An empirical study on the impact of refactoring activities on evolving client-used APIs

Raula Gaikovina Kula[*,a,b], Ali Ouni[d], Daniel M. German[c], Katsuro Inoue[a]

[a] Osaka University, Japan
[b] Nara Institute of Science and Technology, Japan
[c] University of Victoria, Canada
[d] Ecole de Technologie Superieure Montreal, Canada

ARTICLE INFO

ABSTRACT

Context: Refactoring is recognized as an effective practice to maintain evolving software systems. For software libraries, we study how library developers refactor their Application Programming Interfaces (APIs), especially when it impacts client users by breaking an API of the library.
Objective: Our work aims to understand how clients that use a library API are affected by refactoring activities. We target popular libraries that potentially impact more library client users.
Method: We distinguish between library APIs based on their client-usage (referred to as client-used APIs) in order to understand the extent to which API breakages relate to refactorings. Our tool-based approach allows for a large-scale study across eight libraries (i.e., totaling 183 consecutive versions) with around 900 clients projects.
Results: We find that library maintainers are less likely to break client-used API classes. Quantitatively, we find that refactoring activities break less than 37% of all client-used APIs. In a more qualitative analysis, we show two documented cases of where non-refactoring API breaking changes are motivated other maintenance issues (i.e., bug fix and new features) and involve more complex refactoring operations.
Conclusion: Using our automated approach, we find that library developers are less likely to break APIs and tend to break client-used APIs when performing maintenance issues.

1. Introduction

Software libraries are constantly evolving, either responding to client needs, patching bug fixes or addressing other maintainability concerns. Refactoring is a controlled and widely-used technique for improving the design of an existing software, especially with modern and large-scale software systems that depend on a large number of third-party libraries. Fowler recommends refactoring to improve software readability and reusability, while increasing the speed at which developers can write and maintain their code base [1,2].

The Application Programming Interface (API) are specifications that govern interoperability between a client application and a library. External APIs refer to the APIs available for client usage. Since clients solely rely on APIs for 'blackbox' access to the library's functionality, API backward compatibility is an important consideration for both client and library developers. Clients migrating to a newer library version would be particularly concerned with whether previously invoked external APIs in an older version will continue to be invoked without error. This is known as preserving API compatibility.[1] Hence, any API change between two library versions that violates this linkage is known as an API breakage. From a library viewpoint, a developer refactoring an external APIs may not consider the effect it has in affecting a client's chances of adopting the latest version. Conversely, negligence to refactor the code base may increase the complexity and maintainability efforts (Lehman's 2nd law), leading up to an eventual degradation in software quality (Lehman's 7th law) [3].

In this work, we conduct an empirical study to explore the relationship between API refactorings and breakages based on actual API usage by clients. We distinguish between library APIs based on their client-usage (referred to as client-used APIs) in order to get a deeper understanding on the extent to which API breakages can be related to refactoring activities. Our investigation covers over 9700 breaking classes and around 12,900 refactoring operations from eight popular Java libraries, with each library having around 10 ∼ 38 consecutive releases. We observe the following: (i) library maintainers are less likely to break client-used APIs compared to other classes of the library, (ii)

---

detected refactoring operations only breaking less than 37% of client-used APIs, qualitatively finding that the (iii) rest (63%) API breakages are motivated by maintenance issues that are likely to involve more complex refactorings. Finally, we find that (iv) simple refactorings (i.e., `move_method`, `rename_method`, `move_field`) were less frequently applied to client-used API classes compared to other classes.

Our main contributions of this paper are three-fold and can be summarized as follows: (1) our study involves the investigation of APIs that are used by actual client, (2) using automated tooling, we conducted a large scale empirical study to investigate API breakages and refactorings and (3) we present a large dataset of API breakages and refactorings which is publicly available as a replication package at: http://sel.ist.osaka-u.ac.jp/people/raula-k/APIBreakage/.

The rest of the paper is organized as follows. Section 2 describes the background and definitions. Section 3 presents our approach we use in the empirical study. Section 4 details the research questions and what method is used in the study. We then show our results in Section 5, with discussion of implications and threats of the study in Section 6. Section 7 surveys related work. Finally, Section 8 concludes the paper and presents future research directions.

2. Basic concepts & definitions

This section provides the necessary background and concepts that are prerequisites to understand the conducted study.

2.1. Backward compatibility of APIs

The precise definition of backward compatibility depends in part on the Java language's notion of binary compatibility[2]:

"binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error."

Importantly, a class or interface should treat its accessible members (method and fields) and constructors, their existence and behavior, as a *contract* with its users.

In this paper, we define that any changes violating this contract are said to cause an API breakage between the library and its client user. We show two examples of API breakages. The first example of an API breakage is when a method name is modified (i.e., renamed or deleted method). For instance, the removal of the method in a class could break the API linkage, resulting in a `NoClassDefFound` exception error to the client application. Conversely, adding parameters (i.e., adding new fields, methods, or constructors) to an existing class or interface usually does not break an API.

The second example of an API breakage is when third-parties cause an API breakage to the library, which then indirectly breaks the client. In many cases, a library is also a client user of other libraries within their environment. For instance, any changes to the library's environment such as an update to the Java Development Kit (JDK) may break a method in the library, and therefore ripples its effect to any client user of this library API.

2.2. Refactoring activities and API breakages

Refactoring is a disciplined engineering practice that restructures an existing code by altering its internal structure without changing its external behavior [1]. Fowler discusses around seventy various refactorings, which can be either simple or become quite complex. In this paper, we determine if any of the API breakages is related to a refactoring activity. Formally, we define a Refactoring Operation (R_{opt}) as an atomic refactoring change applied between two library versions.

2.3. API categorization based on client usage

In this paper, we are interested in the APIs actually used by a client application, assuming that a code change between a client-used API will cause a breakage to that contract between library and client user. To investigate the extent of which developers are breaking their APIs, we must first define the usage dimension of an API. In reality, not all public entities (APIs) are intended for client usage. Based on a developer's intended use, an API of a library can either be `external` or `internal`.

- **External APIs** - are APIs designed by library maintainers for usage by clients.
- **Internal APIs** - are APIs intended only for internal usage by the library code itself.

An internal API may exist for several reasons. For instance, the `Finalizer` class within `base.internal` package of the `google-guava`$_{17.0}$ documented [4]:

While this class is public, we consider it to be *internal* and not part of our published API. It is public so we can access it reflectively across class loaders in secure environments.

In an ideal world, internal APIs are never used by any client. However, in reality internal APIs may be subjected to client usage. For instance, Businge et al. found that a large proportion of plugins used the Eclipse framework internal APIs [5]. Moreover, concepts such as the Application Binary Interfaces (ABIs) [6] and the OSGi framework [7] have been proposed to differentiate between the two API types. However, unless explicitly documented, it is extremely difficult to distinguish between *external* or *internal* APIs.

As shown in Fig. 1, we describe the different class categories of a library. To distinguish between external and internal APIs, we propose a method to approximate external API classes by mining actual usage by clients, defined as *client-used API classes*. Details of the method are explained in the subsequent methodology subsection. All library class categories are defined as follows:

- **API class** - is a class that has at least one public entity (i.e., method and field members) and accessible by any client user.
- **non API class** - is a class that contains no API entities, i.e., private or protected.
- **client-used API class (clientUse)** - is an API class that is used by at least one client. It is an approximation of the external APIs. The set of client-used API classes should ideally cover all external APIs. However, there exist cases when a client uses an internal API. (i.e., *client-used API classes≈external API classes*)
- **non client-used API class (non clientUse)** - is an API class that is not used by any client. The set of all non client-used API classes should cover all internal APIs. (i.e., *non client-used API classes ≃ internal API classes*)

Henceforth, we classify our API breakages at the class-level. Classes are then classified as either:

- **breaking class** - is a changed class that is breaking its API in either class, method or field levels such as rename/move/delete changes.
- **non breaking class** - a changed class that does not affect API compatibility.

We then explore the extent to which breaking changes to client-used API are caused by refactoring activities. As defined in the Section 2, R_{opt} is an atomic refactoring operation applied between two library versions. We now introduce the following terminologies related to R_{opt}:

- **Ref class** - is a changed API class where at least one R_{opt} has been applied to any of its elements. (i.e., field, methods or class

---

[2] documentation at http://docs.oracle.com/javase/specs/jls/se8/html/jls-13.html.
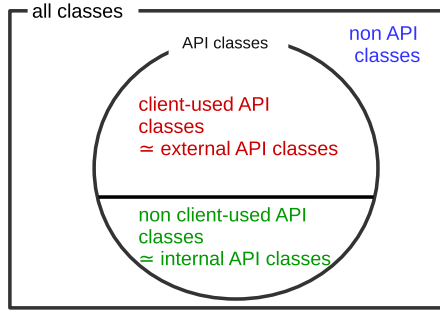
Fig. 1. A conceptual composition of all library class types. The venn diagram shows the relationship between (a) client-used API, (b) non client-used API and (c) non API class types.

attributes).

- **R_{opt} density** - refers to the number of R_{opt} applied per class.

3. Approach

In this section, we first present our case study libraries and methodology used in the empirical study. Our method includes (1) categorization of API based on client usage, (2) API breakage detection and (3) API refactoring detection.

3.1. Subject libraries

We used a systematic method to select our subject libraries. Our selection of these libraries is based on the following criteria: (1) have a large enough client-user API usage and (2) have sufficient evolution history. Additionally, we required diverse libraries that (3) are from different application domains and (4) have been extensively studied in related work. This criteria was used to select libraries from a set of 2500 client projects collected from GitHub.

Table 1 shows all 183 library versions from the eight selected libraries. For each library, we collected 10 to 38 different library versions. All libraries constitute a large client-base and are from different application domains. Moreover, three out of the eight subject libraries were used in prior work [8–10]. The chosen studied libraries range from being testing, logging, utilities and web-based libraries. As shown in the table, we selected GUAVA [11], HTTPCLIENT [12], JAVASSIST [13], JDOM [14], JODA-TIME [15], LOG4J [16], SLF4J [17] and XERCES [18] For all libraries, we only selected consecutive version releases, ignoring release candidates. Only the official binaries and available source code for each library were used in this study.

3.2. Client-used API extraction method

Actual client usage is needed to distinguish between *external* or *internal* APIs. Specifically, we would need to compile each individual client system to know what APIs are used by clients. To enable a large

Table 1
Studied libraries showing the releases range, number of versions, time period, and the range of number of classes per library (min-max).

Library	Release range	#Versions	Releases Time Period	# Classes (min ~ max)
GUAVA	r03 ~ 18.0	22	Apr 10 ~ Aug 14	727 ~ 1763
HTTPCLIENT	4.0 ~ 4.5	25	Aug 09 ~ May 15	230 ~ 460
JAVASSIST	2.5.1 ~ 3.19.0	28	Feb 06 ~ Jan 15	187 ~ 334
JDOM	1.1 ~ 2.0.6	10	Sept 04 ~ Feb 15	73 ~ 258
JODA-TIME	0.95 ~ 2.8	22	No. 05 ~ May 15	191 ~ 246
LOG4J	1.1.3 ~ 1.2.17	17	Jun 01 ~ May 12	242 ~ 974
SLF4J	1.1.0 ~ 1.7.12	38	Dec 06 ~ Mar 15	11 ~ 28
XERCES	1.2.3 ~ 2.11.0	21	Dec 00 ~ No. 10	580 ~ 1652

scale analysis, we use the `jcabi-aether` [19] library and `Java-Compiler` (*ver.1.6*) Eclipse compiler [20] to dynamically compile and log all client-loaded classes. As a result, we are able to extract the fully qualified library class name of all external APIs for many clients.

One of the main challenges to determine client-used API collection is the coverage of all external APIs. Hence, our technique consists of continuously collecting client systems until full coverage is reached (i.e., no more APIs are used). We coin this coverage as the *saturation point* reached for a library version. So instead of trying to compile as many clients are possible, we use the saturation point as a heuristic to show that enough clients have been collected. Fig. 2 and Table 2 shows the saturation point for our case studies. The saturation point is represented as a cumulative count of client-used API classes (x-axis) represented as a function over the number of client projects (y-axis), with the coasting of the curve assuring confidence that a stable number of client-used API classes have been reached. For example, of the 195 collected clients, guava reached a saturation with 98 client systems to cover 184 API classes. It is important to note that each project was selected at random, making the formation of the curve unintentional. The table also summarizes the number of client GitHub projects that we mined for each of the eight subject libraries (total code base size of 600GB). To ensure maturity and quality of the client projects, the projects dataset only includes java projects that had at least 100 commits. We ran experiments for about 30 days. The process of client-used API collection of a single project took between 10 min ~ 3 h.

3.3. API breakage detection method

In recent times, state–of–the–art API breakage detection tools [21–25] have been extensively used by both researchers [26,27] and practitioners [11],[12] alike, especially for a systematic comparison of API checking backward incompatibilities between library versions[3] As noted by Raemaekers [27], these tools are underestimations– as all detected breaking API changes will definitely break an API but some binary compatible APIs could be semantically incompatible.

To identify the API differences between two library binaries, we use the `Japi-cmp` library [28]. Similar to other tools, `Japi-cmp` is able to detect and differentiate changes in instrumented and generated classes to determine binary compatibility as well as public or private accessibility. Using the definitions in Section 2, we then map and label all classes as either breaking or non-breaking. Overall, the resulting dataset consists of over 9700 detected breaking classes from the eight libraries.

3.4. Refactorings detection method

To automatically collect R_{opt} applied between the two versions, we use the state–of–the–art `Ref-Finder` [29] tool. Based on template logic rules, the tool identifies up to different 52 refactoring types between two versions. It is important to note that the collected refactorings are structural, only detectable by mechanical transformations; "*Ref-Finder does not include changes that may either require restricted conditions to be met, or to some degree of additional specification from a developer that could not be automatically inferred by a tool*" [8]. As a result, our dataset consists of 12,900 R_{opt} from all eight libraries.

3.5. Mapping refactorings to API breakages

The study involves a mapping between the collected `Ref` and `breaking` classes, where a `Ref` class contains at least one R_{opt}. Fig. 3 describes this mapping as an intersection between *breaking classes* and *Ref classes*. It is important to note false positives, where the tools detect

---

[3] For instance, developers of the google guava library, use `JDiff` to report changes between two versions, e.g., API changes from guava v18 to v19 are at http://google.github.io/guava/releases/19.0/api/diffs/.
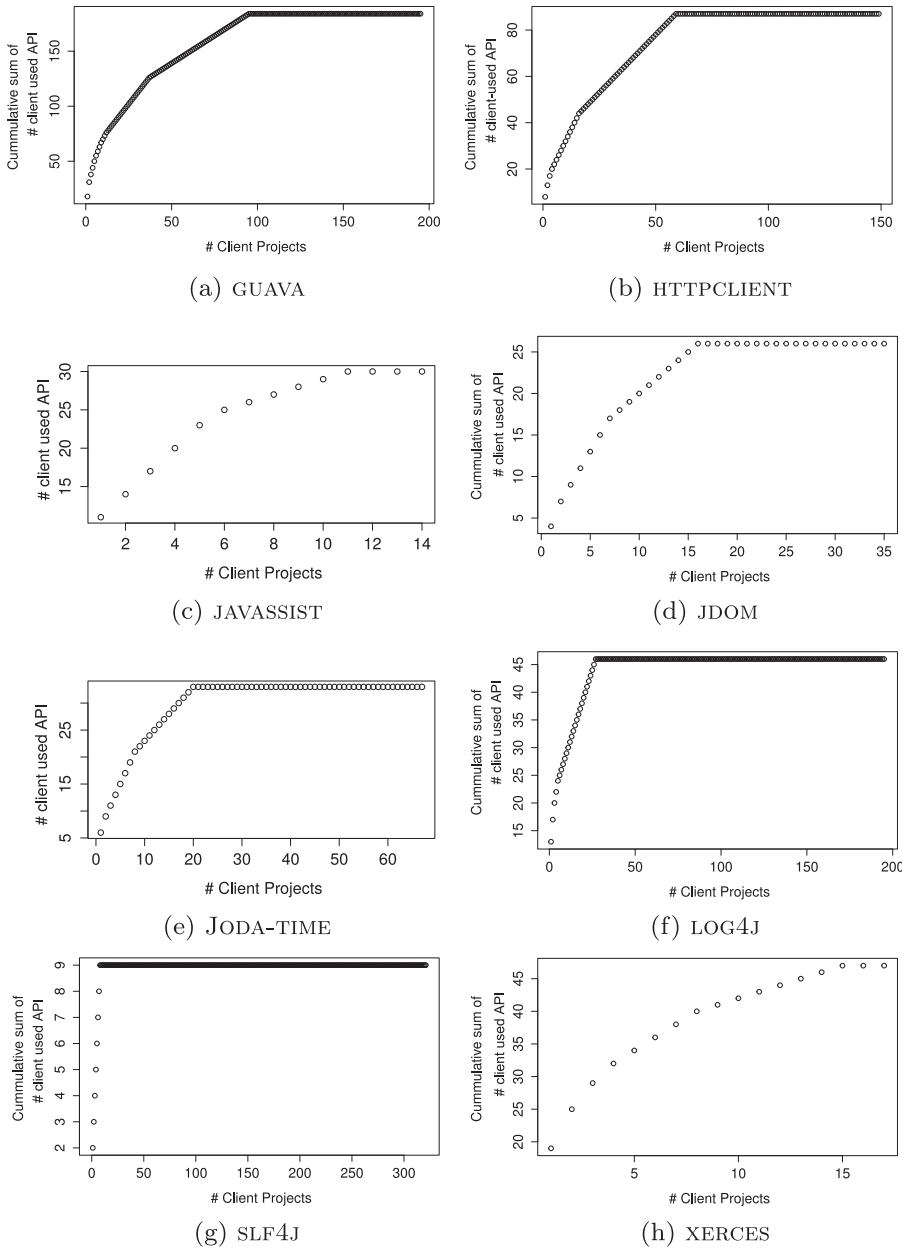
(a) GUAVA

(b) HTTPCLIENT

Fig. 2. Cumulative count of client-used API classes (x-axis) represented as a function over the number of client projects (y-axis). The saturation function (coasting of the curve) indicates that a stable number of client-used API classes classes have been reached (See Table 2).

(c) JAVASSIST

(d) JDOM

(e) JODA-TIME

(f) LOG4J

(g) SLF4J

(h) XERCES

Table 2
Collected client-used API classes as shown in Fig. 2.

	# Collected Clients	At Saturation Point (SP)	
		used clients at SP	client-used API classes at SP
GUAVA	195	98	184
HTTPCLIENT	149	67	87
JAVASSIST	14	11	30
JDOM	35	16	26
JODA-TIME	69	20	27
LOG4J	195	36	46
SLF4J	321	20	9
XERCES	17	15	47
All clients	995		

Fig. 3. Venn diagram of the overlapping relations of refactored and breaking classes.

refactorings in unchanged classes. Upon manual inspection of some cases, we confirmed these were false positives as the classes were unchanged. As a result, we semi-automatically identified and discarded
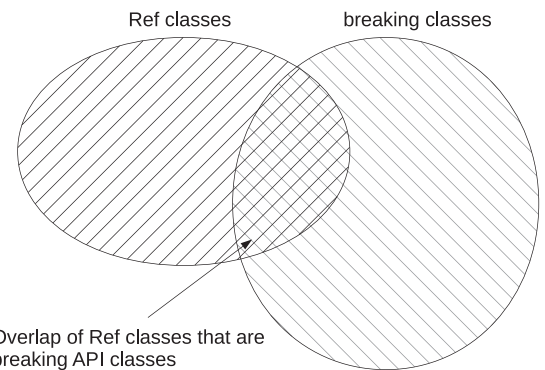
2100 instances of such false positives, finally leaving us with 10,800 R_{opt} from all eight libraries.

A simple example of a refactoring that breaks API can be seen with the `com.google.common.collect.ImmutableMultiset` of the

Table 3
Library Class Categories Incompatibility Matrix.

	Compatible Change	Incompatible Changes ($break_{change}(L_v)$)
CLIENT-USED API	API compatible	API Breaking code change
NON CLIENT-USED API	API compatible	Incompatible change unintended for client
NON API	Not affect client	Incompatible change does not affect client

GUAVA library[4]. According to the API Diff report, the Im-mutableMultiset<E> of(E[]) method (i.e., which takes E[] and returns an immutable multiset) was removed between version 11.0.02 and 12.0. In this example, our approach automatically detects this change as the remove_method R_{opt}. The official Java documentation states that *'deleting a method or constructor from a class breaks compat-ibility with any pre-existing binary that referenced this method or con-structor; a* NoSuchMethodError*may be thrown when such a reference from a pre-existing binary is linked. Such an error will occur only if no method with a matching signature and return type is declared in a super-class'.*

4. Empirical study

In this section, we present the goals and motivation, followed by the method used to address each research question.

4.1. Research questions

Our motivation is to inspect the relationship between refactorings and API breakages. Related, Dig and Johnson [9] manually inspected library release notes for documented API changes to investigate the role of refactoring during API evolution of a library. They cited two reasons why they preferred a manual analysis over the use of automated tools: (1) *'since most API changes follow a long deprecation replace-remove cycle, an obsolete API can coexist with the new API for a long time'* and (2) some behavioral refactoring cases that *'would have been misinterpreted by a tool, but a human expert can easily spot'*. In this study, we find that sta-te–of–the–art tools are now able to detect deprecations, thus negating the first reason. Additionally, we find that the automated approach is not as reliant on documentation.

Our goal in this study is to use an automated approach to investigate how client usage-APIs are affected by the refactoring activities. The automated approach has the benefit of reducing manual inspection and heuristic errors and enables a large-scale empirical study. We designed a rigorous quantitative empirical study, formulating the following re-search questions:

- *(RQ1). To what extent are library maintainers breaking client-used APIs over time?* We want to understand the API breaking tendencies of library maintainers.
- *(RQ2). To what extent are refactoring activities breaking client-used APIs?* Sometimes API breakages are unavoidable, even for the more popular client-used APIs. Prior work indicates that refactoring is common with API changes. Therefore, we want to understand how much of client-used API breakage is related to refactoring activities.

In RQ2, we identified many API breakages not related to refactoring activities. We then formulated RQ3 and RQ4 for a deeper analysis of the detected changes (both refactoring and non refactoring related) that break client-used APIs:

- *(RQ3). What non-refactoring-related code changes are breaking client-*

used APIs?Specifically, our motivation is to understand what API breaking changes are not related to refactorings.
- *(RQ4). What refactoring-related code changes are breaking client-used APIs?*From the perspective of all refactoring activities, we would like to understand (i) how much and (ii) types of refactoring operations that are breaking client-used APIs.

4.2. Research method for RQ1

To answer RQ1, we followed two steps. First, we studied con-secutive versions of a library to understand the library evolution. The goal is to study how (i) client-used API classes, (ii) non client-used API classes and (iii) non API classes evolve over several consecutive ver-sions. Next, we investigate the number of code changes that lead to incompatibility with respect to the different class categories that we defined above. Since the tool is only able to compare two versions at a time, we performed a side-by-side (i.e., each comparison is the current version against the immediate successive library version). We introduce a normalized metric namely $break_{change}$ to describe the rate of the number of breaking changes over all class changes at that version re-lease as defined in Eq. (1):

$$break_{change}(L_v) = \frac{|breaking\ classes|}{|all\ changed\ classes|} \tag{1}$$

where L_v refers to a given library version and ranges from $0 \leqslant break_{change} \leqslant 1$ for each class category of L_v. Values that are closer to 1 indicate that there are more breakages per class changes.

Table 3 shows the $break_{change}(L_v)$ metric interpretation based on the class type. Hence, the $break_{change}(L_v)$ metric has different interpretations based on the class type. For instance, for non API classes, the metric shows significant changes that do not affect clients. We believe that it is important to track which classes are more prone to incompatible code changes. To assess the significance of breakages between the different library class categories, we use the Kruskal Wallis and Mann–Whitney non-parametric test. The null hypothesis would state no statistical dif-ference between the class types. Furthermore, to assess the difference magnitude, we study the effect size based on Cohen's d [30]. The effect size is considered: (1) small if $0.2 \leqslant d < 0.5$, (2) medium if $0.5 \leqslant d < 0.8$, or (3) large if $d \geqslant 0.8$. For the effect size, we use the Man-n–Whitney tests with Bonferroni correction.

4.3. Research method for RQ2

For RQ2, our method is to identify library refactorings that are applied to client-used API classes. We followed two steps. To analyze the impact of the refactoring activities, we first identified for each li-brary the (i) number of Ref classes and (ii) R_{opt} density. We then identified the Ref classes that are breaking. To map refactorings to API breakages as described in Section 3.5, we introduce a normalized metric namely *breaking–to–Ref rate* as Eq. (2):

$$breaking - to - Ref\ (L_v) = \frac{|(Ref \cap breaking\ classes)|}{|Ref|} \tag{2}$$

where L_v refers to a given library version. The metric *breaking − to − Ref rate*(L_v) returns a percentage that ranges from [0..100%] for each class category of L_v. Values that are closer to 100%

---

[4] the API change at http://google.github.io/guava/releases/12.0/api/diffs/changes/com.google.common.collect.ImmutableMultiset.html#methods.

indicate that there are more refactorings that are breaking each of the different class categories. Conversely, from an API breakage perspective, we now introduce a normalized metric namely *Ref–to–breaking rate* to describe the ratio of overlap with respect to all breaking classes as defined in Eq. (3):

$$Ref - to - breaking\ (L_v) = \frac{|(Ref \cap breaking\ classes)|}{|breaking\ classes|} \qquad (3)$$

where L_v refers to a given library version. The metric $Ref - to - breaking rate (L_v)$ returns a score that ranges from [0..1] for each class category of L_v. Values that are closer to 1 indicate that there are more breakages that are related to refactoring activities.

4.4. Research method for RQ3

For RQ3, we used a qualitative approach to investigate the breaking APIs changes that were not detected in our approach as refactoring operations. Results from the prior RQ2 (See Section 5.2, Table 5) indicate that three of the six projects (GUAVA, HTTPCLIENT and XERCES) have many client-used API breakages that were not related to refactoring activities. We consulted related change logs of these three projects; GUAVA,[5] HTTPCLIENT[6] and XERCES[7] to understand the reason why these API breakages were performed by the developer. We manually checked documented change logs of each release to map an API breakage to either a bug fix issue or to accommodate a new enhancement (feature) in the library. To reduce bias, a manual check was carried by a team of three researchers (one postdoctoral and two graduate master students) persons with have an intermediate level of java programming and software development. Since team members do not possess any project-specific knowledge, we solely rely on keywords or issues links (i.e., issueID) in the change log comments to map each API breakage with a bug issue or new features. XERCES was later removed from the analysis as there was too many ambiguous references with no clear linkage to the source code. Analysis will include the aggregation of all API documented changes as either bug fixes or new features and show how many can be mapped to the API breakages that did not involve any refactoring operations.

For a deeper analysis and validation, we will investigate and present some examples of these non-refactoring related API breaking classes.

4.5. Research method for RQ4

For RQ4, we identified what refactoring operations are breaking client-used APIs. We followed two steps in the analysis. For a library, we aggregated the number of *Ref* instances where a certain R_{opt} (e.g., move_method) has been applied. In the second step, we used a normalized metric *prsv* to describe the ratio of overlapped breaking refactorings between client-used API classes and non client-used API classes as defined in Eq. (4).

$$prsv\ (L, R_{opt})$$
$$= \frac{\sum_{L_v \in L} |client - used\ API \cap breaking \cap Ref\ classes|}{\sum_{L_v \in L} |nonclient - used\ API \cap breaking \cap Ref\ classes|} \qquad (4)$$

where L refers to a given library, R_{opt} refers to a certain refactoring operation type.

Our hypothesis is that a *prsv* ratio less than 1 ($0 \leqslant prsv < 1$) indicates that developers are applying less refactoring operations to client-used API classes. Conversely, a high *prsv* ratio ($prsv \geqslant 1$) indicates that more refactoring operations are applied to client-used API classes. A value of 1 indicates that the certain R_{opt} type is equally applied to

both client-used API classes and non client-used API classes.

5. Results

In this section, we present our results of the study by addressing each of the four research questions.

5.1. Findings for RQ1

Figs. 4–6 depict class category analysis of each consecutive library version. Each figure shows the evolution of (i) client-used API classes, (ii) non client-used API classes and (iii) non API classes over consecutive library versions. From these figures, we summarize our findings with three observations (i.e., Figs. 4a–6b). First, we observe that most libraries are composed of non client-used API classes categories (green line), showing that libraries usually have more non client-used API classes than client-used API classes. The exception is LOG4J, which is shown in Fig. 5c to have most APIs intended for external API usage. Interestingly, we see in Fig. 4c that non client-used API classes of JAVASSIST disappears from the more recent libraries. Upon closer inspection, we noticed that this was because developers had changed these non client-used API classes into non API classes. Second, we observe a stable number of client-used API classes (red line) shown across all projects. From a client user viewpoint, the findings indicate that developers of a library are less likely to expand their external APIs. The obtained results show that the number of non API classes (blue line) is constantly changing (i.e., illustrated by various peaks) over time. We find that some of the peak changes can be correlated to different events, such as a major or specially-named releases, beta releases such as xercesImpl2.x.xjaxb and log4j1.3alpha6, or modifying private non API classes into public APIs such as in the case of HTTPCLIENT.

> Library maintainers are less likely to apply client-used API classes changes compared to other class categories.

Fig. 7a shows the $break_{change}$ rates for all eight libraries. From this figure, we observe that except for JAVASSIST and JODA-TIME, library developers are more likely to break non client-used API classes than client-used API classes. Related, Fig. 8 depicts the $break_{change}$ rates grouped by all class categories. The Figure shows that non client-used API classes are more prone to breakages than client-used API classes for all libraries. As shown in the Figure, non client-used API classes are reported to have the most breakages. A Kruskal Wallis test revealed a significant differences between client-used API classes, non client-used API classes and non API classes values ($p < 0.01$). The post-hoc test using Mann–Whitney tests with Bonferroni correction proves the effect size to be medium ($p < 0.01$, $r = 0.54$) when comparing all class categories.

> Findings show that incompatible API code changes are statistically more likely to occur in non client-used API classes compared to client-used API classes.

5.2. Findings for RQ2

Table 4 presents a summary of Ref classes and their R_{opt} density. For instance, we identified 32 GUAVA client-used API classes that were Ref classes. Out of the 32 Ref classes, we report a median of 7 R_{opt} that were applied per Ref class. From this table, we can see that, in general, library maintainers applied more R_{opt} to non client-used API classes and non API classes, as compared to client-used API classes, except for JAVASSIST. For example, the table shows that for XERCES, around

---

[5] and example of Release 11 https://github.com/google/guava/wiki/Release11.

[6] https://archive.apache.org/dist/httpcomponents/httpclient/RELEASE_NOTES-4.5.x.txt.

[7] change logs at https://xerces.apache.org/xerces2-j/releases.html.

(a) GUAVA

(b) HTTPCLIENT

(c) JAVASSIST

Fig. 4. An evolution of changed classes per class types for (a) GUAVA, (b) HTTPCLIENT, (c) JAVASSIST. These figures show the different # of classes identified in chronological order of release.

(a) JDOM

(b) JODA-TIME

(c) LOG4J

Fig. 5. An evolution of # of classes per class types for (a) JDOM, (b) JODA-TIME and (c) LOG4J libraries. Similar to Fig. 4, these figures show the different # of classes identified in chronological order of release.

244 non client-used API classes were refactored, compared to 44 client-used API classes. In more detail, the results show that apart from SLF4J, the median density of R_{opt} per class ranges from 1 to 10 R_{opt} at most. Interestingly, we find that SLF4J had a high number of R_{opt} applied to one non-breaking client-used API classes ($\overline{x} = 22$). LOG4J, SLF4J and JDOM libraries reported only a few breaking classes matched to R_{opt}, which is consistent with recent empirical studies conducted by Cossette et al. [8].

Table 5 reports the median values of R_{opt} that cause API breakages. We use this table to compare between client-used API classes and non client-used API classes. For GUAVA, non client-used API classes ($\overline{x} = 9$) were breaking due to refactorings compared to client-used API classes ($\overline{x} = 2$). From this table, we find that non refactoring changes are more likely to break client-used API classes than non client-used API classes. Moreover, applied refactorings tend to break more non client-used API classes compared to client-used API classes. The results show that many of the API breakages are not mapped to the detected re-factorings (i.e., non Ref classes). We find that more refactoring non

client-used API classes are breaking compared to refactored client-used API classes, with the exception of JAVASSIST.

Table 5 also shows the *breaking–to–Ref* and *Ref–to–breaking* rates. We report that the median *Ref–to–breaking rate* for client-used API classes is up to 37% across all projects ($\overline{x} = 1\% \sim 37\%$). Except for JAVASSIST, the result provides evidence the detected API breakages could not be mapped to refactoring operations. Alternatively, the *breaking–to–Ref rates* reported for client-used API classes in Table 5 indicates that breaking refactorings accounted for a median range of up to 75% of all R_{opt}. The highest *breaking-Ref* rate for non client-used API classes was 86%, reported for the GUAVA library.

> Findings show that up to 75% refactored API classes are breaking their client-used APIs. However, these API breaking refactorings account for less than 37% of all client-used API breakages.
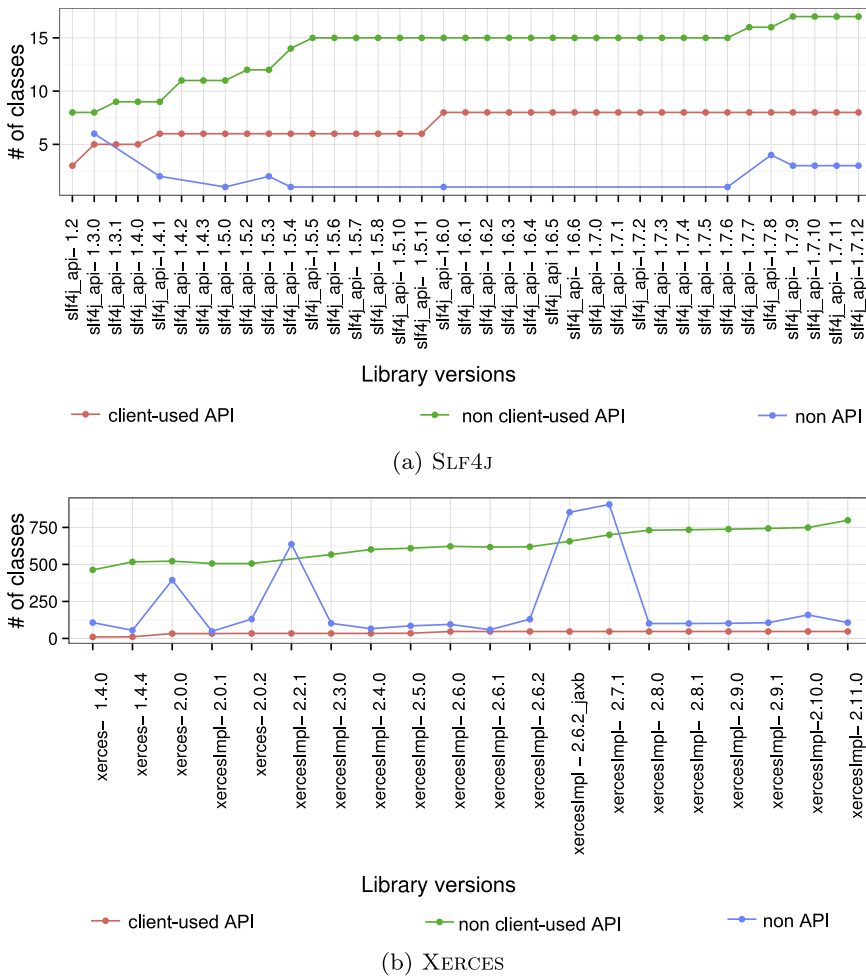
Fig. 6. An evolution of # of classes per class types for (a) SLF4J and (b) XERCES libraries. Similar to Figs. 4 and 5, these figures show the different # of classes identified in chronological order of release.

(a) SLF4J

(b) XERCES

5.3. Findings for RQ3

Table 6 shows results of the manual study of API breakages that did not map to any `Ref Classes` in developer documentation (i.e., change logs). For instance, release 11 of the GUAVA library listed 26 issues and 13 new features.[8] The table confirms our results that find these client-used API class breakages are not only related to refactoring activities (i.e., `non Ref Class`). We find that all four API breakages could be mapped to the API documented changes. From the table, we were able to map 82% of non refactoring client-used API breakages to the API documentation for GUAVA and 58% for HTTPCLIENT. This finding indicates that many of the API breakages not involved in refactorings are most likely motivated by maintenance issues such as bug fixes and for new feature enhancements.

From our manual analysis and similar to a study by Murphy–Hill et al. [31], we find that not all API changes appear in the API change logs. Fig. 9 does show two documented case examples of API breakages that are not mapped to a detected refactoring operation (i.e., `non-Ref`). These examples provide evidence that these many client-used API class breakages are: (a) motivated by a bug fix or new feature or (b) consists of a complex refactoring that is not captured by the automated approach. In the first example (i.e., Fig. 9a, we show an unavoidable API breaking change, especially if it is used to fix a complex defect such as a third party library. This API breaking change was triggered in response to an error reported by a client user *"JDK and Guava*

TypeVariable implementations are no longer compatible under 1.7.0 51-b13".[9] It was widely reported to affect many client users of the library. Developers found that a change in the standard Java library (JDK) causes guava to break API compatibility, as prior guava version implemented an undocumented *internal API* of the JDK (i.e., `Types.TypeVariable.newTypeVariable()`).[10] After much discussion among developers, the accepted API change was documented to *'conditionally work only under the new JDK'*.

In the second example (i.e., Fig. 9b), we acknowledge cases where the automated approach is unable to detect more complex refactoring operations. Soares et al. [32] showed that Ref-Finder is unable to correctly detect all types of refactoring operations, which is a validity threat and is discussed in detail (See Sections 6.3 and Section 6.4). Moreover, this change is listed as a submitted enhancement issue[11] related to *'Move HashCodes static methods to HashCode'* and involves 17 changed files (261 added and 219 deleted lines of code)[12].

> Findings indicate that many client-used API breakages are likely to be motivated by other maintenance issues (i.e., bug fixes and new features) and involve more complex refactoring operations.

---

[8] The release notes are available at https://github.com/google/guava/wiki/Release11.

[9] issue at https://github.com/google/guava/issues/1635 and fix at https://goo.gl/bqDpxU.

[10] A blogger discussions by users is at https://goo.gl/8tcHfY.

[11] https://github.com/google/guava/issues/1495.

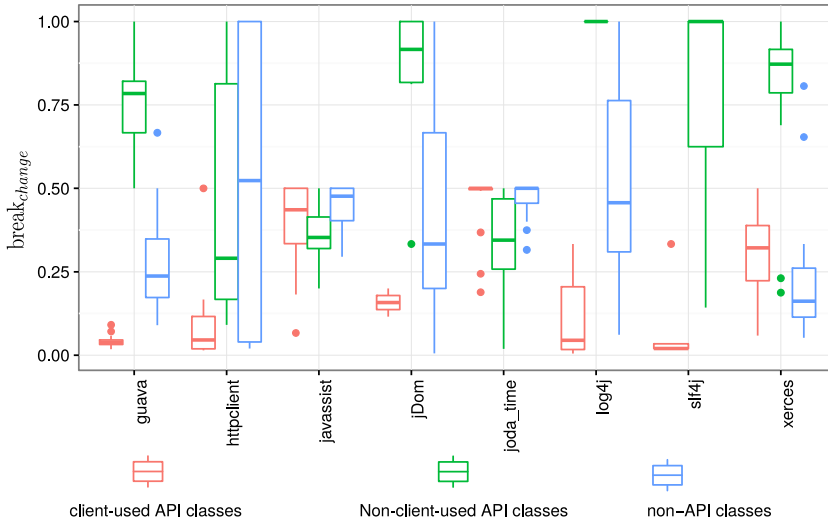[12] the code change is at https://goo.gl/JHVi5J.

Fig. 7. Results of the $break_{change}$ rates for all eight libraries analyzed.

(a) For each of the eight libraries, we show the $break_{change}$ comparing (1) client-used API classes in red, (2) non client-used API classes in green and (3) non API classes in blue.

	# versions	breaking class instances	changed class instances
GUAVA	22	2,215	9,973
HTTPCLIENT	25	113	1,426
JAVASSIST	28	1,017	2,572
JDOM	10	106	445
JODA-TIME	22	1,097	2,922
LOG4J	17	583	3,051
SLF4J	38	21	235
XERCES	21	4,622	7,796
Totals	183	9,774	28,420

(b) Corresponding to Figure 7a, this table shows the # of analyzed (1) library versions, (2) breaking and (3) changed classes collected.
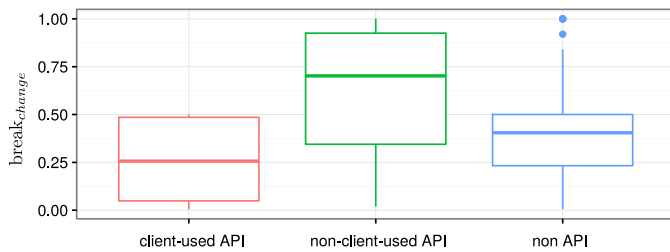

Fig. 8. Summary of $break_{change}$ comparing (1) client-used API classes in red, (2) non client-used API classes in green and (3) non API classes in blue. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.4. Findings for RQ4

Table 7 shows a classification of the applied R_{opt} for the three libraries GUAVA, XERCES, and HTTPCLIENT in our collected dataset. As seen in the table, GUAVA developers applied the `change_parameter` R_{opt} 28 times to breaking client-used API classes. Developers subsequently applied the same R_{opt} and broke 25 non client-used API classes during the library evolution of GUAVA. We find that the GUAVA and XERCES libraries tend to refactor and break their versions during evolution than HTTPCLIENT. Our results align with the findings of Cossette *et al.* [8] on

API transformations, where they also used the same libraries in their experiments. Results in Table 7 show that developers apply specific R_{opt} more frequently when evolving their libraries. For instance, R_{opt} such as `move_method` (GUAVA- 34 R_{opt}), `change_parameter` (HTTPCLIENT- 11 R_{opt}), and `rename_method` (XERCES- 363 R_{opt}) were the most frequently applied that cause API breakages. For client-used API classes, `remove_parameter` (GUAVA- 30 R_{opt}), `move_method` (HTTPCLIENT- 3 R_{opt}) and `rename_method` (XERCES- 146 R_{opt}) are reported as most frequent. Notably, `move_method` (GUAVA- 190 R_{opt}, XERCES- 256 R_{opt}), `remove_parameter` (HTTPCLIENT- 8 R_{opt}) were applied to non client-used API classes.

Table 7 also reports the *prsv* ratio for each library. This metric measures the degree of likelihood to which library developers apply certain R_{opt} to client-used API classes compared to non client-used API classes (i.e., preserving client-used API classes). We use color to highlight the *prsv* scores. Green highlights in the table represents a low preservation of client-used API classes, while the red highlights indicates a high ratio of R_{opt} in non client-used API classes. For example, the library developers of both GUAVA (*prsv* = 0.08) and XERCES (*prsv* = 0.05) tend to apply less `move_method` refactoring operations to client-used API classes. Our results shows library maintainers are less likely to refactor (using the more frequent R_{opt}) client-used API classes than non client-used API classes. For example, 5 out of 10 R_{opt} in GUAVA, 3 out of 5 R_{opt} types in HTTPCLIENT, and 16 out of 17 R_{opt} types in XERCES are less

Table 4
The table reports (a) number of `Ref` classes and (b) R_{opt} density per Ref class ($\bar{x} = |R_{opt}|$). Note that (–) represents no matches.

		breaking classes			non breaking		
		clientUse	non clientUse	non API	clientUse	non clientUse	non API
$|Ref|$	GUAVA	32	143	44	31	24	139
	HTTPCLIENT	3	11	–	6	16	7
	JAVASSIST	111	8	6	29	10	2
	JDOM	1	1	–	–	3	–
	JODA-TIME	30	–	11	29	5	12
	LOG4J	1	–	–	2	–	3
	SLF4J	–	1	–	2	1	–
	XERCES	44	244	31	23	104	66
R_{opt}density (Median)	GUAVA	7	8	5	4	4	8
	HTTPCLIENT	5	5	–	2	3	2
	JAVASSIST	2	5	2	1	1	1
	JDOM	1	1	–	–	4	–
	JODA-TIME	3	–	10	1	1	1
	LOG4J	1	–	–	2	–	2
	SLF4J	–	1	–	3	22	–
	XERCES	8	6	4	4	4	5

Table 5
Matrix that shows the median average # refactored API classes per library. For each library, we summarized the median values across all library versions. Table includes median ($\bar{x}$) of matched refactored classes. 0 represents a value less than 0.01. (–) reports no matched classes.

		# versions	$|Ref|$ (Median)		$|non\ Ref|$ (Median)		Ref–to–breaking rate	breaking–to–Ref rate
			breaking	non breaking	breaking	non breaking		
clientUse	GUAVA	22	**2**	2	166	251	1%	53%
	HTTPCLIENT	25	1	1	46	74	1%	55%
	JAVASSIST	28	2	1	3	1	**37%**	**75%**
	JDOM	10	1	0	1	16	1%	1%
	JODA-TIME	22	15	2	65	45	6%	48%
	LOG4J	17	0	1	87	75	0%	0%
	SLF4J	38	–	1	41	136	0%	0%
	XERCES	21	3	2	24	16	10%	64%
non clientUse	GUAVA	22	9	2	91	20	9%	**86%**
	HTTPCLIENT	25	2	1	8	18	**14%**	58%
	JAVASSIST	28	4	1	2	1	5%	44%
	JDOM	10	1	1	13	3	1%	22%
	JODA-TIME	22	–	1	9	7	–	–
	LOG4J	17	0	–	36	–	0%	–
	SLF4J	38	0	1	2	2	0%	0%
	XERCES	21	14	7	210	22	6%	68%

likely applied to client-used API classes. We find that many high *prsv* ratios (depicted by red in the table) where by the rarely applied R_{opt} types (e.g., `remove_control_flag` (GUAVA- 5 R_{opt}, HTTPCLIENT- 2 R_{opt}) and `pull_up_method` (HTTPCLIENT- 3 R_{opt}, XERCES- 7 R_{opt}).

> Findings show that library maintainers were more likely to refactor non client-used API classes compared to client-used API classes.

6. Discussion

In this section, we first discuss the implications of results and then compare with related work. We then discuss some challenges of our approach and finally present threats to the validity of our study.

6.1. Implications

Our results indicate that when evolving libraries, out of all code changes applied, maintainers are less likely to apply incompatible code changes to external API classes compared to the other classes during the library evolution. This implies that library developers may understand the efforts by clients needed to update their libraries. Complementary to this finding, Bloch mentions the growing awareness of library maintainers to APIs [33]. This is also reinforced by Seo et al. [34] where they found that there are many cases where API breakage changes are only applied when unavoidable (i.e., in response to either vulner- abilities or needed bug fixes etc...). There are benefits to this awareness of client-used API breakages. In particular, the evolution of APIs en- courages trust and reduce the latency of adoption by client projects, which is currently being experienced as a problem by many OSS clients [35]. Larman [36] introduced a notion of the Protected Variation (PV) pattern: identify points of predicted variation and create a stable in- terface around them. This PV pattern could explain how contemporary developers build and evolve libraries in relation to client-used APIs.

6.2. Comparison to literature

It is important to understand that our work cannot be simply compared at face-value to prior studies. As outlined in Section 4.1 there are obvious differences with our approach, compared to the studies of Dig and Johnson [9] and Cossette et al. [8]. Dig and Johnson used the change logs as heuristic to locate all API changes, and other considered

Table 6
Shows for a library release (i) the number of issues and new features per version release analysis and (ii) the number of non-refactoring related API breaking classes. We also show in the number of these API breakages mapped to the change log comments.

Library	Change Log Release	# Issues	# New Features	$\|non\,Ref\| \cap$ breaking $\cap$ clientUse (# mapped to change logs)
GUAVA	v11	26	13	4 (4)
	v12	43	24	4 (4)
	v13	26	28	7 (6)
	v14	64	10	5 (5)
	v15	53	11	7 (6)
	v16	19	8	5 (4)
	v17	11	5	6 (4)
	v18	21	7	3 (2)
		263	106	41 (34) 82%
HTTPCLIENT	v4.1.2	5	–	3 (2)
	v4.1.3	4	–	3 (2)
	v4.2	15	4	3 (1)
	v4.2.1	8	–	4 (2)
	v4.2.2	8	–	4 (2)
	v4.2.3	21	–	2 (2)
	v4.2.4	6	9	3 (1)
	v4.2.5	6	4	2 (2)
		73	17	24 (14) 58%

public entities are APIs. In this study, we detect syntactic changes in classes to infer changes and determine client usage to identify if the change has an effect to its users. As a result, our approach is unable to detect behavioral API breakages. Dig and Johnson's study included behavioral breakages, which we do not consider due to the limitations of our approach. Our definition of an API does differ from prior work. Dig and Johnson considered all public entities. Our work are more similar to the work of Cossette et al., in which we include the detection of protected entities. In this work, we go further and use client usage to focus on API breakages the more popular APIs.

The usage of tools revealed more API breakages, some of which were not reported in the API change logs, which was also consistent with the findings of Murphy-Hill et al. [31]. These undocumented API changes could also explain the disparity in results between manual (i.e., Dig and Johnson study) and machinery refactoring detection. For mechanical refactoring detection, since `Ref-Finder` is template-based refactoring reconstructing approach, we were only able to identify 23 out of 70 of Fowler's catalog. In fact, Cossette et al. [8] also believed that tools would miss some behavioral refactoring, saying that they *'…do not believe that some changes would be easily handled by mechanical transformation tools; instead the API maintainer, or the client developer would need to craft some minimal specification that would describe how to remap classes to accommodate these breaking changes.'* Another difference

(a) This code change in the method RESOLVE was detected as breaking API compatibility for users of the older JDK.

Fig. 9. We show two examples of API breaking changes that were not mapped to detected refactoring operations (i.e., `non-Ref`). We conjecture that these changes are (a) in response to a complex defect in the code and (b) consist of a complex refactoring that is not captured by the automated approach.

(b) This code change breaks API compatibly with the replacement HASHCODES to HASHCODE in the method. This refactoring was missed by the automated approach

```
10  ███ ███  guava/src/com/google/common/base/Optional.java                    View

⚓  @@ -207,19 +207,21 @@
207       * skipping over occurrences of {@link          207       * skipping over occurrences of {@link
          Optional#absent}. Iterators are unmodifiable and are    Optional#absent}. Iterators are unmodifiable and are
208       * evaluated lazily.                             208       * evaluated lazily.
209       *                                               209       *
210   -   * @since 11.0                                   210   +   * @since 11.0 (generics widened in 13.0)
211       */                                              211       */
212       @Beta                                           212       @Beta
213   -   public static <T> Iterable<T> presentInstances(final    213   +   public static <T> Iterable<T> presentInstances(
          Iterable<Optional<T>> optionals) {
                                                          214   +       final Iterable<? extends Optional<? extends T>>
                                                                        optionals) {
214           checkNotNull(optionals);                    215           checkNotNull(optionals);
215           return new Iterable<T>() {                  216           return new Iterable<T>() {
216             @Override public Iterator<T> iterator() { 217             @Override public Iterator<T> iterator() {
217               return new AbstractIterator<T>() {      218               return new AbstractIterator<T>() {
218   -             private final Iterator<Optional<T>> iterator =   219   +             private final Iterator<? extends Optional<?
          checkNotNull(optionals.iterator());                        extends T>> iterator =
                                                          220   +                 checkNotNull(optionals.iterator());
219                                                       221
220               @Override protected T computeNext() {   222               @Override protected T computeNext() {
221                 while (iterator.hasNext()) {          223                 while (iterator.hasNext()) {
222   -               Optional<T> optional = iterator.next();   224   +               Optional<? extends T> optional =
                                                                          iterator.next();
223                   if (optional.isPresent()) {         225                   if (optional.isPresent()) {
224                     return optional.get();            226                     return optional.get();
225                   }                                   227                   }
⚓
```

Fig. 10. Example of a misidentified detected refactoring-related API breaking change (Ref ∩ breaking class). Ref-Finder detects this code change between GUAVA version 12 and 13 as an add_parameter R_{opt}, while the API breaking tool reports it as binary incompatible modified method.

in our method that may have influenced results, is where we analyze API changes between consecutive versions, while prior work analyzed versions that were not consecutive.

6.3. Challenges of the automated approach

Key threats to the automated approach accuracy is when: (i) refactorings are missed by our approach (i.e., Ref-Finder [32]), (ii) developers may not report all API changes [31] and (iii) misidentification of breaking APIs is reported but it did not cause a breakage. Fig. 10 presents an example of a misidentification reported by our automated approach.[13] In this example, Ref-Finder detects this change as an add_parameter R_{opt}, that is also API breaking since it has a change in the method signature. However, according to Java documentation, the superclass extends indicates that the change of this special *'type parameter of the class does not, in itself, have any implications for binary compatibility'*. We believe these limitations will encourage researchers to further investigate and help us understand how developers evolve their libraries, especially in regards to avoidable API breakages.

6.4. Threats to validity

Internal Threats: The most significant internal threat is correctness of the automated tools, especially Ref-Finder. To mitigate this and as a sanity check, we randomly inspected a small sample of the results for validation. Mentioned earlier in the paper, an example of a false positive was when a unchanged file was reported to have a refactoring identified. In the end, we understand that recall is not as obvious to investigate as ground truth is unknown. Ref-Finder is the current state–of–the–art and actively used in research.

Another minor threat to our approach is that API breakages false positives caused by the class-level granularity of analysis. Theoretically, an external API class that has a breakage related to a private entity could be a false positive. However, even with this assumption in mind, our analysis may be underestimations. It is true that the accuracy of the saturation point is fairly dependent on the sample size. We believe our sample clients are sufficient to at least identify the most popular APIs

that reside in the client-used API classes. Sometimes variations between the refactored classpath (originating from source code) and API breakages class path (originating from binary code) may cause a missmatch. To overcome this, we manually validated the consistency of file paths to ensure consistency and completeness. Correct ordering of consecutive library releases is another minor threat. We therefore consider Maven [37] as the ground truth to base our chronological ordering of the released versions of a library. Some of our conclusions are based on the statistical analysis. We believe that due to outliers and nature of the data collected, non parametric statistical tests were deemed appropriate.

External Threats: As an external threat, we understand that our collected clients and the six selected OSS libraries are not necessary complete representations of the real world. However, we believe that the diverse nature (such as size, domain, team) of the six libraries is enough to assume generalization. Although our approximations of external APIs can only be justified through documentation and developers, we believe our method provides sufficient confidence of external client coverage. Another important threat is selection of the more popular libraries. As a results, our findings may not be applicable for less popular libraries. In this study, we consider that both library developers and users are more concerned with popular APIs, as they tend to reach a larger client user-base. Moreover, the same libraries that we study have been used in prior studies by researchers. As future work, we plan to expand our study to investigate more frameworks and libraries. Since our study is focused only on java libraries, we cannot make generalizations to other programming languages. We are confident that our research method is scalable and can be replicated with different sets of clients and subject libraries in other languages.

7. Related work

In this section, we introduce literature related to API usage, library migration support and library evolution.

API usage. There has been different work that have collected clients API usage. For example, work such as De Roover et al. [38] exploit API usage to understand popularity and usage patterns of clients. The data collected is visualized to further explore to provide program comprehension as well as identify patterns in the code. Another set of research use the API usage as a measure of stability or popularity [39,40]. Our

---

[13] commit can be found at goo.gl/CwXoBj and API change at https://goo.gl/VPPTIX.

Table 7

Classification of R_{opt} for API classes with *presver* ratio. Note that one class may be classified under several refactoring types. Note (–) represents no matches. We also show the total of all breakages (cu. + ncu.) and use to colors to highlight when prsv = low and prsv = high.

Classification of R_{opt}	GUAVA breaking clientUse	GUAVA breaking non clientUse	GUAVA breaking prsv	GUAVA non breaking clientUse	GUAVA non breaking non clientUse	HTTPCLIENT breaking clientUse	HTTPCLIENT breaking non clientUse	HTTPCLIENT breaking prsv	HTTPCLIENT non breaking clientUse	HTTPCLIENT non breaking non clientUse	XERCES breaking clientUse	XERCES breaking non clientUse	XERCES breaking prsv	XERCES non breaking clientUse	XERCES non breaking non clientUse
change_parameter	28	25	(53) 1.12	20	4	1	10	(11) 0.10	–	11	56	217	(273) 0.26	–	19
cdcf*	–	–	–	–	–	–	1	–	4	1	44	74	(118) 0.59	8	48
extract_method	8	32	(40) 0.25	3	–	–	1	–	–	2	9	61	(70) 0.15	5	47
extract_subclass	3	–	–	–	–	–	–	–	–	–	–	1	–	–	–
extract_superclass	–	1	–	–	–	–	–	–	–	–	–	2	–	–	–
inline_method	6	3	(9) 2.00	–	2	1	–	–	–	–	7	23	(30) 0.30	1	1
inline_temp	4	4	(8) 1.00	6	1	2	–	–	–	–	8	20	(28) 0.40	–	7
introduce_explaining_variable	–	10	–	6	2	–	–	–	–	–	8	33	(41) 0.24	2	21
introduce_null_object	–	–	–	1	2	–	–	–	–	–	–	–	–	–	–
move_field	10	66	(76) 0.15	9	7	–	4	–	–	–	15	255	(270) 0.06	7	5
move_method	15	19	(34) 0.08	8	14	3	6	(9) 0.50	–	7	12	256	(268) 0.05	16	12
pull_up_constructor_body	–	–	–	1	–	–	–	–	–	–	–	–	–	–	–
pull_up_field	–	3	–	–	3	–	–	–	–	–	–	5	–	–	–
push_down_field	–	2	–	–	–	–	–	–	–	–	–	41	–	–	–
ratp*	–	2	–	–	3	2	–	–	2	5	2	14	(16) 0.14	–	9
remove_control_flag	5	1	(5) 5.00	1	12	2	1	(3) 2.00	–	5	1	11	(12) 0.09	1	6
remove_middle_man	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–
remove_parameter	30	20	(50) 1.50	16	2	1	8	(9) 0.12	1	–	45	170	(62) 0.26	–	18
rename_method	28	54	(82) 0.52	1	4	–	–	–	2	–	146	217	(363) 0.67	1	19
rcwfm*	2	2	(4) 1.00	12	7	–	–	–	–	–	–	–	–	–	–
replace_data_with_object	2	–	–	–	–	–	–	–	1	–	4	11	(15) 0.36	–	2
replace_exception_with_test	–	9	–	–	–	–	1	–	–	–	–	3	–	–	4
rmnwc*	16	34	(50) 0.47	8	20	–	8	–	–	17	49	165	(214) 0.30	9	117
rmwmo*	4	–	–	2	–	–	–	–	1	–	4	40	(44) 0.10	1	5
rncgc*	–	–	–	–	1	–	–	–	–	–	9	27	(36) 0.33	–	14
replace_temp_with_query	–	6	–	1	–	–	–	–	–	–	–	–	–	–	1
pull_up_method	–	–	–	–	–	2	1	(3) 2.00	–	–	5	2	(7) 2.50	–	–
extract_interface[a]	–	–	–	–	–	–	–	–	–	–	–	1	–	9	–
Median ($\bar{x}$)	4	4	1.00	3	3	2	1	0.50	1	5	9	17	0.33	5	11
Mean (μ)	8.4	14.78	1.19	5.14	4.4	1.75	3.2	0.95	1.71	6.17	24.9	62.70	0.46	5.45	19.72

*Note types abbreviations - cdcf = consolidate_duplicate_cond_fragment, rcwfm = replace_constructor_with_factory_method, ratp = remove_assignment_to_parameters, rmnwc = replace_magic_number_with_constant, rmwmo = replace_method_with_method_object, rncgc = replace_nested_cond_guard_clause

previous work [41], among work leveraged popularity to recommend when libraries are deemed safe to use by the masses. Other related work that studied the impact of API evolution on their clients on online forums such as Stack Overflow [42] and the Android App [43], Pharo [44] and Smalltalk ecosystems [45].

Library Migration Support. Much work has been in transformation of the client code to support migration of library API changes. Work by Chow and Notkin [46] and Balaban et al. [47] use a change specification language. There is work that provides the client with automatic tool support to accommodate changes made the APIs of a library. For instance, SemDiff [48] recommends replacements for framework methods that were accessed by clients. Other similar tools were proposed by Xing and Stroulia [49] and Schafer et al. [50]. Other work on reuse support is through code analysis. This area of work considers code clone detection techniques [51] to support which library version is most appropriate candidate for migration. Godfrey et al. [52] proposed origin analysis to recover context of code changes. Our previous work [53] tracked how code is reused cross-projects. Related works [10] focused on support for clients migrating to a newer library version. Likewise, other works [39,54–56] studied how library maintainers balance API compatibility with an evolving library.

Library Evolution. There is similar work with respect to library

maintenance and evolution. Cossette et al. [8] manually illustrated the complexities of library changes and transformations. Other work such as Kim et al. [57] studied the role of refactoring during software evolution. Recently, there has been large-scale empirical studies conducted on library migrations and evolution. Empirical studies by Raemakers et al. [27,58], Jezek et al. [26] and Joel et al. [59] studied in-depth how libraries that reside in the Maven Central super-repository evolve and break APIs.

8. Conclusions and future work

Refactorings is a key maintainability practice, even for library maintainers. When evolving code, we find that library developers are less likely to break APIs. However, we find that many of these API breaking changes relate to bug fixes and new features, with only up to 37% of client-used API breakages related to refactoring operations. The study finds that there are still challenges to improving our tools. The study also reveals challenges faced by the tools. As future work, we envision that this study encourages more research into automated refactoring detection techniques to advance our understanding of refactoring activities on API breakages.

Acknowledgments

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] W.F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis. University of Illinois at Urbana-Champaign, 1992.

[3] M.M. Lehman, Laws of software evolution revisited, Proceedings of the 5th European Workshop on Software Process Technology, EWSPT '96, Springer-Verlag, London, UK, UK, 1996, pp. 108–124.

[4] Source file example on google guava, accessed 2014-09-01, (https://code.google.com/p/guava-libraries/source/browse/guava/src/com/google/common/base/internal/Finalizer.java?r=9de0ce12f1648ee4f190e2ce706f26088727ff89).

[5] J. Businge, A. Serebrenik, M.G.J. van den Brand, Eclipse API usage: the good and the bad, Softw. Q. J. 23 (1) (2015) 107–141.

[6] I.R. Forman, M.H. Conner, S.H. Danforth, L.K. Raper, Release-to-release binary compatibility in som, Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '95, (1995), pp. 426–438.

[7] J. Bauml, P. Brada, Automated versioning in osgi: a mechanism for component software consistency guarantee, Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on, (2009), pp. 428–435.

[8] B.E. Cossette, R.J. Walker, Seeking the ground truth, Proc. ACM SIGSOFT Intrn. Symp. Found. Software Eng. - FSE '12 (2012).

[9] D. Dig, R. Johnson, How do APIs evolve? A story of refactoring, J. Softw. Maintenance Evolut. 18 (2006) 83–107.

[10] P. Kapur, B. Cossette, R.J. Walker, Refactoring references for library migration, ACM SIGPLAN Notices 45 (10) (2010) 726.

[11] Guava-libraries, accessed 2017-04-28, (https://code.google.com/p/guava-libraries/).

[12] The hyper-text transfer protocol client, accessed 2017-04-28, (https://hc.apache.org/httpcomponents-client-ga/).

[13] The java programming assistant, accessed 2017-04-28, (http://jboss-javassist.github.io/javassist/).

[14] Jdom website, accessed 2017-04-28, (http://www.jdom.org/).

[15] Joda-time for java, accessed 2017-04-28, (http://www.joda.org/joda-time/).

[16] Apache log4j, accessed 2017-04-28, (http://logging.apache.org/log4j/1.2/).

[17] Simple logging facade for java (slf4j), accessed 2017-04-28, (http://www.slf4j.org/).

[18] The apache xerces project, accessed 2017-04-28, (https://xerces.apache.org/).

[19] Aether adapter for maven plugins, accessed 2017-04-28, (http://aether.jcabi.com/index.html).

[20] Javacompiler tool, accessed 2017-04-28, (http://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.CompilationTask.html).

[21] The japicmp tool, accessed 2016-04-08, (https://siom79.github.io/japicmp/).

[22] The clirr tool, accessed 2016-04-08, (http://clirr.sourceforge.net/).

[23] The java api compliance checker, accessed 2016-04-08, (http://ispras.linuxbase.org/index.php/Java_API_Compliance_Checker).

[24] Jdiff - an html report of api differences, accessed 2016-04-08, (http://javadiff.sourceforge.net/).

[25] Revapi - fully featured api checker for java and beyond, accessed 2016-04-08, (http://revapi.org/).

[26] K. Jezek, J. Dietrich, P. Brada, How java APIs break - an empirical study, Inf. Softw. Technol. (2015) 129–146, http://dx.doi.org/10.1016/j.infsof.2015.02.014.

[27] S. Raemaekers, A. van Deursen, J. Visser, Semantic versioning versus breaking changes: a study of the maven repository (2014) 215–224.

[28] japicmp tool and library, accessed 2017-04-28, (https://github.com/siom79/japicmp).

[29] K. Prete, N. Rachatasumrit, N. Sudan, M. Kim, Template-based reconstruction of complex refactorings, Software Maintenance (ICSM), 2010 IEEE International Conference on, IEEE, 2010, pp. 1–10.

[30] J. Cohen (Ed.), Front Matter, Academic Press, revised edition edition, 1977.

[31] E. Murphy-Hill, C. Parnin, A.P. Black, How we refactor, and how we know it, 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 287–297, http://dx.doi.org/10.1109/ICSE.2009.5070529.

[32] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, J. Syst. Softw. 86 (4) (2013) 1006–1022, http://dx.doi.org/10.1016/j.jss.2012.10.040.

[33] J. Bloch, How to design a good api and why it matters, Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 506–507.

[34] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, R. Bowdidge, Programmers' build errors: a case study (at google), Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 724–734, http://dx.doi.org/10.1145/2568225.2568255.

[35] R.G. Kula, D.M. German, T. Ishio, K. Inoue, Trusting a library: a study of the latency to adopt the latest maven release, 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, Canada, March 2–6, 2015, (2015).

[36] C. Larman, Protected variation: the importance of being closed, Softw., IEEE 18 (3) (2001) 89–91.

[37] The maven central repository, accessed 2014-09-01, (http://search.maven.org/).

[38] C. De Roover, R. Lammel, E. Pek, Multi-dimensional exploration of API usage, IEEE Int. Conf. Progr. Comprehension (2013) 152–161.

[39] T. McDonnell, B. Ray, M. Kim, An empirical study of API stability and adoption in the android ecosystem, IEEE Int. Conf. Softw. Maint., ICSM (2013) 70–79, http://dx.doi.org/10.1109/ICSM.2013.18.

[40] Y.M. Mileva, V. Dallmeier, M. Burger, A. Zeller, Mining trends of library usage, Proc. Intl and ERCIM Principles of Soft. Evol. (IWPSE) and Soft. Evol. (Evol) Workshops, IWPSE-Evol '09, ACM, New York, NY, USA, 2009, pp. 57–62.

[41] R.G. Kula, C.D. Roover, D.M. German, T. Ishio, K. Inoue, Visualizing the evolution of systems and their library dependencies, Proc. IEEE Work. Conf. Soft. Viz. (VISSOFT) (2014).

[42] M.L. Vásquez, G. Bavota, M.D. Penta, R. Oliveto, D. Poshyvanyk, How do API changes trigger stack overflow discussions? a study on the android SDK, ICPC, ACM, 2014, pp. 83–94.

[43] G. Bavota, M.L. Vásquez, C.E. Bernal-Cárdenas, M.D. Penta, R. Oliveto, D. Poshyvanyk, The impact of API change- and fault-proneness on the user ratings of android apps, IEEE Trans. Software Eng. 41 (4) (2015) 384–407.

[44] A.C. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, M.T. Valente, How do developers react to API evolution? the pharo ecosystem case, ICSME, IEEE Computer Society, 2015, pp. 251–260.

[45] R. Robbes, M. Lungu, D. Röthlisberger, How do developers react to API deprecation?: the case of a smalltalk ecosystem, SIGSOFT FSE, ACM, 2012, p. 56.

[46] K. Chow, D. Notkin, Semi-automatic update of applications in response to library changes, Proceedings of the 1996 International Conference on Software Maintenance, ICSM '96, IEEE Computer Society, Washington, DC, USA, 1996.

[47] I. Balaban, F. Tip, R. Fuhrer, Refactoring support for class library migration(2005) 265–279. http://dx.doi.org/10.1145/1094811.1094832.

[48] B. Dagenais, M.P. Robillard, Semdiff: Analysis and recommendation support for api evolution, Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 599–602.

[49] Z. Xing, E. Stroulia, API-evolution support with diff-catchup, 33 (2007) 818–836, http://dx.doi.org/10.1109/TSE.2007.70747.

[50] T. Schäfer, J. Jonas, M. Mezini, Mining framework usage changes from instantiation code, Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 471–480, http://dx.doi.org/10.1145/1368088.1368153.

[51] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Software Eng. 28 (7) (2002) 654–670, http://dx.doi.org/10.1109/TSE.2002.1019480.

[52] M. Godfrey, L. Zou, Using origin analysis to detect merging and splitting of source code entities, IEEE Trans. Softw. Eng. 31 (2) (2005) 166–181.

[53] N. Kawamitsu, T. Ishio, T. Kanda, R.G. Kula, C.D. Roover, K. Inoue, Identifying source code reuse across repositories using lcs-based source code similarity, Proc. of SCAM, (2014).

[54] G. Malpohl, J.J. Hunt, W.F. Tichy, Renaming detection, Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE '00, IEEE Computer Society, Washington, DC, USA, 2000, p. 73.

[55] M. Mezini, Maintaining the consistency of class libraries during their evolution, SIGPLAN Not. 32 (10) (1997) 1–21, http://dx.doi.org/10.1145/263700.263701.

[56] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, Reuse contracts: managing the evolution of reusable assets, SIGPLAN Not. 31 (10) (1996) 268–285, http://dx.doi.org/10.1145/236338.236363.

[57] M. Kim, D. Cai, S. Kim, An empirical investigation into the role of API-level refactorings during software evolution, 2011 33rd Int. Conf. Softw. Eng. (ICSE) (2011) 151–160.

[58] S. Raemaekers, A. van Deursen, J. Visser, Measuring software library stability through historical version analysis, Proc. of Intl. Comf. Soft. Main. (ICSM), (2012), pp. 378–387.

[59] M.v.E. Joel Cox Eric Bouwers, J. Visser, Measuring dependency freshness in software systems, 2 (2015) 109–118.