APIDiff: Detecting API Breaking Changes

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*—Libraries are commonly used to increase productivity. As most software systems, they evolve over time and changes are required. However, this process may involve breaking compatibility with previous versions, leading clients to fail. In this context, it is important that libraries creators and clients frequently assess API stability in order to better support their maintenance practices. In this paper, we introduce APIDIFF, a tool to identify API breaking and non-breaking changes between two versions of a Java library. The tool detects changes on three API elements: types, methods, and fields. We also report usage scenarios of APIDIFF with four real-world Java libraries.**

***Index Terms*—API Evolution, Breaking Changes, Mining Software Repositories.**

I. Introduction

Change is a common practice in software development. Developers create, remove, and update software systems to accommodate new features, fix bugs, and improve code quality. Nowadays, most modern systems rely on libraries[1], which are widely used by developers to increase productivity and reuse well designed and tested component solutions [1]–[3]. Libraries provide functionalities via *Application Programming Interfaces* (APIs), which are contracts used by clients to communicate with these components [4]. In this context, during software development, ideally, API creators should strive to properly maintain these contracts, avoiding breaking their clients.

However, the literature presents that *API contracts are commonly broken* [5]–[7]. For example, a recent study with over 300 Java libraries shows that API creators often break backward compatibility [5]. In this context, several approaches have been proposed to handle API evolution, for instance, to detect refactoring actions [8], to capture and replay performed refactorings [9], and to track popularity and migration of APIs [10]. However, we still lack approaches and tools to support API creators and clients assessing possible breaking changes. Specifically, important questions often arise after a new library release, for example:

- Are there API breaking changes in this version?
- Which changes may affect clients?
- How stable is this library?

The answers to these questions may motivate API creators to write migration documents in order to help clients when updating their applications or to identify and revert accidental breaking changes. On the client side, one could assess the evolution of an API, analyzing the amount of breaking changes over time, to select more stable libraries to depend on.

To address these challenges, we propose APIDIFF, a tool to identify breaking (and non-breaking) changes between two versions of a Java library. The goal of APIDIFF is to support both API creators and clients in their development activities. The tool analyses libraries hosted on the distributed version control system `git`, which is used by GitHub, a popular repository with more than 74 million projects and a community with more than 26 million developers worldwide (on January, 2018). The tool analyses changes in types, methods and fields, and it was already used in previous studies of our research group [5], [11], [12]. APIDIFF reports several breaking changes (e.g., public field removal, change of parameter type), as well as changes that do not affect clients (e.g., type addition), together with 13 well-known refactoring types (e.g., method rename and move).

Structure of the paper. Section II describes the architecture and features of APIDIFF. Section III presents usage scenarios in four real-world Java libraries. Section IV presents the limitations of our tool and Section V related work. Finally, we conclude the paper in Section VI.

II. Features and Architecture

A. Architecture

APIDIFF identifies breaking and non-breaking changes between versions of a Java library hosted on `git` repositories in a fully automated way. It uses a similarity heuristic based on static analysis to detect changes in API elements (i.e., types, methods, and fields). Figure 1 presents the high level architecture of APIDIFF, which includes three major modules: *processing*, *refactoring*, and *analysis*.

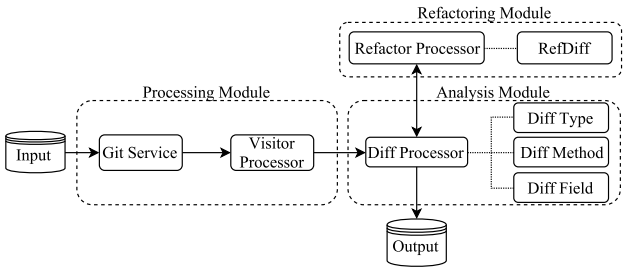


Fig. 1. APIDiff Architecture

[1]We use the term *library* to designate both frameworks and libraries.

Processing Module. This module receives as input the metadata of the analyzed library. In *Git Service*, the project is cloned from its repository, and a directed acyclic graph (DAG) is created with the history of commits. Then, *Visitor Processor* creates instances of the library. These instances are passed as input to the *Analysis Module*.

Refactoring Module. This module detects refactoring actions between two instances of a library. It is reused from REFDIFF, a tool that detects 13 well-known refactoring operations [8]. REFDIFF algorithm finds changes in types, methods, and fields, such as move or rename.

Analysis Module. This module detects the changes in the library history. The final output of APIDIFF contains a list of changes performed in types, methods, and fields, including also commit metadata (e.g., author, commit hash, date) and information about the change (e.g., element type and name).

B. Catalog of Changes

APIDIFF focuses on syntactic changes. We classify as *Breaking Changes* (BC) the changes performed in API elements such as types, methods, and fields that may break client applications. We exclude changes performed on deprecated API elements since clients have been previously warned about possible incompatibilities in such element.

Table I presents the catalog of BCs detected by APIDIFF regarding types, methods, and fields. BCs in types and methods include popular refactoring actions such as rename and move, as well as critical ones such as removal. BCs in fields include, for example, refactorings and changes in default values.

TABLE I
BREAKING CHANGES DETECTED BY APIDIFF

Element	Breaking Changes (BC)
Type	REMOVE TYPE, LOST VISIBILITY, CHANGE IN SUPERTYPE, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER, RENAME TYPE, MOVE TYPE
Method	REMOVE METHOD, LOST VISIBILITY, CHANGE IN RETURN TYPE, CHANGE IN PARAMETER LIST, CHANGE IN EXCEPTION LIST, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER, MOVE METHOD, RENAME METHOD, PUSH DOWN METHOD, INLINE METHOD
Field	REMOVE FIELD, LOST VISIBILITY, CHANGE IN FIELD TYPE, CHANGE IN FIELD DEFAULT VALUE, ADD FINAL MODIFIER, MOVE FIELD, PUSH DOWN FIELD

Changes that do not break clients are classified as *Non-breaking Changes* (NBC), as presented in Table II. Common NBCs in this context involve, for example, type addition and visibility modifier change to public/protected (i.e., gain of visibility). Changes on deprecated API elements are also classified as NBCs.

C. Features

In this section we describe the main features and functionalities provided by APIDIFF.

TABLE II
NON-BREAKING CHANGES DETECTED BY APIDIFF

Element	Non-Breaking Changes (NBC)
Type	ADD TYPE, GAIN VISIBILITY, REMOVE FINAL MODIFIER, ADD STATIC MODIFIER, ADD SUPERTYPE, EXTRACT SUPERTYPE, DEPRECIATE TYPE
Method	ADD METHOD, PULL UP METHOD, GAIN VISIBILITY, REMOVE FINAL MODIFIER, ADD STATIC MODIFIER, DEPRECIATE METHOD, EXTRACT METHOD
Field	ADD FIELD, PULL UP FIELD, GAIN VISIBILITY, REMOVE FINAL MODIFIER, DEPRECIATE FIELD

Detecting changes in version histories. In this functionality, it is possible to analyze changes performed in several commits of a given project. For example, we can select one branch or analyze all branches. The input includes the project path, git url, and the branch name (for all branches the name is not necessary). API creators and clients can use this feature to assess the stability of their libraries.

Detecting changes in specific commit. In this feature, it is possible to analyze changes performed in a specific commit. The input includes the project path, git url, and the commit to be inspected. For example, API creators can use this functionality to analyze their commits or to evaluate pull requests, detecting accidental breaking changes performed by contributors.

Fetching new commits. By using this feature, the tool fetches new commits from a repository. The input includes the project path and git url. API creators may benefit from this functionality by monitoring changes in their own repositories, during a time interval. For example, a library developer can clone and track a repository, and then use APIDIFF to detect changes in new commits. In this way, if contributors introduce breaking changes, he is notified shortly afterwards to accept the change or revert it.

Filtering Packages. For all provided features, it is possible to filter in or filter out packages according to their names, considering keywords such as *internal*, *test*, *experimental*, and *example*. In this way, we can create a proxy to detect changes in internal implementations or to eliminate from the analysis source code that is not intended to be public.

Reading and writing a CSV file. Users can customize the output of APIDIFF, reporting the detected changes in a CSV file. They can also read the input from CSV files.

III. USAGE SCENARIOS

In this section, we present four usage scenarios for APIDIFF. In the first scenario, we investigate API changes considering the complete history of two popular Android libraries: PHILJAY/MPANDROIDCHART[2] (a chart view library) and BUMPTECH/GLIDE[3] (an image loading and caching library). In

[2]https://github.com/PhilJay/MPAndroidChart
[3]https://github.com/bumptech/glide

the second scenario, we present the most popular BCs and NBCs detected in the history of PHILJAY/MPANDROIDCHART. Then, in a third example, we present the changes over time in SQUARE/PICASSO[4] (a downloading and caching library for Android). Lastly, we detect changes in a specific commit of MOCKITO/MOCKITO[5] (a framework to implement unit tests).

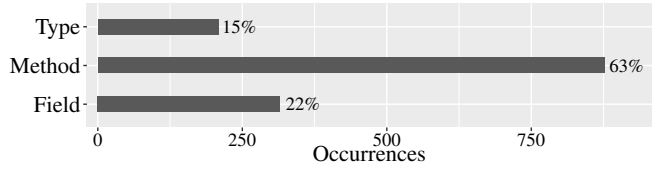
A. Most Common Elements with Changes

In this first example, we assess the whole commit history of the PHILJAY/MPANDROIDCHART and BUMPTECH/GLIDE. The results refer to the *master* branch and to the elements documented with JavaDoc. Figure 2 presents the piece of code necessary to run this functionality in APIDIFF. The input includes the project path, git url, and branch name.

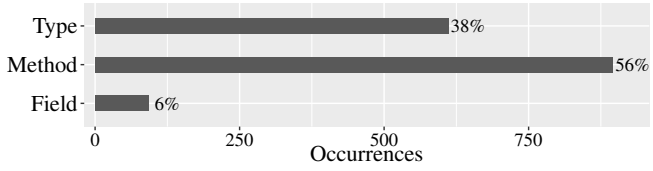
```
APIDiff diff = new APIDiff(
        “bumptech/glide”,
        “https://github.com/bumptech/glide.git”);

Result result =
diff.detectChangeAllHistory(“master”, Classifier.API);
```

Fig. 2. Detecting changes in BUMPTECH/GLIDE

APIDIFF detected 1,401 BCs in the history of PHILJAY/MPANDROIDCHART and 1,599 BCs in BUMPTECH/GLIDE history. The most common BCs are performed at method level. The tool also detected 1,662 NBCs in PHILJAY/MPANDROIDCHART and 1,392 NBCs in BUMPTECH/GLIDE histories. Among the detected NBCs, methods are also the most modified elements. Figures 3 and 4 present the distribution of BCs and NBCs over types, methods, and fields in these two libraries.



(a) PHILJAY/MPANDROIDCHART

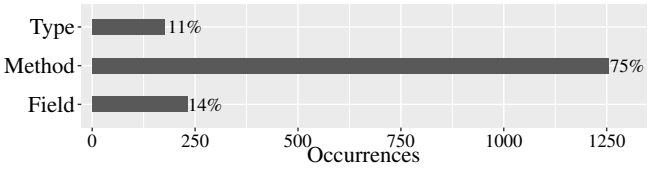


(b) BUMPTECH/GLIDE

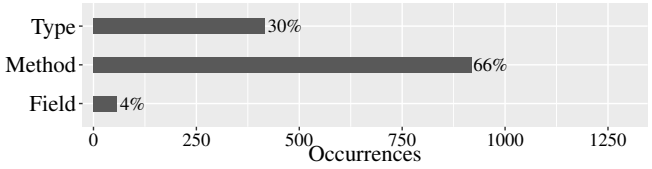
Fig. 3. Breaking Changes in types, methods or fields

Therefore, using this feature, library creators may detect which API elements break more frequently. Consequently, they can discover the elements that are more affected by API instability. On the client side, this feature may support checking the frequency of breaking changes in a library. In other words, the results can help clients to compare similar libraries and select the most stable and well-maintained one.

[4]https://github.com/square/picasso
[5]https://github.com/mockito/mockito



(a) PHILJAY/MPANDROIDCHART



(b) BUMPTECH/GLIDE

Fig. 4. Non-Breaking Changes (NBCs)

B. Most Popular Changes

This example presents the most popular changes performed in PHILJAY/MPANDROIDCHART as detected by APIDIFF (Figure 5). The results refer to the *master* branch and to the elements documented with JavaDoc. Among the 1,401 BCs, 44% (613) are related to method removal and 12% (171) involve field removal. The other three changes involve rename methods, changes in return type, and changes in field default value, with 6% of occurrences each.

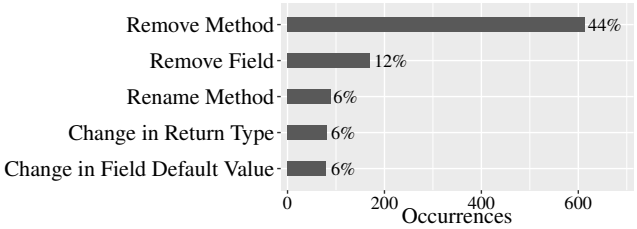


Fig. 5. Top-5 Breaking Changes in PHILJAY/MPANDROIDCHART

Furthermore, APIDIFF detected 1,662 NBCs. As presented in Figure 6, the most popular NBCs involve adding elements: methods (1,115 occurrences, 67%), fields (186 occurrences, 11%), and types (151 occurrences, 9%).

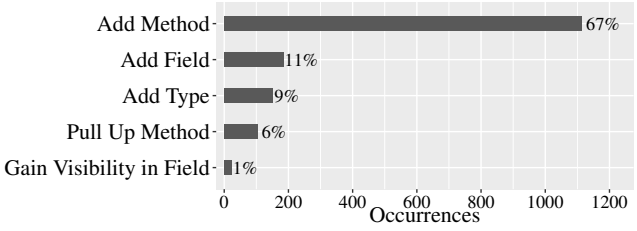


Fig. 6. Top-5 Non-breaking Changes in PHILJAY/MPANDROIDCHART

C. API Changes Over Time

In this example, we use APIDIFF to detect changes in the history of SQUARE/PICASSO. The results include changes performed in the *master* branch and in JavaDoc elements.

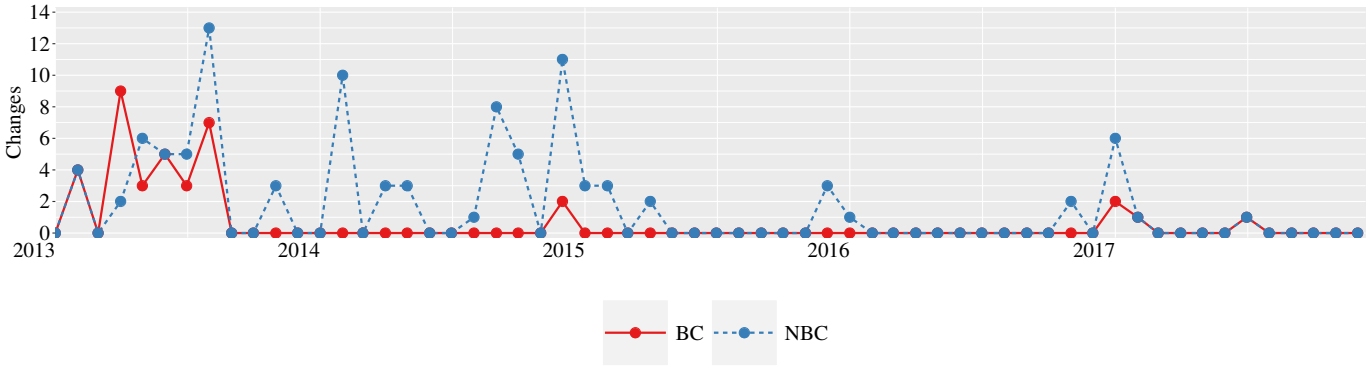


Fig. 7. BCs and NBCs Over Time in SQUARE/PICASSO:

Figure 7 presents the distribution of changes since 2013 (when the repository was created) until 2017. The first two years were more unstable, including many BCs (red line) and also NBCs (blue line). By contrast, APIDIFF detected fewer changes after 2015, showing that the latest versions had little impact on clients, providing more stable APIs.

D. Changes at Commit

This last example focuses on a specific commit of the MOCKITO/MOCKITO framework. Figure 8 shows the piece of code in APIDIFF to detect changes in a given commit.

```
APIDiff diff = new APIDiff(
          “mockito/mockito”,
          “https://github.com/mockito/mockito.git”);

Result result =
diff.detectChangeAtCommit(
     "4ad5fdc14ca4b979155d10dcea0182c82380aefa",
     Classifier.API);
```

Fig. 8. Detecting Changes in a Specific Commit

The input includes the project path, git url, and the commit hash. In this case, APIDIFF detected the addition of a method, as presented in Figure 9.

Change: Addition Method

Description: Method *answersWithDelay(long, Answer<T>)* added in class *org.mockito.AdditionalAnswers*

Fig. 9. Addition method detected by APIDIFF

Therefore, library creators may use this feature to analyze a commit and detect accidental changes that may harm client applications. In GitHub, for example, library creators could automatically analyze pull requests and check whether there are breaking changes in contributions.

IV. LIMITATIONS

Breaking changes detected by APIDIFF do not necessarily have an impact on clients applications. For example, the modified API element may be a public low level or internal API, without any usage by clients. Furthermore, the results may include false positives, for example, due to *casting*.

V. RELATED WORK AND TOOLS

We organized this section in two topics: (a) studies on tools and approaches to handle API evolution and migration, and (b) studies on API evolution and breaking changes.

A. API Changes Tools

Several tools have been proposed to deal with the impact of software evolution. For example, Dagenais and Robillard [13] present *SemDiff*, a tool that suggests replacements for client systems based on changes in their own framework code. Kim and Notkin [14] introduce *Logical Structural Diff* (*LSdiff*), which computes differences between two versions of a system. Hora *et al.* [10] present *apiwave*, which focuses on library migration. Wu *et al.* [15] present *AURA* (AUtomatic change Rule Assistant), which generates automatic change rules, helping developers migrating to new releases. Xing and Stroulia [16] present *Diff-CatchUp*, which recommends features to replace an obsolete API implementation. Additionally, the same authors present *UMLDiff*, which detects structural changes between two version of a software system [17] and that is implemented in the *JDEvAn* tool, an Eclipe Plugin. However, *UMLDiff* does not focus on non-breaking or breaking changes at API level. For example, it does not verify deprecated elements or implementations in an internal package.

Silva *et al.* [8] present REFDIFF, an approach to detect refactorings between two versions of a `git` repository. REFDIFF itself does not detect other changes (i.e., addition or removal of API elements, deprecation operations, changes in visibility modifiers, etc) nor report whether the structure has JavaDoc or deprecation annotations. Therefore, we integrated REFDIFF with our tool, and its output is merged with the changes detected by APIDIFF. Still in the context of refactoring, Henkel and Diwan [9] present *CatchUp*, an approach that captures and replays performed refactorings.

B. Studies on API Evolution

In previous work [5], we investigate breaking changes performed in 317 real-world Java libraries on GitHub. The results include 9K releases and show that library owners frequently

break contracts over time. However, by using the language and infrastructure BOA [18], they report that few clients are impacted by these changes. In another work [11], we show that breaking changes are mainly motivated by the implementation of new features, to simplify API, and to improve maintainability. Bogart *et al.* [6] conducted a study involving three ecosystems: Eclipse, R/CRAN, and Node.js/npm. They investigate how developers manage and negotiate breaking changes, reporting answers from key developers in each ecosystem.

Dig and Johnson [7] also investigate the reasons behind API changes, studying five frameworks. They report that more than 80% of the changes involve refactorings. In fact, refactoring operations is a common practice in software evolution [19]. APIDIFF catalog includes some breaking changes from Dig and Johnson [7] study (e.g., move and remove method), and adds new changes (e.g., change in access modifier and `final` modifier). However, APIDIFF focuses on syntactic changes. Behavior breaking changes are studied in another work [20].

Some studies investigate the usage of internal APIs, i.e., details of implementation that should not be used by system clients. Mastrangelo *et al.* [21] report that clients frequently use internal APIs. This study focused on the internal interface `sun.misc.Unsafe` provided by JDK. Businge *et al.* [22], [23] also report the usage of internal APIs in Eclipse plugins. Additionally, the same authors investigated the reasons why internal API are used by client systems [24]. In this context, Hora *et al.* [25] detected that some internal APIs are likely to be promoted to public.

VI. Conclusion

In this paper, we introduced APIDIFF, a tool to detect changes in API versions. Our tool includes a combination of similarity heuristics and static analysis of Java source code to detect API breaking and non-breaking changes. The changes include removal and addition of API elements, 13 well-known refactoring operations, and also changes in visibility, static and final modifiers, exceptions, field default values, superclasses, and deprecated APIs. As an additional contribution, we made the source code of APIDIFF publicly available on GitHub.[6]

As future work, we plan to assess the precision of APIDIFF with real-world Java libraries. We also plan to improve the tool to support other popular languages. Finally, we plan to provide a new feature to analyze changes at *release level*.

Acknowledgments

This work is supported by FAPEMIG and CNPq.

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] 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.

[6] 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.

[7] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of Software Maintenance and Evolution (JSME)*, vol. 18, no. 2, pp. 83–107, 2006.

[8] 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.

[9] 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.

[10] A. Hora and M. T. Valente, "apiwave: Keeping track of API popularity and migration," in *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 321–323, 2015.

[11] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how Java developers break APIs," in *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1–11, 2018.

[12] 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.

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

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

[15] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," 32nd International Conference on Software Engineering (ICSE), pp. 325–334, 2010.

[16] Z. Xing and E. Stroulia, "API-evolution support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[17] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *20th International Conference on Automated Software Engineering (ASE)*, pp. 54–65, 2005.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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