UNDERSTANDING THE MOTIVATIONS FOR BREAKING CHANGES IN JAVA APIS

ALINE NORBERTA DE BRITO

UNDERSTANDING THE MOTIVATIONS FOR BREAKING CHANGES IN JAVA APIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: André Cavalcante Hora

Belo Horizonte

Julho de 2018

ALINE NORBERTA DE BRITO

UNDERSTANDING THE MOTIVATIONS FOR BREAKING CHANGES IN JAVA APIS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: André Cavalcante Hora

Belo Horizonte
July 2018

Brito, Aline Norberta de

B862u Understanding the motivations for breaking changes in Java APIs / Aline Norberta de Brito. — Belo Horizonte, 2018
xxiii, 66 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação

Orientador: Marco Túlio de Oliveira Valente
Coorientador: André Cavalcante Hora

1. Computação — Teses. 2. Engenharia de Software. 3. Interfaces (Computador). I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Understanding the motivations for breaking changes in Java APIs

ALINE NORBERTA DE BRITO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

André Cavalcante Hora

Prof. André Cavalcante Hora
Faculdade de Computação - UFMS

[illegible]

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Maurício Finavaro Aniche
Faculty of Electrical Engineering, Mathematics and Computer Science - TU Delft

Belo Horizonte, 10 de julho de 2018.

Acknowledgments

Agradeço aos meus amigos, familiares, professores e mentores! A presença de vocês foi imprescindível para a conclusão desta jornada. Agradeço em especial:

A Deus, por permitir a conclusão desta etapa.

Aos meus familiares, pelo incentivo durante este curso de mestrado. Especialmente, aos meus pais Bernadete e José Norberto, e ao meu irmão Rodrigo, por todo o carinho, apoio e conselhos recebidos.

Aos meus orientadores, Prof. Marco Tulio Valente e Prof. André Hora, pela paciência, suporte, ensinamentos, orientação e confiança no meu trabalho.

Aos meus amigos, pela amizade, apoio e por todos os momentos felizes proporcionados.

Aos meus amigos do Aserg, pela parceria, amizade e aprendizado. Em especial, ao Laerte Xavier, pelo auxílio nas pesquisas realizadas no decorrer deste curso.

Aos meus amigos da Concert, pelo companheirismo e pela contribuição para o meu crescimento pessoal e profissional.

Aos meus professores, pela inspiração, incentivo e conhecimento compartilhado durante o mestrado, graduação e ensino básico.

Aos membros da banca, Prof. Eduardo Figueiredo e Prof. Maurício Aniche, pela disponibilidade em participar deste trabalho.

Ao DCC/UFMG, FAPEMIG e CNPq, pelo suporte financeiro, logístico e profissional.

"Either write something worth reading or do something worth writing."
(Benjamin Franklin)

Resumo

Bibliotecas e APIs são comumente usadas no desenvolvimento de software, visto que permitem o reuso de código fonte e melhoram a produtividade. Assim como a maioria dos sistemas de software, elas também evoluem, o que pode quebrar contratos previamente estabelecidos com sistemas clientes, introduzindo *breaking changes* que podem afetar a compilação dos projetos. No entanto, as principais razões para introduzir *breaking changes* em APIs não são claras. Portanto, nesta dissertação de mestrado, são reportados os resultados de um longo estudo de campo de aproximadamente 4 meses com bibliotecas e *frameworks* Java populares. Foi configurada uma infraestrutura para observar todas as mudanças em 400 bibliotecas e para detectar *breaking changes* logo após sua introdução no código. Detectou-se possíveis *breaking changes* em 61 projetos. Depois de identificar as alterações, os desenvolvedores foram contatados, com o objetivo principal de entender as razões por trás das alterações nas APIs. Durante o estudo, identificou-se 59 *breaking changes*, confirmadas pelos desenvolvedores de 19 projetos. Analisando as respostas dos desenvolvedores, observou-se que (i) 39 % das alterações investigadas no estudo podem ter um impacto nos clientes, sendo porém mudanças que exigem pouco esforço na migração para as novas versões da API, (ii) *breaking changes* são motivadas principalmente pela necessidade de implementar novos recursos, pelo desejo de tornar as APIs mais simples, e para melhorar a manutenção do código fonte, (iii) os desenvolvedores não depreciam elementos antes de uma mudança devido ao aumento no esforço de manutenção, e (iv) a maioria dos desenvolvedores planejam documentar as mudanças, geralmente por *release notes* e *changelogs*. Na dissertação, também são fornecidas sugestões para projetistas, desenvolvedores de ferramentas, pesquisadores da área Engenharia de Software e desenvolvedores de APIs. Para concluir o trabalho, a ferramenta APIDIFF usada nesta dissertação para identificar *breaking changes*, foi atualizada com recursos importantes, por exemplo, suporte a operações de refatoração e integração com o sistema de controle de versões `git`.

Palavras-chave: Evolução de APIs, Compatibilidade, Mineração de repositórios de software.

Abstract

Libraries and APIs are commonly used in software development to reuse code and increase productivity. As most software systems, they also evolve, which may break contracts previously established with client systems, introducing *breaking changes* that can affect the compilation of the projects. However, the main reasons to introduce *breaking changes* in APIs are unclear. Therefore, in this master dissertation, we report the results of an almost 4-month long field study with popular Java libraries and frameworks. We configured an infrastructure to observe all changes in 400 libraries and to detect *breaking changes* shortly after their introduction in the code. We detected possible breaking changes in 61 projects. After identifying *breaking changes*, we contacted the developers. Our main goal is to understand the reasons behind APIs changes. During the study, we identified 59 *breaking changes*, confirmed by the developers of 19 projects. By analyzing the developers' answers, we report that (i) 39% of the changes investigated in the study may have an impact on clients, but requiring a minor effort to migrate to the new API versions, (ii) *breaking changes* are mostly motivated by the need to implement new features, by the desire to make the APIs simpler, and to improve maintainability, (iii) developers do not deprecate elements before a change due to increase on maintainability effort, and (iv) most developers plan to document the *breaking changes*, usually by *release notes* and *changelogs*. We also provide suggestions to language designers, tool builders, software engineering researchers, and API developers. To conclude the work, we describe a new version of APIDIFF tool, which was used in this dissertation to identify API *breaking changes*. This version includes important new features, like support to refactoring operations and integration with the `git` version control system.

Palavras-chave: API evolution, Breaking changes, Mining software repositories.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

Software libraries are commonly used nowadays to support development. For example, there are more than 200K unique libraries registered on Maven's central repository, a popular package management and build tool for Java. They cover distinct usage scenarios in modern software development, from mobile and Web programming to math and statistical analysis. These functionalities are provided to client systems via *Application Programming Interfaces* (APIs), which are contracts that clients rely on [Reddy, 2011], granting several benefits to them:

- Decrease of development time and costs [Moser and Nierstrasz, 1996];
- Reuse of well designed and tested component solutions [Brito et al., 2018c];
- Increase of productivity [Konstantopoulos et al., 2009].

In principle, APIs should be stable and backward-compatible when evolving, so that clients can confidently rely on them. In practice, however, the literature shows the opposite: APIs are often unstable and backward-incompatible [Wu et al., 2010a; Robbes et al., 2012; McDonnell et al., 2013; Bogart et al., 2016]. A recent study points that 28% out of 500K API changes break backward compatibility, that is, they may cause side effects on client systems [Xavier et al., 2017a]. This includes not only small and less critical libraries, but also world wide ones such as Android, JUnit, and Eclipse, in which the impact is magnified to millions of clients. API breaking changes comprise from simple modifications, such as the change of a method signature or return type, to more critical and dangerous ones, such as the removal of a public type, method, or field. In this context, one important question is not completely answered by the literature: *despite being recognized as a programming practice that may harm client applications, why do developers break APIs?* A better understanding of these reasons may support the development of new language features and software engineering approaches and tools to improve library maintenance practices.

In this dissertation, we study the motivations driving API breaking changes from the perspective of library developers. Using a tool designed to detect API breaking changes, we mined daily commits of 400 relevant libraries and frameworks hosted on GitHub to better understand the reasons behind the changes, the possible impact on client systems, and the practices adopted to warn clients about the disruption in the contract. During 116 days, we detected 282 possible breaking changes, sent emails to 102 developers, and received 56 responses, which represents a response rate of 55%.

Furthermore, several approaches have been proposed to handle API evolution, for instance, to detect refactoring operations [Silva and Valente, 2017], to capture and replay refactorings [Henkel and Diwan, 2005], and to track popularity and migration of APIs [Hora and Valente, 2015]. 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 (breaking) changes may affect clients?
- How stable is this library?

Tools that automatically provide answers to these questions may support API creators when writing migration documents in order to help clients when updating their applications or to identify and revert accidental breaking changes. On the client side, such tools 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, in this master dissertation, we also worked on a new version of APIDIFF [Xavier et al., 2017a,b], a tool to detect API breaking and non-breaking changes between two versions of a Java library by inspecting the API elements (types, methods, and fields). We made its source code publicly available, and we added several new features. For example, we included support to the distributed `git` version control system, and the detection of refactoring operations.

1.2 Proposed Work

First, in this master dissertation we conducted an empirical study with real developers to understand the motivations to perform API breaking changes. Additionally, we extended APIDIFF with relevant new features, based on our own usage experience. Next, we describe the empirical study as well as the new tool version:

Breaking Change Motivations. We performed a investigation with API creators from 400 popular Java libraries and frameworks. Specifically, we questioned them regarding the reasons to break API contracts, the possible impact on external client systems, and the strategies to document these modifications. To support this first study, we asked the following research questions:

1. **How often do changes impact clients?** In this question, we investigate the rate of API changes with impact on client systems. We discover that 39% of the changes may have an impact on clients. However, a minor migration effort is required in most cases, according to the surveyed developers.

2. **Why do developers break APIs?** In this second question, we analyze the reasons to perform breaking changes. We identified three major motivations to break APIs, including changes (i) to support new features, (ii) to simplify the APIs, and (iii) to improve maintainability.

3. **Why do not developers deprecate broken APIs?** In this question, we investigate the motivation to API creators do not deprecate API elements before a change in the contract. Among the received answers, most developers mentioned the increase on maintenance work as the reason for not deprecating broken APIs.

4. **How do developers document breaking changes?** In this final question, we survey the methods used by developers to document the modifications in API elements. Among the received answers, most developers plan to document the detected breaking changes, mainly using release notes or changelogs.

APIDIFF 2.0. APIDIFF is a tool to detect API breaking and non-breaking changes, which was proposed and used in a previous work [Xavier et al., 2017a,b]. We selected this tool to perform the survey with the developers because it worked at three API levels (types, methods, and fields). During our study, however, we identified some limitations on APIDIFF; therefore, we extended it with a set of new features, resulting in APIDIFF 2.0.

1.3 Contributions

We highlight the main contributions of the empirical study presented in this master dissertation as follows:

- We provide a large-scale study to understand the reasons developers break backward API compatibility; the strategies used to document these breaking changes, and the impact on client systems;
- We present an extensive list of empirically-justified implications of our study, involving important implications to: language designers, researchers, tool builders, and practitioners.

The APIDIFF new version includes the following new features:

- Integration with the git version control system, providing options to analyze commits, project history, and an output with information about the changes and developers;
- Integration with REFDIFF, a tool to detect refactoring operations, and, consequently, new changes in the catalog;
- Options to filter out changes from specific packages and a customizable output;
- Automatic dependency management, using Maven Central Repository.

1.4 Publications

This master dissertation produced the following publications, and, therefore, it contains material of them:

- Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018b). Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265 (Qualis A2)
- Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018a). APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*, pages 507–511

The following publications represent earlier research efforts during this master's work:

- Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017a). Historical and impact analysis of API breaking changes: A large scale study. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147 (Qualis A2)

- Brito, A., Hora, A., and Valente, M. T. (2016b). Um estudo em larga escala sobre o uso de APIs internas. 4th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), pages 1–8 (Qualis B5)

- Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2016). Um estudo em larga escala sobre estabilidade de APIs. 4th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), pages 1–8 (Qualis B5)

- Brito, A., Hora, A., and Valente, M. T. (2016a). JAVALI: Uma ferramenta para analise de popularidade de APIs Java. 7th Brazilian Conference on Software: Theory and Practice (CBSoft, Tools Track), pages 1–8

1.5 Outline of the Dissertation

The remaining of this dissertation is organized as follows:

- **Chapter 2** introduces the main concepts related to this dissertation, covering topics related to Application Programming Interfaces (APIs). Then, we discuss API popularity, evolution and migration, and backward compatibility. Importantly, we define core terms for this study, such as *breaking changes* and *non-breaking changes*.

- **Chapter 3** describes our empirical study, a survey with developers to understand the reasons behind breaking API contracts. We start by presenting the APIDIFF tool, which was used to monitor the repositories and to detect possible breaking changes. Then, we detail the study design and results, summarizing the received answers. Finally, we present practical implications and threats to validity.

- **Chapter 4** presents the new version of APIDIFF tool, which was proposed after our experience using the initial tool version in the empirical study. This chapter describes the limitations on APIDIFF 1.0, and the added features. Finally, we present usage examples and conclusions.

- **Chapter 5** covers the state of the art. We grouped related work in four categories: studies on API evolution, studies on breaking changes, API changes tools, and studies using the *firehouse* interview method. We also discuss and compare the major differences between each study and ours.
- **Chapter 6** presents the main contributions of this master dissertation, including an overview about the study, limitations, and future work.

Chapter 2

Background

This chapter provides the background required to understand the study presented in this dissertation. First, we present API-related concepts (Section 2.1). Then, we discuss API evolution and popularity (Section 2.2), providing an overview about breaking and non-breaking changes. Finally, we conclude the chapter in Section 2.3.

2.1 Application Programming Interfaces

Software libraries provide functionalities via Application Programming Interfaces (APIs), which are contracts commonly used by clients to support software development [Moser and Nierstrasz, 1996; Konstantopoulos et al., 2009; Raemaekers et al., 2012]. These functionalities are exposed to clients using visibility modifiers. For example, in Java, we can control access to API elements (i.e., types, methods, and fields) using the modifiers *public* and *protected*. Figure 2.1 illustrates the communication between client systems and a software component via an API.

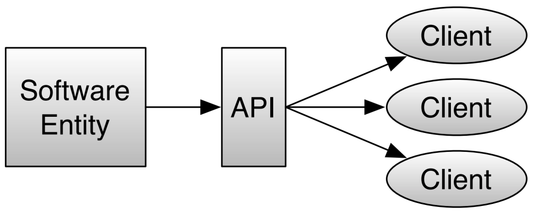


Figure 2.1: API acting as interface for software clients [Montandon, 2013]

Listing 2.1 shows an example of a Java API: a code snippet of the `Observable` class, which is available in the package `java.util`. This API provides functionalities to implement the observable object in the Observer design pattern [Gamma et al., 1994]. In this example, the methods `addObserver` (Line 7), `deleteObserver`

(Line 18), `countObservers` (Line 22) have the visibility modifier `public`, and the methods `clearChanged` (Line 26) and `setChanged` (Line 30) have the visibility modifier `protected`. In this dissertation, these methods are called as API elements. By contrast, the fields `changed` (Line 3) and `obs` (Line 5) have the modifier `private`, thus they are not accessible to external clients. Additionally, the class `Observable` itself is public for external systems, consequently it is also an API element.

```
 1 public class Observable {
 2
 3     private boolean changed = false;
 4
 5     private Vector obs;
 6
 7     public synchronized void addObserver(Observer o) {
 8
 9         if (o == null){
10             throw new NullPointerException();
11         }
12
13         if (!obs.contains(o)) {
14             obs.addElement(o);
15         }
16     }
17
18     public synchronized void deleteObserver(Observer o) {
19         obs.removeElement(o);
20     }
21
22     public synchronized int countObservers() {
23         return obs.size();
24     }
25
26     protected synchronized void clearChanged() {
27         changed = false;
28     }
29
30     protected synchronized void setChanged() {
31         changed = true;
32     }
33
34 }
```

Listing 2.1: Example of API elements in class `java.util.Observable`

The literature confirms that modern software development heavily depends on APIs, since they bring several benefits [Moser and Nierstrasz, 1996; Konstantopoulos et al., 2009; Raemaekers et al., 2012]. Particularly, developers use APIs to provide source code reuse, improve productivity, and, consequently, decrease development costs. According to JAVALI[1], a tool to measure and assess the popularity of Java libraries, there are more than 250K Java clients hosted on GitHub. The `java.util` package, which provides utility functions, has more than 200K client systems. In the Android context, the class `android.view.View` (an API commonly used to create *widgets* in Android apps) has about 49K clients.

2.2 API Evolution and Compatibility

Change is a common practice in software development. Daily, developers create, remove, and update source code to accommodate new features, fix bugs, and improve code quality. As most software systems, libraries and frameworks also evolve, as well as their APIs.

During software development, ideally, API creators should strive to properly maintain API contracts, avoiding side effects and negative impact on their clients. However, the literature presents that *API contracts are commonly broken* [Xavier et al., 2017a; Bogart et al., 2016; Dig and Johnson, 2006]. Particularly, a recent study from our research group with over 300 Java libraries shows that API creators often break backward compatibility [Xavier et al., 2017a]. In this context, there are two API change categories [Xavier et al., 2017a; Dig and Johnson, 2005] as follows. Sections 2.2.1 and 2.2.2 present more details about these concepts.

- ***Breaking Changes:*** API change that breaks backward compatibility with clients.
- ***Non-breaking Changes:*** API change that does not break their clients.

2.2.1 Breaking Changes

We named as Breaking Changes (BC) the changes performed in API elements that break backward compatibility [Xavier et al., 2017a; Dig and Johnson, 2006; Bogart et al., 2016; Kula et al., 2018]. Theses changes may generate (or not) compilation errors. Listing 2.2 shows an evolution example of the `Observable` class, which was presented in Section 2.1, with breaking changes. In the first part, the `addObserver` method was renamed to `includeObservable` (Lines 5–12). In this case, the method renaming would result in a compilation error on client systems, that would need to

[1]`http://java.labsoft.dcc.ufmg.br/javali`

modify their source code to use the new name. Listing 2.2 shows another syntactic change: the method `countObservers` was removed from class `Observable` (Lines 20–22). In this case, removing an element would also result in a compilation error on the client side, that would have an inconsistent method call.

In contrast to syntactic changes, some modifications do not affect the compilation of client systems; however, their behavior may change [Dig and Johnson, 2006; Mostafa et al., 2017]. In this case, it is required to perform tests on the client applications to identify possible effects. Listing 2.2 shows an example of a behavioral modification (Lines 16–18); the reserved word *synchronized* was removed from method `deleteObserver`. In this instance, the clients would not not face any compilation error. However, *synchronized* is used to avoid access to an element in Java. Consequently, after the change, the `Observable` class might not ensure the same state of the vector `obs` for all threads. For example, two threads can remove the same object at the same time, causing unexpected behavior. Behavioral backward incompatibilities are outside of the scope of this dissertation. Instead, we focus on changes that cause compilation errors in client applications.

```
 1   public class Observable {
 2
 3       ...
 4
 5   -   public synchronized void ~~add~~includeObserver(Observer o) {
 6
 7           if (o == null){
 8               throw new NullPointerException();
 9           }
10
11           if (!obs.contains(o)) {
12               obs.addElement(o);
13           }
14       }
15
16   -   public ~~synchronized~~ void deleteObserver(Observer o){
17           obs.removeElement(o);
18       }
19
20   -   ~~public synchronized int countObservers(){~~
21   -       ~~return obs.size();~~
22   -   ~~}~~
23
24       ...
25
26   }
```

Listing 2.2: BCs performed in the `Observable` class presented in Listing 2.1

2.2.2 Non-breaking Changes

We named as Non-breaking Changes (NBC), the changes performed in API elements that do not break backward compatibility [Xavier et al., 2017a; Dig and Johnson, 2006]. Listing 2.3 presents examples of non-breaking changes. The first one shows a visibility gain in methods `clearChanged` and `setChanged`. The access modifier `protected` was changed to `public` (Lines 5 and 9). In this way, these methods can be accessed by other elements, besides their packages classes and subclasses.

The second example of non-breaking change is a method addition. The method `deleteObservers`, which removes a list of observers from the vector `obs`, was added. In this case, the clients can use the new method after the migration for the new library version.

```
1   public class Observable {
2
3       ...
4
5   -   ~~protected~~public synchronized void clearChanged() {
6           changed = false;
7       }
8
9   -   ~~protected~~public synchronized void setChanged() {
10          changed = true;
11      }
12
13  +   public synchronized void deleteObservers(List<Observer> observers) {
14  +       obs.removeAll(observers);
15  +   }
16
17  }
```

Listing 2.3: NBCs performed in the `Observable` class presented in Listing 2.1

It is important to mention that changes on deprecated API elements are classified as NBCs, since clients were warned about possible incompatibilities in previous releases [Brito et al., 2016c]. There are three strategies provided by the Java language to warn clients about deprecated elements, helping them dealing with the upgrade:

- `@Deprecated` annotation: generates a warning by the compiler when the developer uses the annotated element;
- `@deprecated` tag on JavaDoc: warns the developer about the deprecated elements;
- `@link` tag on JavaDoc: informs the replacement element on JavaDoc (available since Java 1.2).

Listing 2.4 presents an example of a deprecated element. The method `addObserver` (Line 9) was renamed to `includeObserver` (Line 20). In addition to including the observer in the vector `obs`, the new method throws an exception if the element already exists in the vector (Line 26). As we can notice, the example shows the usage of the reserved words in Java (`@deprecated`, `@link`, `@Deprecated`) to handle deprecated elements. Instead of simply removing the original method `addObserver`, it has been deprecated (Line 8), and then a JavaDoc has been added with a replacement element suggestion (Lines 5–7). Therefore, in future versions, API creators may remove the deprecated method `addObserver` without breaking clients, since they were previously warned.

```
 1  public class Observable {
 2
 3      ...
 4
 5      /**
 6      * @deprecated Use {@link #includeObserver(Observer)} instead.
 7      */
 8      @Deprecated
 9      public synchronized void addObserver(Observer o) {
10
11          if (o == null){
12              throw new NullPointerException();
13          }
14
15          if (!obs.contains(o)) {
16              obs.addElement(o);
17          }
18      }
19
20      public synchronized void includeObserver(Observer o) {
21
22          if (o == null){
23              throw new NullPointerException();
24          }
25
26          if(obs.contains(o)){
27              throw new ObserverConflictException();
28          }
29
30          obs.addElement(o);
31      }
32
33  }
```

Listing 2.4: Example of deprecation API elements

2.3 Final Remarks

In this chapter, we presented key concepts related to this dissertation. Specifically, we detailed the concepts of Application Programing Interfaces (APIs). Finally, we defined and illustrated two important concepts to this dissertation: *breaking changes*, changes that break backward compatibility, and (ii) *non-breaking changes*, changes that do not break clients.

Chapter 3

Why and How Java Developers Break APIs

In this chapter, we present the main study of this dissertation, which intends to reveal the motivations driving API breaking changes, from the perspective of library developers. By daily mining commits of relevant libraries, we searched for API breaking changes, and, when detected, we sent emails to developers to better understand the reasons behind the changes, the real impact on client applications, and the practices adopted to alleviate the breaking changes. We also characterize the most common program transformations that lead to breaking changes. This chapter is organized as follows. Section 3.1 presents the investigated research questions. Section 3.2 introduces the tool and approach used to detect API breaking changes. Section 3.3 details our study design and Section 3.4 presents our results. We discuss the implications of the study in Section 3.5. Finally, Section 3.6 states threats to validity and Section 3.7 concludes the chapter.

3.1 Research Questions

In order to better understand the reasons behind breaking API contracts, we investigate four research questions:

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

improve maintainability.

3. *Why don't developers deprecate broken APIs?* Most developers mentioned the increase on maintainability effort as the reason for not deprecating broken APIs.

4. *How do developers document breaking changes?* Most developers plan to document the detected breaking changes, mainly using *release notes* and *changelogs*.

The goal of this study is to support the development of new language features and software engineering approaches and tools to improve library maintenance practices. By following a firehouse interview method [Rogers, 2003], we monitored 400 real world Java libraries and frameworks hosted on GitHub during 116 days.

3.2 APIDiff 1.0

To detect breaking changes, we use a tool named APIDIFF (version 1.0), which was implemented and used by Xavier et al. [2017a] in a study about the frequency and impact of breaking changes. Essentially, APIDIFF compares two versions of a library and lists all changes in the signature of public methods, constructors, fields, annotations, and enums. In this paper, the results produced by APIDIFF are named *Breaking Change Candidates* (BCC). The reason is that changes in public elements—as identified by APIDIFF—do not necessarily have an impact on API clients. For example, the changed elements may denote internal or low-level services, which are designed only for local usage. To clarify this question, we conducted a survey with API developers, to confirm whether the BCCs detected by APIDIFF are indeed *breaking changes* (see Section 3.3).

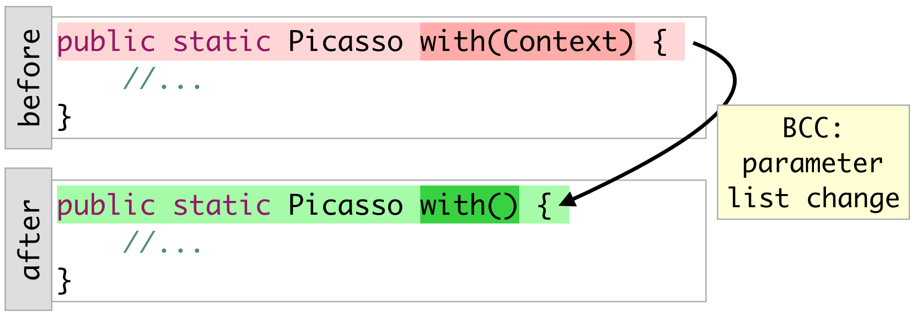
Definition: Changes detected by APIDIFF in public API elements are named Breaking Change Candidates (BCC).

Table 3.1 lists the BCCs detected by APIDIFF. These changes refer to the following API elements: types, methods, or fields. BCCs on types include, for example, drastic changes, like the removal of a type from the code. But subtle changes in public types are also detected, including changing a type visibility from public to another modifier, changing the supertype of a type, adding a *final* modifier to a type (to disable inheritance), or removing the *static* modifier of an inner class. Besides the changes detected to types, BCCs in methods include changes in return types or parameter lists. Changes in fields include, for example, changing the default value of a field.

Figure 3.1 shows an example of BCC detected by APIDIFF in a method of SQUARE/PICASSO (an image downloading library). According to the developer who performed this change, he removed the parameter `Context` from method `with` to simplify the API, since this parameter can be retrieved in other ways.

Table 3.1: BCCs detected by APIDIFF

Element	**BCC**
Type	REMOVE CLASS, CHANGE IN ACCESS MODIFIERS, CHANGE IN SUPERTYPE, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER
Method	REMOVE METHOD, CHANGE IN ACCESS MODIFIERS, CHANGE IN RETURN TYPE, CHANGE IN PARAMETER LIST, CHANGE IN EXCEPTION LIST, ADD FINAL MODIFIER, REMOVE STATIC MODIFIER
Field	REMOVE FIELD, CHANGE IN ACCESS MODIFIERS, CHANGE IN FIELD TYPE, CHANGE IN FIELD DEFAULT VALUE, ADD FINAL MODIFIER



```
public static Picasso with(Context) {
    //...
}
```

```
public static Picasso with() {
    //...
}
```

Figure 3.1: Example of BCC detected by APIDIFF at method level

As implemented by the current APIDIFF version, changes in deprecated API elements (i.e., elements annotated with `@Deprecated`) are not BCCs. The rationale is that clients of these elements were previously warned that they are no longer supported, and, therefore, subjected to changes or even to removal. Finally, APIDIFF warns if a BCC is performed in an experimental or internal API [Businge et al., 2015; Mastrangelo et al., 2015]. For this purpose, the tool checks if the qualified name of the changed API element includes a package named `internal`, as in this example: `io.reactivex.internal.util.ExceptionHelper`. With this warning, the goal is to alert users that the identified BCC is probably a false breaking change.

As presented in Table 3.1, APIDIFF does not use the term refactoring to name BCCs. For example, the RENAME of an API element `A` to `B` is identified as the removal of the element `A` from the code. Similarly, a MOVE CLASS/METHOD/FIELD from location `C` to a new location `D` is identified as the removal of the element from its original location `C`. In order to use the most appropriate names to identify these operations, we manually inspected the BCCs detected by APIDIFF. For each commit with a BCC, we analyzed its textual diff, as generated by GitHub. The detection of refactorings performed on classes (RENAME/MOVE CLASS) was facilitated because these operations are automatically indicated in the textual diff computed by GitHub. For example, Figure 3.2 shows a screenshot of a diff in FACEBOOK/FRESCO that includes a MOVE CLASS.[1] At the top of the figure, there is an indication that class `DrawableFactory` was moved from package `com.facebook.drawee.backends.pipeline` to package `com.facebook.imagepipeline.drawable`. By contrast, to detect RENAME/MOVE METHOD/FIELD we needed to perform a detailed inspection on the diffs results.

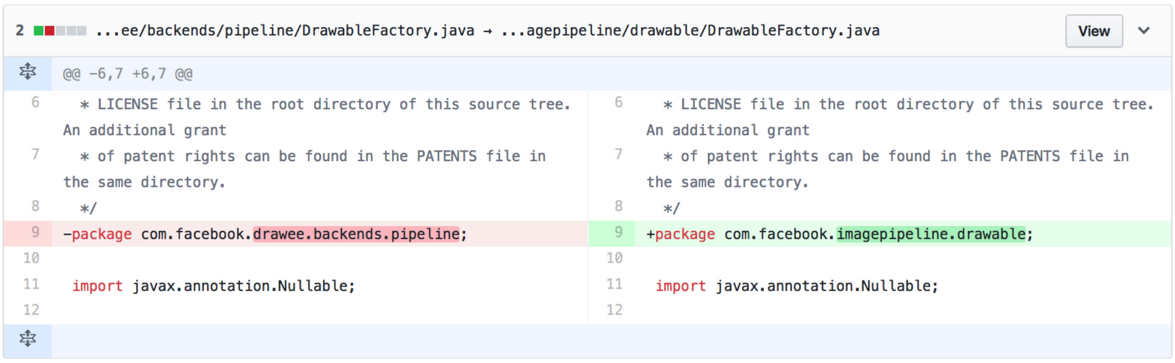

```
2 ...ee/backends/pipeline/DrawableFactory.java → ...agepipeline/drawable/DrawableFactory.java
@@ -6,7 +6,7 @@
 * LICENSE file in the root directory of this source tree.      * LICENSE file in the root directory of this source tree.
An additional grant                                            An additional grant
 * of patent rights can be found in the PATENTS file in         * of patent rights can be found in the PATENTS file in
the same directory.                                            the same directory.
 */                                                             */
-package com.facebook.drawee.backends.pipeline;                +package com.facebook.imagepipeline.drawable;

 import javax.annotation.Nullable;                              import javax.annotation.Nullable;

```

Figure 3.2: Screenshot of a textual diff produced by GitHub in FACEBOOK/FRESCO. A MOVE CLASS is indicated in the header line.

3.3 Study Design

3.3.1 Selection of the Java Libraries

First, we selected the top-2,000 most popular Java projects on GitHub, ordered by number of stars and that not are forks (on March, 2017). We used this criteria because stars is a common and easily accessible proxy for the popularity of GitHub projects [Borges et al., 2016]. Next, we discarded projects that do not have the following keywords in their short description: *library(ies)*, *API(s)*, *framework(s)*. We also manually removed *deprecated* projects from this list, i.e., projects that have deprecated in their short description, to focus the study on active repositories.

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

These steps resulted in a list of 449 projects. Then, we manually inspected the documentation, wiki, and web pages of these projects to guarantee they are libraries or similar software. As a result, we removed 49 projects. For example, GOOGLESAMPLES/ANDROID-VISION has the following short description: *Sample code for the Android Mobile Vision API.* Despite having the keyword API in the description, this repository is neither a library nor a framework, but just a tutorial about a specific Android API. Thus, the final list consists of 400 GitHub projects, including well-known systems such as JUNIT-TEAM/JUNIT4 (a testing framework), SQUARE/PICASSO (an image downloading and caching framework), and GOOGLE/GUICE (a dependency injection library).

3.3.2 Detecting Breaking Changes Candidates

During 116 days, from May 8th to August 31th, 2017, we monitored the commits of the selected projects to detect BCCs. To start the study, on May 8th, 2017 we cloned the selected 400 libraries and frameworks to a local repository. Next, on each work day, we ran scripts that use the *git fetch* operation to retrieve the new commits of each repository. We discarded a new commit when it did not modify Java files. Furthermore, on Git, developers can work locally in a change and just submit the new revision (via a *git push*) after a while. Therefore, we also discarded commits with more than seven days, to focus the study on recent changes, which is important to increase the chances of receiving feedback from developers (see Section 3.3.3). We also discarded commits representing merges because these commits usually do not include new features; moreover, merges have two or more parent commits, which leads to a duplication of the BCCs identified by APIDIFF [Xavier et al., 2017a,b]. Finally, we manually discarded commits in branches that only contain test code.

APIDIFF identified 282 BCCs in 110 commits, distributed over 61 projects (47% of the set of 130 libraries and frameworks with commits detected during the study period). Figure 3.3 presents the distribution of number of stars, age (in years), number of contributors, and number of commits of the initial selection of 400 libraries and frameworks (labeled as *Libraries*) and of the 61 projects with BCCs (labeled as *Libraries with BCCs*).

The distributions of *Libraries with BCCs* are statistically different from the initial selection of 400 libraries in age, number of contributors, and number of commits, but not regarding the number of stars (according to Mann-Whitney U Test, p-value $\leq 5\%$). To show the effect size of this difference, we computed Cliff's delta (or d) [Cliff, 2014]. The effect is medium for age, and large for number of contributors and commits.

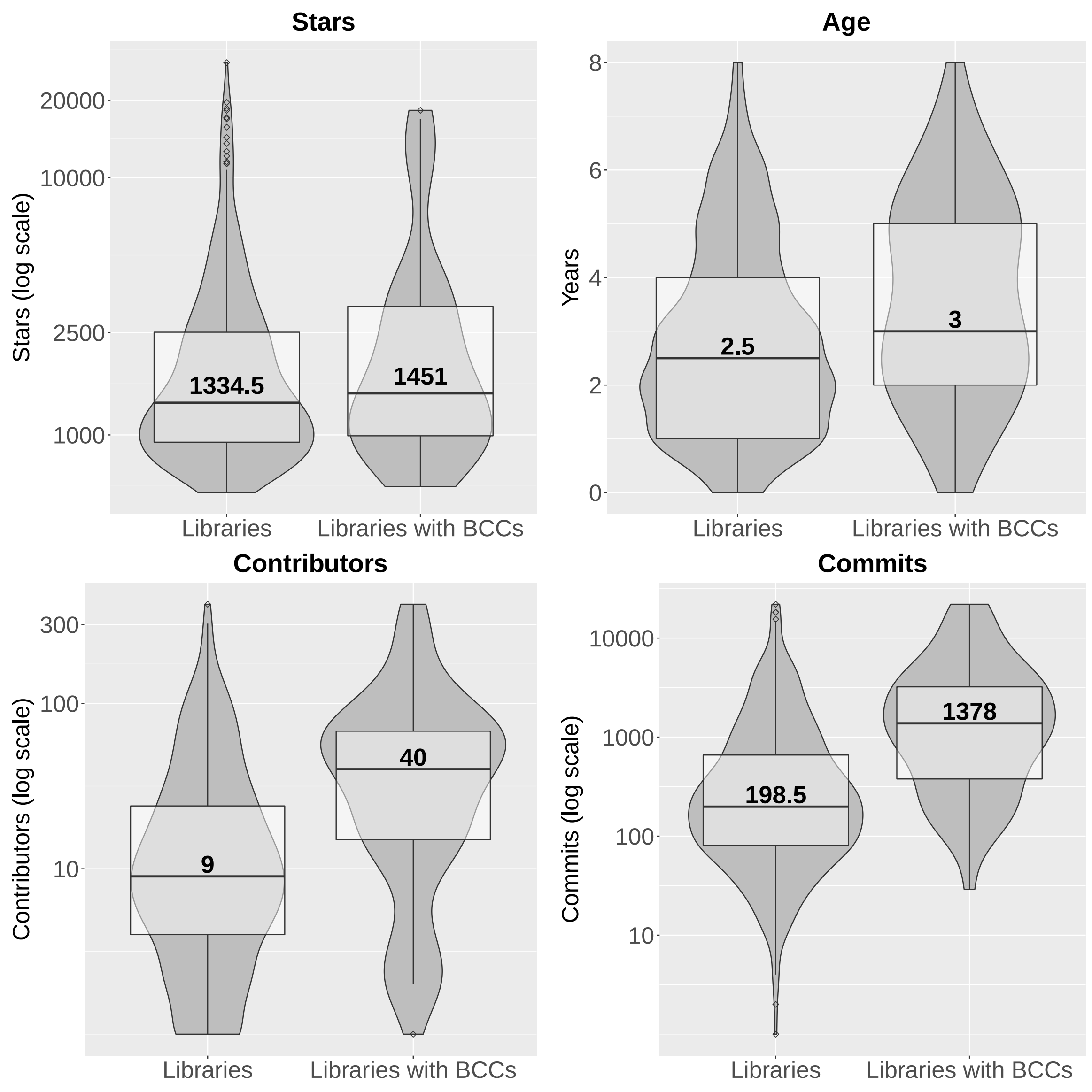


Figure 3.3: Distribution of number of stars, age, number of contributors, and number of commits of the initial 400 *Libraries* and of the 61 *Libraries with BCCs*

In other words, libraries with BCCs are moderately older (3 vs 2.5 years, median measures), have more contributors (40 vs 9) and more commits (1,378 vs 198.5) than the original list of libraries selected for the study.

Finally, Figure 3.4 shows the distribution of BCCs per project, considering only *Libraries with BCCs*. The median is two BCCs per project and the system with the highest number of BCCs is ROBOLECTRIC/ROBOLECTRIC, with 38 BCCs (including 35 BCCs where public API elements were changed to protected visibility).

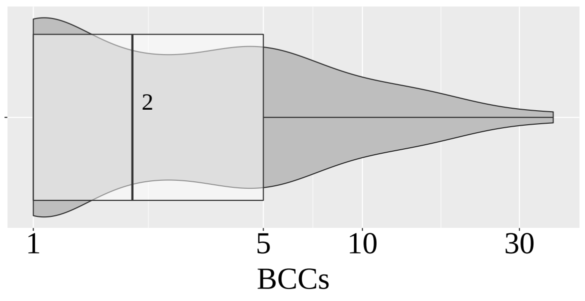


Figure 3.4: BCCs per project

3.3.3 Contacting the Developers

Among the 282 BCCs considered in the study, 268 (95%) were detected in commits that contain a public email. Therefore, on each day of the study, after detecting such BCCs, we contacted the respective developers. In the emails sent to them (see a template in Figure 3.5), we added a link to the GitHub commit and a description of the BCC. Then, we asked four questions. With the first question, we intended to shed light on the real motivation behind the detected changes. With the second question, we intended to confirm whether the BCC detected by APIDIFF can break existing clients. With the third question, our interest was to understand why the developers have not deprecated the API element where the BCC was detected. Finally, with the last question, our interest was to investigate how often developers document BCCs.

We sent only one email to each developer. Specifically, whenever we detected BCCs by the same developer, but in different commits, we only sent one email to him, about the BCC detected in the first commit. In this way, we reduced the chances that developers perceived our emails as spam. Figure 3.6 shows the number of emails sent (blue line) and received (red line) on each day of the study. We detected BCCs and sent emails during 71 days (out of 116 days of the study).

It is also important to mention that before sending each email we inspected the respective commit description to guarantee it did not include an answer to the proposed questions. In the case of six commits, we found answers to the first question (*why did you perform these changes?*). As an example, we have the following commit description:

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

Dear [developer name],

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

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

[BCCs list] and [commit links]

Could you please answer the following questions:

1. Why did you perform these changes?

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

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

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

Figure 3.5: Mail to the authors of commits with BCCs detected by APIDIFF

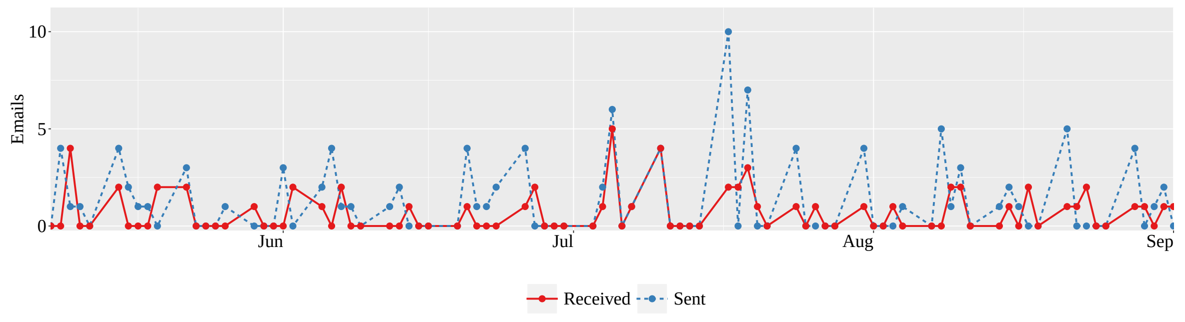


Figure 3.6: Number of emails sent and received per day

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

Now, [Class Name] can be configured to apply to different use cases ... Breaking changes: Remove [Class Name] (D22)

During the 116 days of the study, we sent 102 emails and received 56 responses, which represents a response ratio of 55%. Table 3.2 summarizes the numbers and statistics about the study design phase, as previously described in this section. After receiving all emails, we analyzed the answers using thematic analysis [Cruzes and Dyba, 2011], a technique for identifying and recording *themes* (i.e., patterns) in textual documents. Thematic analysis involves the following steps: (1) initial reading of the answers, (2) generating a first code for each answer, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. Steps 1 to 4 were performed independently by two authors of this paper. After this, a sequence of meetings was held to resolve conflicts and to assign the final themes (step 5). When quoting the answers, we use labels D1 to D60 to indicate the respondents (including four developers with answers coming from commits).

Table 3.2: Numbers about the study design

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

3.4 Results

3.4.1 How Often do Changes Impact Clients?

To answer this question, we first define breaking changes:

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

As presented in Figure 3.7, only 59 BCCs (39%) detected by APIDIFF are BCs. The remaining BCCs—which have not been confirmed by the respective developers—are called *unconfirmed BCCs*. Next, we characterize the BCs investigated in this study; we also reveal the reasons for the high percentage of unconfirmed BCs.

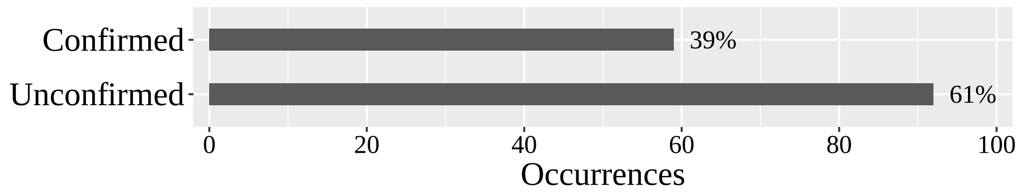


Figure 3.7: Confirmed and unconfirmed BCCs; confirmed BCCs are called BCs

Breaking Changes (BC). The 59 BCs detected in the study are distributed over 19 projects and 24 commits, including 20 commits with BCs confirmed by email and 4 commits with BCs declared in the commit description. Figure 3.8 shows the most common BCs. Among the Top-5, three are refactorings, including MOVE METHOD (11 occurrences), RENAME METHOD (8 occurrences), and MOVE CLASS (8 occurrences). The second most common BCs are the removal of an entire class (10 occurrences), which can be viewed as a drastic API change. The third most popular BCs are CHANGES IN METHOD PARAMETERS (9 occurrences). Considering the 17 types of BCs detected by APIDIFF (see Table 3.1), only 8 appeared in our study.

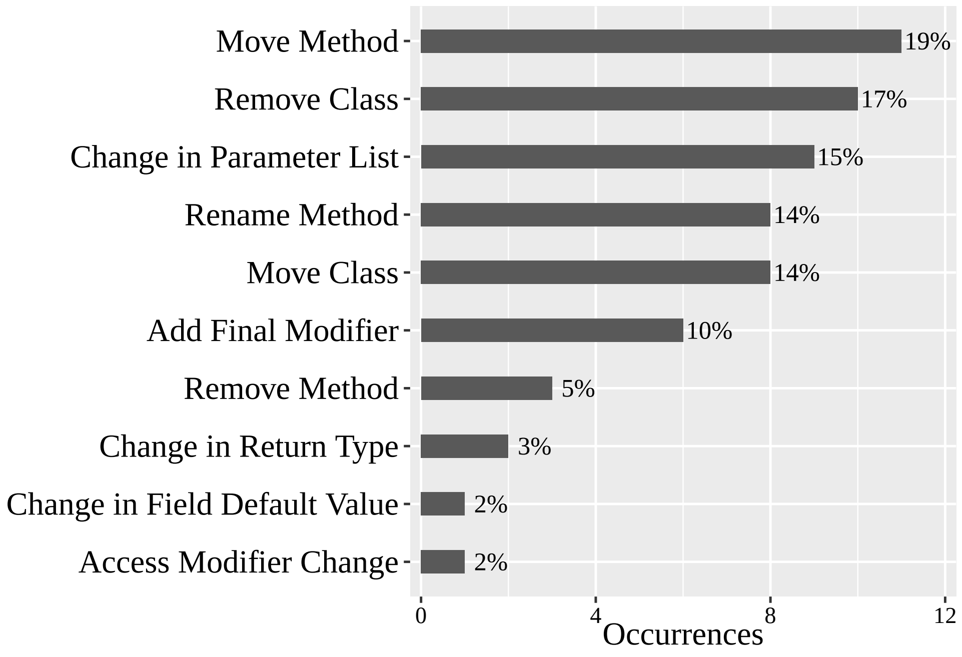


Figure 3.8: Most common breaking changes

Regarding the elements affected by the changes, Figure 3.9 shows the BCs grouped by API element: 35 BCs (59%) are performed on methods, followed by BCs on types (21 instances, 36%) and fields (3 instances, 5%).

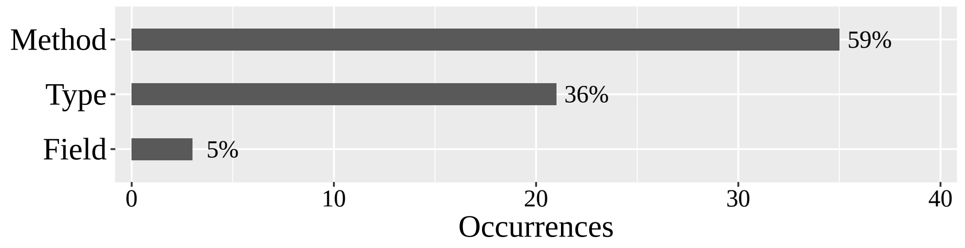


Figure 3.9: Most common breaking changes per API element

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

Unconfirmed BCCs. By contrast, in the case of 92 changes (61%), the surveyed developers did not agree they have an impact on clients. We organized the reasons mentioned by these developers on two major themes: internal APIs and experimental branches/new releases. Regarding the first theme, APIDIFF gives a warning about APIs that are likely to be internal; specifically, the ones implemented in packages containing the string `internal`, as recommended in the related literature [Businge et al., 2015; Mastrangelo et al., 2015]. Nonetheless, 21 developers mentioned that the BCCs occurred at internal (or low-level) APIs that do not include `internal` in their names, as in the following answers:

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

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

The second cause of unconfirmed BCCs are due to experimental branches. As described in Section 3.3.2, we monitored all branches of the analyzed repositories to contact the developers just after the changes. Consequently, in some cases, we considered BCCs in branches that do not represent major developments, e.g., branches dedicated to experiments, etc. Ten developers mentioned that the BCC occurred in such branches, as in the following answer:

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

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

3.4.2 Why do Developers Break APIs?

As reported in Table 3.3, we found four distinct reasons for breaking APIs: New Feature, API Simplification, Improve Maintainability, and Bug Fixing. In the following paragraphs, we describe and give examples of each of these motivations.

Table 3.3: Why do we break APIs?

Motivation	Description	Occur.
NEW FEATURE	BCs to implement new features	19
API SIMPLIFICATION	BCs to simplify and reduce the API complexity and number of elements	17
MAINTAINABILITY	BCs to improve the maintainability and the structure of the code	14
BUG FIXING	BCs to fix bugs in the code	3
OTHER	BCs not fitting the previous cases	6

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

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

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

In the first answer, D01 moved some classes from packages, before starting the implementation of a new feature. Therefore, clients should update their `import` statements, to refer to the new class locations. In the second answer, D13 renamed a method to better reflect its purpose after implementing a new feature. The rename should then be propagated to the method calls in the API clients.

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

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

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

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

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

Improve Maintainability. With 14 instances (24%), BCs performed to improve maintainability, i.e., internal software quality aspects, are the third most frequent ones. As examples, we have the following answers:

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

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

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

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

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

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

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

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

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

Figure 3.10 shows the top-3 most common BCs due to Feature Addition, API Simplification, and to Improve Maintainability. MOVE CLASS is the most common BC when implementing a new feature, with 7 occurrences. Specifically, when working on a new major release, developers tend to start by performing structural changes in the code, which include moving classes between packages. To simplify APIs, developers usually REMOVE CLASSES (5 instances) and also add a `final` modifier to methods (4 instances). The latter is considered a simplification because it restricts the usage of API methods; after the change, the API methods cannot be redefined in subclasses, but only invoked by clients. Finally, it is not a surprise that BCs performed to improve maintainability are refactorings. In this case, the three most popular BCs are due to MOVE METHOD (11 instances), RENAME METHOD (2 instances), and MOVE CLASS (1 instance). Interestingly, MOVE CLASS is also used when implementing a new feature.

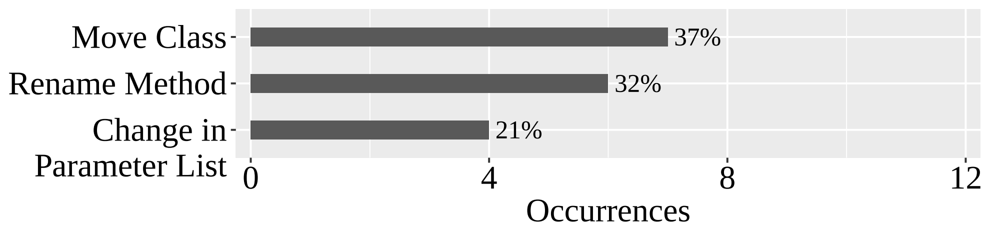
Summary: BCs due to refactorings are performed both to improve maintainability and to enable and facilitate the implementation of new features.

3.4.3 What Is the Effort on Clients to Migrate?

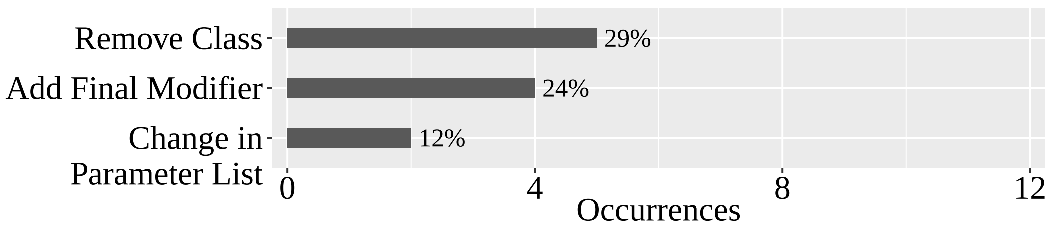
We organized the answers of this survey question in three levels: *minor*, *moderate*, or *major effort*. Seven developers answered the question. As presented in Figure 3.11, six developers estimated that the effort to use the new version is minor, while one answered with a *moderate* effort; none of them considered the update effort as a *major* one.

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

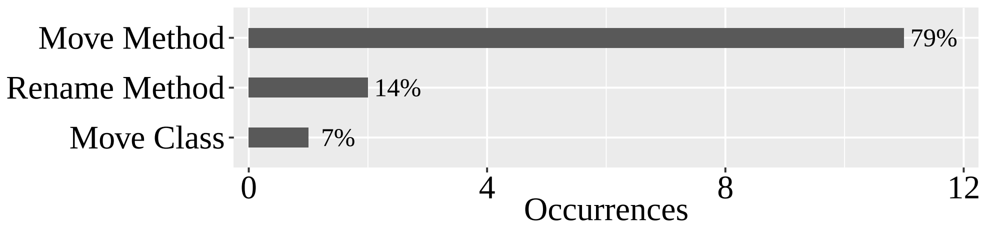
Work required should be minor, since it is just a change of a package. (D04, Move Class)



(a) BCs to implement new features



(b) BCs to simplify APIs



(c) BCs to improve maintainability

Figure 3.10: Top-3 most common BCs, grouped by motivation

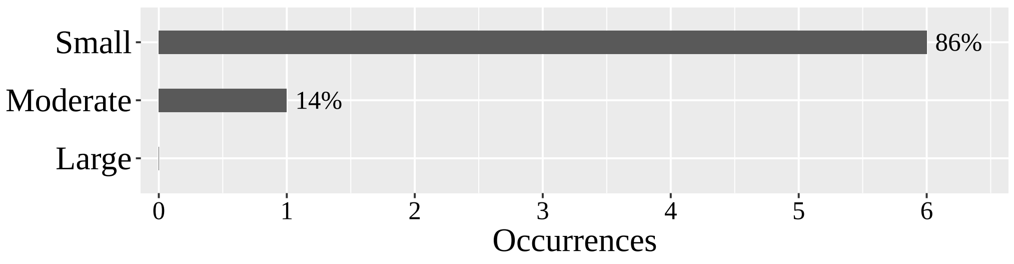


Figure 3.11: Effort required on clients to migrate

A single developer (D09) answered that a class removal may require a *moderate* effort on clients:

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

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

3.4.4 Why didn't you Deprecate the Old Implementation?

17 developers answered this survey question. As presented in Figure 3.12, they presented five reasons for not deprecating the API elements impacted by the BCs.

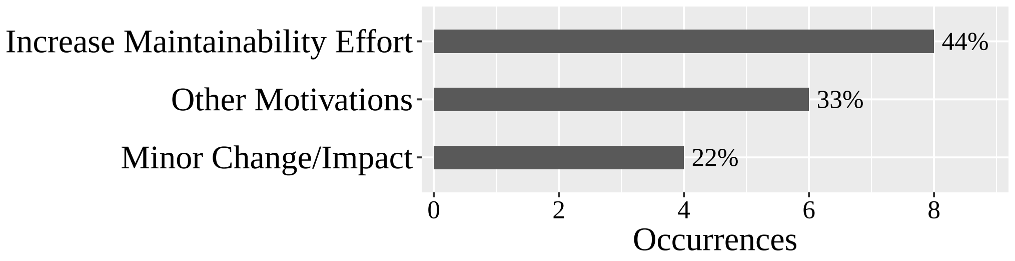


Figure 3.12: Reasons for not deprecating the old versions

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

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

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

Because the fix is so easy. (D15)

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

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

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

3.4.5 How do Developers Document Breaking Changes?

This question was answered by 18 developers. Among the received answers, 14 developers stated they intend to document the BCs. We analyzed these answers and extracted seven different documents they plan to use to this purpose (see Figure 3.13). Release Notes and Changelogs are the most common documents, mentioned by four developers each, followed by JavaDoc (3 developers).

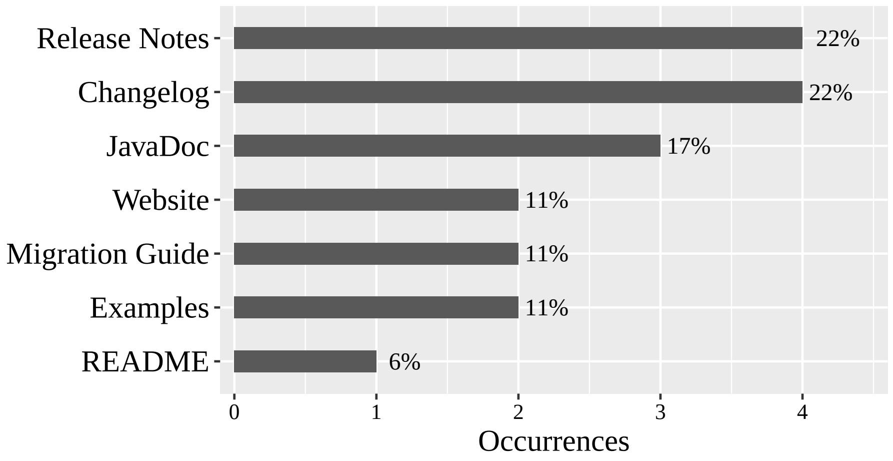


Figure 3.13: How do you plan to document the detected BCs?

Finally, four developers do not plan to document the BCs. For example, two of them considered the changes trivial and self-explained, as in the following answer:

The usage is simple so the code explains itself. (D02)

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

3.5 Implications

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

Language Designers. Among the 151 Breaking Changes Candidates (BCCs) with developers' answers, only 59 were classified as Breaking Changes (BCs). The other BCCs are mostly changes in internal or low-level APIs or changes performed in experimental branches. Since they are designed for internal usage only, developers do not view changes in these APIs as BCs. However, previous research has shown that occasionally internal APIs are used by external clients [Hora et al., 2016; Boulanger and Robillard, 2006; Businge et al., 2012, 2013, 2015; Dagenais and Robillard, 2008; Mastrangelo et al., 2015]. For example, clients may decide to use internal APIs to improve performance, as a workaround for bugs, or to benefit from undocumented features. This usage is only possible because internal APIs are public, as the official and documented ones; and their usage is not checked by the Java compiler. To tackle this problem, a new module system is being proposed to Java, which will allow developers to explicitly declare the module elements they want to make available to external clients.[2] The Java compiler will use these declarations to properly encapsulate and check the usage of internal APIs. Therefore, our study reinforces the importance of introducing this new module system in Java, since we confirmed that changes in internal API elements are frequent. We also confirmed that API developers use the `public` keyword in Java with two distinct semantics ("public only to my code" *vs* "public to any code, including clients").

Tool Builders. APIDIFF is an useful tool both to API developers and clients. API developers can use the tool to document changes in their APIs, e.g., to automatically generate changelogs or release notes. API clients can also rely on APIDIFF to produce these documents, in order to better assess the effort to migrate to API versions that are not properly documented. However, we also faced an important limitation when dealing with the output produced by APIDIFF. Currently, MOVE/RENAME operations are detected as a removal (REMOVE) followed by an addition (ADD) of an API element. As described in Section 3.2, to generate the correct names for these operations, we had to manually inspect the output produced by APIDIFF and the textual diff of the respective commits. Thus, we consider that APIDIFF implementation can follow existing approaches and tools [Kim et al., 2005; Silva and Valente, 2017; Silva et al., 2016] and automatically detect the cases where REMOVE followed by an ADD is indeed a RENAME (when confined to the same class) or a MOVE (when involving different classes) refactoring.

Researchers. Although based on a limited number of 59 BCs, our study reveals opportunities to improve the state-of-the-art on API design, evolution, and analysis. First, the study suggests that BCs are often motivated by the implementation of new features and that refactorings are usually performed at that moment, to support the implementation of the new code. In fact, a recent study on refactoring practices considering all types of GitHub projects, i.e., not restricted to libraries and frameworks, also shows that refactoring is mainly driven by the implementation of new requirements [Silva et al., 2016]. Therefore, we envision a new research line on techniques and tools to recommend refactorings and related program redesign operations, when a new version of an API is under design. In other words, the focus should be on API-specific remodularization techniques, instead of global remodularization approaches, as commonly proposed in the literature [Mitchell and Mancoridis, 2006; Praditwong et al.,

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

2011; Abdeen et al., 2009; Anquetil and Lethbridge, 1999; Terra et al., 2015]. Second, the study suggests that BCs are also motivated by a desire to reduce the number of API elements or reduce the possible usages of some elements (e.g., by making them `final`). Therefore, we envision research on API-specific static analysis tools (or API-specific linter tools), which could for example recommend the removal of useless parameters in API methods (as we found in 2 BCs), the insertion of a `final` modifier (as we found in 6 BCs) or even the removal of underused methods and classes (as we found in 2 BCs). The benefit in this case would be the recommendation of these changes at design time or during early usage phases, before the affected API elements gain clients and the change costs and impact increase. Third, the answers of the third survey question suggest that BCs may have a minor impact on clients (but according to a small sample of six developers). Thus, we envision further research on migration tools, which could help API clients to move to new API versions by providing recommendations on how to deal with trivial BCs [Dagenais and Robillard, 2008; Nguyen et al., 2010; Wu et al., 2010a; Zhang et al., 2012]. Fourth, the answers of the last survey question show that some API developers can be reluctant to use the deprecation mechanism provided by Java. Essentially, they argue that deprecation increases maintenance burden, by requiring updates on multiple versions of the same API element. Therefore, we also envision research on new and possibly lightweight mechanisms to API versioning. It is also possible to recommend the traditional mechanism only in special cases, particularly when the BCs might impact a large number of clients or require complex changes.

Practitioners. The study also provides actionable results and guidelines to practitioners, especially to API developers. First, we detected many unconfirmed BCCs in packages that do not have the terms `internal` or `experimental` (or similar ones) in their names. We recommend the usage of these names to highlight to clients the risks of using internal and unstable APIs. Second, the study also reveals that some BCs are caused by trivial programming mistakes, e.g., when a developer changes a method, removing a parameter from the body, but forgetting this parameter without use in the method signature. Since APIs are the external communication ports of libraries and frameworks, it is important that they are carefully designed and implemented. Third, most BCs detected in the study require trivial changes in clients, at least according to six surveyed developers. Thus, API developers should carefully evaluate the introduction of BCs demanding complex migration efforts, which can trigger a strong rejection by clients. Fourth, we listed good practices used by developers to document BCs, for example, changelogs and release notes.

3.6 Threats to Validity

External Validity. As usual in empirical software engineering, our findings are restricted to the studied subjects and cannot be generalized to other scenarios. Nevertheless, we daily monitored a large dataset of 400 Java libraries and frameworks, during a period of 116 days (almost 4 months). During this time, we questioned 102 developers about the motivations of breaking changes right after they had been performed. Due to such numbers, we consider that our findings are based on representative libraries, which were assessed during a large period of time, with answers provided by developers while the subject was still fresh in their minds. Moreover, our analysis is restricted to syntactical breaking changes, which result on compilation errors in clients. BCs that modify the API behavior without changing its signature, usually named Behavioral Backward Incompatibilities [Mostafa et al., 2017], are outside of the scope of this paper.

Internal Validity. First, we use APIDIFF to detect breaking changes between two versions of a Java library. Although this tool was implemented and used in our previous research [Xavier et al., 2017a], an error on its result would introduce false positives in our analysis. To mitigate this threat, we considered the breaking changes provided by the tool as *candidates* and only assessed those confirmed by their developers, which represents 39% of BCCs (see Sections 3.4.1). Second, we reinforce the subjective nature of this study and its results. As discussed in Section 3.3.3, a thematic analysis was performed to elicit the reasons that drive API developers to introduce BCs. Although this process was rigorously followed by two authors of the paper, the replication of this activity may lead to a different set of reasons. To alleviate this threat, special attention was paid during the sequence of meetings held to resolve conflicts and to assign the final themes. Third, against our belief, the trustworthiness and correctness of the responses is also a threat to be reported. To mitigate it, we strictly sent emails in no more than few days after the commits. This was important to guarantee a higher response rate and reliable answers, once the modifications were still fresh on developers' minds.

Construct Validity. The first threat relates to the selection of the Java libraries. As discussed in Section 3.3.1, we automatically discarded, from the top-2,000 most popular Java projects on GitHub, the ones that do not have the following keywords in their short description: *library(ies), API(s), framework(s).* Next, we manually discarded those that, although containing such words, do not actually represent a library. Since this process is conservative in providing a reliable dataset of projects that are libraries, we can not guarantee that we retrieved the whole set of actual libraries from the 2,000 projects. Second, our results stand on the agreement of developers on the detected BCCs. As observed in Section 3.4.1, most developers pointed out that the detected changes refer to internal or low-level APIs, mentioning that it is unlikely that they could break clients. However, previous research has shown that occasionally internal APIs are used by external clients [Hora et al., 2016; Boulanger and Robillard, 2006; Businge et al., 2012, 2013, 2015; Dagenais and Robillard, 2008; Mastrangelo et al., 2015]. Therefore, we might have excluded BCCs that could actually impact clients, but we decided to follow the conservative decision of only considering BCCs perceived by developers as having a high potential to break existing clients.

3.7 Final Remarks

In this chapter, we described a large-scale empirical study (400 libraries, 4-month long period, 282 possible breaking changes, 56 developers contacted by email) to understand *why* and *how* developers break APIs in Java. By using a firehouse interview method, we found that BCs are mainly motivated by the implementation of new features, to simplify the number of API elements, and to improve maintainability. The most common BCs are due to refactorings (47%); regarding the programming elements affected by BCs, most are methods (59%). According to the surveyed developers, the effort on clients to migrate to new API versions, after BCs, is minor.

We also listed some strategies to document BCs, like release notes and changelogs. Last but not least, we presented an extensive list of empirically-justified implications of our study, targeting four distinct audiences: programming languages designers, tool builders, software engineering researchers, and API developers. However, such implications should be viewed and interpreted with care, since they are derived from considering only 59 BCs and a single programming language (Java).

Chapter 4

APIDiff 2.0

In this chapter, we introduce a new version of the APIDiff, a tool to identify breaking and non-breaking changes between two versions of a Java library. APIDIFF 1.0 was used in a previous study about breaking changes performed over time [Xavier et al., 2017a]. However, during the survey study described in Chapter 3 we detected that APIDIFF 1.0 presents some limitations. Therefore, based on our learning, we improved APIDIFF to mitigate these issues. In Section 4.1, we describe the detected limitations and the new features added to APIDIFF 2.0. Section 4.2 presents the high level architecture and Section 4.3 presents a catalog of breaking and non-breaking changes detected by the APIDIFF. Section 4.4 presents usage scenarios in four real-world Java libraries. Finally, we detail the tool limitations in Section 4.5.

4.1 Features

During our survey study, we detected some limitations faced by APIDIFF 1.0. Specifically, we noticed the lack of features to support both API creators and clients. Thus, based on our own experience using the first version of the tool, we implemented new and relevant features, which resulted in a new APIDIFF version.

4.1.1 Git Operations

APIDIFF 1.0 does not provide any integration with the `git` version control system, which is used by GitHub (the most popular software repository with more than 80 million projects and a community with more than 27 million developers worldwide).[1] In fact, APIDIFF 1.0 simply receives as input two folders with different project versions.

[1]On March, 2018

As a first improvement, we added the `git` support to APIDIFF 2.0, which facilitates access to several commit metadata, such as author name, author email, commit date, and changed files. Listing 4.1 shows the interface `DiffDetector` from APIDIFF 2.0, which provides features to support `git` operations and is implemented by `APIDiff` class.

```
1  public interface DiffDetector {
2  
3      public Result detectChangeAtCommit(String commitId, Classifier classifier)
4      throws Exception;
5  
6      public Result detectChangeAllHistory(String branch, List<Classifier>
           classifiers)
7      throws Exception;
8  
9      public Result detectChangeAllHistory(String branch, Classifier classifier)
10     throws Exception;
11 
12     public Result detectChangeAllHistory(List<Classifier> classifiers)
13     throws Exception;
14 
15     public Result detectChangeAllHistory(Classifier classifier)
16     throws Exception;
17 
18     public Result fetchAndDetectChange(List<Classifier> classifiers)
19     throws Exception;
20 
21     public Result fetchAndDetectChange(Classifier classifier)
22     throws Exception;
23 }
```

Listing 4.1: DiffDetector interface

There are different signatures for the methods `detectChangeAllHistory` (Lines 6–15) and `fetchAndDetectChange` (Lines 18–21), allowing the analysis of all branches of a project or a specific one. In addition, we can filter out packages according to their names using the parameter `Classifier`. More specifically, the `DiffDetector` interface includes the following features, with the purpose of supporting `git` operations:

- **Detecting changes in specific commit.** By using this feature, which is provided by method `detectChangeAtCommit`, 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 feature to analyze their commits or to evaluate pull requests, detecting accidental breaking changes performed by contributors.

- **Detecting changes in version histories.** By using method `detectChange-AllHistory`, 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). Basically, API creators and clients can use this feature to assess the stability of their libraries.

- **Fetching new commits.** By using this feature, which is provided by method `fetchAndDetectChange`, the tool fetches new commits from a repository. The input includes the project path and git url. API creators may benefit from this functionality to monitor 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, whenever contributors introduce breaking changes, he is notified shortly afterwards to accept the change or revert it.

4.1.2 Filtering Packages

APIDIFF 1.0 does not include any option to filter out specific packages during the analysis. This first version only has the capability to remove elements implemented in packages with the names *example*(*s*) and *test*(*s*), not including options to customize the filter or to remove irrelevant packages. For example, by adopting this rigid strategy, the tool does not allow the removal of elements implemented in *internal* packages, which contain unstable APIs [Businge et al., 2012, 2013, 2015; Mastrangelo et al., 2015; Hora et al., 2016] that may bias the analysis.

In APIDIFF 2.0, for all provided features, it is possible to filter in or filter out packages according to their names, considering specific keywords. Among the possible options, we included the keywords *experimental*, *sample*, *demo*, and *internal*. In this way, we can create a filter to detect changes in internal implementations or to eliminate from the analysis source code that are not intended to be public.

Listing 4.2 shows the `Classifier` enum included in APIDIFF 2.0, which is used to create the filter. We provide six enum values:

- **EXPERIMENTAL.** enum value to analyze elements in packages with the keyword *experimental*;

- **EXAMPLE.** enum value to analyze elements in packages with the keywords *example(s)*, *sample(s)*, and *demo*;

- **TEST.** enum value to analyze elements in packages with the keyword *test(s)*, and types with prefix or suffix *test(s)*;
- **INTERNAL.** enum value to analyze elements in packages with the keyword *internal*;
- **NON_API.** this enum value groups the enum values EXPERIMENTAL, EXAMPLE, TEST and INTERNAL. In other words, it analyzes non-API elements;
- **API.** enum value to analyze elements without the keywords commonly used in source code that is not intended to be public.

```
public enum Classifier {
    API,
    NON_API,
    INTERNAL,
    TEST,
    EXAMPLE,
    EXPERIMENTAL;
}
```

Listing 4.2: Enum `apidiff.enums.Classifier` in APIDIFF

4.1.3 Customizing Metadata

In APIDIFF 1.0, it is not possible to customize the output because the result is not stored as an object (in fact, the output is simply displayed in the console). We identify this characteristic as a limitation due to two major reasons: (i) we cannot select the parameters to build the output (e.g., element name or change description); and (ii) we cannot use APIDIFF 1.0 as part of a large data flow where the output of a module is the input for another module.

By contrast, APIDIFF 2.0 output contains the `Result` type, which includes the list of performed changes. In this way, it is possible to customize or reuse the output. Section 4.2 presents more details about this type. Besides that, users can customize the output of APIDIFF 2.0, reporting the detected changes in a CSV file. Moreover, it is also possible to read the input from CSV files.

4.1.4 Dependency Management

APIDiff 1.0 does not support dependency management. For example, in order to include the Eclipse JDT library, which provides features to create and handle ASTs, the user has to manually download it and import the binary (jar) in the project. We identify this characteristic as a limitation because it is necessary an extra effort to compile the source code and to generate the executable.

By contrast, APIDiff 2.0 uses Maven, a popular package manager for Java, which allows automatic source code compilation. Beside that, we made APIDIFF 2.0 available in Central Maven[2], therefore, we are able to declare it as a dependency in the build system (e.g., Maven or Gradle), as illustrated in Listing 4.3

```
<dependency>
    <groupId>com.github.aserg-ufmg</groupId>
    <artifactId>apidiff</artifactId>
    <version>2.0.0</version>
</dependency>
```

Listing 4.3: APIDIFF as a dependency in the build system

4.2 Architecture

APIDIFF identifies breaking and non-breaking changes between versions of a Java library hosted on `git` repositories in a fully automated way. Figure 4.1 presents the high level architecture of APIDIFF 2.0. This improved version includes two new modules (*processing* and *refactoring*) and the module *analysis* was refactored to be adapted to the new ones. We also refactored the output to include the new metadata and the types `Result` and `Change`.

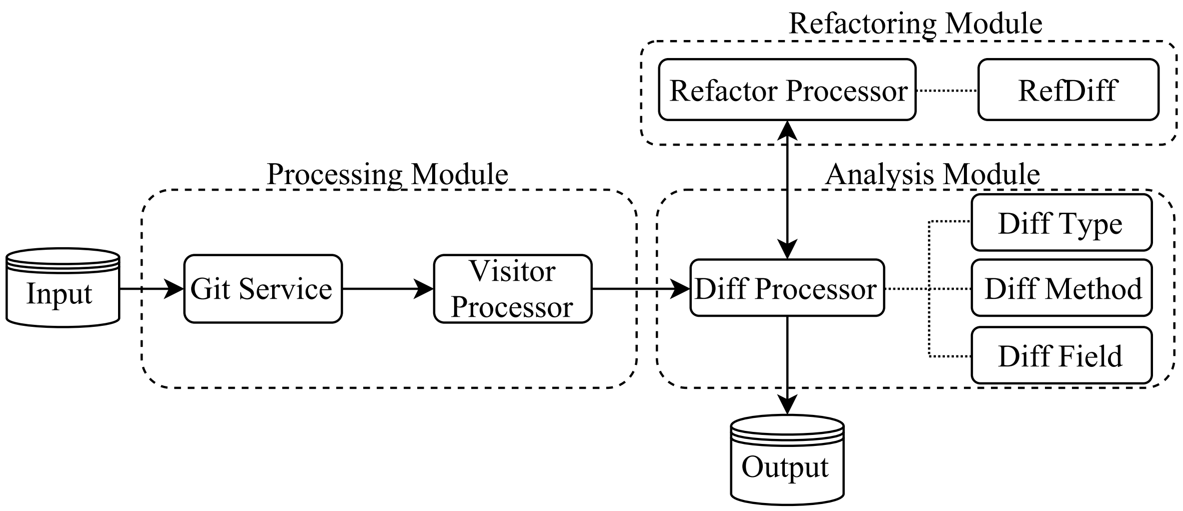


Figure 4.1: APIDiff architecture

[2]https://mvnrepository.com/artifact/com.github.aserg-ufmg/apidiff

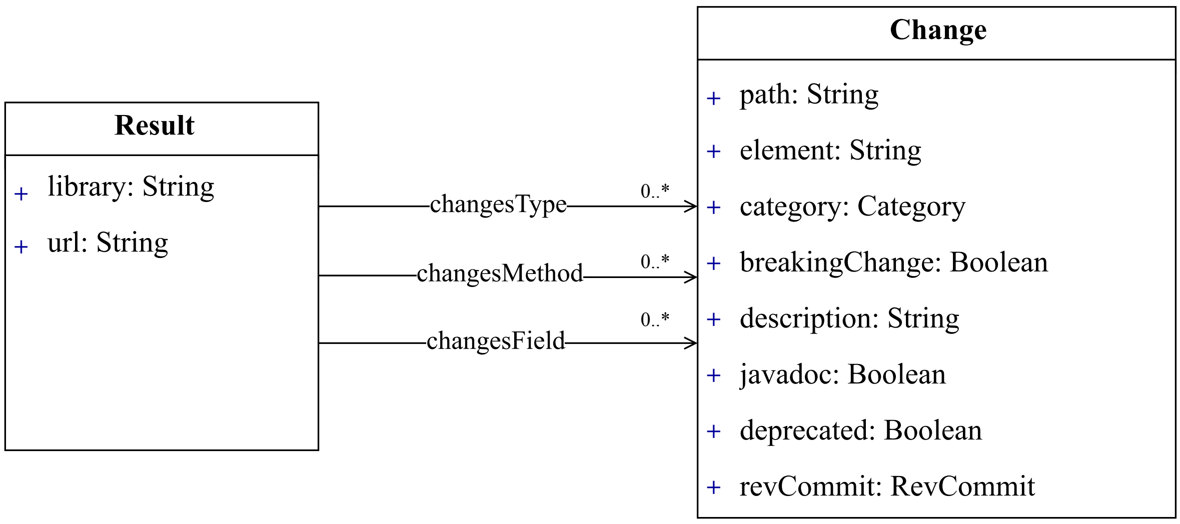


Figure 4.2: Class diagram of the APIDIFF output

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, which are the project states to compare. 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 [Silva and Valente, 2017]. REFDIFF algorithm detects refactoring involving types, methods, and fields, such as move or rename.

Analysis Module. This module detects breaking and non-breaking changes in the library history. It finds changes in types, methods, and fields, which are then merged with results of the *Refactoring Module*

Output. The final output contains the list of performed changes in types, methods, and fields. Figure 4.2 presents a part of the class diagram related to the output. Notice that `Result` includes the list of performed changes, the library name, and the git url. Each change has the following elements:

- **path.** location of the changed element (e.g., the class name);
- **element.** element name;
- **category.** enum of the detected changes (e.g., Rename Method, Add Field);

- **breakingChange.** field to indicate whether the change involves a breaking contract;

- **description.** text with the change description (e.g., "*Method* `M1` *was added in the class* `C1`");

- **javadoc.** field to indicate whether the element contains JavaDoc;

- **deprecated.** field to indicate whether the element contains the annotation `@Deprecated`, or whether the element is located in a deprecated class;

- **revCommit.** information about commit metadata (e.g., author, author's email, commit hash, date).

4.3 Catalog of Changes

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

We faced an important limitation when analyzing the output produced by APIDIFF 1.0. During the study described in Chapter 3, we detected that APIDIFF 1.0 did not support refactoring operations. For example, the operations involving rename of elements were classified as elements removal and addition (i.e., two distinct operations). In this way, the users needed to manually inspect the results to infer some operations. To overcome this limitation, we added the support to refactoring operations in APIDIFF 2.0.

Table 4.1 presents the catalog of BCs detected by APIDIFF 2.0 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.

Changes that do not break clients are classified as *Non-breaking Changes* (NBC), as presented in Table 4.2. Common NBCs in this context involve, for example, type additions and visibility modifiers changes to public/protected (i.e., gain of visibility). Changes on deprecated API elements are also classified as NBCs.

Table 4.1: 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

Table 4.2: 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, DEPRECATED TYPE
Method	ADD METHOD, PULL UP METHOD, GAIN VISIBILITY, REMOVE FINAL MODIFIER, ADD STATIC MODIFIER, DEPRECATED METHOD, EXTRACT METHOD
Field	ADD FIELD, PULL UP FIELD, GAIN VISIBILITY, REMOVE FINAL MODIFIER, DEPRECATED FIELD

4.4 Usage Scenarios

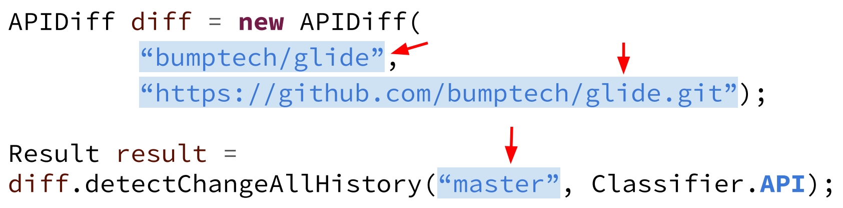
In this section, we present four usage scenarios for APIDIFF. In the first scenario, we investigate API changes considering the complete history of two popular Android libraries: PHILJAY/MPANDROIDCHART[3] (a chart view library) and BUMPTECH/GLIDE[4] (an image loading and caching library). In the second 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[5] (a downloading and caching library for Android). Lastly, we detect changes in a specific commit of

[3]https://github.com/PhilJay/MPAndroidChart
[4]https://github.com/bumptech/glide
[5]https://github.com/square/picasso

mockito/mockito[6] (a framework to implement unit tests).

4.4.1 Most Common Elements with Changes

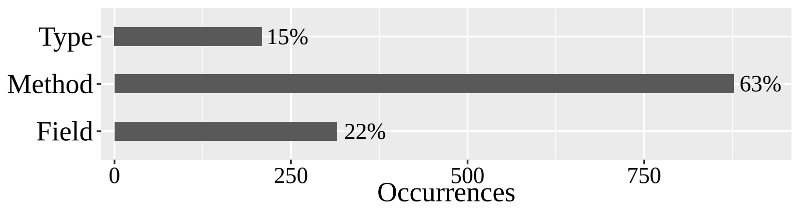
In this first example, we assess the whole commit history of the philjay/mpandroidchart and bumptech/glide. The results refer to the *master* branch and to the elements documented with JavaDoc. Figure 4.3 presents the piece of code necessary to run this functionality in apidiff. The input includes the project path, git url, and branch name.



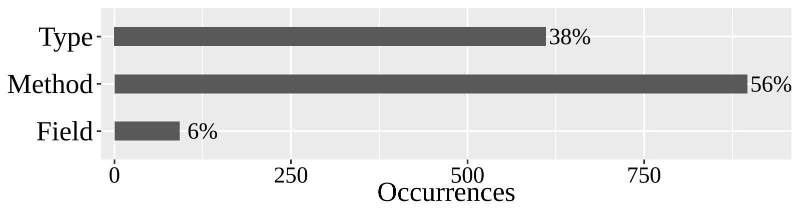
```
APIDiff diff = new APIDiff(
        "bumptech/glide",
        "https://github.com/bumptech/glide.git");

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

Figure 4.3: 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 4.4 and 4.5 present the distribution of BCs and NBCs over types, methods, and fields in these two libraries.



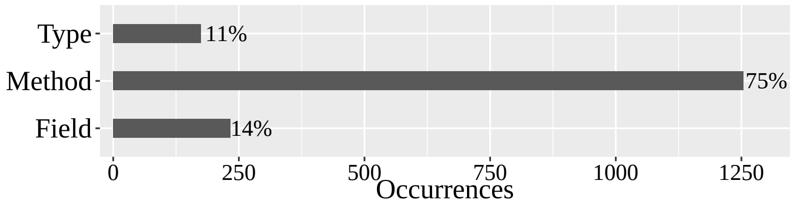
(a) philjay/mpandroidchart



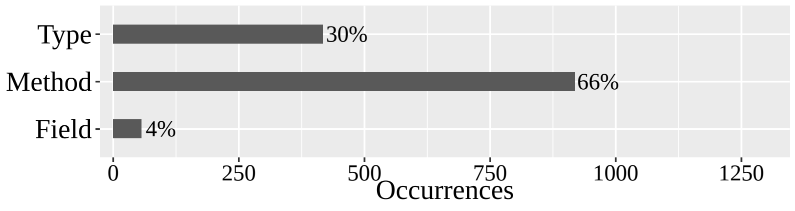
(b) bumptech/glide

Figure 4.4: Breaking changes in types, methods or fields

[6] https://github.com/mockito/mockito



(a) PHILJAY/MPANDROIDCHART



(b) BUMPTECH/GLIDE

Figure 4.5: Non-breaking changes (NBCs)

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.

4.4.2 Most Popular Changes

This example presents the most popular changes performed in PHILJAY/MPANDROID-CHART as detected by APIDIFF (Figure 4.6). 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.

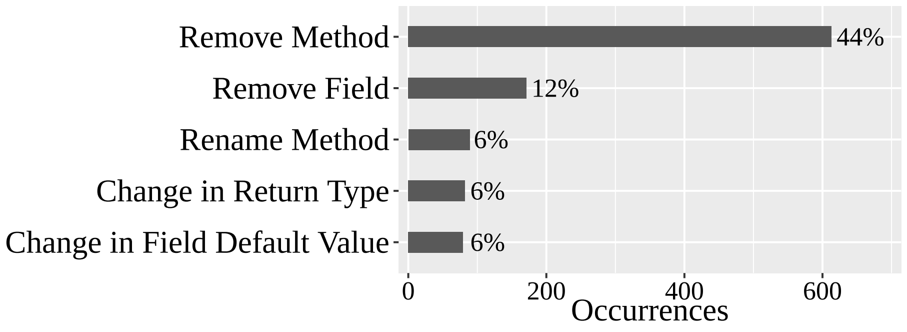


Figure 4.6: Top-5 breaking changes in PHILJAY/MPANDROIDCHART

Furthermore, APIDIFF detected 1,662 NBCs. As presented in Figure 4.7, the most popular NBCs involve adding elements: methods (1,115 occurrences, 67%), fields (186 occurrences, 11%), and types (151 occurrences, 9%).

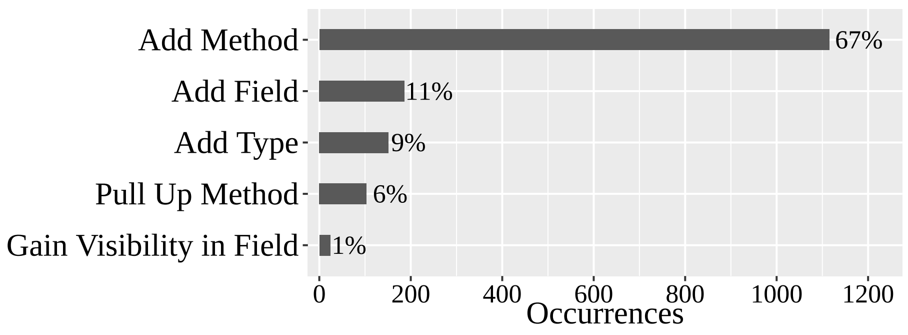


Figure 4.7: Top-5 non-breaking changes in PHILJAY/MPANDROIDCHART

4.4.3 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.

Figure 4.8 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.

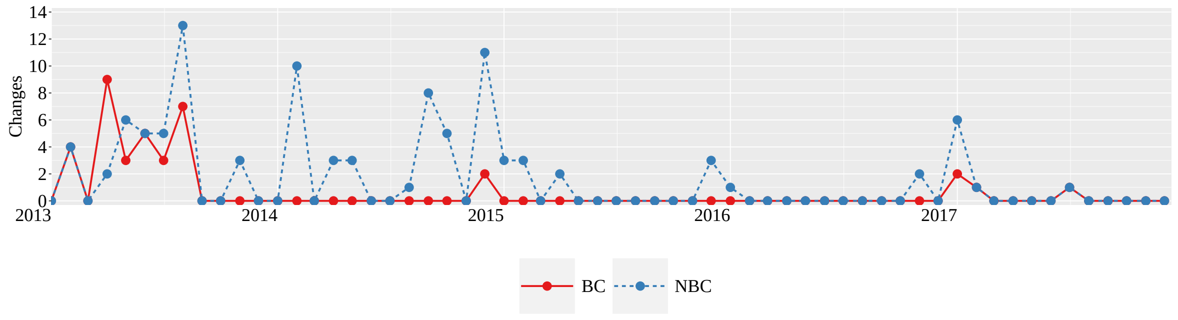
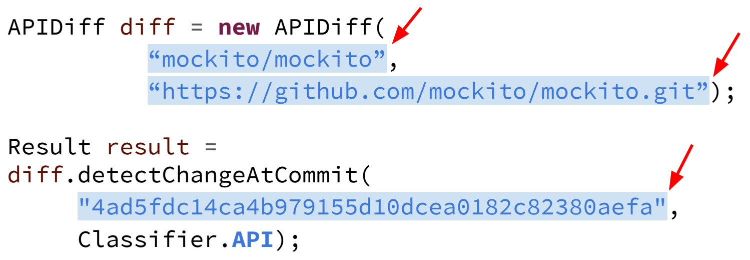


Figure 4.8: BCs and NBCs over time in SQUARE/PICASSO:

4.4.4 Changes at Commit

This last example focuses on a specific commit of the MOCKITO/MOCKITO framework. Figure 4.9 shows the piece of code to detect changes in a given 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 4.10.



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

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

Figure 4.9: Detecting changes in a specific commit

Change: Addition Method

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

Figure 4.10: 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. Particlyarly, in GitHub, library creators can automatically analyze pull requests and check whether there are breaking changes in contributions.

4.5 Final Remarks

In this chapter, we introduced a new release of APIDIFF, a tool to detect breaking and non-breaking changes in Java APIs. We discussed about the limitations in APIDIFF 1.0. next, we presented the new features to mitigate these limitations, and provided an overview about the new architecture and some usage scenarios. Among its most important new features, APIDIFF 2.0 adds support to refactoring operations and integration with `git`. The current catalog of API changes includes the removal and the addition of API elements, 13 well-known refactoring operations, as well as 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 2.0 publicly available on GitHub.[7]

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

Chapter 5

Related Work

Performing changes is a common practice in software evolution, including addition of new features, bugging fixes or code improvement [Moser and Nierstrasz, 1996; Konstantopoulos et al., 2009; Raemaekers et al., 2012]. Software libraries and consequently their APIs also evolve over time [Xavier et al., 2017a]. In this context, the literature presents several studies and approaches to handle with API evolution and stability. In this chapter, we present related work on this theme, specifically on breaking contracts in APIs during evolution. We separate this chapter into four related subjects: studies on API evolution (Section 5.1); studies on breaking changes in APIs (Section 5.2); studies on tools and approaches to handle API evolution and migration (Section 5.3); field studies using the firehouse interview method, which is used in this dissertation to contact the surveyed developers (Section 5.4).

5.1 Studies on API Evolution

Several studies have been conducted to support API evolution and client developers. In a large-scale study, Robbes et al. [2012] assess the impact of API deprecation in a Smalltalk ecosystem. Recently, the authors also evaluated the impact in the context of the Java programming language [Sawant et al., 2016, 2017]. In this study, they found that some API deprecations have a large impact on the ecosystem under analysis and that the quality of deprecation messages should be improved. Jezek et al. [2015] study 109 Java open-source programs and 564 program versions, showing that APIs are commonly unstable. Raemaekers et al. [2012] investigate API stability with the support of four metrics, based on method removal and implementation change. In the context of mobile development, McDonnell et al. [2013] investigate stability and adoption of the Android API. In this study, the authors show that APIs are updated on average 115 times per month, representing a rate faster than clients' update.

Other studies investigate the usage and evolution of internal APIs, i.e., public but unstable and undocumented APIs that should not be used by client applications [Businge et al., 2012, 2013, 2015; Mastrangelo et al., 2015; Hora et al., 2016]. In this context, Businge et al. [2012] study the survival of Eclipse plugins, and classify them in two categories: plugins depending on internal APIs and plugins depending only on official APIs. Despite being a bad practice, the authors reveal that client developers often use internal APIs. In an extended study [Businge et al., 2015], the authors report that 44% of 512 Eclipse plugins depend on internal APIs. In addition, the same authors investigate the reasons why developers do use internal APIs [Businge et al., 2013]. For example, they detect cases where developers do not read documentation (so they are not aware of the risks), but also cases where developers deliberately use internal APIs to benefit from advanced features, not available in the official APIs. Mastrangelo et al. [2015] show that clients commonly use internal APIs provided by JDK. Even though it is unsafe, the authors found several usage cases of the internal API *sun.misc.Unsafe*. Recently, Hora et al. [2016] studied the transition of internal APIs to public ones, aiming to support library developers to deliver better API modularization. The authors also performed a large analysis to assess the usage of internal APIs. In our survey (Chapter 3), several developers mentioned that the breaking changes happened in public but internal or low-level APIs that clients should not rely on. Notice, however, that the related literature points in the opposite direction: client developers tend to use internal APIs.

5.2 Studies on Breaking Changes

In a short paper, Xavier et al. [2017b] report a preliminary study to reveal the reasons of API breaking changes in Java. In this study, they also use APIDIFF (version 1.0) to detect breaking changes. The authors contacted the principal developers of 49 libraries, asking them about the reasons of all breaking changes detected by APIDIFF in previous releases of these libraries. By contrast, in the study described in Chapter 3 contacted the precise developers responsible by a breaking change, right after it was introduced in the code; and we asked them to reveal the reasons for this specific breaking change. Furthermore, to identify breaking changes, we monitored all commits of a list of 400 Java libraries, during 116 days. As a consequence of the distinct methodologies, Xavier et al. [2017b] received valid answers of only seven developers (while in this dissertation we received 56 answers). From these seven answers, they extracted five reasons for breaking changes: API Simplification, Refactoring, Bug Fix, Dependency Changes, and Project Policy. The first four are also detected in this dissertation. However, the major reason for breaking changes reported in the present study (New Feature) was not detected in the preliminary one.

In another related study [Xavier et al., 2017a], the authors investigate breaking changes in 317 real-world Java libraries, including 9K releases and 260K client applications. They show that 15% of the API changes break compatibility with previous versions and that the frequency of breaking changes increases over time. Using data from the BOA ultra-large dataset [Dyer et al., 2013], they report that less than 3% of the breaking changes impact clients. To reach this result, the authors considered all breaking changes detected by APIDIFF. However, in the present dissertation, we found that only 39% of the BCCs are viewed by developers as having a major potential to break clients.

Kula et al. [2018] focus in the impact of API refactoring on system clients, analyzing versions of eight popular libraries (GUAVA, HTTPCLIENT, JAVASSIST, JDOM, JODA-TIME, LOG4J, SLF4J, and XERCES). In this empirical study, Japi-cmp[1] library was used to compute the difference between two library version, and REF-FINDER was used to detect refactoring actions [Prete et al., 2010]. The study includes approximately 9,700 classes with breaking changes and 2,900 refactoring operations. The authors classified as internal APIs, implementations that have no use by client systems, and they classified as external APIs, the classes with least one client. Among their major results, the authors show that 75% of refactoring operations break client applications, and that breaking changes are more likely on internal APIs. In our study, the most common reason to break contracts is refactoring too (47%) and most breaking changes were classified as internal by the surveyed developers (61%). However, Kula et al. [2018] did not contact developers to understand the reasons behind the breaking changes and to confirm whether they am indeed true positives. Beside that, some APIs with clients could be internal implementation, since the literature shows that internal APIs are often used by clients [Businge et al., 2015; Mastrangelo et al., 2015; Hora et al., 2016].

Dig and Johnson [2005] studied API changes in five frameworks and libraries (Eclipse, Mortgage, Struts, Log4J, and JHotDraw). They report that more than 80% of the breaking changes in these systems were due to refactorings. By contrast, using a large dataset of 400 popular Java libraries and frameworks, we also found that BCs are usually related to refactorings, but at a lower rate (47%). Moreover, we listed two other important motivations for breaking changes: to support the implementation of

[1]`https://github.com/siom79/japicmp`

new features and to simplify and reduce the number of API elements.

Bogart et al. [2016] conducted a study to understand how developers plan, negotiate, and manage breaking changes in three software ecosystems: Eclipse, R/CRAN, and Node.js/npm. After interviewing key developers in each ecosystem, they report that a core value of the Eclipse community is long-term stability; therefore, breaking changes are rare in Eclipse. R/CRAN values snapshot consistency, i.e., the newest version of every package should be always compatible with the newest version of every other package in the ecosystem. Once snapshot consistency is preserved, breaking changes are not a major concern in R/CRAN. Finally, breaking changes in Node.js/npm are viewed as necessary for progress and innovation. In the interviews, the participants also mentioned three general reasons for breaking changes: technical debt (i.e., to improve maintainability), to fix bugs, and to improve performance. The first two motivations appear in our study, but we did not detect breaking changes motivated by performance improvements. However, these answers should be interpreted as general reasons for breaking changes, as perceived by the interviewed developers. By contrast, in this dissertation the goal was to reveal reasons for specific breaking changes, as declared by developers right after introducing them in the source code of popular Java libraries and frameworks.

5.3 API Changes Tools

Several tools and approaches have been proposed to deal with the impact of software evolution. For example, Chow and Notkin [1996] present an approach where library developers themselves annotate the changed methods with replacement rules. Nguyen et al. [2010] use graph-based techniques to help developers migrate from one library version to another. Kim and Notkin [2009] introduce *Logical Structural Diff* (*LSdiff*), which computes differences between two versions of a system. Hora and Valente [2015] present *apiwave*, which focuses on library migration. Wu et al. [2010b] present *AURA* (AUtomatic change Rule Assistant), which generates automatic change rules, helping developers migrating to new releases.

Xing and Stroulia [2007a] present *Diff-CatchUp*, which recommends features to replace an obsolete API implementation. The approach is based in the UMLDiff algorithm to recover the changes [Xing and Stroulia, 2007b]. Additionally, the same authors present *UMLDiff*, which detects structural changes between two version of a software system [Xing and Stroulia, 2005] 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 internal packages. Other studies focus on extracting API evolution rules from source code. For example, Schäfer et al. [2008] mine library change rules from client systems, while Dagenais and Robillard [2008] present *SemDiff*, a tool that suggests replacements for client systems based on changes in their own framework code. Also in this context, Meng et al. [2012] propose a history-based matching approach to support API evolution.

Silva and Valente [2017] 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. Besides that, REFDIFF does not warn when changes may break API contracts. 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 [2005] present *CatchUp*, an approach that captures and replays performed refactorings.

5.4 Studies Using Firehouse Interviews

A *firehouse interview* is one that is conducted right after the event of interest has happened [Rogers, 2003]. The term relates to the difficulty of performing qualitative studies about unpredictable events, like a fire. In such cases, researchers should act like firemen after an alarm; they should rush to the firehouse, instead of waiting the event to be concluded to start their research. In our study, the events of interest are API breaking changes; and firehouse interviews allowed us to collect the reasons for theses changes right after they were committed to GitHub repositories.

In software engineering research, firehouse interviews were previously used to investigate bugs just fixed by developers [Murphy-Hill et al., 2013; Murphy-Hill et al., 2015], but using face-to-face interviews with eight Microsoft engineers. Silva et al. [2016] were the first to use firehouse interviews to contact GitHub developers by email. Their goal was to reveal the reasons behind refactorings applied by these developers; in this case, they also used a tool to automatically detect refactorings performed in recent commits. They sent e-mails to 465 developers and received 195 answers (42% of response ratio). Mazinanian et al. [2017] used a similar approach, but to understand the reasons why developers introduce lambda expressions in Java. They sent emails to 351 developers and received 97 answers (28% of response ratio). In this dissertation, we contacted 102 developers and received 56 answers (55% of response ratio).

5.5 Final Remarks

In this chapter, we present work related with the major themes of this dissertation, involving break contracts in APIs. Specifically, we discussed subjects witch motivated our study about the reasons behind a breaking changes, and studies which motivated the development of the new version of the APIDIFF tool. In Section 5.4, we presented studies based on firehouse interview method, the technique used in this dissertation to contact the survey developers. To the best of our knowledge, this is the first large-scale field study that reveals the reasons of concrete breaking changes introduced by practitioners in the source code of popular Java APIs. Chapter 6 presents the conclusions and contributions of our study. Additionally, we propose future work on API evolution.

Chapter 6

Conclusion

This chapter presents our conclusions for this dissertation. First, Section 6.1 presents an overview about the performed empirical study and the APIDIFF 2.0 tool, highlighting the main contributions. Section 6.2 shows the limitations of our work. Finally, we suggest future work in Section 6.3.

6.1 Overview and Contributions

In this dissertation, we presented an empirical study on the reasons that drive API breaking changes. In this study, we employ the *firehouse* technique for interviews and we used the APIDIFF tool to monitor 400 popular libraries and frameworks hosted on GitHub. Whenever we detected a breaking change candidate in these repositories, we sent an email to the developers asking the reasons to break the compatibility, the possible impact of the change, and the strategies to document the performed changes. Specifically, we investigated four main questions: (i) how often do changes impact clients, (ii) why do developers break APIs, (iii) why do not developers deprecate broken APIs, and (iv) how do developers document breaking changes. In addition, based on our experience using APIDIFF, we provided a new version of the tool to mitigate a list of identified limitations. We summarize the results and the major contributions of this master dissertation in the following subsections.

6.1.1 Breaking Changes and Library Developers

We conducted a large-scale empirical study, including 400 Java libraries and frameworks, 282 possible breaking changes, and answers from 56 developers to understand *why* and *how* API changes are performed. Based on the answers provided by developers, we found that BCs are mainly motivated by the implementation of new features (32%), to simplify the number and the complexity of API elements (29%), and to improve maintainability (24%). Based on the received answers, we provided an extensive list of empirically-justified implications to language designers, tool builders, researchers, and practitioners. In this context, the main findings of this study are as follow:

- **Internal APIs.** Only 59 breaking changes candidates (39%) detected by APIDIFF are confirmed as breaking contracts by the survey developers. The most unconfirmed breaking changes are internal or low-level implementations. However, these elements are accessible to external systems and some studies show that clients commonly use internal APIs [Businge et al., 2015; Mastrangelo et al., 2015; Hora et al., 2016]. Therefore, this rate of internal APIs with breaking changes highlights the importance of the new module system proposed by Java to better encapsulate structures.

- **Refactoring.** Among the top-5 most common confirmed breaking change, three are refactorings: MOVE METHOD (11 occurrences), RENAME METHOD (8 occurrences), MOVE CLASS (8 occurrences), representing 47% of our results.

- **API elements.** After grouping the results by type, method, and field, we found that most API element affected by breaking changes are methods, representing 59% of our results.

- **Breaking Changes and Migration.** We also questioned the migration effort on clients facing the breaking changes; according to the surveyed developers, this effort is minor.

- **Documenting Breaking Changes.** 18 developers answered the questions about documenting the breaking changes; 14 developers stated they intend n to document the performed modifications; usually, using *release notes* or *changelogs*.

6.1.2 APIDiff 2.0

Based on our experience during the performed study, we identified some limitations in APIDIFF 1.0. To mitigate these problems, we upgraded the tool, incorporating important new features:

- **Git Operations.** We implemented support to `git` operations. Among the new operations, there are options to clone projects and to analyze the changes performed in a branch or in a specific commit. The support to `git` also allows users to access commit metadata (e.g., name author, date, commit hash).

- **Catalog of Changes.** We included refactoring support to the detected breaking changes (for example, to detect rename and move operations) based on the approach provided by REFDIFF.

- **Management Dependencies.** APIDIFF 2.0 includes support to Maven, a popular package manager, which allows automatic source code compilation. Besides that, the source code of the tool is publicly available on GitHub.[1]

- **Classifier for Specific Packages.** We implemented a classifier to filter specific packages. For example, it is possible analyze changes performed at internal packages or to filter out possible non-APIs from the analysis.

- **Customizing Results.** Unlike the previous version, which only shows the result on the console, APIDIFF 2.0 is more flexible: the results are stored as an object, allowing the output customization.

6.2 Limitations

Our study has the following limitations:

- We do not cover all possible syntactic breaking and non-breaking changes in the Java language. For example, the APIDIFF catalog does not include changes performed in annotation classes.

- APIDIFF users can only use the package names from the enum classifier. In other words, it is not possible to create rules to filter out specific packages during the analysis. We reported the supported package names in Section 4.1.2. Moreover, some libraries use annotations to warn about specific API groups (e.g., unstable APIs or internal APIs). However, APIDIFF has no support to these strategies.

- APIDIFF output may include false positives. For example, the detected breaking changes can be in low-level or internal APIs. Moreover, the refactoring operations at API level can be false positives reported by REFDIFF.

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

6.3 Future Work

In this dissertation, we performed an empirical study, contacting API creators and asking them about the reasons behind API breaking changes. We suggest as extension of this work:

Behavioral Breaking Changes. This dissertation focuses in syntactical breaking changes; in other words, changes that generate compilation errors on client systems. However, there are changes that do not affect the compilation on client systems, but that can change their behavior, which is an under unexplored research topic [Mostafa et al., 2017]. In this way, as future work, we suggest an extension of this dissertation by focusing on behavioral breaking changes, since the detection of this change type is not trivial and the impact on API clients may be high.

Ecosystems and Programming Languages. Our results are derived from popular Java libraries and frameworks hosted on GitHub and should not be generalized to other scenarios. In this context, we recommend further studies considering other software ecosystems and programming languages. Particularly, we suggest studies focusing on dynamic typed languages. For example, studies using JavaScript systems, since there is an increase in the popularity of this language[2]; additionally, JavaScript's flexibility allows changes with high impact on client systems.

Research Methodologies. In this dissertation, we sent emails to developers to better understand the reasons behind the performed breaking changes detected by APIDIFF. However, there are other research methodologies to perform this study, collecting more complete answers, for example semi-structured interviews with API creators and clients.

API Clients. We contacted the API creators to understand the reasons to break library compatibility. In this context, we suggest a quantitative and qualitative assessment of the breaking change impact on the other protagonists of this story: the developers who depend on APIs affected by breaking changes.

Additionally, we provided a new version of APIDIFF, with the aim of mitigating some limitations identified in its first release. In this context, we suggest work related to the development of new features on APIDIFF, and on current approaches to deal with API evolution:

[2]`https://insights.stackoverflow.com/survey/2018/#most-popular-technologies`

Precision of apidiff and New Features. We suggest to evaluate the precision of apidiff 2.0 with real-world Java libraries. Besides that, we suggest the addition of a new feature to apidiff to analyze changes at *release level*, since changes at commits are not necessarily modifications incorporated to new releases.

Visualization of API changes. apidiff plots textual data. A graphical visualization of API changes can be useful to both API creators and clients to observe the library stability over time. The new tool can be inspired by popular techniques in the literature to represent massive datasets, for example, visualization in city format [Wettel and Lanza, 2008]. The tool can also include features to plot the charts presented in this dissertation (as in Figure 4.8) to view changes over time.

Multi-platform Tool. apidiff only supports a single language (Java). A multi-language tool would be very useful to detect and compare breaking and non-breaking changes in others popular languages.

Bibliography

Abdeen, H., Ducasse, S., Sahraoui, H., and Alloui, I. (2009). Automatic package coupling and cycle minimization. In *16th Working Conference on Reverse Engineering (WCRE)*, pages 103–112.

Anquetil, N. and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255.

Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). 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)*, pages 109–120.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Boulanger, J.-S. and Robillard, M. P. (2006). Managing concern interfaces. In *22nd IEEE International Conference on Software Maintenance (ICSME)*, pages 14–23.

Brito, A., Hora, A., and Valente, M. T. (2016a). JAVALI: Uma ferramenta para analise de popularidade de APIs Java. 7th Brazilian Conference on Software: Theory and Practice (CBSoft, Tools Track), pages 1–8.

Brito, A., Hora, A., and Valente, M. T. (2016b). Um estudo em larga escala sobre o uso de APIs internas. 4th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), pages 1–8.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018a). APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*, pages 507–511.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018b). Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016c). Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In *23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 360–369.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2018c). On the use of replacement messages in api deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321.

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

Businge, J., Serebrenik, A., and van den Brand, M. (2013). Analyzing the Eclipse API usage: Putting the developer in the loop. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 37–46.

Businge, J., Serebrenik, A., and van den Brand, M. G. J. (2015). Eclipse API usage: the good and the bad. *Software Quality Journal*, 23(1):107–141.

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

Cliff, N. (2014). *Ordinal methods for behavioral data analysis*. Psychology Press.

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

Dagenais, B. and Robillard, M. P. (2008). Recommending adaptive changes for framework evolution. In *30th International Conference on Software Engineering (ICSE)*, pages 481–490.

Dig, D. and Johnson, R. (2005). How do APIs evolve? a story of refactoring. In *22nd International Conference on Software Maintenance (ICSM)*, pages 83–107.

Dig, D. and Johnson, R. (2006). How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution (JSME)*, 18(2):83–107.

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

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

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

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

Jezek, K., Dietrich, J., and Brada, P. (2015). How Java APIs break - an empirical study. *Information and Software Technology*, 65(C):129–146.

Kim, M. and Notkin, D. (2009). Discovering and representing systematic code changes. In *31st International Conference on Software Engineering (ICSE)*, pages 309–319.

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

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

Kula, R. G., Ouni, A., German, D. M., and Inoue, K. (2018). An empirical study on the impact of refactoring activities on evolving client-used APIs. *Information and Software Technology*, 93(C):186–199.

Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., and Nystrom, N. (2015). 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), pages 695–710.

Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D. (2017). Understanding the use of lambda expressions in Java. In *32nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 85:1–85:31.

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

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

Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208.

Montandon, J. (2013). Documenting application programming interfaces with source code examples. Master's thesis, UFMG.

Moser, S. and Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9):45–51.

Mostafa, S., Rodriguez, R., and Wang, X. (2017). Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *26th International Symposium on Software Testing and Analysis (ISSTA)*, pages 215–225.

Murphy-Hill, E., Zimmermann, T., Bird, C., and Nagappan, N. (2015). The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81.

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

Nguyen, H. A., Nguyen, T. T., Jr., G. W., Nguyen, A. T., Kim, M., and Nguyen, T. N. (2010). A graph-based approach to API usage adaptation. In *25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 302–321.

Praditwong, K., Harman, M., and Yao, X. (2011). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282.

Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10.

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

Reddy, M. (2011). *API Design for C++*. Morgan Kaufmann Publishers.

Robbes, R., Lungu, M., and Röthlisberger, D. (2012). How do developers react to API deprecation? the case of a Smalltalk ecosystem. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 56:1–56:11.

Rogers, E. M. (2003). *Diffusion of Innovations*. Free Press, 5th edition.

Sawant, A. A., Robbes, R., and Bacchelli, A. (2016). On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In *32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–410.

Sawant, A. A., Robbes, R., and Bacchelli, A. (2017). On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK. *Empirical Software Engineering*, pages 1–40.

Schäfer, T., Jonas, J., and Mezini, M. (2008). Mining framework usage changes from instantiation code. In *30th International Conference on Software Engineering (ICSE)*, pages 471–480.

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

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

Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342.

Wettel, R. and Lanza, M. (2008). Codecity: 3D visualization of large-scale software. In *30th International Conference on Software Engineering (ICSE)*, pages 921–922.

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

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

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2016). Um estudo em larga escala sobre estabilidade de APIs. 4th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), pages 1–8.

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017a). Historical and impact analysis of API breaking changes: A large scale study. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147.

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

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

Xing, Z. and Stroulia, E. (2007a). API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836.

Xing, Z. and Stroulia, E. (2007b). Differencing logical UML models. *Automated Software Engineering*, 14(2):215–259.

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