 6.0, permission behaviour has changed to runtime. To use a feature that requires a permission, one should check rst if the permission is granted previously. ... If permission isn't granted or it is rst time, a request for permission should be made. (A657)

Next, Idiom Changes (13 occurrences, 34%) includes answers recommending minor idiomatic modi cations on clients in order to adapt to breaking changes, e.g., adding or removing parameters, con guration options or usage properties. As an example, the accepted answer A847[21] details a solution that involves only adding a new parameter on clients to x the issue:

Adding the following to OnModelCreating in ApplicationDbContext.cs xed the problem for me .... (A847)

[17] The users pro le are unclear in 8 questions classi ed as general discussions.
[18] https://stackoverflow.com/questions/35993598
[19] https://stackoverflow.com/questions/32442605
[20] https://stackoverflow.com/questions/32151603#32151901
[21] https://stackoverflow.com/questions/45725330#45743449

Finally, in the case of four questions (11%) is was not clear the category of the change suggested in the accepted answer.

Do new version have breaking changes? In 14 questions (13%), developers use Stack Over ow to discuss the costs and risks to migrate to new versions, due to possible breaking changes. In these questions, the ultimate goal is to quantify the e ort required by the migration and therefore better support their decision on whether to migrate (or not) to a new version. For instance, in question Q086,[22] the developer asks about possible breaking changes in .NET:

My application is based on .NET 4.0 and EF 4. I'm now looking at upgrading to the latest versions. Are there any breaking changes or behavioral di erences that may adversely a ect my application? How easy is the upgrade path? Does upgrading to EF 5 require any code changes or other work? Are there any new features related to code- rst that would be worth upgrading for? (Q086)

As a second example, a developer uses question Q109[23] to discuss possible breaking changes in a new NHibernate version, a framework to mapper objects in .NET:

What kinds of considerations are there for migrating an application from NHibernate 1.2 to 2.0? What are breaking changes vs. recommended changes? Are there mapping issues? (Q109)

Interestingly, in three questions the developers look for tools and techniques to detect breaking changes. As an example, in question Q156,[24] a developer asks for a tool to check the compatibility between assemblies on .NET :

I'm looking for a tool that will be able to automate the process of verifying the backward compatibility between .NET assemblies. (Q156)

How to avoid breaking changes? In 9 questions (8%), the developers|of them owners|ask for help on how to maintain backward compatibility, when evolving their systems. For instance, a developer opened question Q072[25] to discuss error handling strategies. He mentions a plan to change a method's signature by adding a new exception; however, as this exception represents a breaking change, he is looking for alternative solutions:

That would mean adding a new Exception to the method's signature and that would be a breaking change to the Java API. We would like to have a more robust solution that would not result in breaking changes. (Q072)

How to organize and manage dependencies? Assuming that breaking changes in some cases are inevitable, 17 developers (15%) ask for solutions on

[22] https://stackoverflow.com/questions/12137939
[23] https://stackoverflow.com/questions/27243
[24] https://stackoverflow.com/questions/8279299
[25] https://stackoverflow.com/questions/19315263

how to organize and manage multiple versions (we were also able to con rm that 12 of such developers are owners). As an example, we present question Q042[26], where a developer asks for help on handling dependencies to Go libraries:

In Golang, we can specify open source libraries on GitHub as dependencies. This will try to look for a branch based on your Go version and default to master if I understand correctly. So there is no way to import a speci c release of a dependency. What is the best practice to manage dependencies in Go then? Is it to create new modules for major versions with breaking changes? (Q042)

In another example (question Q162[27]), the developer asks for a suggestion on to avoid npm packages with broken contracts:

How can I install the most recent package that does not have breaking changes? (Q162)

How to deal with deprecated elements? In six cases (5%), developers ask questions related to deprecated elements. For example, in question Q075[28], a developer asks for a replacement to a deprecated element in Angular:

I'm updating my unit tests for Angular2 RC5 (...) The changelog notes the following breaking change: addProviders is deprecated, use TestBed.con gureTestingModule instead. But that now throws an error (...) (Q075)

Q340[29] (an API owner) asks a very speci c questions related to breaking changes and deprecated elements in C macros:

I'm trying to handle a breaking change in a library gracefully. I want users to have a nice, clear warning whenever they use an old macro, so it will be clear that they need to migrate their code to using the new macro. (Q340)

General Discussions. 15 questions (14%) are basically general discussions and comments about breaking changes. For example, a developer opened a post Q017[30] with a list of breaking changes that can impact .NET clients (and asked other users to comment, extend, or revise his list). This post|entitled A de nitive guide to API-breaking changes in .NET|has 193 upvotes, 180 favorite votes, and it has been visualized more than 20K times:

I would like to gather as much information as possible regarding API versioning in .NET/CLR, and speci cally on how API changes do or do not break client applications. (Q017)

[26] https://stackoverflow.com/questions/30300279
[27] https://stackoverflow.com/questions/39533030
[28] https://stackoverflow.com/questions/38985159
[29] https://stackoverflow.com/questions/44642302
[30] https://stackoverflow.com/questions/1456785

Fig. 16 Which ecosystems have more questions about breaking changes?

Despite the opposite answers reported in the rehouse interviews, breaking changes tend to have an important impact on API clients. For example, more than 40% of the studied Stack Over ow questions from developers asking how to overcome speci c breaking changes. Other posts refer to management and organization of multiple versions (15%), general discussions and commentaries (14%), and the prevalence of breaking changes in new releases (13%).

5.4 Which ecosystems have more questions about breaking changes?

As reported in Figure 16, most studied questions are about breaking changes in three ecosystems: .NET, Javascript, and Java. We reinforce, however, that these results are very speci c to the collected SO dataset and they are not representative of the real state of a airs.

The ecosystem with more questions is .NET (54 questions, 49%), covering breaking changes on several .NET frameworks and tools. Q259[31] show an example, a developer asks tips to handle breaking changes on Entity Framework, an object-relational mapper on this plataform, because he had a problem when migrating from version 5 to 6:

\I'm in the process of migrating a code base from the Entity Framework 5 to 6... EF6 contains breaking changes that include removing classes out of System.Spatial over to System.Data.Entity.Spatial "... Anyone with any suggestion?" (Q259)

JavaScript is the second ecosystem with the highest number of questions (24 questions, 22%); interestingly, most of these questions refer to Angular (10 questions), a MVC-based front-end framework which recently evolved to a new release with major design changes. This category also includes questions about npm, a popular package manager for JavaScript. The third most mentioned ecosystem is Java (11 questions, 10%); covering questions about

[31] https://stackoverflow.com/questions/19614954

Fig. 17 Which documents are mentioned in posts about breaking changes?

breaking changes on Java frameworks, as well as in the Android apps. Other questions involve ecosystems with few occurrences, such as Ruby on Rails (two questions), R (one question), and Go (one question) as well as questions where the developers did not point the underlying ecosystem.

Despite the higher concentration of questions in the .NET Platform, we cannot|and is not our goal to|assure this ecosystem has more breaking changes than others; further analysis should be done in this direction.

5.5 Which documents are mentioned in posts about breaking changes?

In only 13 questions (12%), Stack Overflow users refer to official documents in their posts to confirm breaking changes or to recommend migration strategies, as presented in Figure 17. These references include an official website (5 questions, 38%), migration guides (4 questions, 31%), release notes and changelog files (both with two questions and 15%, each one). In the other cases, the questions are answered by using code snippets and/or by providing links to other non-official documents, such as questions in other Q&A platforms (e.g., Google Groups) or to specific commits. For example, in question Q004[32] the user refers to a problem he is facing in an Internet Explorer (IE) plug-in:

> It appears that `typeof (window.ActiveXObject)` results in `undefined`, whereas in IE10 mode, it results in `function (:::)`. Does anybody know why this changed, or where I can find a list of these types of differences between IE10 and IE11 so that I can figure out what other breaking changes there are? (Q004)

Another developer answered by linking to Internet Explorer's API website:

> You can't use that check for IE11: <link to doc> ... window.ActiveXObject property is hidden from the DOM. (This means you can no longer use the property to detect IE11.) (A496)

These documents are also among the ones mentioned by API developers in the rehouse interviews (Section 4.5). However, the different contexts do not allow a comparison of both studies. Mostly because the rehouse interviews

[32] https://stackoverflow.com/questions/19638981

focus on API owners' answers, and the Stack Over ow study includes questions reported by both owners and clients. Furthermore, the results presented in Figure 17 refer to di erent programming languages and ecosystems (e.g, Java, .NET, JavaScript), while the rst study focused only on Java APIs.

The most common documents reported by Stack Over ow users to con rm breaking changes or to recommend migration strategies are websites (5 questions, 38%) and migration guides (4 questions, 31%).

6 Implications

This section presents the study implications to language designers, tool builders, researchers, and practitioners.

Language DesignersIn the rst study, among the 151 Breaking Changes Candidates (BCCs) with developers' answers, only 59 were classi ed as true Breaking Changes (BCs). The other BCCs are mostly changes in internal or low-level APIs or changes performed in experimental branches. Since they are designed for internal usage only, developers do not view changes in these APIs as BCs. However, previous research has shown that occasionally internal APIs are used by external clients [9, 12{14, 18, 23, 28]. For example, clients may decide to use internal APIs to improve performance, as a workaround for bugs, or to bene- t from undocumented features. This usage is only possible because internal APIs are public, as the o cial and documented ones; and their usage is not checked by the Java compiler. To tackle this problem, a new module system was recently introduced in Java (since version 9), which allows developers to explicitly declare the module elements they want to make available to external clients. The Java compiler uses these declarations to properly encapsulate and check the usage of internal APIs. Therefore, our rst study reinforces the importance of this new module system, since we con rmed that changes in internal API elements are frequent. We also con rmed that API developers use the public keyword in Java with two distinct semantics (\public only to my code" vs \public to any code, including clients").

Tool Builders: apidiff is an useful tool both to API developers and clients. API developers can use the tool to document changes in their APIs, e.g., to automatically generate changelogs or release notes. API clients can also rely on apidiff to produce these documents, in order to better assess the e ort to migrate to API versions that are not properly documented. In fact, in the Stack Over ow study, we identi ed two questions where developers ask for tools similar to apidiff . However, these questions refer to .NET platform.

Researchers: Although based on a limited number of 59 BCs, the rehouse interviews reveal opportunities to improve the state-of-the-art on API design, evolution, and analysis. First, the study suggests that BCs are often moti-

vated by the implementation of new features and that refactorings are usually performed at that moment, to support the implementation of the new code. In other words, the implementation of new features does not only extend the API with new elements, it may require changes in the signature of existing ones. In fact, a recent study on refactoring practices considering all types of GitHub projects, i.e., not restricted to libraries and frameworks, also shows that refactoring is mainly driven by the implementation of new requirements [51]. Therefore, we envision a new research line on techniques and tools to recommend refactorings and related program redesign operations, when a new version of an API is under design. In other words, the focus should be on API-specific remodularization techniques, instead of global remodularization approaches, as commonly proposed in the literature [1,3,33,42,52]. Second, the study suggests that BCs are also motivated by a desire to reduce the number of API elements or reduce the possible usages of some elements (e.g., by making them final). Therefore, we envision research on API-specific static analysis tools (or API-specific linter tools), which could for example recommend the removal of useless parameters in API methods (as we found in two BCs), the insertion of a final modifier (as we found in six BCs) or even the removal of underused methods and classes (as we found in two BCs). The benefit in this case would be the recommendation of these changes at design time or during early usage phases, before the affected API elements gain clients and the change costs and impact increase. Third, the answers of the last survey question show that some API developers might be reluctant to use the deprecation mechanism provided by Java. Essentially, they argue that deprecation increases maintenance burden, by requiring updates on multiple versions of the same API element. Therefore, we also envision research on new and possibly lightweight mechanisms to API versioning. It is also possible to recommend the traditional mechanism only in special cases, particularly when the BCs might impact a large number of clients or require complex changes. Fourth, the StackOverflow study suggests that researchers should also consider the impact of breaking changes in other ecosystems, e.g., .NET and JavaScript.

Practitioners: Our studies also provide actionable results and guidelines to practitioners, especially to API developers. First, we detected many unconfirmed BCCs in packages that do not have the terms *internal* or *experimental* (or similar ones) in their names. We recommend the usage of these names to highlight to clients the risks of using internal and unstable APIs. Second, the study also reveals that some BCs are caused by trivial programming mistakes, e.g., parameters that are never used. Since APIs are the external communication ports of libraries and frameworks, it is important they are carefully designed and implemented. Third, we listed good practices followed by developers to document BCs, for example, changelogs and release notes, which were also confirmed in the StackOverflow study.

7 Threats to Validity

External Validity. As usual in empirical software engineering, our findings are restricted to the studied subjects and cannot be generalized to other scenarios. Nevertheless, we daily monitored a large dataset of 400 Java libraries and frameworks, during a period of 116 days (almost 4 months). During this time, we questioned 102 developers about the motivations of breaking changes right after they had been performed. Due to such numbers, we consider that our findings are based on representative libraries, which were assessed during a large period of time, with answers provided by developers while the subject was still fresh in their minds. Moreover, our analysis is restricted to syntactical breaking changes, which result on compilation errors in clients. BCs that modify the API behavior without changing its signature, as studied by Mostafa et al. [36] and Mezzetii et al. [32], are outside of the scope of this paper. We also can not guarantee the 59 BCs considered in the first study are part of new releases. However, the chances of the reverting such changes are small, since they were confirmed as BCs by the survey developers, which also estimated their impact on clients. Besides that, our follow up study focusing on the client perspective only comprises Stack Overflow questions and the English idiom. However, there are a significant number of studies conducted on Stack Overflow [2, 34, 41, 57], since it is a popular Q&A platform, having a community with more than 50M developers and different profiles.

Internal Validity. First, we use apidiff to detect breaking changes between two versions of a Java library. Although this tool was implemented and used in our previous research [55], an error on its result would introduce false positives in our analysis. To mitigate this threat, we considered the breaking changes provided by the tool as candidates and only assessed those confirmed by their developers, which represents 39% of BCCs (see Sections 4.1). Second, we reinforce the subjective nature of this study and its results. As discussed in Section 3.3 and Section 5.1, a thematic analysis was performed to elicit the reasons that drive API developers to introduce BCs, to identify the recurrent themes faced by clients due to breaking changes on Stack Overflow questions, and to classify the users in clients or owners. Although this process was rigorously followed by two authors of the paper, the replication of this activity may lead to a different set of reasons. To alleviate this threat, special attention was paid during the sequence of meetings held to resolve conflicts and to assign the final themes. Third, against our belief, the trustworthiness and correctness of the responses in the survey study is also a threat to be reported. To mitigate it, we strictly sent emails in no more than few days after the commits. This was important to guarantee a higher response rate and reliable answers, once the modifications were still fresh on developers' minds.

Construct Validity. The first threat relates to the selection of the Java libraries. As discussed in Section 3.1, we automatically discarded, from the top-2,000 most popular Java projects on GitHub, the ones that do not have the following keywords in their short description: library(ies), API(s), frame-

work(s). Next, we manually discarded those that, although containing such words, do not actually represent a library. Since this process is conservative in providing a reliable dataset of projects that are libraries, we can not guarantee that we retrieved the whole set of actual libraries from the 2,000 projects. Second, our results stand on the agreement of developers on the detected BCCs. As observed in Section 4.1, most developers pointed out that the detected changes refer to internal or low-level APIs, mentioning that it is unlikely that they could break clients. However, previous research has shown that occasionally internal APIs are used by external clients [9, 12{14, 18, 23, 28]. Therefore, we might have excluded BCCs that could actually impact clients, but we decided to follow the conservative decision of only considering BCCs perceived by developers as having a high potential to break existing clients. Third, we discarded commits with more than seven days from our analysis, inspired by similar decisions in other Firehouse Interview studies [29, 51]. In such studies, this strategy is commonly adopted to increase the chances of receiving answers. As a downside, it discards from the analysis pull-requests including long and lengthy discussions, before being accepted. Fourth, despite we mined the complete Stack Over ow dataset, we only inspected questions with the term breaking changes(s) on the elds Body or Title . Then, we manually investigated all questions to discard the ones not necessarily focusing on broken contracts. In addition, we focused on questions with good evaluation by the community (favorite votes and scores). Consequently, we can not guarantee that we retrieved the whole set of Stack Over ow questions about breaking changes, however, the ones analyzed do represent the most relevant according to the community.

8 Related Work

We organized related work in four subsections: (a) studies about breaking changes in APIs; (b) eld studies using the rehouse interview method; (c) eld studies exploring Q&A websites; and (d) other studies on API evolution.

8.1 Studies on Breaking Changes

In a previous short paper, we report a preliminary study to reveal the reasons of API breaking changes in Java [56]. In this rst study, we also use apidiff to detect breaking changes. We contacted the principal developers of 49 libraries, asking them about the reasons of all breaking changes detected by apidiff in previous releases of these libraries. By contrast, in this new study we contacted the precise developers responsible by a breaking change, right after it was introduced in the code; and we asked them to reveal the reasons for this speci c breaking change. Furthermore, to identify breaking changes, we monitored all commits of a list of 400 Java libraries, during 116 days. As a consequence of the distinct methodologies, in the rst study we received valid answers of

only seven developers (while in the present study we received 56 answers). From these seven answers, we extracted five reasons for breaking changes: API Simplification, Refactoring, Bug Fix, Dependency Changes, and Project Policy. The first four are also detected in the present study. However, the major reason for breaking changes reported in the present study (New Feature) was not detected in the preliminary one.

In another related study [55], we investigate breaking changes in 317 real-world Java libraries, including 9K releases and 260K client applications. We show that 15% of the API changes break compatibility with previous versions and that the frequency of breaking changes increases over time. Using data from the BOA ultra-large dataset [20], we report that less than 3% of the breaking changes impact clients. To reach this result, we considered all breaking changes detected by *apidiff*. However, in the present paper, we found that only 39% of the BCCs are viewed by developers as having a major potential to break existing clients.

Kula et al. [27] focus on the impact of API refactoring on client systems, by analyzing the versions of eight popular libraries (*guava*, *httpclient*, *javassist*, *jdom*, *joda-time*, *log4j*, *slf4j*, and *xerces*). In this empirical study, Japi-cmp[33] library is used to compute the differences between two library versions, while *ref-finder* is used to detect refactoring actions [43]. Among their major results, the authors show that 75% of the refactoring operations break client applications, and that breaking changes are more likely to happen on internal APIs. In our study, we found that the most common reason to break contracts was also refactoring (47%) and most breaking changes were classified as internal by the surveyed developers (61%). However, the authors did not contact developers to understand the reasons behind the breaking changes nor to confirm whether they were indeed true positives. Dig and Johnson [19] studied API changes in five frameworks and libraries (Eclipse, Mortgage, Struts, Log4J, and JHotDraw). They report that more than 80% of the breaking changes in these systems were due to refactorings. By contrast, using a large dataset of 400 popular Java libraries and frameworks, we also found that BCs are usually related to refactorings, but at a lower rate (47%). Moreover, we listed two other important motivations for breaking changes: to support the implementation of new features and to simplify and reduce the number of API elements. Bogart et al. [7] conducted a study to understand how developers plan, negotiate, and manage breaking changes in three software ecosystems: Eclipse, R/CRAN, and Node.js/npm. After interviewing key developers in each ecosystem, they report that a core value of the Eclipse community is long-term stability; therefore, breaking changes are rare in Eclipse. R/CRAN values snapshot consistency, i.e., the newest version of every package should be always compatible with the newest version of every other package in the ecosystem. Once snapshot consistency is preserved, breaking changes are not a major concern in R/CRAN. Finally, breaking changes in Node.js/npm are viewed as necessary for progress and innovation. In the interviews, the participants also

[33] https://github.com/siom79/japicmp

mentioned three general reasons for breaking changes: technical debt (i.e., to improve maintainability), to fix bugs, and to improve performance. The first two motivations appear in our study, but we did not detect breaking changes motivated by performance improvements. However, these answers should be interpreted as general reasons for breaking changes, as perceived by the interviewed developers. By contrast, in our study the goal was to reveal reasons for specific breaking changes, as declared by developers right after introducing them in the source code of popular Java libraries and frameworks.

8.2 Studies using Firehouse Interviews

A firehouse interview is one that is conducted right after the event of interest has happened [47]. The term relates to the difficulty of performing qualitative studies about unpredictable events, like a fire. In such cases, researchers should act like firemen after an alarm; they should rush to the firehouse, instead of waiting the event to be concluded to start their research. In our study, the events of interest are API breaking changes; and firehouse interviews allowed us to collect the reasons for theses changes right after they were committed to GitHub repositories. In software engineering research, firehouse interviews were previously used to investigate bugs just fixed by developers [37, 38], but using face-to-face interviews with eight Microsoft engineers. Silva et al. [51] were the first to use firehouse interviews to contact GitHub developers by email. Their goal was to reveal the reasons behind refactorings applied by these developers; in this case, they also used a tool to automatically detect refactorings performed in recent commits. They sent e-mails to 465 developers and received 195 answers (42% of response ratio). Mazinanian et al. [29] used a similar approach, but to understand the reasons why developers introduce lambda expressions in Java. They sent emails to 351 developers and received 97 answers (28% of response ratio). In our study, we contacted 102 developers and received 56 answers (55% of response ratio).

8.3 Studies on Stack Overflow

Stack Overflow is a popular question and answer platform for software developers, with more than 14M questions and 19M answers (as in October, 2018). It covers distinct topics, from software tools and programming languages, to software architecture and code design. Beside that, the community is diverse; there are software developers with different ages and skills [5, 34, 53]. For these reasons, several studies assess the data provided by this platform to understand development practices [4–6, 41, 57].

In the context of software libraries, Ahasanuzzaman et al. [2] provided a method to classify topics related APIs on Stack Overflow, while Wang et al. [53] combined techniques to detect topics related API design. Zhang et al. [57] focused on assessing posts reporting API usage. By comparing API examples

from Stack Over ow and patterns extracted from GitHub, the authors show that approximately 30% of the posts contain examples with possible problems (e.g., change in API behavior). Stack Over ow is also used in studies outside of the scope of APIs [4{6, 41]. For instance, Pintoet al. [41] presented an empirical study to assess energy consumption questions. By analyzing approximately 300 questions, they presented ve major themes reported by developers. As in our study, the authors used thematic analysis technique to extract the themes and mined the Stack Over ow dataset using speci c keywords. Baruaet al. [5] presented a method to extract the major themes in Stack Over ow questions. They highlighted the importance of Stack Over ow to better understand relevant topics reported by developers and popular technologies. Among the topics found by the proposed methodology, the authors cited the .NET ecosystem in questions about development platforms. In our study, we also found the .NET platform among the most popular ecosystems related to API breaking changes. Bajajet al. [4] focused on the analysis of web developers questions. Beyer et al. [6] presented a technique to label the posts in seven de ned categories. The authors pointed the relevance of the API change category, which is useful for owners better understanding the clients needs (e.g, improving documentation). In fact, in our Stack Over ow study, some API clients requested migration documents and tips to deal with backward incompatibility.

8.4 Studies on API Evolution

Several studies have been proposed to support API evolution and client developers. Chow and Notkin [15] present an approach where library developers themselves annotate the changed methods with replacement rules. Henkel and Diwan [21] propose a tool that captures and replays API evolution refactorings. Kim et al. [25] support computing di erences between two versions of a system. Nguyen et al. [39] use graph-based techniques to help developers migrate from one library version to another. Other studies focus on extracting API evolution rules from source code. For example, Schafer et al. [50] mine library change rules from client systems, while Dagenais and Robillard [18] suggest API replacements based on how libraries adapt to their own changes. Also in this context, Meng et al. [31] propose a history-based matching approach to support API evolution.

In a large-scale study, Robbes et al. [46] assess the impact of API deprecation in a Smalltalk ecosystem. Recently, the authors also evaluated the impact in the context of the Java programming language [48, 49]. In this study, they found that some API deprecation have large impact on the ecosystem under analysis and that the quality of deprecation messages should be improved. Jezek et al. [24] study 109 Java open-source programs and 564 program versions, showing that APIs are commonly unstable. Raemaekers et al. [44] investigate API stability with the support of four proposed metrics, based on method removal and implementation change. In the context of mobile development, McDonnell et al. [30] investigate stability and adoption of the Android

API. In this study, the authors show that APIs are updated on average 115 times per month, representing a rate faster than clients' update.

Some studies investigate the usage and evolution of internal APIs, i.e., public but unstable and undocumented APIs that should not be used by client applications [12{14, 23, 28]. In this context, Businge et al. [12] study the survival of Eclipse plugins, and classify them in two categories: plugins depending on internal APIs and plugins depending only on o cial APIs. In an extended study [14], the authors present that 44% of 512 Eclipse plugins depend on internal APIs. In addition, the same authors investigate the reasons why developers do use internal APIs [13]. For example, they detect cases where developers do not read documentation (so they are not aware of the risks), but also cases where developers deliberately use internal APIs to bene t from advanced features, not available in the o cial APIs. Mastrangelo et al. [28] show that clients commonly use the internal API sun:misc:Unsafe provided by JDK. Recently, Hora et al. [23] studied the transition of internal APIs to public ones, aiming to support library developers to deliver better API modularization.The authors also performed a large analysis to assess the usage of internal APIs. In our survey, several developers mentioned that the breaking changes happened in public but internal or low-level APIs that clients should not rely on. Notice, however, that the related literature points in the opposite direction: client developers tend to use internal APIs.

9 Conclusion

Libraries and frameworks are key instruments to promote reuse an increase productivity in modern software development. Ideally, software libraries and frameworks should provide stable and backward-compatible APIs to their clients. However, the practice reveals that breaking changes (BCs) are common. In this study, we described a large-scale empirical study (400 libraries, 4-month long period, 282 possible breaking changes, 56 developers contacted by email) to understand why and how developers break APIs in Java. We use a tool named apidiff [34] to detect these possible breaking changes. By using a rehouse interview method, we found that BCs are mainly motivated by the implementation of new features, to simplify the number of API elements, and to improve maintainability. The most common BCs are due to refactorings (47%); regarding the programming elements a ected by BCs, most are methods (59%). According to the surveyed developers, the e ort on clients to migrate to new API versions, after BCs, is minor. We also listed some strategies to document BCs, such as release notes and changelogs.

Besides that, this paper includes a follow up study, focusing on the other protagonist of this story: the developers who depend on components a ected by backward incompatibility. By mining 110 questions on Stack Over ow related to breaking changes in di erent programming languages and ecosystems, we

[34] https://github.com/aserg-ufmg/apidiff

found that the most common topics involve developers asking help to overcome breaking changes, discussions about management and organization of multiple versions, and general topics about breaking changes. The most common ecosystems a ected by BCs are .NET (49%) and JavaScript (22%). Among the 110 questions, 70 (64%) are reported by clients, while 32 (29%) are opened by owners, showing that backward compatibility is a real and challenging issue faced by both the clients and the owners.

Last but not least, we presented an extensive list of empirically-justi ed implications of our study, targeting four distinct audiences: programming languages designers, tool builders, software engineering researchers, and API developers. However, such implications should be viewed and interpreted with care, since they are derived from considering only 59 BCs, and a single programming language (Java). Moreover, the study focusing on the client systems is limited to the assessment of 110 Stack Over ow question.

Further studies about motivations to break APIs can consider other software ecosystems and programming languages (particularly, dynamic languages), and other research methodologies (e.g., semi-structured interviews). Future work may also expand our Stack Over ow study, including a broader set of breaking change terms (e.g.,\broken contract" , \backward incompatibility" , etc) and problems reported on other Q&A platforms (e.g., Quora).

Acknowledgements We thank the 56 GitHub developers who participated in our study and shared their ideas and practices about breaking changes. This research is supported by grants from FAPEMIG, CNPq, and CAPES.

References

1. Abdeen, H., Ducasse, S., Sahraoui, H., Alloui, I.: Automatic package coupling and cycle minimization. In: 16th Working Conference on Reverse Engineering (WCRE), pp. 103{112 (2009)
2. Ahasanuzzaman, M., Asaduzzaman, M., Roy, C.K., Schneider, K.A.: Classifying Stack Over ow posts on API issues. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 244{254 (2018)
3. Anquetil, N., Lethbridge, T.C.: Experiments with clustering as a software remodularization method. In: 6th Working Conference on Reverse Engineering (WCRE), pp. 235{255 (1999)
4. Bajaj, K., Pattabiraman, K., Mesbah, A.: Mining questions asked by web developers. In: 11th Working Conference on Mining Software Repositories (MSR), pp. 112{121 (2014)
5. Barua, A., Thomas, S.W., Hassan, A.E.: What are developers talking about? an analysis of topics and trends in Stack ver ow. Empirical Software Engineering pp. 619{654
6. Beyer, S., Macho, C., Pinzger, M., Penta, M.D.: Automatically classifying posts into question categories on stack over ow. In: 26th Conference on Program Comprehension (ICPC), pp. 211{221 (2018)
7. Bogart, C., Kastner, C., Herbsleb, J., Thung, F.: How to break an API: cost negotiation and community values in three software ecosystems. In: 24th International Symposium on the Foundations of Software Engineering (FSE), pp. 109{120 (2016)
8. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHub repositories. In: 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 334{344 (2016)
9. Boulanger, J.S., Robillard, M.P.: Managing concern interfaces. In: 22nd IEEE International Conference on Software Maintenance (ICSME), pp. 14{23 (2006)

10. Brito, A., Xavier, L., Hora, A., Valente, M.T.: APIDiff: Detecting API breaking changes. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track, pp. 507–511 (2018)
11. Brito, A., Xavier, L., Hora, A., Valente, M.T.: Why and how Java developers break APIs. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 255–265 (2018)
12. Businge, J., Serebrenik, A., van den Brand, M.: Survival of Eclipse third-party plug-ins. In: 28th IEEE International Conference on Software Maintenance (ICSM), pp. 368–377 (2012)
13. Businge, J., Serebrenik, A., van den Brand, M.: Analyzing the Eclipse API usage: Putting the developer in the loop. In: 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 37–46 (2013)
14. Businge, J., Serebrenik, A., van den Brand, M.G.J.: Eclipse API usage: the good and the bad. Software Quality Journal **23**(1), 107–141 (2015)
15. Chow, K., Notkin, D.: Semi-automatic update of applications in response to library changes. In: 12th International Conference on Software Maintenance (ICSM), pp. 359–368 (1996)
16. Cliff, N.: Ordinal methods for behavioral data analysis. Psychology Press (2014)
17. Cruzes, D.S., Dyba, T.: Recommended steps for thematic synthesis in software engineering. In: 5th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 275–284 (2011)
18. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: 30th International Conference on Software Engineering (ICSE), pp. 481–490 (2008)
19. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. In: 22nd International Conference on Software Maintenance (ICSM), pp. 83–107 (2005)
20. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 35th International Conference on Software Engineering (ICSE), pp. 422–431 (2013)
21. Henkel, J., Diwan, A.: Catchup!: Capturing and replaying refactorings to support API evolution. In: 27th International Conference on Software Engineering (ICSE), pp. 274–283 (2005)
22. Hora, A., Robbes, R., Valente, M.T., Anquetil, N., Etien, A., Ducasse, S.: How do developers react to API evolution? a large-scale empirical study. Software Quality Journal **26**(1), 161–191 (2018)
23. Hora, A., Valente, M.T., Robbes, R., Anquetil, N.: When should internal interfaces be promoted to public? In: 24th International Symposium on the Foundations of Software Engineering (FSE), pp. 280–291 (2016)
24. Jezek, K., Dietrich, J., Brada, P.: How Java APIs break - an empirical study. Information and Software Technology **65**(C), 129–146 (2015)
25. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: 31st International Conference on Software Engineering (ICSE), pp. 309–319 (2009)
26. Konstantopoulos, D., Marien, J., Pinkerton, M., Braude, E.: Best principles in the design of shared software. In: 33rd International Computer Software and Applications Conference (COMPSAC), pp. 287–292 (2009)
27. Kula, R.G., Ouni, A., German, D.M., Inoue, K.: An empirical study on the impact of refactoring activities on evolving client-used APIs. Information and Software Technology **93**(C), 186–199 (2018)
28. Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., Nystrom, N.: Use at your own risk: The Java unsafe API in the wild. In: 30th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 695–710 (2015)
29. Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in Java. In: 32nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 85:1–85:31 (2017)
30. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the Android ecosystem. In: 29th International Conference on Software Maintenance (ICSM), pp. 70–79 (2013)

31. Meng, S., Wang, X., Zhang, L., Mei, H.: A history-based matching approach to identification of framework evolution. In: 34th International Conference on Software Engineering (ICSE), pp. 353–363 (2012)
32. Mezzetti, G., Møller, A., Torp, M.T.: Type regression testing to detect breaking changes in Node.js libraries. In: 32nd European Conference on Object-Oriented Programming (ECOOP), pp. 7:1–7:24 (2018)
33. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the Bunch tool. IEEE Transactions on Software Engineering **32**(3), 193–208 (2006)
34. Morrison, P., Murphy-Hill, E.: Is programming knowledge related to age? an exploration of Stack Overflow. In: 10th Working Conference on Mining Software Repositories (MSR), pp. 69–72 (2013)
35. Moser, S., Nierstrasz, O.: The effect of object-oriented frameworks on developer productivity. Computer **29**(9), 45–51 (1996)
36. Mostafa, S., Rodriguez, R., Wang, X.: Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In: 26th International Symposium on Software Testing and Analysis (ISSTA), pp. 215–225 (2017)
37. Murphy-Hill, E., Zimmermann, T., Bird, C., Nagappan, N.: The design space of bug fixes and how developers navigate it. IEEE Transactions on Software Engineering **41**(1), 65–81 (2015)
38. Murphy-Hill, E.R., Zimmermann, T., Bird, C., Nagappan, N.: The design of bug fixes. In: 35th International Conference on Software Engineerin (ICSE), pp. 332–341 (2013)
39. Nguyen, H.A., Nguyen, T.T., Jr., G.W., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to API usage adaptation. In: 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 302–321 (2010)
40. Pinto, G., Castor, F., Liu, Y.D.: Mining questions about software energy consumption. In: Working Conference on Mining Software Repositories (MSR), pp. 22–31 (2014)
41. Pinto, G., Castor, F., Liu, Y.D.: Mining questions about software energy consumption. In: 11th Working Conference on Mining Software Repositories (MSR), pp. 22–31 (2014)
42. Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering **37**(2), 264–282 (2011)
43. Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: 26th International Conference on Software Maintenance (ICSM), pp. 1–10 (2010)
44. Raemaekers, S., van Deursen, A., Visser, J.: Measuring software library stability through historical version analysis. In: 28th International Conference on Software Maintenance (ICSM), pp. 378–387 (2012)
45. Reddy, M.: API Design for C++. Morgan Kaufmann Publishers (2011)
46. Robbes, R., Lungu, M., Röthlisberger, D.: How do developers react to API deprecation? the case of a Smalltalk ecosystem. In: 20th International Symposium on the Foundations of Software Engineering (FSE), pp. 56:1–56:11 (2012)
47. Rogers, E.M.: Diffusion of Innovations, 5th edn. Free Press (2003)
48. Sawant, A.A., Robbes, R., Bacchelli, A.: On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In: 32nd International Conference on Software Maintenance and Evolution (ICSME), pp. 400–410 (2016)
49. Sawant, A.A., Robbes, R., Bacchelli, A.: On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK. Empirical Software Engineering pp. 1–40 (2017)
50. Schäfer, T., Jonas, J., Mezini, M.: Mining framework usage changes from instantiation code. In: 30th International Conference on Software Engineering (ICSE), pp. 471–480 (2008)
51. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of GitHub contributors. In: 24th International Symposium on the Foundations of Software Engineering (FSE), pp. 858–870 (2016)
52. Terra, R., Valente, M.T., Czarnecki, K., Bigonha, R.S.: A recommendation system for repairing violations detected by static architecture conformance checking. Software: Practice and Experience **45**(3), 315–342 (2015)

53. Wang, W., Malik, H., Godfrey, M.: Recommending posts concerning API issues in developer Q&A sites. In: 12th Working Conference on Mining Software Repositories (MSR), pp. 224–234 (2015)
54. Wu, W., Gueheneuc, Y.G., Antoniol, G., Kim, M.: AURA: a hybrid approach to identify framework evolution. In: 32nd International Conference on Software Engineering (ICSE), pp. 325–334 (2010)
55. Xavier, L., Brito, A., Hora, A., Valente, M.T.: Historical and impact analysis of API breaking changes: A large scale study. In: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 138–147 (2017)
56. Xavier, L., Hora, A., Valente, M.T.: Why do we break APIs? first answers from developers. In: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 392–396 (2017)
57. Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M.: Are code examples on an online Q&A forum reliable? a study of API misuse on stack overflow. In: 40th International Conference on Software Engineering (ICSE), pp. 886–896 (2018)