Why and How Java Developers Break APIs

Aline Brito*, Laerte Xavier*, Andre Hora†, Marco Tulio Valente*
*ASERG Group, Department of Computer Science (DCC), Federal University of Minas Gerais, Brazil
{alinebrito, laertexavier, mtov}@dcc.ufmg.br
† Faculty of Computer Science (FACOM), Federal University of Mato Grosso do Sul, Brazil
hora@facom.ufms.br

***Abstract*—Modern software development depends on APIs to reuse code and increase productivity. As most software systems, these 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 report the results of an almost 4-month long field study with the developers of 400 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. After identifying breaking changes, we asked the developers to explain the reasons behind their decision to change the APIs. During the study, we identified 59 breaking changes, confirmed by the developers of 19 projects. 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. We conclude by providing suggestions to language designers, tool builders, software engineering researchers and API developers.**

***Index Terms*—API Evolution, Breaking Changes, Field Study.**

I. Introduction

Software libraries are commonly used nowadays to support development, providing source code reuse, improving productivity, and, consequently, decreasing costs [1]–[3]. For example, there are more than 200K libraries registered on Maven's central repository, a popular package management for Java. They cover distinct scenarios, from mobile and web programming to math and statistical analysis. These functionalities are provided to client systems via *Application Programming Interfaces* (APIs), which are contracts that clients rely on [4]. In principle, APIs should be stable and backward-compatible when evolving, so that clients can confidently rely on them.

In practice, however, the literature shows the opposite: APIs are often unstable and backward-incompatible (e.g., [5]–[8]). A recent study points that 28% out of 500K API changes break backward compatibility, that is, they may cause side effects on client systems [9]. API breaking changes comprise from simple modifications such as the change of a method signature or return type to more critical and dangerous 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.

In this paper, we 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 characterize the most common program transformations that lead to breaking changes. Specifically, we investigate four research questions:

1) *How often do changes impact clients?* 39% of the changes investigated in the study may have an impact on clients. However, a minor migration effort is required in most cases, according to the surveyed developers.
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) *Why don't developers deprecate broken APIs?* Most developers mentioned the increase on maintainability effort as the reason for not deprecating broken APIs.
4) *How do developers document breaking changes?* Most developers plan to document the detected breaking changes, mainly using release notes and changelogs.

By following a firehouse interview method [10], 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%. With the study, we provide the following contributions: (1) to the best of our knowledge, this is the first large-scale field study that reveals 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 provide an extensive list of implications of our study, including implications to language designers, tool builders, software engineering researchers, and practitioners.

Structure of the paper. Section II introduces the tool and approach used to detect breaking changes. Section III details our experiment design, while Section IV presents our results. We discuss the implications of the study in Section V. Section VI states threats to validity and Section VII presents related work. Finally, we conclude the paper in Section VIII.

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

II. APIDIFF TOOL

To detect breaking changes, we use a tool named APIDIFF, which was implemented and used by Xavier et al. [9] in a study about the frequency and impact of breaking changes. Essentially, APIDIFF compares two versions of a library and lists all changes in the signature of public methods, constructors, fields, annotations, and enums. 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 III).

Definition: Changes detected by APIDIFF in public API elements are named Breaking Change Candidates (BCC).

Table I 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 *final* 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.

TABLE I
BCCS DETECTED BY APIDIFF

Element	BCC
Type	REMOVE CLASS, 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

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. Finally, APIDIFF warns if a BCC is performed in an experimental or internal API [11], [12]. For this purpose, the tool checks if the qualified name of the changed API element

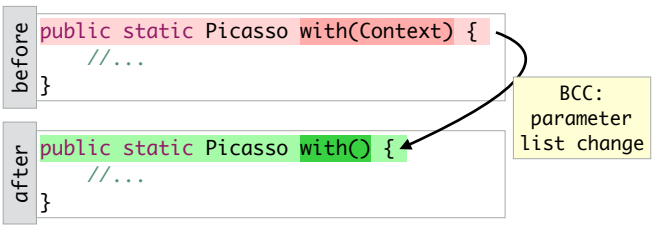


Fig. 1. Example of BCC detected by APIDIFF at method level

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 I, 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.[1] 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.

III. STUDY DESIGN

A. 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 [13]. 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

[1] 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.

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*

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).

B. 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 the study on recent changes, which is important to increase the chances of receiving feedback from developers (see Section III-C). 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 [9], [14]. 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 set of 130 libraries and frameworks with commits detected during the study period). 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) [15]. The effect is medium for age, and large for number of contributors and commits. In other words, libraries with BCCs are moderately 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).

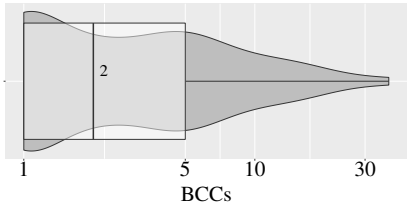


Fig. 4. BCCs per project

C. 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 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.

> 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

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 Modifier)

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 configured to apply to different 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 II 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 [16], 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 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).

TABLE II
NUMBERS ABOUT THE STUDY DESIGN

Days	116
Projects	400
Projects with commits	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%

IV. RESULTS

A. How Often do Changes Impact Clients?

To answer this question, we first define breaking changes:

> *Definition:* BCCs confirmed by the surveyed developers are named Breaking Changes (BC).

As presented in Figure 6, only 59 BCCs (39%) detected by APIDIFF are 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.

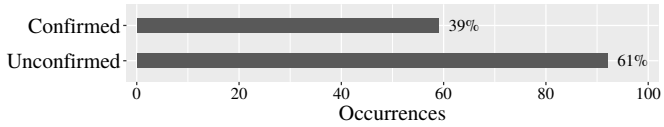


Fig. 6. Confirmed and unconfirmed BCCs; confirmed BCCs are called BCs

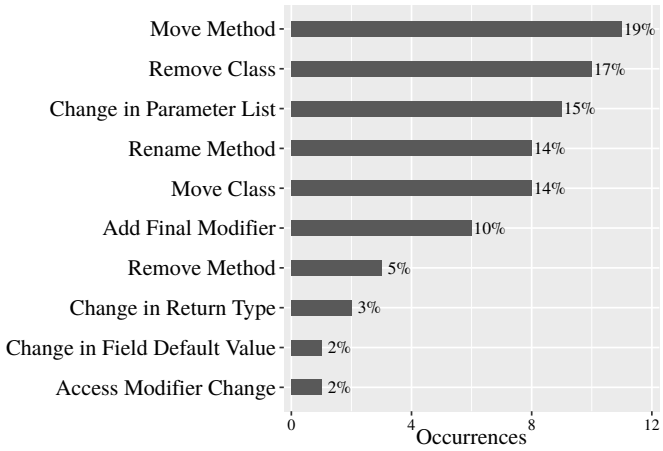


Fig. 7. Most common breaking changes

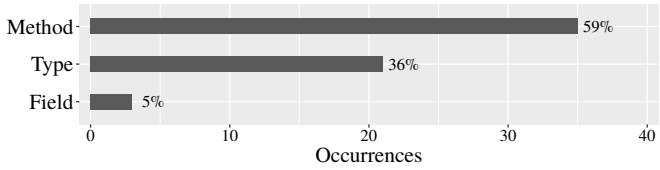


Fig. 8. Most common breaking changes per API element

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 I), 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%).

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 testing 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 [11], [12]. 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 testing branches. As described in Section III-B, we monitored all branches of the analyzed repositories 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., testing branches, 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.

B. Why do Developers Break APIs?

As reported in Table III, 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.

TABLE III
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

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 reflect 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 the second answer, D13 renamed a method to better reflect its purpose after implementing a new feature. The rename should then be propagated to the method calls in the API clients.

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, 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.

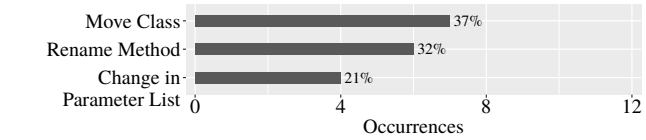
Bug Fixing. In the case of 3 BCs (5%), the motivation is related with fixing a bug, as in the following answers:

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)

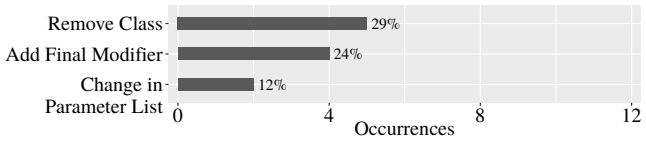
The API element could cause serious memory leaks. (D12, Change in Parameter List)

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.

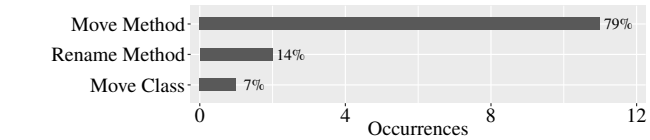
Other Motivations. This category includes six BCs whose motivations do not fit the previous cases. For example, BCs performed to remove deprecated dependencies (2 instances), BCs to adapt to changes in requirements and specification (2 instances), BCs to eliminate trademark conflicts (1 instance), and one BC with an unclear motivation, i.e., we could not understand the specific answer provided by the developer.



(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

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 Simplification, and to Improve Maintainability. MOVE CLASS is the most common BC when implementing a new feature, with 7 occurrences. Specifically, 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` modifier to methods (4 instances). The latter is considered a simplification because it restricts the usage of API methods; after the change, the API methods cannot be redefined 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.

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

C. What Is the Effort on Clients to Migrate?

We organized the answers of this survey question in three levels: *minor*, *moderate*, or *major effort*. Seven developers answered the question. As presented in Figure 10, six developers estimated that the effort to use the new version is minor, while one answered with a *moderate* effort; none of them considered the update effort as a *major* one. For example, developer D04—who moved a class between

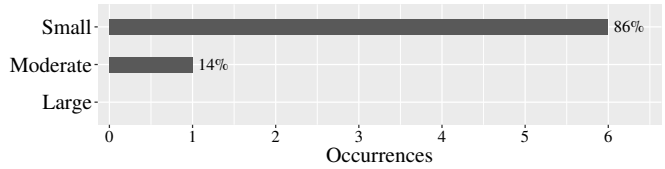


Fig. 10. Effort required on clients to migrate

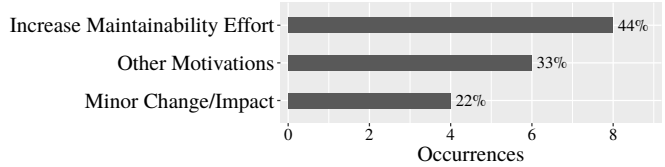


Fig. 11. Reasons for not deprecating the old versions

packages with the purpose of improving maintainability —estimates a minor effort 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* effort on clients:

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 effort on clients to migrate to the new API versions is minor.

D. Why didn't you Deprecate the Old Implementation?

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

Increase Maintenance Effort. 8 developers mentioned that deprecated elements increase the effort 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 fix 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 conflicts (1 answer). Finally, one developer forgot to add deprecated annotations.

Summary: Developers do not deprecate elements affected by BCs mostly due to the extra effort to maintain them.

E. 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 different 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).

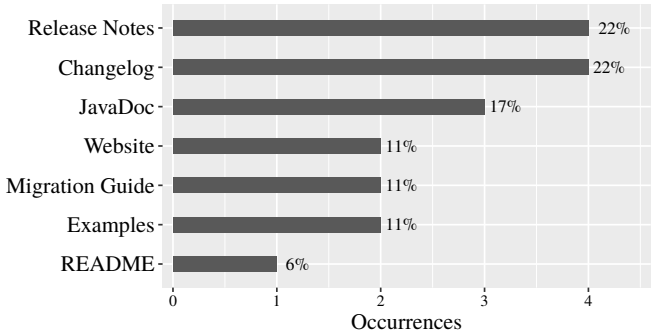


Fig. 12. How do you plan to document the detected BCs?

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.

V. Implications

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

Language Designers: Among the 151 Breaking Changes Candidates (BCCs) with developers' answers, only 59 were classified as 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 [11], [12], [17]–[21]. For example, clients may decide to use internal APIs to improve performance, as a workaround for bugs, or to benefit from undocumented features. This usage is only possible because internal APIs are public, as the official and documented ones; and their usage is not checked by the Java compiler. To tackle this problem, a new module system is being proposed to Java, which will allow developers to explicitly declare the module elements they want to make available to external clients.[2] The Java compiler will use these declarations to properly encapsulate and check the usage of internal APIs. Therefore, our study reinforces the importance of introducing this new module system in Java, since we confirmed that changes in internal API elements are frequent. We

[2]http://openjdk.java.net/projects/jigsaw

also confirmed 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 effort to migrate to API versions that are not properly documented. However, we also faced an important limitation when dealing with the output produced by APIDIFF. Currently, MOVE/RENAME operations are detected as a removal (REMOVE) followed by an addition (ADD) of an API element. As described in Section II, to generate the correct names for these operations, we had to manually inspect the output produced by APIDIFF and the textual diff of the respective commits. Thus, we consider that APIDIFF implementation can follow existing approaches and tools [22]–[24] and automatically detect the cases where REMOVE followed by an ADD is indeed a RENAME (when confined to the same class) or a MOVE (when involving different classes) refactoring.

Researchers: Although based on a limited number of 59 BCs, our study reveals opportunities to improve the state-of-the-art on API design, evolution, and analysis. First, the study suggests that BCs are often motivated by the implementation of new features and that refactorings are usually performed at that moment, to support the implementation of the new code. 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 [24]. 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 [25]–[29]. 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 2 BCs), the insertion of a `final` modifier (as we found in 6 BCs) or even the removal of underused methods and classes (as we found in 2 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 third survey question suggest that BCs may have a minor impact on clients (but according to a small sample of six developers). Thus, we envision further research on migration tools, which could help API clients to move to new API versions by providing recommendations on how to deal with trivial BCs [5], [21], [30], [31]. Fourth, the answers of the last survey question show that some API developers can 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.

Practitioners: The study also provides 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 that they are carefully designed and implemented. Third, most BCs detected in the study require trivial changes in clients, at least according to six surveyed developers. Thus, API developers should carefully evaluate the introduction of BCs demanding complex migration efforts, which can trigger a strong rejection by clients. Fourth, we listed good practices used by developers to document BCs, for example, changelogs and release notes.

VI. 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, usually named Behavioral Backward Incompatibilities [32], are outside of the scope of this paper.

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 [9], 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 IV-A). Second, we reinforce the subjective nature of this study and its results. As discussed in Section III-C, a thematic analysis was performed to elicit the reasons that drive API developers to introduce BCs. 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 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 III-A, 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), framework(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 IV-A, 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 [11], [12], [17]–[21]. 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.

VII. Related Work

We organized related work in three subsections: (a) studies about breaking changes in APIs; (b) field studies using the firehouse interview method; (c) other studies on API evolution.

A. Studies on Breaking Changes

In a previous short paper, we report a preliminary study to reveal the reasons of API breaking changes in Java [14]. In this first 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 specific 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 first 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 [9], 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 [33], 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.

Dig and Johnson [34] 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.* [8] 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 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.

B. Studies using Firehouse Interviews

A *firehouse interview* is one that is conducted right after the event of interest has happened [10]. 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 [35], [36], but using face-to-face interviews with eight Microsoft engineers. Silva *et al.* [24] 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.* [37] 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).

C. Studies on API Evolution

Several studies have been proposed to support API evolution and client developers. Chow and Notkin [38] present an approach where library developers themselves annotate the changed methods with replacement rules. Henkel and Diwan [39] propose a tool that captures and replays API evolution refactorings. Kim *et al.* [40] support computing differences between two versions of a system. Nguyen *et al.* [30] 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.* [41] mine library change rules from client systems, while Dagenais and Robillard [21] suggest API replacements based on how libraries adapt to their own changes. Also in this context, Meng *et al.* [42] propose a history-based matching approach to support API evolution.

In a large-scale study, Robbes *et al.* [6] 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 [43], [44]. 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.* [45] study 109 Java open-source programs and 564 program versions, showing that APIs are commonly unstable. Raemaekers *et al.* [3] 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.* [7] 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 [11], [12], [17], [19], [20]. In this context, Businge *et al.* [19] study the survival of Eclipse plugins, and classify them in two categories: plugins depending on internal APIs and plugins depending only on official APIs. In an extended study [11], 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 [20]. 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 benefit from advanced features, not available in the official APIs. Mastrangelo *et al.* [12] show that clients commonly use the internal API `sun.misc.Unsafe` provided by JDK. Recently, Hora *et al.* [17] 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.

VIII. 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 paper, 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. By using a firehouse 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 affected by BCs, most are methods (59%). According to the surveyed developers, the effort on clients to migrate to new API versions, after BCs, is minor. We also listed some strategies to document BCs, like release notes and changelogs. Last but not least, we presented an extensive list of empirically-justified 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).

Further studies can consider other software ecosystems and programming languages (particularly, dynamic languages); other research methodologies (e.g., semi-structured interviews); and also provide a quantitative and qualitative assessment of the impact of breaking changes in the other protagonists of this story: the developers who depend on APIs affected by breaking changes.

Acknowledgments

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 (process CEX-PPM-00490-17) and CNPq (process 306554/2015-1).

REFERENCES

[1] S. Moser and O. Nierstrasz, "The effect of object-oriented frameworks on developer productivity," *Computer*, vol. 29, no. 9, pp. 45–51, 1996.

[2] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *33rd International Computer Software and Applications Conference (COMPSAC)*, pp. 287–292, 2009.

[3] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *28th International Conference on Software Maintenance (ICSM)*, pp. 378–387, 2012.

[4] M. Reddy, *API Design for C++*. Morgan Kaufmann Publishers, 2011.

[5] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "AURA: a hybrid approach to identify framework evolution," in *32nd International Conference on Software Engineering (ICSE)*, pp. 325–334, 2010.

[6] R. Robbes, M. Lungu, and D. Röthlisberger, "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.

[7] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *29th International Conference on Software Maintenance (ICSM)*, pp. 70–79, 2013.

[8] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "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.

[9] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "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.

[10] E. M. Rogers, *Diffusion of Innovations*. Free Press, 5th ed., 2003.

[11] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Eclipse API usage: the good and the bad," *Software Quality Journal*, vol. 23, no. 1, pp. 107–141, 2015.

[12] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "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.

[13] H. Borges, A. Hora, and M. T. Valente, "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.

[14] L. Xavier, A. Hora, and M. T. Valente, "Why do we break APIs? first answers from developers," in *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 392–396, 2017.

[15] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[16] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 275–284, 2011.

[17] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?," in *24th International Symposium on the Foundations of Software Engineering (FSE)*, pp. 280–291, 2016.

[18] J.-S. Boulanger and M. P. Robillard, "Managing concern interfaces," in *22nd IEEE International Conference on Software Maintenance (ICSME)*, pp. 14–23, 2006.

[19] J. Businge, A. Serebrenik, and M. van den Brand, "Survival of Eclipse third-party plug-ins," in *28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 368–377, 2012.

[20] J. Businge, A. Serebrenik, and M. van den Brand, "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.

[21] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *30th International Conference on Software Engineering (ICSE)*, pp. 481–490, 2008.

[22] S. Kim, K. Pan, and E. J. Whitehead, "When functions change their names: automatic detection of origin relationships," in *12th Working Conference on Reverse Engineering (WCRE)*, pp. 143–152, 2005.

[23] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *14th International Conference on Mining Software Repositories (MSR)*, pp. 1–11, 2017.

[24] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of GitHub contributors," in *24th International Symposium on the Foundations of Software Engineering (FSE)*, pp. 858–870, 2016.

[25] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[26] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[27] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization," in *16th Working Conference on Reverse Engineering (WCRE)*, pp. 103–112, 2009.

[28] N. Anquetil and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *6th Working Conference on Reverse Engineering (WCRE)*, pp. 235–255, Oct 1999.

[29] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, vol. 45, no. 3, pp. 315–342, 2015.

[30] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "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.

[31] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *34th International Conference on Software Engineering (ICSE)*, pp. 826–836, 2012.

[32] S. Mostafa, R. Rodriguez, and X. Wang, "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.

[33] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering (ICSE)*, pp. 422–431, 2013.

[34] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," in *22nd International Conference on Software Maintenance (ICSM)*, pp. 83–107, 2005.

[35] E. R. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *35th International Conference on Software Engineerin (ICSE)*, pp. 332–341, 2013.

[36] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, 2015.

[37] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "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.

[38] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *12th International Conference on Software Maintenance (ICSM)*, pp. 359–368, 1996.

[39] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE)*, pp. 274–283, 2005.

[40] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *31st International Conference on Software Engineering (ICSE)*, pp. 309–319, 2009.

[41] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *30th International Conference on Software Engineering (ICSE)*, pp. 471–480, 2008.

[42] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *34th International Conference on Software Engineering (ICSE)*, pp. 353–363, 2012.

[43] A. A. Sawant, R. Robbes, and A. Bacchelli, "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.

[44] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK," *Empirical Software Engineering*, pp. 1–40, 2017.

[45] K. Jezek, J. Dietrich, and P. Brada, "How Java APIs break - an empirical study," *Information and Software Technology*, vol. 65, no. C, pp. 129–146, 2015.