HISTORICAL AND IMPACT ANALYSIS OF API BREAKING CHANGES

JOSÉ LAERTE PIRES XAVIER JÚNIOR

HISTORICAL AND IMPACT ANALYSIS OF API BREAKING CHANGES

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 Tulio de Oliveira Valente
Coorientador: André Cavalcante Hora

Belo Horizonte
Maio de 2017

JOSÉ LAERTE PIRES XAVIER JÚNIOR

HISTORICAL AND IMPACT ANALYSIS OF API BREAKING CHANGES

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 Tulio de Oliveira Valente
Co-Advisor: André Cavalcante Hora

Belo Horizonte

May 2017

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Xavier Júnior, José Laerte Pires.

X3h Historical and impact analysis of API breaking changes / José Laerte Pires Xavier Júnior – Belo Horizonte, 2017.
xxiv, 66 f.: il.; 29 cm.

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

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

1. Computação – Teses. 2. Redes de computadores. 3.Software – manutenção. 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

Historical and impact analysis of API breaking changes

JOSE LAERTE PIRES XAVIER JUNIOR

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 - Coorientador
Faculdade de Computação - UFMS

[illegible]

PROFA. KÉCIA ALINE MARQUES FERREIRA
Departamento de Computação - CEFET-MG

[illegible]

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 02 de maio de 2017.

To all those who inspired me to go further.

Acknowledgments

Na vida, nenhuma conquista vale a pena se não for dividida com aqueles que deram algo de si para sua realização. Por isso, agradeço a cada um dos familiares, amigos e colegas que generosamente contribuíram nessa jornada. Agradeço, em especial:

A Deus, que planeja cuidadosamente cada um dos meus passos e me ensina, pelo exemplo de sua mãe Maria, a confiar cada vez mais nos Seus propósitos.

Aos meus pais e ao meu irmão, que foram os suportes necessários nos momentos de indecisão, as palavras de incentivo nas horas das batalhas e as vozes acolhedoras nos momentos de saudade.

Aos meus avós e demais familiares, que sempre me inspiraram a ir além.

Aos amigos Derek e Jordana, que nunca permitiram que eu me perdesse de mim mesmo, recordando sempre o valor da minha essência. Pelas boas risadas e infinitas ligações nos momentos de descanso.

Aos amigos do EJC São Judas Tadeu e do Projeto Jovem Fanuel, que contribuíram para que eu nunca deixasse de sentir o amor e o cuidado de Deus.

Aos colegas do ASERG, pelas boas risadas e momentos de crescimento intelectual.

Ao meu orientador Marco Túlio, pela acolhida, atenção e paciência em conduzir este trabalho. Pelas revisões, sugestões e suporte durante todo esse ano.

Ao meu co-orientador André Hora, com quem sempre tive a abertura para pedir ajuda, sugestões e conselhos. Pelo apoio também indispensável em todos os momentos.

Aos membros da banca, Kecia Ferreira e Mariza Bigonha, pela disponibilidade em participar deste trabalho.

Ao DCC/UFMG e a CAPES, pelo suporte financeiro, logístico e profissional.

"It is our choices, Harry, that show what we truly are, far more than our abilities."

(J.K. Rowling)

Resumo

Mudança é uma constante em desenvolvimento de software. Assim como em qualquer sistema, bibliotecas também estão sujeitas a diversas mudanças, compelindo seus clientes a atualizarem-se e, então, aproveitarem as melhorias providas por suas APIs. Entretanto, algumas dessas mudanças não preservam compatibilidade, quebrando contratos previamente estabelecidos. Este tipo de mudança é referenciado como *breaking change*. Assim, clientes podem enfrentar erros de compilação ou mudanças comportamentais ao atualizarem bibliotecas que possuem *breaking changes*. Diversas soluções têm sido propostas a fim de mitigar o impacto dessas mudanças em aplicações clientes; poucos estudos focam nas motivações reais que as ocasionam. Dessa forma, pouco se sabe a respeito do tamanho real desse problema, dos seus possíveis efeitos, e das razões específicas que motivam tais mudanças. Nesta dissertação, propõe-se a ferramenta `APIDiff`, cujo objetivo é comparar duas versões de uma biblioteca Java e identificar *breaking* e *non-breaking changes* entre elas. Essa ferramente é utilizada para realização de dois estudos empíricos sobre *breaking changes* em APIs. No primeiro estudo, objetiva-se analisar, quantitativamente, (i) a frequência dessas mudanças, (ii) sua evolução ao longo do tempo, (iii) o impacto nos clientes, e (iv) as características de bibliotecas com alta frequência de mudanças desse tipo. No segundo estudo, objetiva-se entender, qualitativamente, (v) as razões que levam desenvolvedores a introduzirem essas mudanças, e (vi) a consciência dos mesmos sobre os seus efeitos. Foram analisadas 317 bibliotecas Java, 9K versões e 260K clientes. Assim, observou-se que (i) 14.78% das mudanças em API quebram compatibilidade, (ii) tal frequência cresce ao longo dos seus ciclos de vida, (iii) 2.54% dos clientes são afetados, (iv) sistemas com alta frequência de *breaking changes* são maiores, mais populares e mais ativos, (v) desenvolvedores de APIs normalmente introduzem tais mudanças com objetivos específicos, e (vi) a maioria deles são conscientes dos seus efeitos. Por fim, são providas sugestões de ferramentas e estudos para auxiliar desenvolvedores de bibliotecas e seus clientes.

Palavras-chave: Compatibilidade, Evolução de software, Manutenção de software.

Abstract

Change is a routine in software development. As any other system, libraries also evolve over time. As a consequence, clients are compelled to update and, thus, benefit from the available API improvements. However, some of these changes are *backward incompatible*, breaking contracts previously established with client applications. As a result, they may face compilation errors and behavioral changes when updating to library versions enclosing *breaking changes* in their API elements (types, fields, and methods). Several solutions have been proposed to mitigate the impact on clients; few other studies focus on the real motivations driving these changes. However, we are still unaware about the real size of this problem, the impact of these changes on clients, and the specific reasons driving API developers to break such contracts. In this dissertation, we propose an `APIDiff` tool, whose purpose is to compare two versions of a Java library and identify *breaking* and *non-breaking changes* between them. Additionally, we use this tool to perform two empirical studies on API breaking changes. In the first study, our goal is to quantitatively assess (i) the frequency of breaking changes, (ii) their behavior over time, (iii) the impact on clients, and (iv) the characteristics of libraries with high frequency of breaking changes. In the second one, we aim to qualitatively understand (v) the specific reasons why developers introduce breaking changes, and (vi) their awareness about the risks associated to these changes. Our large-scale analysis on 317 real-world Java libraries, 9K releases, and 260K client applications shows that (i) 14.78% of the API changes break compatibility, (ii) their frequency increases over time, (iii) 2.54% of API clients are impacted, (iv) systems with higher frequency of breaking changes are larger, more popular, and more active, (v) library developers usually break contracts with specific motivations, and (vi) most developers are aware of the risks of breaking changes and, in some cases, adopt strategies to mitigate them. Therefore, we provide insights for the development of tools and studies to support library and client developers in their maintenance activities.

Palavras-chave: API breaking changes, Software evolution, Software maintenance.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software libraries promote the reuse of common functionalities by providing *Application Programming Interfaces (APIs)* to client applications [Reddy, 2011; Robillard et al., 2013]. In this context, APIs have become extremely popular (*e.g.,* the `java.utils.ArrayList` library has more than 140K clients[1] [Dyer et al., 2013, 2015]), with a huge number of systems developed on the top of them [Tourwé and Mens, 2003]. Such phenomenon may be explained by the benefits that APIs provide to their clients [Konstantopoulos et al., 2009; Moser and Nierstrasz, 1996], such as:

- Increase of software quality by providing well-adopted and tested components;
- Reduction of development time and budget by avoiding the effort of re-implementing source code already available;
- Increase of software reliability by providing constant updates to improve non-functional requirements, such as safety and performance.

As any other system, libraries are usually changing. While evolving, they are subject of a variety of modifications, such as addition, removal, or modification of their API elements (types, fields, and methods) [Raemaekers et al., 2012]. In theory, these changes should be *backward compatible*, preserving contracts with client applications. However, previous studies indicate that this is *not* a common practice [Wu et al., 2010; Robbes et al., 2012; Hora et al., 2015; Brito et al., 2016]. In this context, API changes are classified into *breaking changes* and *non-breaking changes* [Dig and Johnson, 2006]. *Breaking changes* are those modifications that break backward compatibility with client applications, possibly causing them to face compilation errors or behavioral changes

[1]According to JAVALI, a tool to measure popularity of Java libraries in Boa infrastructure, available at: `http://java.labsoft.dcc.ufmg.br/javali`

after updating. On the other hand, *non-breaking changes* preserve compatibility and do not cause negative effects when clients migrate between versions.

Listings 1.1 and 1.2 illustrate a real case of API *breaking changes* observed in the version 5.2.0 of the HIBERNATE/HIBERNATE-ORM library (all modifications are highlighted in red). First, we observe that the type `Query<R>` changed its supertype from `org.hibernate.BasicQueryContract` to `CommonQueryContract` and, thus, updated the implemented contracts of their methods. Additionally, a package refactoring was also performed in this library version. As a consequence, the methods of the type `Query<R>` had their return types updated to the new package signatures. For example, the method `setBoolean` modified its return type from `org.hibernate.query.Query<R>` to `Query<R>`, breaking a contract previously established with its clients. Therefore, a *major* issue was open on the library's bug tracker, and the library released a *fix version* (5.2.1), restoring compatibility with legacy code, as follows (see more details in Figure 1.1):

> *"The clients of the org.hibernate.Query can no longer use the API in "DSL Style" as the builder methods no longer return the org.hibernate.Query contract, but force updating to the new contract. We can restore the previous backwards compatibility, which was lost during the refactoring work of version 5.2.0."*

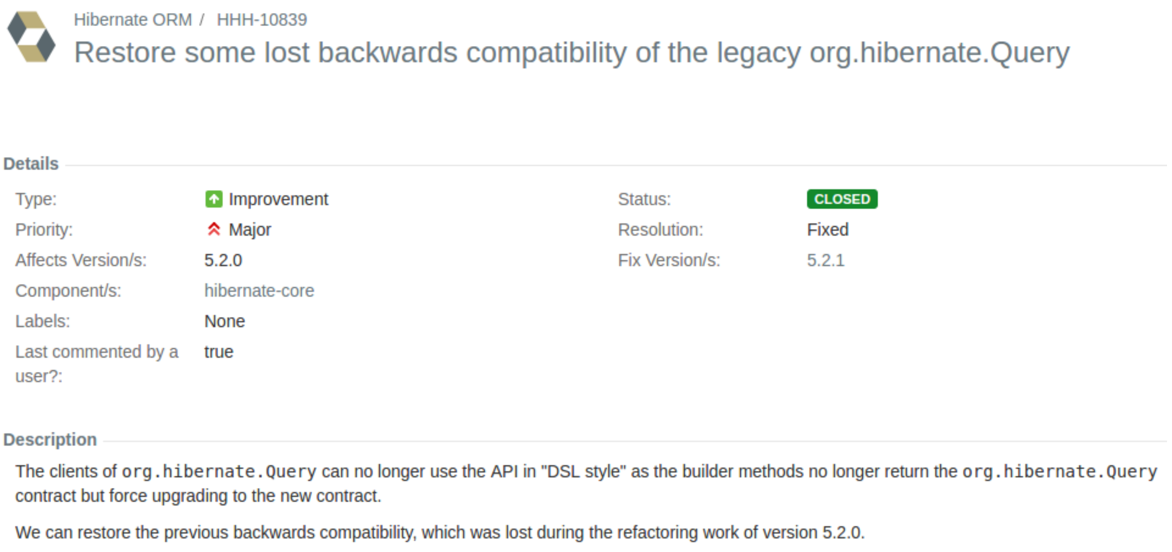
```java
public interface Query<R> extends TypedQuery<R>, org.hibernate.BasicQueryContract
     {
    default org.hibernate.query.Query<R> setBoolean(int position, boolean val) {
        setParameter( position, val, determineProperBooleanType( position, val,
            BooleanType.INSTANCE ) );
        return (org.hibernate.query.Query) this;
    }
}
```

Listing 1.1. Fragment of type `org.hibernate.Query` on its version 5.0

```java
public interface Query<R> extends TypedQuery<R>, CommonQueryContract {
    default Query<R> setBoolean(int position, boolean val) {
        setParameter( position, val, determineProperBooleanType( position, val,
            BooleanType.INSTANCE ) );
        return this;
    }
}
```

Listing 1.2. Fragment of type `org.hibernate.Query` on its version 5.2.0

There are several solutions proposed in the literature to mitigate the impact of API *breaking changes* on client applications (*e.g.,* [Henkel and Diwan, 2005; Kingsum and Notkin, 1996; Meng et al., 2012; Hora et al., 2014; Hora and Valente, 2015]). For



Hibernate ORM / HHH-10839

Restore some lost backwards compatibility of the legacy org.hibernate.Query

Details

Type:	Improvement	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	5.2.0	Fix Version/s:	5.2.1
Component/s:	hibernate-core		
Labels:	None		
Last commented by a user?:	true		

Description

The clients of `org.hibernate.Query` can no longer use the API in "DSL style" as the builder methods no longer return the `org.hibernate.Query` contract but force upgrading to the new contract.

We can restore the previous backwards compatibility, which was lost during the refactoring work of version 5.2.0.

Figure 1.1. Issue opened on HIBERNATE/HIBERNATE-ORM library to restore compatibility.

example, by mining version history, some studies suggest how client applications should be updated due to broken API elements (*e.g.*, a public method removed from an old library version). However, even though there are solutions to alleviate the impact of library evolution, we are still unaware about ***the real number of clients affected by API breaking changes, and unsure whether backward-incompatibility tends to get better (or worse) over time***. Additionally, few studies investigate the real motivations driving such changes. For instance, Bogart et al. [2016] performed a general-purpose case study with 28 developers to study how they plan, manage, and negotiate breaking changes. However, they report developers general views and conceptions on such changes, leaving a gap on ***the specific reasons that motivate breaking changes in the wild***. Therefore, there are still open questions, such as:

- To what extent are clients affected by backward-incompatibility?
- Is backward-incompatibility a problem only faced by newer (and possibly "unstable") libraries or older (and "stable") ones should also take special care?
- Why API developers, who are supposed to be careful about compatibility, break API contracts?
- When developers break API contracts, are they aware of the risks to client applications?

In this dissertation, we aim to investigate these questions, advancing the knowledge on API breaking changes. We provide insights for the development of tools as well as studies to support library and client developers in their maintenance activities. For that, we propose an API change catalog and implement a tool to compare versions of Java libraries, assessing the cataloged changes. Additionally, we use this tool to perform two empirical studies in the context of 317 real-world Java libraries, 9K releases, and 260K client applications. In the next section, we detail these studies, as well as their main results and contributions.

1.1 Proposed Tool and Studies

In order to support our investigations on API breaking changes, we propose an API change catalog based on the previous work of Dig and Johnson [2006]. Besides, we use this proposed catalog to implement an APIDIFF tool, whose purpose is to identify both *breaking* and *non-breaking changes* between two versions of a Java library (Chapter 3). We use this tool to perform two empirical studies with the purpose of assessing (i) the frequency and the impact of API breaking changes, and (ii) the reasons why developers introduce such changes in their libraries. In both studies, Java was chosen to be investigated due to the popularity of the language and their libraries. In this section, we describe each of these studies, as follows:

Historical and Impact Analysis. In our first study, we investigate a set of questions regarding API breaking changes. Our goal is twofold: to measure the amount of breaking changes on real-world libraries and its impact on clients at a large-scale level. Therefore, we propose the following research questions to support this study:

- ***RQ1. What is the frequency of API breaking changes?*** In this research question, we analyze the frequency of API breaking changes in the two latest releases of popular Java libraries. We observe that 28.99% of all API changes break backward compatibility. On the median, this percentage hits 14.78% of changes per library.

- ***RQ2.*** **How do API breaking changes evolve over time?** In this research question, we investigate the behavior of API breaking changes along libraries life cycle by analyzing the frequency of such changes during their first five years. We conclude that the percentage of breaking changes tends to increase over time by a rate of 20% when comparing their first and fifth years.

- ***RQ3. What is the impact of API breaking changes in client applications?*** In this research question, we analyze the impact of API breaking changes on client applications by estimating the number of possible clients of each modified API element. We observe that, on the median, only 2.54% of API clients are potentially impacted by breaking changes.

- ***RQ4. What are the characteristics of libraries with high and low frequency of breaking changes?*** In this research question, we analyze the characteristics of libraries with high frequency of breaking changes. Finally, we conclude that libraries with higher frequency of such changes are larger, more popular, and more active.

In this first study, we analyze 317 real-world Java libraries, 9K releases, and 260K client applications. Based on its results, we provide a set of lessons to better support library and client developers in their maintenance tasks. Therefore, the contributions of our historical and impact analysis are summarized as follows:

- We provide a large-scale study to better understand the extension and the impact of API breaking changes;
- We provide lessons learned from our API analysis to support library/client developers in maintenance activities.

API Breaking Changes Motivations. Next, in our second study we perform a qualitative investigation with library developers and real instances of API breaking changes. Specifically, we aim to elicit from library developers a list of motivations for API breaking changes, as well as verify their awareness on the impact on client applications. To support this study, we investigate the following research questions:

- ***RQ5. Why do developers break API contracts?*** In this research question, we analyze the reasons why developers introduce breaking changes in their libraries. For this purpose, we survey the top contributors of these libraries. We elicit a list of five main motivations, that are: Library Simplification, Refactoring, Bug Fix, Dependency Changes, and Project Policy.
- ***RQ6. Are developers aware of the impact of breaking changes on client applications?*** In this research question, we investigate whether API developers are aware about the risks of breaking changes for client applications. We observe that most of them are conscious of such risks, and, in some cases, they also adopt strategies to alleviate them (*e.g.,* deprecation annotations).

To answer these questions, we conducted a survey with the developers of popular Java libraries with the higher frequency of breaking changes observed in our first analysis. Based on its results, we suggest a future study to strengthen our current findings and also to support the development of tools to better assess the risks and impacts of API breaking changes. Thus, the contributions of this second study are summarized as follows:

- We provide a qualitative study to elicit the motivations of API breaking changes and to understand developers concerns with their impact on client applications.
- We prospect a study based on *firehouse interviews* [Murphy-Hill et al., 2015] to strengthen our current findings.

1.2 Publications

This dissertation generated the following publications and therefore contains material from them:

- Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and Impact Analysis of API Breaking Changes: A Large Scale Study. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, p. 138–147. *(Qualis A2)*
- Xavier, L., Hora, A., and Valente, M. T. (2017). Why do We Break APIs? First Answers from Developers. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER), ERA Track*, p. 392–396.
- Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2016). Um Estudo em Larga Escala sobre Estabilidade de APIs. In *4th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*, p. 1–8.

1.3 Outline of the Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** covers the main subjects related to this dissertation. We begin by explaining the concepts of *Application Programming Interfaces (APIs)*. Next, we discuss library changes and backward compatibility. Finally, we conclude by presenting the API change catalog proposed by Dig and Johnson [2006] that motivated our own catalog.

- **Chapter 3** describes in details our first empirical study: a quantitative historical and impact analysis of API breaking changes. For that, we begin by detailing our API change change and the APIDiff tool implemented to support our analysis. Next, we present the methodology of this study, followed by the results obtained for each proposed research question. Then we conclude the chapter by discussing our findings and presenting threats to validity.

- **Chapter 4** details our second empirical study: a survey with the major contributors of libraries with highest frequency of breaking changes found in the study described in Chapter 3. We begin by detailing the design of this study. Next, we provide answers to each research question, summarizing our results and presenting threats to validity in the conclusion.

- **Chapter 5** discusses the state of the art by presenting related work in the subjects of library evolution and breaking changes impact. We separate this chapter into these two subjects, highlighting the limitations of such works and discussing the contributions of our empirical studies.

- **Chapter 6** presents final considerations, including a summary of our contributions. Additionally, we provide a discussion on the prospected future work.

Chapter 2

Background

In this chapter, we discuss background subjects required to understand the work carried out in this dissertation. We start by detailing API concepts (Section 2.1), and their increasing usage nowadays. Next, we discuss library change and compatibility (Section 2.2), detailing the notion of *breaking change* and *non-breaking change*, and presenting an API change catalog previously proposed in the literature. Finally, we conclude the chapter by presenting final remarks in Section 2.3.

2.1 Application Programming Interfaces

Libraries provide interfaces to software components created to be reused by multiple client applications: the *Application Programming Interfaces (APIs)* [Reddy, 2011; Robillard et al., 2013], which are illustrated in Figure 2.1. They expose services meant to be stable by using visibility modifiers. In Java, for instance, APIs use `public` and `protected` modifiers. As a result, API elements (*i.e.,* types, fields, and methods) provide contracts on which clients (either external or internal) rely, accessing services and avoiding re-work.

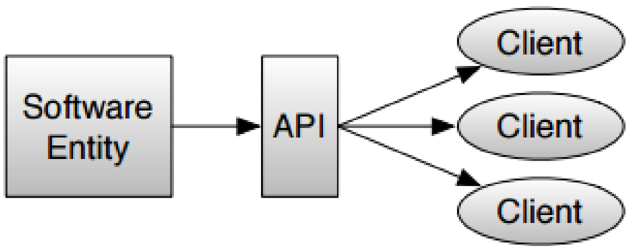


Figure 2.1. API acting as interface for client applications [Montandon, 2013].

As an example, Listing 2.1 shows a fragment of a widely used Java API, `ArrayList`. The class belongs to the `java.utils` package and provides services for a data structure known as *list*. In this example, the methods `size`, `contains`, `get`, and `add` have a `public` modifier, being available for clients as API elements. By contrast, the method `rangeCheck` and the fields `elementData` and `size` are `private`, thus they are not API elements. Finally, the class `ArrayList` itself is considered an API element once that it is public for external clients.

```java
public class ArrayList<E> extends AbstractList<E> implements List<E>,
    RandomAccess, Cloneable, Serializable {

    private transient Object[] elementData;
    private int size;

    public int size() {
        return size;
    }
    public boolean isEmpty() {
        return size == 0;
    }
    public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }
    public E get(int index) {
        rangeCheck(index);
        return elementData(index);
    }
    public boolean add(E e) {
        ensureCapacity(size + 1);
        elementData[size++] = e;
        return true;
    }
    private void rangeCheck(int index) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
}
```

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

In this context, the use of APIs in software development is increasing significantly due to the advantages they bring in terms of quality and productivity [Konstantopoulos et al., 2009; Moser and Nierstrasz, 1996]. By reusing API services, clients may benefit from the quality of components developed by experts and tested by a number of other applications. They may also save time and budget by avoiding the effort of developing services already available. Furthermore, applications may take advantage from the constant updates, when non-functional requirements are improved, such as safety and performance.

There are several examples of successful APIs, such as Java API, Android API, and .NET Framework Class Library. Some of them are used by thousands of clients worldwide. For example, Table 2.1 shows the top-10 most used Java APIs, with their corresponding number of clients, in the context of a large-scale software dataset[1] [Dyer et al., 2013, 2015]. In this case, the number of clients ranges from 60K (`java.util.Iterator`) to 143K (`java.util.ArrayList`).

Table 2.1. Top-10 most popular Java APIs.

Position	**Name**	**Number of Clients**
1	`java.util.ArrayList`	143,454
2	`java.io.IOException`	136,058
3	`java.util.List`	134,053
4	`java.util.HashMap`	94,220
5	`java.io.File`	88,703
6	`java.util.Map`	87,417
7	`java.io.InputStream`	68,000
8	`java.util.Date`	64,460
9	`android.os.Bundle`	63,434
10	`java.util.Iterator`	60,172

2.2 Library Change and Compatibility

As any other software system, during their life cycle, libraries are subjected to evolutionary changes, such as addition, removal, or modification of their API elements. Changes are usually necessary to fix critical bugs, improve performance, decrease technical debt, and release new features [Bogart et al., 2016]. Ideally, they should keep *backward compatibility*, *i.e.,* do not break contracts with client applications.

However, breaking contracts is a common practice: previous studies indicate that APIs are usually *backward incompatible* [Wu et al., 2010; Robbes et al., 2012; Hora et al., 2015; Brito et al., 2016]. Thus, migrating between versions requires extra effort, once that clients will be forced to update their code and accommodate the novelties. Actually, in most cases, clients remain hesitant to evolve and tend to delay API migration, keeping obsolete, and sometimes, faulty code [McDonnell et al., 2013].

In this context, API changes are classified into *breaking changes* and *non-breaking changes* [Dig and Johnson, 2006], as follows:

[1]According to JAVALI, a tool to measure popularity of Java libraries in Boa, available at: `http://java.labsoft.dcc.ufmg.br/javali`

- *Breaking changes.* Changes that break backward compatibility through removal or modification of API elements. As a consequence, clients may face compilation errors or behavioral changes after updating.

- *Non-breaking changes.* Changes that preserve compatibility and usually involve addition of new functionalities to the library. Thus, migrating between API versions including only non-breaking changes does not cause side effects on clients.

Listing 2.2 shows a hypothetical evolution of the `ArrayList` class previously presented in Listing 2.1. In this example, *breaking changes* are highlighted in red, while *non-breaking changes* are in green. With the purpose of broadening the discussion about API changes, we separately analyze the example in Sections 2.2.1 and 2.2.2.

```java
public class ArrayList<E> extends AbstractList<E> implements List<E>,
    RandomAccess, Cloneable, Serializable {

    private transient Object[] elementData;
    private int size;

    public int size() {
        return size;
    }
    public boolean isEmpty() {
        return size == 1;
    }
    public int contains(Object o) {
        if indexOf(o) >= 0
            return 1;
        return 0;
    }
    public E get(int index) {
        rangeCheck(index);
        return elementData(index);
    }
    public boolean add(E e) {
        ensureCapacity(size + 1);
        elementData[size++] = e;
        return true;
    }
    public void rangeCheck(int index) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
    public E getLast(){
        return get(size - 1);
    }
}
```

Listing 2.2. Evolution of the `ArrayList` class presented in Listing 2.1

2.2.1 Breaking Changes

By definition, *breaking changes* are all API modifications that break backward compatibility, changing or modifying services previously available for clients. In Listing 2.2, we observe two of them. The first one is the modification of the signature of the method `contains` (Lines 12–16). In this case, the return type was modified from `boolean` to `int`. Therefore, clients of this method will face compilation errors when updating the version of this API; then, they will be forced to re-work on their code, handling with the modification of the return type.

Most of the breaking changes are detected in compilation or linking time (*i.e.,* after updating and re-compiling, the change will cause an error). However, some changes are harder to identify and their possible effects may be more disturbing to clients, once they may cause the application to behave differently at runtime. Therefore, the functional behavior (*i.e.,* the output for a set of inputs) will change and, unless the client application has a good set of tests, the effects may be felt by its end-users. As an example, the second *breaking change* observed in Listing 2.2 is the modification of a conditional expression of the method `isEmpty` (Line 10). In this case, the value of the field `size` is compared to be equal to 1, rather than 0 in the previous version. After migrating, clients of this method will not face any compilation error to build their applications; instead, due to the modification of its behavior, some bugs may arise afterwards, compromising both system quality and reliability. However, in this dissertation we do not focus on this kind of breaking change (*i.e.,* we focus only on those that cause compilation errors after clients migration).

2.2.2 Non-breaking Changes

Non-breaking changes are defined to be all API modifications whose purpose is to add new services for clients, without compromising those previously available. Listing 2.2 presents two examples of *non-breaking changes.* The first one is the addition of the method `getLast` (Lines 30–32). As a new service added to the API, after migrating between versions, clients may take advantage of this new method to improve their code or to add new features. Therefore, in this case they do not face neither compilation nor behavioral issues.

The second *non-breaking change* is the modification of the visibility modifier of the method `rangeCheck` (from `private` to `public`). In this case, the visibility gain represents an additional service, since the method becomes available for external use. Thus, clients of this new version will also be able to access and reuse this new API element in their applications.

Moreover, changes in deprecated elements are also classified as *non-breaking changes*, once that clients have been previously alerted about the risks of using them. Deprecation annotation is one of the recommended mechanisms to mitigate the impact of *breaking changes* in client applications [Dig and Johnson, 2006]. In theory, before performing such changes, developers should annotate their API elements as deprecated and guide their clients through replacement messages. Therefore, deprecated elements are kept in the API while clients migrate and adapt their code to remove references to such elements.

Listing 2.3 exemplifies the use of deprecation in the method `contains`. Instead of simply changing its return type, a recommended approach is to keep the old method with both deprecated annotation and replacement messages containing guidelines to the adoption of the new method version (Lines 2–8). In the future, when the deprecated method is definitely removed, clients will be warned, but theoretically most of them will have already migrated to the new version. Therefore, both the addition of the method `includes` (Lines 9–13), and the removal of `contains` (in a future version) will be considered *non-breaking changes*.

```
1  public class ArrayList<E> extends AbstractList<E> implements List<E>,
       RandomAccess, Cloneable, Serializable {
2      /**
3      * @deprecated Use {@link #includes(Object)} instead.
4      */
5      @Deprecated
6      public boolean contains(Object o) {
7          return indexOf(o) >= 0;
8      }
9      public int includes(Object o) {
10         if indexOf(o) >= 0
11             return 1;
12         return 0;
13     }
14 }
```

Listing 2.3. Deprecation to mitigate the impact of the changes in method `contains`

2.2.3 Change Catalog

In a previous work, Dig and Johnson [2006] elicited a catalog of API modifications, based on refactoring operations. They manually investigated the release notes and change logs of five known Java libraries, and elicited a catalog of 20 breaking changes and 4 non-breaking changes. The proposed breaking operations are: MOVED METHOD, MOVED FIELD, DELETED METHOD, CHANGED ARGUMENT TYPE, CHANGED RETURN TYPE, REPLACED METHOD CALL, RENAMED METHOD, New Hook Method, Extra Argument, Deleted Class, Extracted Interface, Renamed Field, Renamed Class, Method Object, Push Down Method, Moved Class, Pulled up Method, Renamed Package, Split Package, Split Class. The non-breaking changes are: New Method Contract, Implement New Interface, Changed Event Order, New Enum Constant.

In this dissertation, we define and implement an API change catalog based on the one previously proposed by Dig and Johnson. However, since refactoring is not in the scope of this work, we used the Java Specification Language to analyze possible modifications on the syntax of the studied API elements (types, fields, and methods). As result, we propose a catalog of 12 breaking changes, focused on modifications that may cause compilation errors to client applications (*i.e.,* behavioral breaking changes are not included), and 9 non-breaking changes. This catalog, as well as the corresponding tool implemented to analyze it, are described in Section 3.2.

2.3 Final Remarks

In this chapter, we presented the central topics related to this dissertation, detailing the concepts of *Application Programing Interfaces (APIs)*, discussing its usage, and going deeper through their evolutionary changes. More specifically, we discussed *breaking changes* and *non-breaking changes*, detailing their effects to client applications. Finally, we presented the API change catalog proposed by Dig and Johnson [2006], detailing their operations and briefly discussing the catalog proposed in this dissertation.

In the next chapters, we will adopt these definitions with the purpose of analyzing (i) the frequency of API *breaking changes*; (ii) the behavior ot these changes over time; (iii) the impact on client applications; (iv) the characteristics of libraries with high and low frequency of *breaking changes*; (v) the reasons why developers break API contracts; and (vi) whether developers are aware of the risks of such changes.

Chapter 3

Historical and Impact Analysis

In this chapter, we present the first study of this dissertation: a historical and impact analysis of API breaking changes. The purpose is to investigate the real impact of such changes and its behavior along libraries life cycle. For that, in Sections 3.1, 3.2, and 3.3, we present the research questions that guide this study, the proposed API change catalog, and the APIDiff tool implemented to support their investigation, respectively. The design of our experiments is detailed in Section 3.4, and the results are presented in Section 3.5. Moreover, we discuss our results and present a summary of this study in Section 3.6. Threats to validity are discussed in Section 3.7. Finally, we conclude with final remarks in Section 3.8.

3.1 Research Questions

With the purpose of quantitatively investigating the frequency and impact of backward *incompatibility*, we perform a historical and impact analysis of API breaking changes. We analyze (i) the frequency of API breaking changes, (ii) the behavior of these changes over time, (iii) the impact on client applications, and (iv) the characteristics of libraries with high frequency of such changes. Therefore, our main goal is to investigate the following research questions:

- *RQ1.* What is the frequency of API breaking changes?
- *RQ2.* How do API breaking changes evolve over time?
- *RQ3.* What is the impact of API breaking changes in client applications?
- *RQ4.* What are the characteristics of libraries with high and low frequency of breaking changes?

3.2 API Change Catalog

In this dissertation, we use the definitions of *breaking changes* and *non-breaking changes* presented in Section 2.2 to define a catalog of API modifications. For this catalog, we consider only *breaking changes* that may cause compilation errors to client applications (*i.e.,* behavioral changes are not included). In the total, 21 modifications are cataloged (12 *breaking*, and 9 *non-breaking changes*). In this section, we present these changes in context of the studied API elements (*i.e.,* types, fields, and methods).

Table 3.1. Catalog of changes for *types* (classes, interfaces, and enums).

Change	**Classification**	**Description**
REMOVAL	Breaking	Removal of `public` or `protected` types that were not previously deprecated
VISIBILITY LOSS	Breaking	Visibility change (from `public` or `protected` to `private`) of types that were not previously deprecated
SUPERTYPE CHANGE	Breaking	Inheritance change of `public` or `protected` types that were not previously deprecated
ADDITION	Non-Breaking	Addition of new `public` or `protected` types in a new version
VISIBILITY GAIN	Non-Breaking	Visibility change from `private` to `public` or `protected`
DEPRECATED OPERATIONS	Non-Breaking	Modifications (*e.g.,* removal or visibility loss) in `public` or `protected` deprecated types

Table 3.1 presents the catalog of changes for *types* (*i.e.,* classes, interfaces, and enums). In this case, breaking changes include removal of a type, change on its visibility modifier (*e.g.,* from `public` or `protected` to `private`), and change in the type's supertype. By contrast, non-breaking changes include addition of new elements and change on visibility modifiers (*e.g.,* from `private` to `public` or `protected`). Finally, changes in deprecated elements (*e.g.,* removal of deprecated methods) are also classified as *non-breaking changes.*

For *fields*, Table 3.2 details the *breaking changes* and *non-breaking changes* cataloged. Besides the trivial ones, also listed for types in Table 3.1 (*i.e.,* REMOVAL and VISIBILITY LOSS), breaking changes in fields include, for example, modifications in the field's type or default value. Additionally, *all* non-breaking changes cataloged for fields are similar to the ones listed for types, involving operations in deprecated fields, visibility gain, and addition of new ones.

Table 3.2. Catalog of changes for *fields*.

Change	**Classification**	**Description**
Removal	Breaking	Removal of `public` or `protected` fields that were not previously deprecated
Visibility Loss	Breaking	Visibility change (from `public` or `protected` to `private`) of fields that were not previously deprecated
Type Change	Breaking	Change of the type of `public` or `protected` fields that were not deprecated
Default Value Change	Breaking	Addition, removal, and modification of the default initializer value of `public` or `protected` fields that were not deprecated
Addition	Non-Breaking	Addition of new `public` or `protected` fields in a new version
Visibility Gain	Non-Breaking	Visibility change from `private` to `public` or `protected`
Deprecated Operations	Non-Breaking	Changes (*e.g.*, type or default value change) in `public` or `protected` deprecated fields

Table 3.3. Catalog of changes for *methods*.

Change	**Classification**	**Description**
Removal	Breaking	Removal of `public` or `protected` methods that were not previously deprecated
Visibility Loss	Breaking	Visibility change (from `public` or `protected` to `private`) of methods that were not previously deprecated
Return Type Change	Breaking	Modification of the type returned by `public` or `protected` methods that were not deprecated
Parameter List Change	Breaking	Parameters addition, removal, and type modifications of `public` or `protected` methods that were not deprecated
Exceptions Change	Breaking	Addition, removal, and type change of exceptions thrown by `public` or `protected` methods that were not previously deprecated
Addition	Non-Breaking	Addition of new `public` or `protected` methods in a new version
Visibility Gain	Non-Breaking	Visibility change from `private` to `public` or `protected`
Deprecated Operations	Non-Breaking	Modifications (*e.g.*, removal or visibility loss) in `public` or `protected` deprecated methods

Finally, Table 3.3 details the catalog of changes for *methods*. Breaking changes in methods include some of the changes listed before, such as REMOVAL and VISIBILITY LOSS; but also embrace other related to their formal signature, such as modification of the method's return type, changes in the parameters list (*i.e.,* addition, removal, and type change), and modifications on thrown exceptions (*i.e.,* addition, removal, and type change). Non-breaking changes, by contrast, include all modifications previously discussed for types and fields (*i.e.,* ADDITION, VISIBILITY GAIN, and DEPRECATED OPERATIONS).

3.3 APIDiff Tool

In order to support our investigation on the frequency of API breaking changes, we define and implement an APIDIFF tool whose purpose is to identify *breaking* and *non-breaking changes* between two versions of a Java library. In this section, we present this tool by providing an overview of its approach (Section 3.3.1), and detailing its architecture (Section 3.3.2).

3.3.1 Overview

To identify and classify API changes between two versions of a Java library, our APIDIFF tool evaluates each of the changes detailed in our API change catalog (see Section 3.2). For example, breaking changes in types include removal of a type, change on its visibility modifier (*e.g.,* from `public` to `protected`), and change in the type's supertype. Breaking changes in fields include, for example, changes in the field's type or default value. Breaking changes in methods include, for example, changes in their signatures. By contrast, non-breaking changes include addition of new elements and change on visibility modifiers (*e.g.,* from `private` to `public` or `protected`). Furthermore, changes in deprecated elements (*e.g.,* deprecated method removal) are classified as *non-breaking changes* by our APIDIFF tool, because developers in this case have been previously alerted about the risks of using deprecated entities.

To accomplish that, our tool implements a parser based on the Eclipse JDT library. We focus on public and protected API elements (types, fields, and methods) once that they represent the external contract between libraries and clients. It takes as input the path of both versions (named version 1 and 2), via the following *command*:

```
java -jar APIDiff.jar [path_version_1] [path_version_2]
```

Next, to compute the frequency of changes, the tool acts as illustrated in Figure 3.1. First, it parses both library versions to find all *.java* files to be analyzed by the JDT Library (Steps 1 and 2). The library constructs the Abstract Syntax Trees (ASTs) for each version (Step 3). Then, with both ASTs, the tool starts the analysis by checking all API elements on version 1, verifying whether any of the cataloged API changes occurs in version 2 (Step 4). Next, it analyzes the API elements on version 2 with the purpose of verifying additions and new deprecations to report as *non-breaking changes* (Step 5). Finally, the tool reports all detected changes in a *.xls* file (Step 6) containing each modified element, its enclosing type, and the change description as presented in Section 3.2.

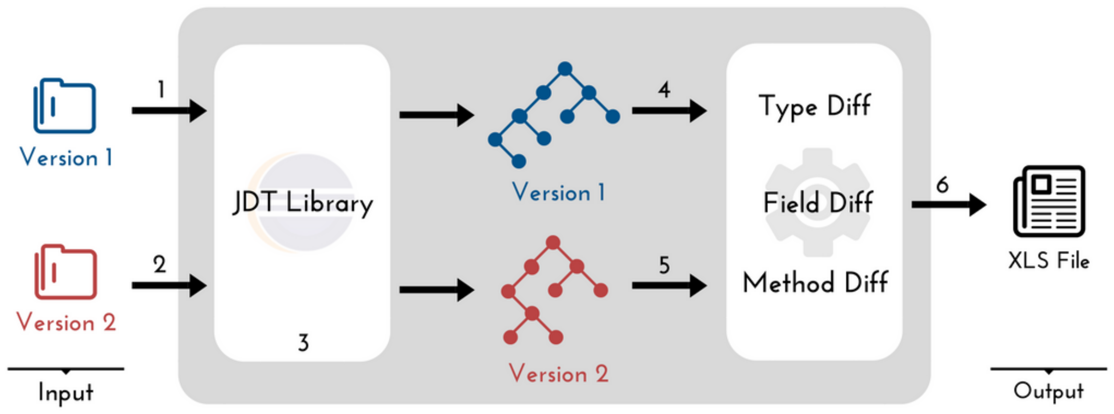


Figure 3.1. APIDiff tool approach overview.

To illustrate the usage of the tool, Listings 3.1 and 3.2 present a hypothetical scenario of evolution of a famous Java library for the *map* data structure: `java.utils.HashMap`. Suppose that between versions 1 and 2 the class has its supertype changed (from `AbstractMap` to `AbstractHashMap`), a new argument added to the paramenter list of the method `clear`, and a `deprecated` annotation added to the same method. In this example, *breaking changes* are highlighted in red, and *non-breaking changes* in green. As a result, the analysis of our APIDiff tool would detect two *breaking changes* (Supertype Change and Paramenter List Change), and one *non-breaking change* (Deprecated Operations). Figure 3.2 presents the output of the tool.

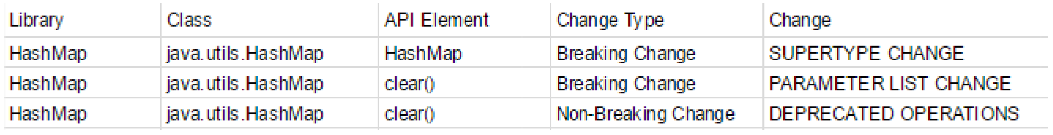
```java
public class HashMap<k, V> extends
    AbstractMap<K, V>{

    public void clear() {
        //...
    }
}
```

Listing 3.1. Version 1 of `HashMap`.

```java
public class HashMap<k, V> extends
    AbstractHashMap<K, V>{
    @Deprecated
    public void clear(boolean flag) {
        //...
    }
}
```

Listing 3.2. Version 2 of `HashMap`.



Library	Class	API Element	Change Type	Change
HashMap	java.utils.HashMap	HashMap	Breaking Change	SUPERTYPE CHANGE
HashMap	java.utils.HashMap	clear()	Breaking Change	PARAMETER LIST CHANGE
HashMap	java.utils.HashMap	clear()	Non-Breaking Change	DEPRECATED OPERATIONS

Figure 3.2. APIDIFF tool output for Listings 3.1 and 3.2.

3.3.2 Architecture

Our tool is mainly implemented over the JDT Eclipse Library, which is responsible for parsing all *.java* files and creating the ASTs of both analyzed versions. Besides that, each modification in our API change catalog is analyzed by comparing the ASTs in the corresponding element diff (*e.g.*, type modifications are investigated in the `TypeDiff` class). The results are stored in a `HashMap` structure at the class `Finder` and, at the end, saved in a *.xls* file by the class APIDIFF. Figure 3.3 presents the UML class diagram of the tool. The responsibility of each class is also briefly described as follows:

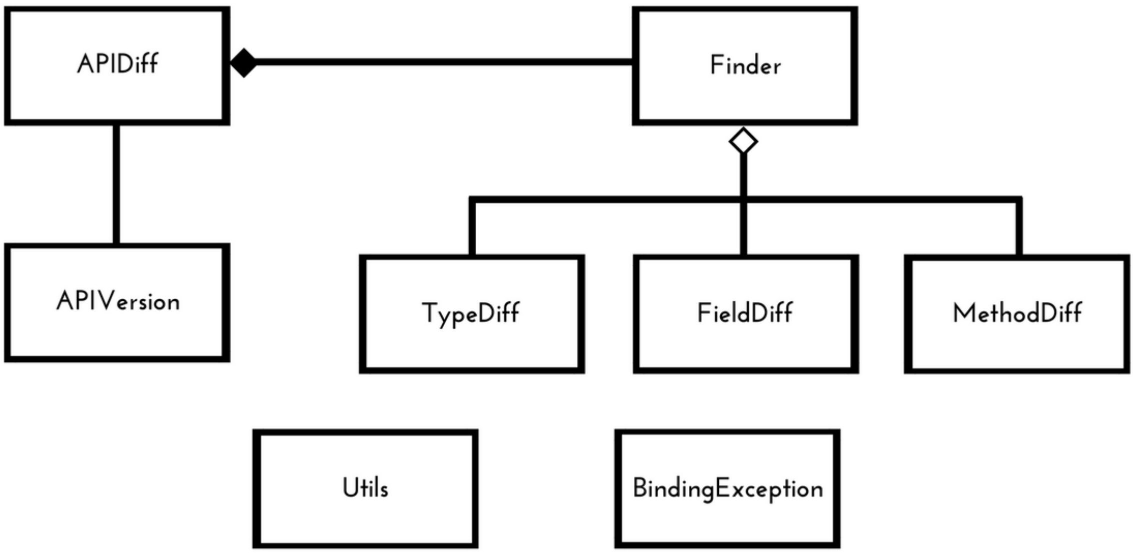


Figure 3.3. UML class diagram of our APIDIFF tool.

- `APIDiff`: Main class, responsible for receiving the input, storing the ASTs (`APIVersions`), and interacting with the `Finder` class;
- `APIVersion`: Abstraction of the AST for an API Version;
- `Finder`: Stores the changes observed in each API element finder;
- `TypeDiff`: Analyzes the changes in all `public` and `protected` types;
- `FieldDiff`: Analyzes the modifications in all `public` and `protected` fields;
- `MethodDiff`: Analyzes the changes in all `public` and `protected` methods;
- `Utils`: Provides auxiliary utility functions;
- `BindingException`: Deals with exceptions caused by binding issues during the AST construction.

3.4 Study Design

To answer the proposed research questions, we conducted a large-scale experiment with the most relevant libraries hosted on GitHub. In this section, we present the design of this study, detailing our dataset in Section 3.4.1, and discussing the methodologies we adopted for each research question (Sections 3.4.2, 3.4.3, and 3.4.4).

3.4.1 Selecting Java Libraries

We analyzed the most popular Java libraries hosted on GitHub (collected in August, 2016). First, we selected the top 1,000 repositories ordered by number of stars. Then, we manually classified them into *library* (554 repositories, 55.40%) and *non-library* (446 repositories, 44.60%). Finally, from the *library* group, we discarded the ones in the first quartile of number of releases and age, in order to filter out irrelevant projects [Kalliamvakou et al., 2014], as follows:

- ***Number of releases.*** We selected libraries with two or more releases (*i.e.,* first quartile equals to 1). We applied this criteria to focus on active libraries and to ensure at least one pair of releases to be compared in each library.
- ***Age.*** We selected systems with more than 515 days from the first commit (*i.e.,* first quartile equals to 515 days). We use this criteria to assess libraries with a relevant evolution and to ensure historical data to our analysis.

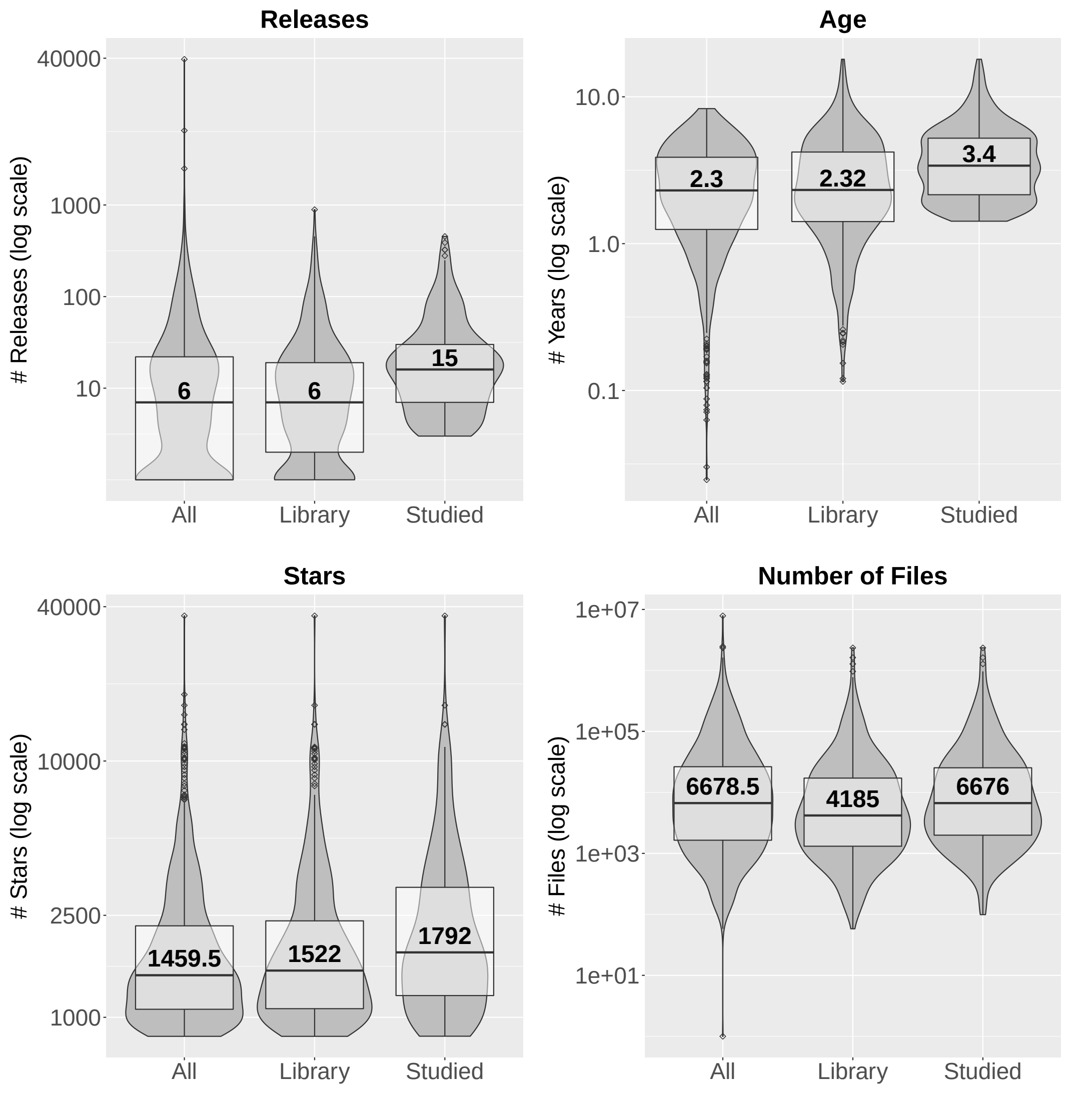


Figure 3.4. Repositories distributions by releases, age, stars, and files.

Based on this filtering criteria, the final selection has *317 libraries*, including well-known ones such as FACEBOOK/REACTNATIVE, GOOGLE/GUAVA, and JUNIT-TEAM/JUNIT4. To better characterize these libraries, Figure 3.4 presents the distribution of number of releases and age (in years) as well as number of stars and files. We provide violin plots for *all* 1,000 initial repositories, for the 554 repositories categorized as *library*, and for the 317 *studied* libraries. Violin plots are useful for presenting the distribution of data because besides embedding a box plot, they also show the probability density of the data at different values.

Considering the *studied* libraries, we have the following results. For number of releases, the first quartile, median, and third quartile are 6, 15, and 29 releases. The top-3 repositories with more releases are: PROCESSING/PROCESSING (453 releases), druid-io/druid (387), and h2oai/h2o-2 (324). For age, the quartiles are 2.2, 3.4, and 5.2 years. The top-3 older repositories are: java-native-access/jna (18 years), junit-team/junit4 (15.8), and processing/processing (15.2). For number of stars, the first quartile, median, and third quartile are 1,216, 1,792 and 3,215 stars. The top-3 systems with more stars are: facebook/react-native (36,856 stars), ReactiveX/RxJava (16,493), and square/okhttp (13,910). For number of files, the quartiles are 1,298, 6,676, and 25,335 files. The top-3 larger repositories are: processing/processing (1,625,224 files), apereo/cas (1,277,502), and libgdx/libgdx (964,580). Finally, we observe that the *studied* libraries are statistically significant different for number of releases, age, and number of stars (*p-value* < 0.05 for Mann-Whitney test), when compared to *all* repositories and also to the *libraries* repository. However, they are not statistically different for number of files.

3.4.2 Extracting API Breaking Changes (*RQ1* and *RQ2*)

To measure the frequency of API breaking changes, we use the APIDiff tool presented in Section 3.3. Let R_n be the last release and R_1 the first one of a given library. To answer *RQ1 (What is the frequency of API breaking changes?)*, we computed the *diff* between releases R_n and R_{n-1}, *i.e., diff(R_n,R_{n-1})*. Figure 3.5 illustrates this approach. In this case, we use the two latest versions to estimate the current state of each studied library, despite of the evolution of their release history.

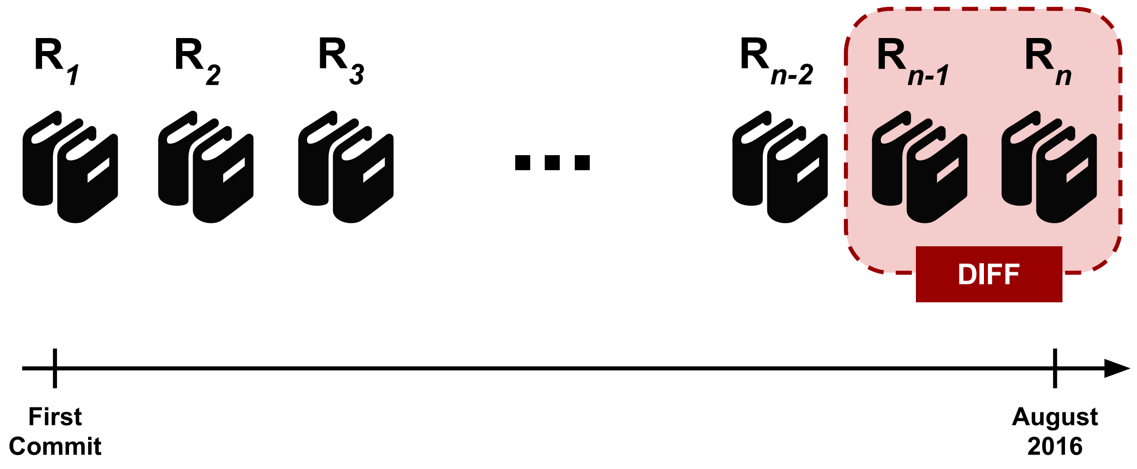


Figure 3.5. *diff* approach to answer *RQ1*.

Moreover, to answer *RQ2 (How do API breaking changes evolve over time?)*, we compared *all* subsequent releases (from R_1 to R_n) of a library, *i.e., diff(R_i,R_{i-1})* for $i = 2, \ldots, n$. To better understand this data, we summarized the breaking changes over time by calculating the mean of the amount of their occurrences per year. For that, we grouped the changes observed in the libraries first five years, and compared results on each period. Figure 3.6 details this methodology.

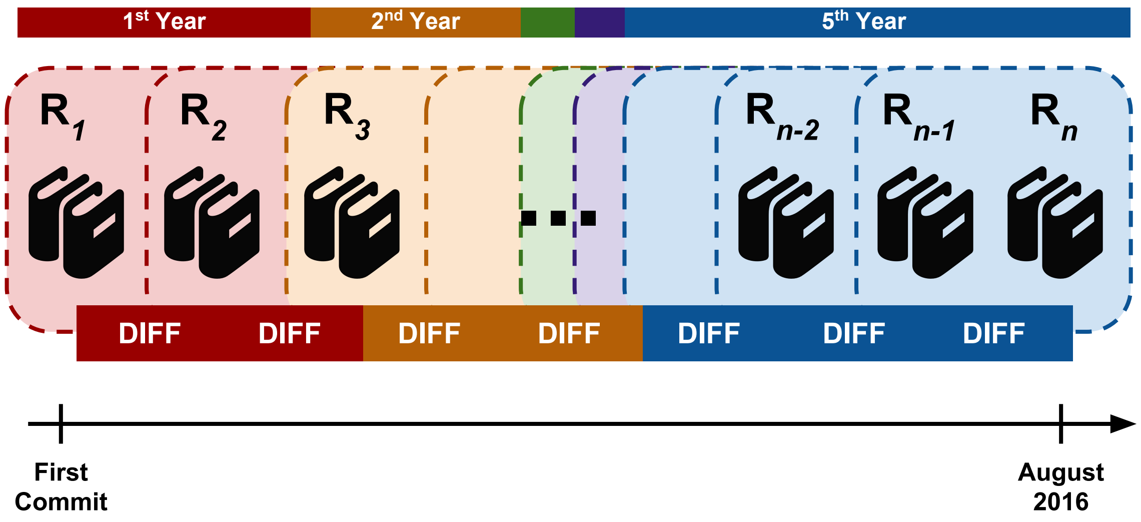


Figure 3.6. *diff* approach to answer *RQ2*.

3.4.3 Measuring API Breaking Changes Impact (*RQ3*)

To support answering *RQ3 (What is the impact of API breaking changes in client applications?)*, we calculated the impact of the breaking changes identified in *RQ1* on client systems. Using an ultra-large dataset of Java systems [Dyer et al., 2013, 2015], we counted the ones that feature an *import* statement to the detected breaking changes. For types, we perform a direct analysis by looking for their qualified names. For methods and fields, on the other hand, we assess the *imports* of their enclosing type. In other words, if a breaking change is detected in a method `m` of a class `C`, we count as potentially impacted all clients that import `C`. This approach at least retrieves the worst case scenario of the potential impact measure.

Listings 3.3 and 3.4 illustrate this decision. Consider the hypothetical change scenario between versions 1 and 2 of the class `ArrayList`: the return type of the method `contains` was modified from `boolean` to `int`. In this case, to calculate the possible impact of this change, we would consider the total of clients that import the class `ArrayList` in their projects. Thus, we would estimate that 143,454 clients (see Table 2.1 in Chapter 2) are *possibly* impacted by this change. A similar result would happen if class `ArrayList` were removed or renamed.

```java
public class ArrayList{

    public boolean contains(Object o){
        return indexOf(o) >= 0;
    }

}
```

Listing 3.3. Version 1 of `ArrayList`.

```java
public class ArrayList {
    public int contains(Object o) {
        if indexOf(o) >= 0
            return 1;
        return 0;
    }
}
```

Listing 3.4. Version 2 of `ArrayList`.

We used the JAVALI (Java Libraries and Interfaces)[1] tool with the purpose of calculating the number of imports of a given API and, as a result, collecting client systems information. The tool aims to measure the popularity of Java libraries by processing a dataset with more than 260K Java systems, 16M files, and 131M APIs. To collect this data, JAVALI uses a Domain-Specific Language (DSL) and infrastructure that aims to ease mining software at a ultra-scale level: the Boa language [Dyer et al., 2013, 2015]. This DSL leverages distributed computing techniques to execute queries about open-source projects mined from software repositories. In the JAVALI case, the Boa language was used to query projects hosted on GitHub, analyzing information about import statements. We use this information to measure the possible impact of *breaking changes* in types with at least one import statement.

3.4.4 Comparing Libraries with High and Low Frequency of Breaking Changes (*RQ4*)

In order to distinguish libraries with low and high rates of API breaking changes, we classified the studied libraries in two groups (*top* and *bottom*) to answer *RQ4 (What are the characteristics of libraries with high and low frequency of breaking changes?)*. We then collected a set of project metrics (such as activity, size, etc) to compare both groups. The goal is to verify whether these metrics have an impact on the number of API breaking changes. This process is summarized in the following three steps:

1. Defining metrics likely to impact breaking changes. To analyze libraries with high and low rate of breaking changes, we define five dimensions related to open-source development and social coding: popularity, size, community, activity, and maturity. For each, we define specific metrics to measure and characterize the studied libraries. These metrics were also used in a previous study about the adoption of replacement messages in API deprecation [Brito et al., 2016]. Each dimension and the corresponding metrics are described in Table 3.4.

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

Table 3.4. Metrics likely to impact *breaking changes*, divided in five dimensions.

Dimension	**Description**	**Metrics**
Popularity	Represents how popular a library is on GitHub	*number of stars* *number of watchers* *number of forks*
Size	Characterizes the library volume of artifacts	*number of files* *number of API elements*
Community	Represents the library community size	*number of contributors* *average files per contributor* *average API elements per contributor*
Activity	Characterizes the activity level of a library development team	*number of commits* *number of releases* *average days per release*
Maturity	Represents the age of a library	*number of years*

2. Selecting Top and Bottom libraries. We consider two groups of libraries: the ones with low rate of breaking changes, labeled as *top libraries*, and a second group, labeled as *bottom libraries*, with higher rates of breaking changes.

We first identified the active libraries, *i.e.,* the ones with at least one API change (either breaking or non-breaking), resulting in 235 libraries. Then, we sorted these 235 libraries, in ascending order, by the percentage of API breaking changes. Finally, we ended up with two groups: top-25% (*i.e.,* libraries with the lowest percentage of breaking changes) and bottom-25% (*i.e.,* libraries with the highest percentage); each group with 58 libraries. Figure 3.7 shows the distribution of breaking changes in both groups. As expected, the median percentage of changes is low (0%) for *top libraries* and very high (73.75%) for *bottom* ones. Table 3.5 shows the name of five top and five bottom libraries. All top libraries in this table have no breaking changes; by contrast, in the bottom libraries, all detected API changes are classified as breaking changes.

3. Extracting metrics and comparing libraries. We extracted the metrics in Table 3.4 for both *top* and *bottom* libraries and then compared the obtained values. First, we analyze the statistical significance of the difference between both groups by applying the Mann-Whitney test at *p-value* = 0.05. To show the effect size of the difference between them, we compute Cliff's Delta (or d). Following guidelines previously adopted in the literature [Grissom and Kim, 2005; Tian et al., 2015; Linares-Vásquez et al., 2013], we interpret the effect size values as small for $0.147 < d < 0.33$, medium for $0.33 < d < 0.474$, and large for $d > 0.474$.

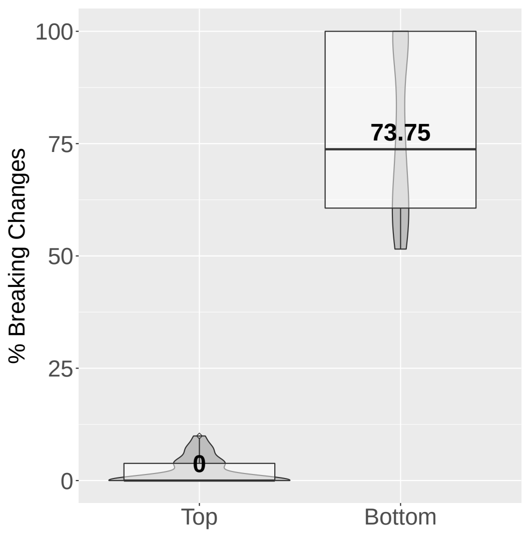


Figure 3.7. Breaking changes distribution in top-25% and bottom-25% libraries.

Table 3.5. Example of *Top* and *Bottom* libraries ordered by number of breaking changes.

Group	Library	Breaking Changes
Top	CHRISBANES/ACTIONBAR-PULLTOREFRESH	0 (0%)
	GOOGLEMAPS/ANDROID-MAPS-UTILS	0 (0%)
	FACEBOOK/CONCEAL	0 (0%)
	CHRISJENX/CALLIGRAPHY	0 (0%)
	MIKEPENZ/ABOUTLIBRARIES	0 (0%)
Bottom	KYMJS/KJFRAMEFORANDROID	4 (100%)
	GRAILS/GRAILS-CORE	5 (100%)
	MONGODB/MONGO-HADOOP	5 (100%)
	LIAOHUQIU/CUBE-SDK	19 (100%)
	ZEROMQ/JEROMQ	23 (100%)

3.5 Results

In this section, we answer and analyze the results of the proposed research questions.

RQ1: What is the frequency of API breaking changes?

We analyze the frequency of changes for types, fields, and methods between the two latest releases, *i.e., diff(R_n,R_{n-1})*, of the 317 studied libraries. We identified at least one change in 235 libraries (74.13%). From this total, 198 libraries (62.46%) have at least one breaking change, while 218 (92.77%) have at least one non-breaking change. Table 3.6 presents the number of changes per API element (*i.e.,* types, fields, and methods). Considering all elements, 501,645 changes were identified, from which 27.99% are breaking changes and 72.01% are non-breaking changes. Methods are the API elements with more changes, including breaking changes. Considering the 140,460 breaking changes, 27.81% are in methods.

Table 3.6. Number of API breaking and non-breaking changes.

Element	Total	Breaking Change	Non-Breaking Change
Types	61,897	11,712 (18.92%)	50,185 (81.08%)
Fields	66,953	25,044 (37.41%)	41,909 (62.59%)
Methods	372,795	103,704 (27.81%)	269,091 (72.19%)
All	501,645	140,460 (27.99%)	361,185 (72.01%)

To understand the stability of the studied libraries, Figure 3.8 presents the distribution of absolute and relative breaking changes (*i.e.,* the percentage of breaking change per API modification). A logarithmic scale is applied to absolute plots so we can better visualize outlier libraries.

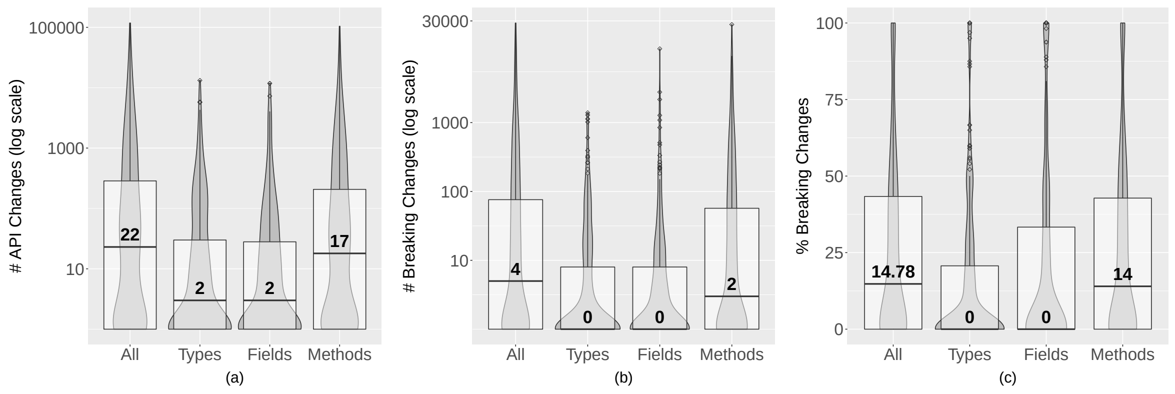


Figure 3.8. Distribution of API changes for all elements, types, fields, and methods. (a) Absolute number of all changes, (b) absolute number of breaking changes, and (c) relative number of breaking changes.

Absolute analysis. Figure 3.8(a) shows the absolute distribution of the number of changes (breaking and non-breaking) per library. Considering all API elements, the first quartile is 0, the median is 22, and the third quartile is 285 changes. On the median, types and fields have two changes while methods have 17. The third quartile for types, fields, and methods is 29, 27, and 206 changes, respectively.

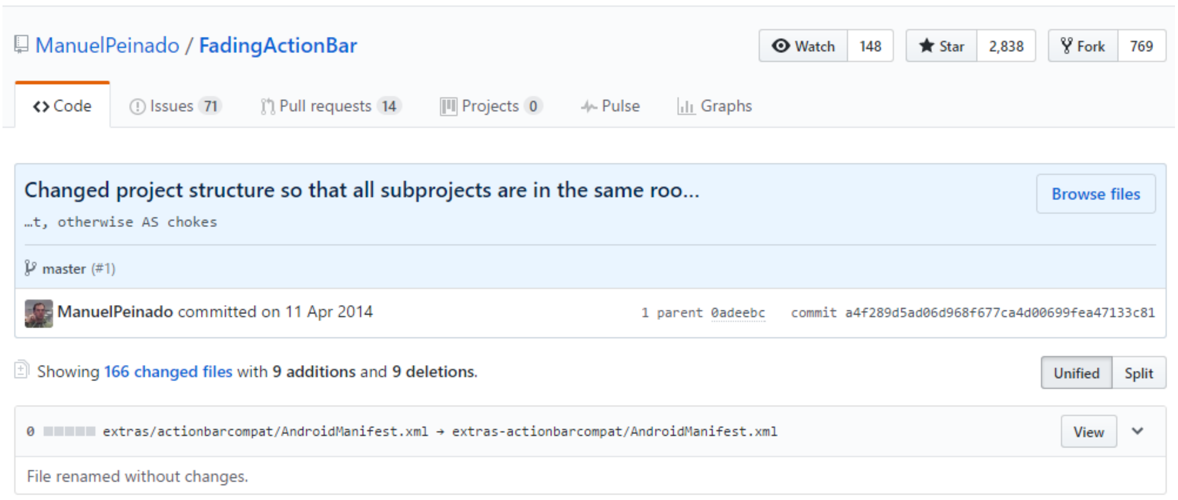
Figure 3.8(b) details the previous analysis by exploring the absolute distribution of *breaking changes* per library. Considering all API elements, the first quartile is 0, the median is 4, and the third quartile is 75. In absolute terms, we note that types and fields present similar distributions (median equal to 0). However, outlier values are very different: we observe a library with 11,816 breaking changes for fields, and another one with 1,392 breaking changes for types. We manually analyzed both cases. The first one happened in the Android library ManuelPeinado/FadingActionBar, when the project structure faced a major change, as described in the commit message:

> *"Changed project structure so that all subprojects are in the same root."*

The second one happened in the graph library neo4j/neo4j, when several changes were inserted to improve its design. One example is found in the pull request (PR) that removed the type `SchemaRuleContent`:

> *"This PR makes sure that all types of schema rules are properly checked and simplifies duplicates checking by removal of SchemaRuleContent."*

Figures 3.9, and 3.10 present the screenshot of these repositories, detailing both commit message and pull request, respectively.



Changed project structure so that all subprojects are in the same roo…

…t, otherwise AS chokes

master (#1)

ManuelPeinado committed on 11 Apr 2014 1 parent 0adeebc commit a4f289d5ad06d968f677ca4d00699fea47133c81

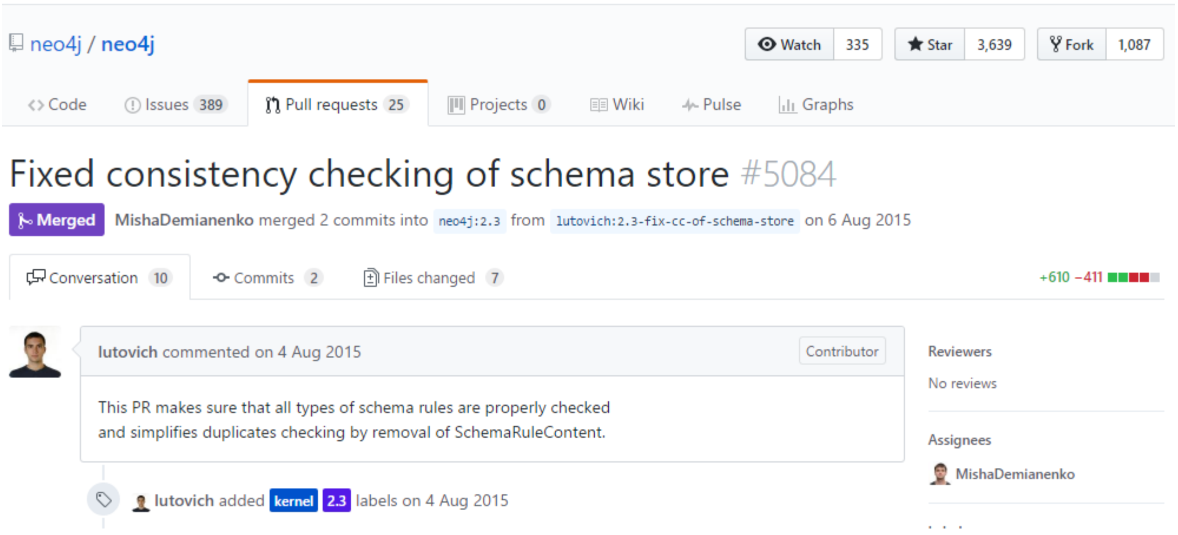
Showing 166 changed files with 9 additions and 9 deletions.

0 extras/actionbarcompat/AndroidManifest.xml → extras-actionbarcompat/AndroidManifest.xml

File renamed without changes.

Figure 3.9. Commit message in the library ManuelPeinado/FadingActionBar reporting the breaking changes observed.

Relative analysis. Figure 3.8(c) presents the distribution of the relative number of breaking changes per library. For all API elements, the first quartile is 0%, the median is 14.78%, and the third quartile is 43.35%. Moreover, we found 17 libraries (5.35%) with 100% of breaking changes, such as Netflix/astyanax, nathanmarz/storm, and grails/grails-core. But in all these cases, the absolute number of changes is also small (at most 23 changes in Netflix/astyanax).


neo4j / neo4j

Fixed consistency checking of schema store #5084

Merged MishaDemianenko merged 2 commits into neo4j:2.3 from lutovich:2.3-fix-cc-of-schema-store on 6 Aug 2015

lutovich commented on 4 Aug 2015

This PR makes sure that all types of schema rules are properly checked and simplifies duplicates checking by removal of SchemaRuleContent.

lutovich added kernel 2.3 labels on 4 Aug 2015

Figure 3.10. Pull request message in the library NEO4J/NEO4J exemplifying the breaking changes observed.

Summary: From the 501,645 analyzed API changes, we observe a relevant rate of breaking changes (27.99%). On the median, 14.78% of the API changes in a library break contracts with clients; the higher ratio of breaking changes occurs on methods.

RQ2: How do API breaking changes evolve over time?

To answer this second research question, we verify *all* releases (from R_1 to R_n) of the 317 studied libraries. The goal is to analyze the frequency of breaking changes over time and, thus, to investigate the impact of software evolution on library stability. To accomplish that, we verify *9,329 releases* and summarize the frequency of breaking changes per year. Because the third quartile of the studied libraries age is 5.2 years, we decided to analyze at most five years of their evolution. In addition, due to our selection criteria discussed in Section 3.4.1, the studied libraries have at least one year.

Figure 3.11 presents the relative distribution of the means of breaking changes per library and per year. Those with no versions released in a given year were discarded. For each library, we calculate the mean number of breaking changes in each year, by considering only the releases in the year. In this way, we generate distributions per library and per year. In the first year of existence, 232 libraries released public versions. The first quartile of the means is 16.65%; the median, 29.02%; and the third quartile, 42.74%.

For breaking changes in releases during the second year, the first quartile is 15.32%, the median is 31.46%, and the third quartile is 47.72%. From the total, 212 libraries registered at least one release during their second year. From the first to the

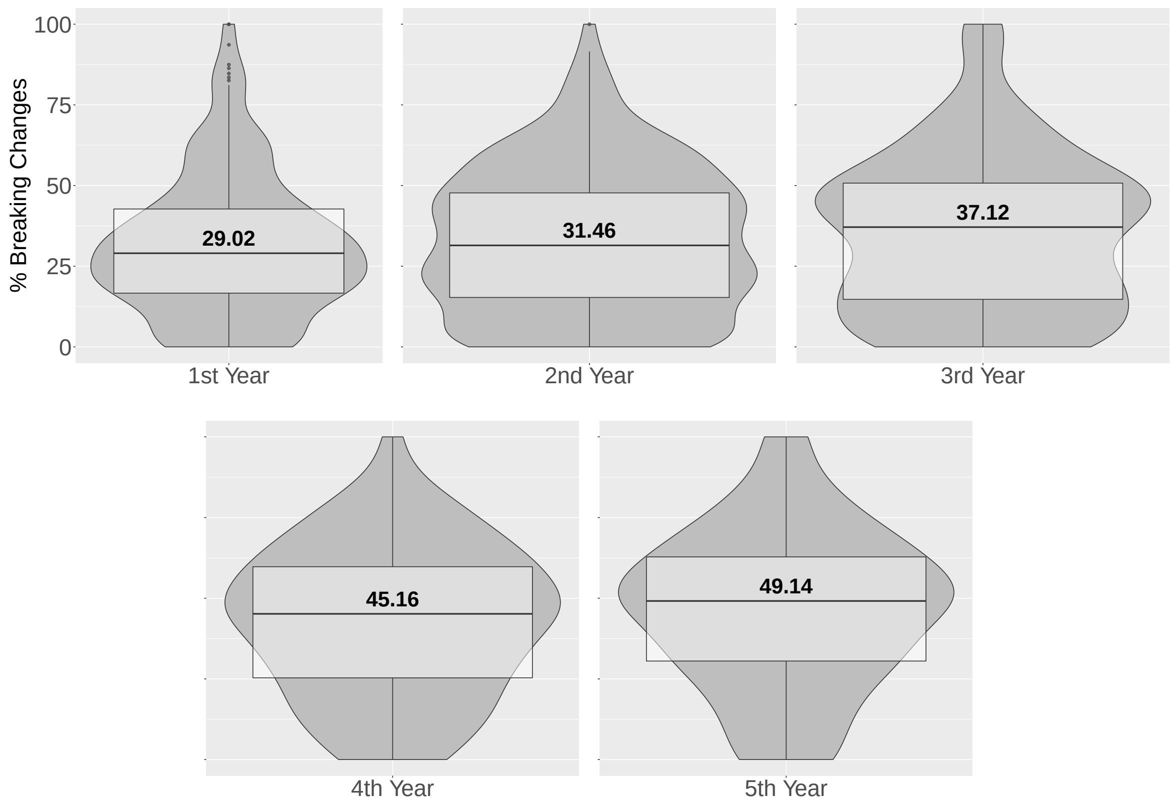


Figure 3.11. Distribution of API breaking changes per year. The distribution values are the mean rate of changes in a year, considering the releases produced in this year.

second year, we observe a light increase of 2.44% in the median value. However, the Mann-Whitney test reveals no statistical significant difference between both groups.

In the third year, 149 libraries were analyzed. The first quartile, the median, and the third quartile are, respectively: 14.73%, 37.12%, and 50.75%. Comparing to the previous years, the median is slightly higher, increasing 5.66% and 8.10% when compared to the second and first years, respectively. Despite of that, the Mann-Whitney test does not show a statistical significant difference between the three years.

In our dataset, 106 libraries have version released during their fourth year. The quartile values are 25.33%, 45.16%, and 59.76%, respectively. The frequency of breaking changes increases by 8.04%, 13.70%, and 16.14% when compared to the third, second, and first years, respectively. In this case, the Mann-Whitney test reveals that this group is statistically significant different from the three previous ones.

Finally, in the fifth year, 83 libraries have released versions. The first quartile is 30.53%, the median is 49.14%, and the third quartile is 62.80%. From the fifth to the fourth years, we do not observe statistical significant difference.

Therefore, the historical analysis of the breaking changes frequency reveal that they increase by 20% in five years (median values). This may be explained by the fact that as time passes, libraries tend to provide API elements that are harder to manage and more likely to change.

As an illustrative example, Figure 3.12 plots the curve of breaking changes frequency for three of the studied libraries: GOOGLEMAPS/ANDROID-MAPS-UTILS, DROPWIZARD/METRICS, and ROBOGUICE/ROBOGUICE. For the first, we plot the evolution along its four years of existence, once that the library has less than five years. During this time, we observe a small increase of breaking changes (from 3.91% to 8.70%), against the tendency observed in the dataset. For the others, we register values for their first five years: in the first case, the curve for ROBOGUICE/ROBOGUICE grows in a variation of more than 60% of breaking changes (from 37.88% to 99.30%); in the second (DROPWIZARD/METRICS), we observe that the frequency ranges from an initial growth (from 21.02% to 42.10%), to a subsequent decrease to 38.51%, increasing again in the fourth and fifth years. In both cases, the percentage of increase comparing the first and fifth years represents a variation higher than the median of 20%.

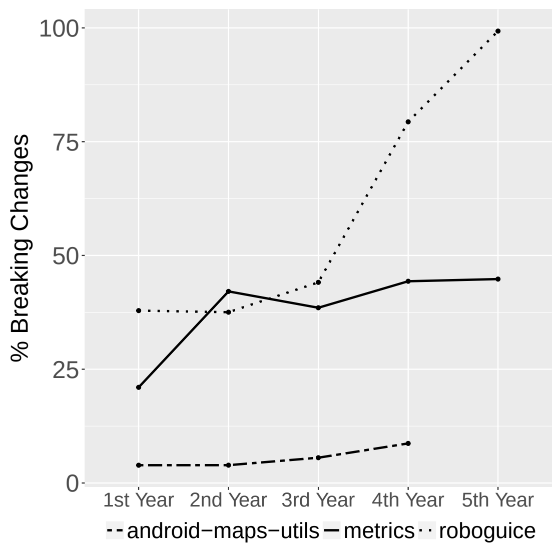


Figure 3.12. Breaking changes evolution for three libraries: GOOGLEMAPS/ANDROID-MAPS-UTILS, DROPWIZARD/METRICS and ROBOGUICE/ROBOGUICE.

Summary: The frequency of breaking changes increases over time. Comparing the first to the fifth year, this number increases by 20% (from 29.02% to 49.14%). This may occur because as libraries evolve, they become larger and more likely to change.

RQ3: What is the impact of API breaking changes in client applications?

In this third research question, we investigate the impact of the breaking changes reported in *RQ1* on client applications. For that, we analyze both the types with breaking changes and the types declaring fields and methods with breaking changes. The goal is to assess the *potential impact* of breaking changes by analyzing *import* statements in client systems. As detailed in Section 3.4.3, our dataset of client applications has around 260K Java systems.

Considering all API elements, 140,460 breaking changes were detected (see Table 3.6), referring to 16,291 types. From such types, 1,290 (7.91%) potentially impacted at least one client application, *i.e.,* clients with at least one *import* to these types in our dataset. For the remaining types with breaking changes, we did not find clients in the Javali/Boa dataset; therefore, they were discarded. Figure 3.13 presents the distribution of absolute and relative number of impacted clients per library/type. A logarithmic scale is applied to absolute plots to ensure outliers visualization.

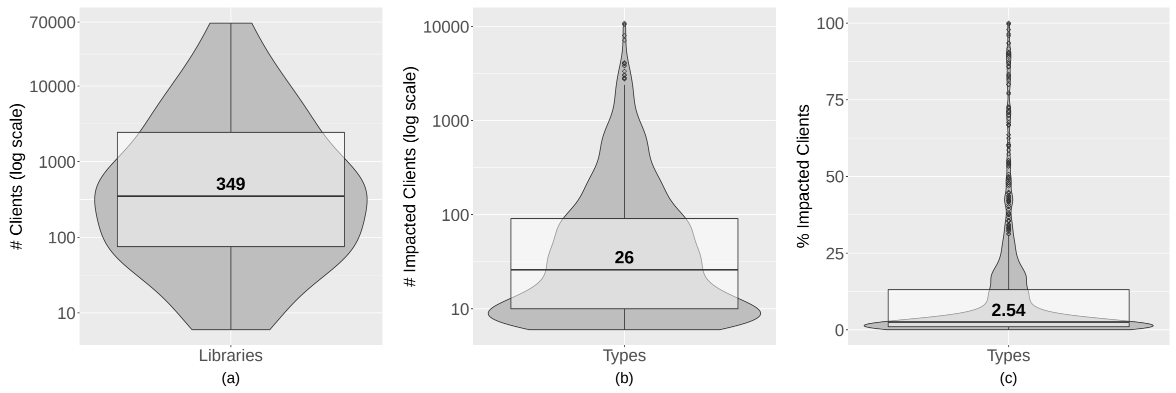


Figure 3.13. Impact of API breaking changes in client applications: (a) number of clients of APIs with breaking changes, (b) number of clients impacted by each type with a breaking change, and (c) relative number of clients impacted by each type with a breaking change.

Figure 3.13(a) presents the absolute distribution of the number of impacted clients per library. The first quartile is equal to 75 clients; the median, 349; and the third quartile is 2,245 clients. In this context, the top-3 libraries with more impacted clients are junit-team/junit4, with 54,217 clients; spring-projects/spring-framework, with 23,793 clients; and google/guava, with 12,524 clients.

Figure 3.13(b) shows the absolute distribution of the number of clients impacted per type with breaking change. The first quartile, the median, and the third quartile are 10, 26 and 90.75 clients, respectively. Despite the low numbers registered by the quartiles, the top-3 types with more impacted clients belong to well-known libraries: `org.junit.Assert` (imported on 10,857 clients), `junit.framework.Assert` (imported on 10,535 clients), and `org.bukkit.plugin.java.JavaPlugin` (imported on 8,005 clients).

Finally, Figure 3.13(c) details the relative distribution of the impacted clients, *i.e.,* number of clients impacted by a breaking change in a given API divided by the total number of clients of this API. The first quartile is 1%, the median is 2.54%, and the third quartile is 13.10%. The top-3 types with higher rates of breaking change impact are: `⋆.streaming.video.VideoQuality`, from FYHERTZ/LIBSTREAMING, with 100%; `⋆.chronicle.Excerpt`, from PETER-LAWREY/JAVACHRONICLE, also with 100%; and `⋆.scene2d.ui.Label`, from LIBGDX/LIBGDX, with 99.64%. However, in all these three cases, the number of clients of each type is low (at most 280 clients in LIBGDX/LIBGDX).

Therefore, the impact of breaking changes in terms of impacted clients tends to be low (2.54%, on the median). This may indicate that (i) library developers are careful before inserting breaking changes on heavily-used types, or (ii) the changed types are for internal usage only (*i.e.,* APIs not created for external clients, but for the system itself) [Hora et al., 2016; Businge et al., 2013]. However, we also notice many outliers (123 types) with more than 32.03% of clients impact. Table 3.7 lists the breaking changes with the highest impact on clients.

Table 3.7. Top-10 breaking changes with the highest impact on clients.

Type	**Impact**
`⋆.streaming.video.VideoQuality`	100.00%
`⋆.chronicle.Excerpt`	100.00%
`com.badlogic.gdx.scenes.scene2d.ui.Label`	99.64%
`android.content.res.AssetManager`	97.87%
`⋆.mustachejava.DefaultMustacheFactory`	96.47%
`android.telephony.TelephonyManager`	95.86%
`com.android.volley.RequestQueue`	93.52%
`org.bukkit.plugin.java.JavaPlugin`	93.45%
`org.pegdown.PegDownProcessor`	91.67%
`android.widget.AbsListView`	90.51%

Summary: 2.54% of the client applications are potentially impacted by breaking changes, on the median. One possible explanation is that developers may be careful to break widely-used types ot that most APIs elements may not be widely used by client application.

RQ4: What are the characteristics of libraries with high and low frequency of breaking changes?

To analyze libraries with high and low percentage of breaking changes, we compare *top* and *bottom* libraries as described in Section 3.4.4. The purpose is to verify whether library popularity, size, community, activity, and maturity impact the frequency of breaking changes.

Table 3.8 details the metrics related to each characteristic and the respective *p-values* and d coefficients obtained for *top* and *bottom* libraries. Metrics in bold have *p-value* < 0.05, and $d > 0.147$, *i.e.,* they are statistically significant different with at least a small effect size in *top* and *bottom* libraries. As can be observed in the table, the selected *top* and *bottom* libraries are statistically significant different in 6 out of the 12 metrics. The effect size is small in three metrics (number of watchers, number of API elements, and number of releases), and medium in other three (number of watchers, number of contributors, and average API elements per contributor). Next, we analyze each group:

Table 3.8. Metrics and their respective *p-values* and d on *top* and *bottom* libraries. Bold means *p-value* < 0.05 (statistically significant different) and $d > 0.147$ (at least a small effect size). Direction: "↑" = *top* libraries have significantly higher value on this metric. "↓" = *bottom* libraries have significantly higher value on this metric.

Dimension	Metric	p-value	d-value	Size	Direction
	number of stars	0.490	0.272	small	↓
Popularity	***number of watchers***	**0.016**	**0.377**	**medium**	↓
	number of forks	0.679	0.247	small	↓
Size	*number of files*	0.010	0.017	negligible	↓
	number of API elements	**< 0.001**	**0.149**	**small**	↓
	number of contributors	**0.014**	**0.330**	**medium**	↓
Community	*average files per contributor*	0.454	0.192	small	↓
	average API elements per contributor	**< 0.001**	**0.335**	**medium**	↓
	number of commits	**< 0.001**	**0.219**	**small**	↓
Activity	***number of releases***	**0.001**	**0.251**	**small**	↓
	average days per release	0.003	0.088	negligible	↑
Maturity	*age (in number of days)*	0.350	0.215	small	↓

- ***Popularity.*** Libraries with higher measures for *number of watchers* are on the *bottom* group, *i.e.,* they have higher values of breaking changes. Thus, our results suggests that popular libraries (at least, in number of watchers) are more likely to break compatibility. This contradicts our initial conjecture, once we believed that popular libraries would be more careful before inserting breaking changes. In fact, based on these results, we hypothesize that popular libraries have more pressure to evolve, including the need to make design decisions that lead to breaking changes.

- ***Size.*** Libraries with higher measures for *number of API elements* are also on the *bottom* group. Indeed, libraries with more API elements tend to be harder to maintain and evolve, increasing the probability of breaking changes.

- ***Community.*** Libraries with higher measures for *number of contributors* and *average API elements per contributors* also appear on the *bottom* group. Thus, our results suggest that libraries with more contributors tend to have more breaking changes than the others.

- ***Activity.*** Libraries with higher measures for *number of commits* and *number of releases* are on the *bottom* group. Thus, our results suggest that more code changes are more likely to break compatibility.

- ***Maturity.*** Finally, we detected that there is no statistical significant difference between *top* and *bottom* libraries with respect to their *age* (in number of days).

As an illustrative example, Table 3.9 details a comparison between a *top* and a *bottom* library. The *top* library is DAIMAJIA/ANIMATIONEASINGFUNCTIONS, with no breaking change at all (among 17,550 changes). On the other hand, the *bottom* one is ZEROMQ/JEROMQ, with 100% of API changes classified as *breaking change* (among 23 changes). Indeed, it is clear the difference between both libraries regarding the metrics we found a statistically significant result.

Table 3.9. Comparison between a top and bottom library, respectively: DAIMAJIA/ANIMATIONEASINGFUNCTIONS and ZEROMQ/JEROMQ.

Metric	**Top Library**	**Bottom Library**
number of watchers	94	142
number API elements	139	3,139
number of contributors	3	40
avg API elements per contributor	46.33	78.48
number of commits	22	528
number of releases	2	8

Summary: Bottom libraries are statistically significantly different from top ones in 6 out of 12 metrics. Libraries with more contributors and more API elements per contributor have more breaking changes, with medium effect size. Also, the number of API elements, the number of commits, and the number of releases affect breaking changes, with small effect. Maturity, though, has no effect on breaking changes.

3.6 Summary and Findings

Based on our historical large-scale study on 317 real-world Java libraries, their 9K releases, and 260K client applications, we derive the following findings and implications into API breaking changes impact and evolution:

1. Libraries often break backward compatibility. We show that 27.99% of all API changes break backward compatibility. On the median, the percentage of breaking changes per library hits 14.78%. In this context, we observe that API breaking changes are recurrent and occur in a relevant percentage. This may occur due to several reasons, for example, (i) unawareness of breaking change risks, (ii) development by naive or less experienced programmers, or (iii) need to restructure the library and, consequently, change the API elements. Therefore, we point out the need for further investigation on reasons developers break contracts with client applications. In the next chapter, we provide an analysis on this subject.

2. Breaking changes frequency increases over time. Our study shows that the percentage of breaking changes tends to increase over time by a rate of 20% when comparing their first and fifth years (from 29.02% to 49.14%). This result shows that as time passes, libraries do not become more reliable and stable, as expected. Thus, we suggest the adoption of historical analysis to measure library stability, warning API developers about the increase of breaking changes on their libraries. This analysis would also provide useful information for client developers when reasoning whether to depend or not on a library.

3. Most breaking changes do not have a massive impact on clients. Despite the high number of verified breaking changes, we observe that, on the median, only 2.54% of clients are potentially impacted. This low percentage may indicate that (i) library developers pay especial attention on the usage of types before breaking contracts or (ii) the changed types are for internal usage, *i.e.,* not intended to be used by client applications. However, the ratio of impacted clients increases to 13% in a quarter of the studied libraries. Moreover, the analysis of outlier values shows that this impact can be very large, reaching 100% of clients in some cases. Based on that, an impact analysis tool can be helpful for library developers to support their decisions before changing highly used APIs.

4. Development and social coding measures are associated with API breaking changes. We show that libraries with higher frequency of breaking changes have specific project characteristics. We found statistically significant higher values for the following metrics: *number of watchers*, *number of API elements*, *number of contributors*, *average API elements per contributor*, *number of commits*, and *number of releases*. Thus, libraries with higher frequency of breaking changes are larger, more popular, and more active. Moreover, notice that the relative measure on the workload of API elements per contributor is also associated with high frequency of breaking changes: the more API elements a contributor has to maintain, the more unstable is likely to be the library. Thus, we suggest the usage of relative development metrics (such as *average API elements per contributor*) as a proxy that developers should use to assess the "health" of their libraries.

3.7 Threats to Validity

3.7.1 Construct Validity

Construct validity is related to whether the measurements in the study reflect real-world situations.

Classification of Repositories. One possible threat of our study is that repositories may have been incorrectly classified into *library* and *non-library*. *Non-library* systems in our studied dataset may bias the results obtained. However, an especial attention was dedicated to this manual classification through the analysis of each repository web page and documentation.

Historical Analysis. In our historical analysis, we consider the first five years of each studied library which represents the third quartile of their age (5.2 years). Therefore, this value can be considered a representative threshold, although not covering the entire life cycle of the studied libraries.

Impact Analysis. To calculate the impact of breaking changes, we count the number of client applications that feature an *import* statement to types that hold a breaking change. A known threat of this decision relates to the impact of breaking changes in fields and methods, since an *import* to their enclosing type does not implies in real usage. However, this measure at least represents the worst case scenario.

3.7.2 Internal Validity

Internal validity is related to uncontrolled aspects that may affect the experimental results.

Parser Implementation. A possible threat is the possibility of errors in the implementation of our APIDiff tool, which identifies breaking and non-breaking changes in Java API elements. However, to mitigate this threat, the implementation of APIDiff is largely based on a well-known Eclipse library: JDT.

Findings Validation. We paid special attention to the appropriate use of statistical tests (*i.e.,* Mann-Whitney test and Cliff's Delta effect size), specially when reporting the results in *RQ4*. This reduces the possibility that these results are due to chance.

Association and Causation. In *RQ4*, we examined whether there are metrics correlated with high and low frequency of breaking changes. However, it is important to acknowledge that correlation does not imply causation [Couto et al., 2014].

3.7.3 External Validity

External validity is related to the possibility to generalize our results. We focused on 317 popular Java libraries hosted in GitHub, the most used code repository nowadays. Therefore, they are credible and representative case studies, with source code easily accessible. In addition, our client applications dataset has more than 260K Java systems. Despite these observations, our findings—as usual in empirical Software Engineering—cannot be generalized to other libraries, specifically those implemented in other programming languages. Moreover, we only consider syntactical breaking changes, which result in compilation errors. Behavioral API changes are outside the scope of this dissertation.

3.8 Final Remarks

In this chapter, we presented the first study of this dissertation, performed in the context of 317 real world Java libraries, 9K releases, and 260K clients. Four research questions were investigated in order to support library/client developers in maintenance activities. Specifically, we applied historical and impact analysis to assess: (i) the frequency of breaking changes, (ii) the behavior of these changes over time, (iii) the impact on client applications, and (iv) the characteristics of libraries with high and low frequency of breaking changes.

Therefore, based on our results we could observe that: (i) libraries often break backward compatibility (at a percentage of 27.99% of all API changes), (ii) breaking changes frequency increase over time (at an increasing rate of 20%), (iii) most breaking changes do not have a massive impact on clients (only 2.54% of clients are potentially impacted, on the median), and (iv) development and social coding measures are associated with API breaking changes (in relation to size, popularity, and activity).

In the next chapter, we present a qualitative analysis with developers that aims to better investigate the reasons of API breaking changes in practice. Based on the results observed in this first study, we selected a subset of libraries to survey their developers and, as a consequence, (i) we elicit a list of reasons that motivate them to introduce breaking changes, and (ii) we verify whether they are aware of their risks.

Chapter 4

API Breaking Changes Motivations

In this chapter, we present the second study of this dissertation: a survey with API developers, whose purpose is to investigate the reasons why *breaking changes* are introduced, and the awareness of developers about their risks for client applications. In Section 4.1, we begin by detailing each research questions investigated. Next, we present the methodology of our survey and the obtained results in Sections 4.2 and 4.3, respectively. In Section 4.4, we discuss our results and present a summary of this study. In Section 4.5, we list the threats to validity, and discuss the strategies adopted to mitigate them. Finally, we conclude the chapter with some final remarks in Section 4.6.

4.1 Research Questions

In order to investigate the specific reasons that drive API developers to introduce *breaking changes* in their libraries, we perform a qualitative study with such developers and real instances of API breaking changes. For that, we selected the libraries with the highest amount of breaking changes in our quantitative study (see Chapter 3), and contacted their contributors to understand the reason of such changes. Therefore, we investigate (i) the reasons why developers implement breaking changes, and (ii) whether they are aware about the risks of these changes. Specifically, our main goal is to elicit a list of motivations for API breaking changes based on library developers answers, and to verify whether they are aware of the impact on client applications. To guide this investigation, we propose the following research questions:

- *RQ5.* Why do developers break API contracts?
- *RQ6.* Are developers aware of the impact of breaking changes on client applications?

4.2 Study Design

In this section, we describe the methodology of our survey by summarizing its four main steps: selecting surveyed developers (Section 4.2.1), contacting developers (Section 4.2.2), filtering responses (Section 4.2.3), and analyzing data (Section 4.2.4).

4.2.1 Selecting Surveyed Developers

First, we selected the repositories with more than *50 breaking changes* collected in our previous study, described in Chapter 3. Out of the 317 libraries, 90 (28.39%) filled this selection criteria. Then, we accessed each repository with the purpose of retrieving the email address of their major contributor (*i.e.,* the developer with the highest amount of commits in the repository according to GitHub statistics, as shown in Figure 4.1).

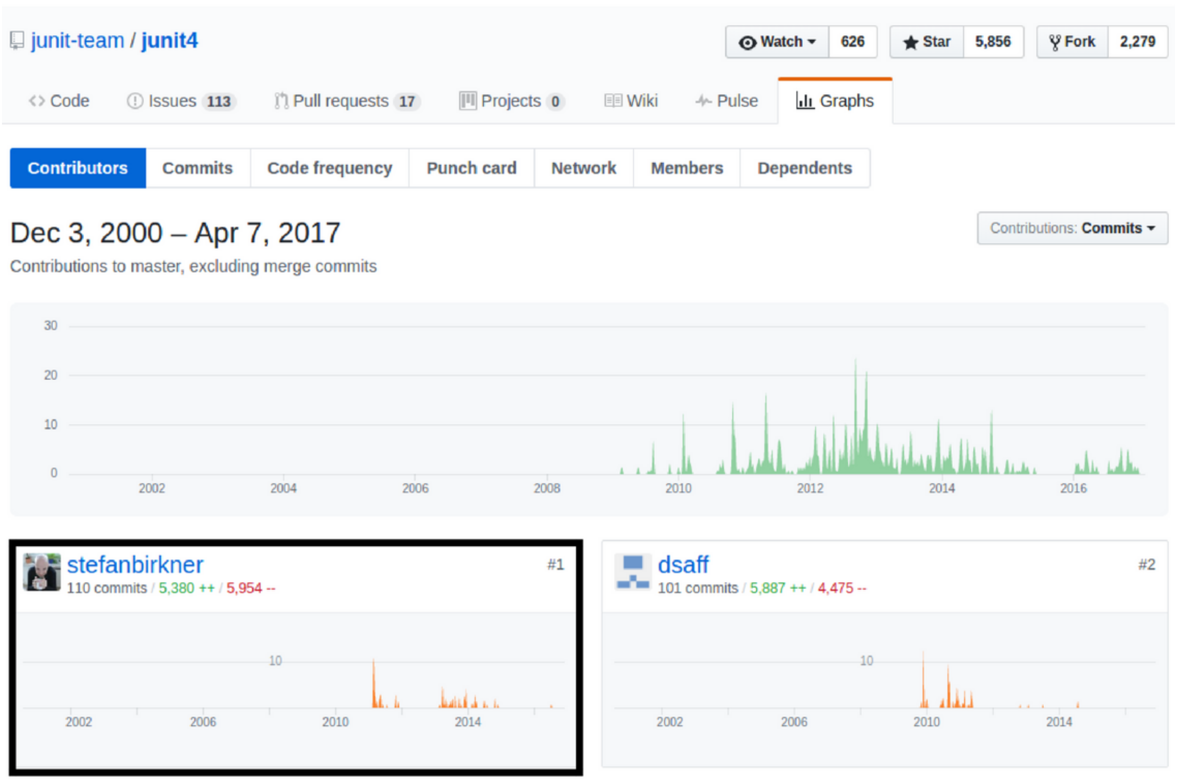


Figure 4.1. Major contributor of JUNIT-TEAM/JUNIT4.

From the 90 selected libraries, 49 (54.44%) of their major contributors shared their email on GitHub profile. Therefore, our initial dataset consists of the corresponding *49 libraries*, including well-known and worldwide used projects, such as BITCOINJ/BITCOINJ, JAVASLANG/JAVASLANG, and JUNIT-TEAM/JUNIT4.

4.2.2 Contacting Developers

For each selected library, we sent an email to its corresponding major contributor (between November 12th and 25th 2016). Figure 4.2 presents the sent email, as well as the proposed questions. In each email, we presented the number of collected breaking changes and an external link describing each of them (*i.e.*, the filtered output of our APIDiff tool, containing each modified element, its enclosing type, and the *breaking changes* description as presented in Section 3.2). Figure 4.3 illustrates a fragment of this list for the OBLAC/JODD library. Moreover, we proposed three questions with the goal of (i) verifying whether developers are aware about the listed breaking changes; (ii) investigating their reasons for implementing breaking changes; and (iii) verifying whether they are aware about the risks of breaking changes to client applications. Specifically, the first question was used as a filtering criteria (*i.e.*, developers who said not being aware of the breaking changes had their emails discarded).

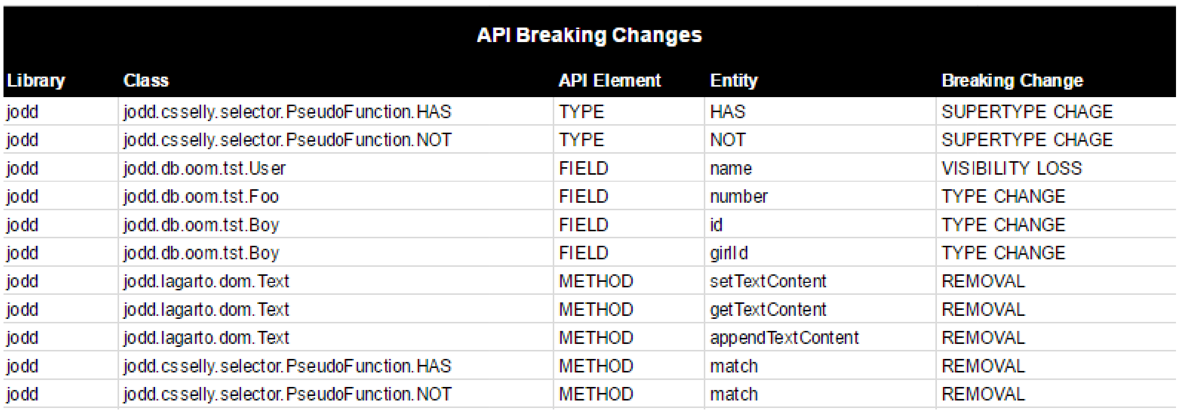
Dear [developer name],

*I figured out that you are a major contributor of [*REP/PROJECT*], from which we found [n] API breaking changes, for instance, in classes A and B (further details here [link]).*

I kindly ask you to answer the following questions to support our research:

1. Are you aware about these API breaking changes?
2. Could you describe why were these API breaking changes introduced?
3. Are you aware about the risks of breaking backward compatibility with your clients?

Figure 4.2. Email sent to the major contributors of the studied libraries



API Breaking Changes

Library	Class	API Element	Entity	Breaking Change
jodd	jodd.csselly.selector.PseudoFunction.HAS	TYPE	HAS	SUPERTYPE CHAGE
jodd	jodd.csselly.selector.PseudoFunction.NOT	TYPE	NOT	SUPERTYPE CHAGE
jodd	jodd.db.oom.tst.User	FIELD	name	VISIBILITY LOSS
jodd	jodd.db.oom.tst.Foo	FIELD	number	TYPE CHANGE
jodd	jodd.db.oom.tst.Boy	FIELD	id	TYPE CHANGE
jodd	jodd.db.oom.tst.Boy	FIELD	girlId	TYPE CHANGE
jodd	jodd.lagarto.dom.Text	METHOD	setTextContent	REMOVAL
jodd	jodd.lagarto.dom.Text	METHOD	getTextContent	REMOVAL
jodd	jodd.lagarto.dom.Text	METHOD	appendTextContent	REMOVAL
jodd	jodd.csselly.selector.PseudoFunction.HAS	METHOD	match	REMOVAL
jodd	jodd.csselly.selector.PseudoFunction.NOT	METHOD	match	REMOVAL

Figure 4.3. Fragment of the breaking change list sent to OBLAC/JODD developer.

4.2.3 Filtering Responses

We received 14 answers, which represents a response rate of 28.6%. From these answers, 6 were considered unclear or invalid (*e.g.,* responses reporting that the developer is no longer engaged with the project). The following answer from a former developer of LMAX-Exchange/disruptor illustrates it:

> *"I am no longer involved with the Disruptor and do not engage on the topic."*

Additionally, the developer of avast/android-styled-dialogs library stated that he was not aware of the breaking changes reported (first question). Thus, we executed our APIDiff tool a second time and manually analyzed the results with the purpose of verifying false positives. As a result, we confirmed the existence of all *breaking changes* reported. As the developer stated he was not aware of them, his answer (illustrated in the following fragment) was also discarded.

> *"I'm not aware about breaking changes recently. The library is very stable, doesn't change often."*

Therefore, *7 answers* were considered for analysis in this study. Table 4.1 describes the libraries whose responses were analyzed, as well as basic information about them (*i.e.,* number of stars and total of breaking changes). The number of stars ranges from 857 (bitcoinj/bitcoinj) to 1,568 (mogobd/mongo-java-driver), showing they are popular libraries. The number of breaking changes ranges from 53 (oblac/jodd) to 3,117 (mogobd/mongo-java-driver).

Table 4.1. Libraries with valid answers

	Library	**Stars**	**Breaking Changes**
D1	mongodb/mongo-java-driver	1,568	3,117
D2	oblac/jodd	1,445	53
D3	Zielony/Carbon	1,380	358
D4	square/assertj-android	1,354	2,218
D5	javaslang/javaslang	1,108	663
D6	davideas/FlexibleAdapter	975	157
D7	bitcoinj/bitcoinj	857	1,940

4.2.4 Analyzing Data

After collecting and filtering all emails, we analyzed the responses in order to investigate the proposed research questions. To answer *RQ5 (Why do developers break API contracts?)*, we followed a thematic analysis, which is a technique whose goal is to identify themes (or codes) within a set of documents [Cruzes and Dyba, 2011]. Thus, each response to the second question in the survey email was manually analyzed and a list of reasons that may explain why developers break API contracts was cataloged. The advisor and co-advisor of this dissertation reviewed the analysis and confirmed the proposed codes. Finally, to answer *RQ6 (Are developers aware of the impact of breaking changes on client applications?)*, we analyzed responses to the third question, and, as a result, we identified insights on how developers deal with the impact of API breaking changes.

4.3 Results

In this section, we present the results obtained in our survey. We separate the section in each research question investigated, answering to them and providing further details observed in our analysis.

RQ5. Why do developers break API contracts?

We identified *five* main reasons that suggests the motivations that drive API developers to introduce *breaking changes* in their libraries, as well as their recurrent explanations for such changes. Table 4.2 describes these reasons, providing a brief description, and detailing the number of occurrences for each of them.

Table 4.2. Reasons why developers break API contracts

Theme	**Description**	**Occur.**
Library Simplification	Redesign to make APIs easier for clients	3
Refactoring	Remodularization to improve quality code	2
Bug Fix	Resolution of issues	2
Dependency Changes	Switch of libraries on which the library depend on	1
Project Policy	Maintenance policy of the project	1

Next, we discuss each theme, detailing fragments of the obtaining answers, and analyzing the changes that motivated them.

LIBRARY SIMPLIFICATION. The most frequent reason for breaking API contracts is related to LIBRARY SIMPLIFICATION (3 instances). In this case, the change is mainly motivated by the need of making APIs easier to use (*e.g.,* developer-friendly code); and also by the remotion of redundant (and more complex) elements. Developers D6, D2, and D3 mentioned this motivation:

> *"Useless item. If a problem can be solved using another simple method, the library can be simplified by removing the redundant solution."* [D6]

> *"The change leads to better and more developer-friendly code (for example, to more fluent code). For example, we recently had one important API change in BeanUtil, where we moved from utility class with static methods to a bean class. Software is living and changing thing, and we constantly are looking Jodd to be more efficient for developers."* [D2]

> *"A problem was solved in a different way. This is the case of addStateAnimator and removeStateAnimator methods. These methods were removed because StateAnimator was rewritten and the functionality was refactored. Another examples are RadioButton and CheckBox classes."* [D3]

REFACTORING. Differently from LIBRARY SIMPLIFICATION theme, whose intents are related to improving the library for external clients, the second most frequent motivation relates to REFACTORING (2 instances). In this case, developers pointed the need of internally improving the code of their APIs (*e.g.,* by moving elements between packages), and also the modification on the way that problems are solved. D6 illustrates this motivation, detailing a refactoring on his library with the purpose of better organizing package signatures:

> *"The classes/methods/fields are not removed all, they are just refactored to a better package signature (many months ago/last year) when the library was know a little but not famous as now... When possible they are initially deprecated and then removed completely."* [D6]

In addition, developer D3 explains that a REFACTORING was applied to change the way a problem was solved, moving functionalities to classes that could provide the same services in a more general way. More specifically:

> *"A problem was solved in a more general way. For example carbon.widget.SVGView was a class used for displaying .svg files. With an introduction of VectorDrawable that functionality was moved to ImageView and there was no need for a separate SVGView class. StateAnimator, FloatingActionButton and RippleDrawable are another examples of such change."* [D3]

Other studies also indicate REFACTORING as a reason for breaking changes. For example, Dig and Johnson [Dig and Johnson, 2006] found that 80% of the breaking changes are due to refactoring (more details in Chapter 5).

Bug Fix. The third most commented reason that motivates API breaking changes is related to Bug Fix (2 instances). In this case, developers are guided by the need of solving some issue in their libraries and, as a consequence, end up breaking contracts with client applications. Developers D3 and D2 discuss this motivation, as illustrated in the following fragments:

> *"Bugfix. For example some of the items shouldn't be accessible and were made private."* [D3]

> *"Our approach is that we are going to make such changes if there is an issue that has to be fixed, [...]"* [D2]

In a recent study, performed with the purpose of analyzing the relationship between project policies and API breaking changes, Bogart et al. [2016] also cite Bug Fix as a possible reason of such changes (more details in Chapter 5).

Dependency Changes. In addition, developer D4 discussed another motivation related to changing the library dependencies: Dependency Changes. According to him, the breaking changes reported were caused by the need of changing one library that they depend on and was not being maintained anymore:

> *"We switched the assertion library on which the library was based since FEST library was no longer being developed and AssertJ was a maintained and updated fork."* [D4]

Project Policy. Finally, developer D6 comments that introducing *breaking changes* is a deliberated maintenance practice in their project. In this case, we analyzed their repository and found that they treat them by documenting all changes and keeping a well defined release versioning. The following fragment illustrates it:

> *"It's a deliberate policy. bitcoinj has never done a 1.0 release that would have posted API stability."* [D7]

RQ6. Are developers aware of the impact of breaking changes on client applications?

To verify whether developers are aware about the risks of breaking APIs, we analyze the answers to the third question proposed in our initial email. Out of the seven responses, in *five* instances developers affirmed being aware of the risks. In the two remaining, we identified unclear or vague answers. Thus, *all* developers who gave valid responses recognized being aware of the impact and costs of breaking changes. In some cases, they also discussed alternatives to mitigate them.

A first and natural alternative to decrease the impact of breaking changes is to use deprecation annotations and replacement messages [Brito et al., 2016; Robbes et al., 2012]. Both developers D3 and D2 cited this strategy. However, developer D2 discusses the lack of human resources to maintain deprecated methods.

> *"I always try to mark things I would like to remove as deprecated, give replacements and document changes to make transitions easy."* [D3]

> *"Once one client asked to use @Deprecated on old methods, but we simply dont have enough resources to maintain all deprecated methods."* [D2]

In addition, both developers justified their breaking changes by highlighting the small number of clients of their libraries. Our previous study (Chapter 3) confirms this fact. The library maintained by D3 has no client affected by the collected breaking changes, while only one class of D2's library impacts 7 clients (which represents 9% of the total). The following comments illustrates their statements:

> *"Carbon is not a commercial, production-quality library, so I'm not as concerned about potential problems as Google is with their libraries. I'm just working on my ideas and I'm giving my solutions to the public."* [D3]

> *"Yes. But we are not Spring yet. [...] Being a small-to-middle library has it's benefits."* [D2]

Finally, developer D4 illustrated an interesting strategy to mitigate the risks of breaking changes. With the purpose of rebuilding the library due to DEPENDENCY CHANGE reasons, and reusing most of the initial code, the decision was to create a new library in the Maven Central Repository.[1] Thus, clients interested in migrating had to switch libraries (and update their code to the new API contracts). This is illustrated in the following comment:

[1] `https://maven.apache.org`

> *"From the consumer perspective it's a totally different library, not just a new version of an existing one that has a new API. In order to upgrade, consumers would have to change their build to point at the new coordinates. If all they were doing was looking for a new version of the old coordinates they would never see it. Additionally, because it's separate coordinates you can even have both versions installed side-by-side and do incremental migration. We decided to keep the same repository despite essentially creating a new library because they solve the same problem, we could re-use 90% of the code, and there never would be releases made of the old version once we switched."* [D4]

4.4 Summary and Findings

In this section, we summarize the results and answer the investigated research questions.

RQ5. Why do developers break API contracts? We elicit a list of *five* specific reasons pointed by developers as motivation for API breaking changes: LIBRARY SIMPLIFICATION, REFACTORING, BUG FIX, DEPENDENCY CHANGES, and PROJECT POLICY. Some of them were recurrent between respondents. For instance, LIBRARY SIMPLIFICATION was discussed in three out of seven analyzed answers. This may reveal that developers are more concerned about the usability of their APIs, despite the possible costs caused by breaking changes.

RQ6. Are developers aware of the impact of breaking changes on client applications? Our study shows that *all* developers are aware of the risks attached to breaking contracts with clients. However, in most of the cases, they highlighted the adoption of alternatives to mitigate them. This result suggests that developers have conscious about the costs for client applications but rather than planning changes and deprecating elements, they prefer adopting strategies to alleviate their side-effects.

4.5 Threats to Validity

The results presented in this study provide initial insights on the reasons why developers break APIs and whether they are aware of the consequences for client applications. Despite a response rate of 28.6%, only seven answers were considered for analysis, which impacts the generalization of our results. However, the studied libraries are representative especially due to their popularity (*i.e.,* at least 857 stars), and high number of breaking changes (*i.e.,* more than 50 ones). Another threat is related to the manual inspection of the answers to provide the reasons for breaking changes. Although this activity has been done with special attention and support of both advisor and co-advisor, its subjective nature may bias the presented results. Finally, against our belief, the trustworthiness of the responses may also be a threat to be reported.

4.6 Final Remarks

In this chapter, we presented the second study of this dissertation: a survey with the major contributors of popular libraries hosted on GitHub about real instances of API breaking changes collected in their repositories. Two research questions were investigated in order to elicit a list of reasons that motivate such changes, and to verify their consciousness on the risks for client applications. Specifically, we performed a qualitative study in order to investigate: (i) the reasons why developers implement breaking changes, and (ii) whether they are aware about the risks of these changes.

As a result, we proposed a list of *five* reasons that motivate developers to break API contracts, including: LIBRARY SIMPLIFICATION, REFACTORING, BUG FIX, DEPENDENCY CHANGES, and PROJECT POLICY. Moreover, we showed that developers are usually aware of the impact on clients and, in some cases, adopt strategies to alleviate them. In the next chapter, we discuss the state of the art, presenting the related work, and highlighting their major limitations which motivated this dissertation.

Chapter 5

Related Work

API evolution and stability have been largely studied in the literature. Many approaches were proposed to support this activity and reduce its costs for client applications. In this chapter, we present related work, providing the state of the art, and discussing their major limitations which motivated this dissertation. To accomplish that, we separate these studies into two related subjects: Library Evolution (Section 5.1), and Breaking Changes Impact (Section 5.2). Then, we conclude the chapter with final remarks (Section 5.3).

5.1 Library Evolution

Library evolution and backward compatibility is a major concern for both library developers and client applications. There are several studies that focus on this area, analyzing library changes and measuring their frequency during their life cycle. As an initial effort to understand and measure this phenomenon, Dig and Johnson [2006] studied a dataset of five known Java libraries, observing their changes and classifying them according to their possible effect on client applications: *breaking changes* and *non-breaking changes* (see the definitions in Section 2.2). Besides, with the purpose of identifying the reasons of breaking changes, they elicited a catalog of 24 corresponding breaking operations (*e.g., Moved Method*, *Deleted Method*, and *Extra Argument*), and manually verified them in the change logs and release notes of their libraries dataset. As a result, the authors found that 80% of all API changes are refactorings. In this dissertation, we applied this classification of changes and we also identified REFACTORING as a motivation for breaking API contracts. However, we also discovered other four reasons for breaking changes: LIBRARY SIMPLIFICATION, BUG FIX, DEPENDENCY CHANGES, and PROJECT POLICY. Indeed, the most common reason identified in our study (LIBRARY SIMPLIFICATION) is not discussed by Dig and Johnson. Additionally, we performed a larger study, with a higher number of libraries, taking into consideration the real impact of these changes on client applications.

In a recent work, Bogart et al. [2016] performed a general-purpose case study to understand how, when, and by whom changes are applied in three important software ecosystems: Eclipse,[1] R/CRAN,[2] and Node.js/npm.[3] They contacted 28 experienced developers, and conducted semistructured recorded interviews in phone calls that lasted 30–60 minutes. The authors structured their questions in a way that the respondents discussed their roles as library developers and third-party clients. As a result, they report the differences in practices, polices, and tools applied when performing or avoiding breaking changes. They conclude that in Eclipse, developers usually do not break APIs; in R/CRAN and Node.js/npm, they adopt strategies to deal with breaking changes: in the first, they reach out affected clients with documentation, and in the second, they simply increase the major version number. Additionally, the authors also argue that community values take an important role in this process, helping both clients and developers to understand and solve conflicts about design decisions and breaking changes. Although this is also a qualitative study, we observe that the conclusions stated by the authors do not reflect developers explanations about concrete breaking changes. Instead, they reflect general perceptions and views about breaking changes in the considered software ecosystems. By contrast, we based our analysis on a real set of breaking changes collected in a larger dataset of libraries, considering developers considerations about them.

There are also studies in the context of API deprecation. Raemaekers et al. [2014] investigated deprecation annotations usage when studying the frequency of breaking changes on major, minor, and patch API versions. The authors observe that methods are commonly deleted without applying deprecation annotations, and also that methods with such annotations are never deleted. In a recent study, Brito et al. [2016] measured the usage of deprecation messages at a large-scale level. They point that 64% of the studied API elements (types, fields, and methods) are deprecated with replacement messages, and that this percentage does not increase over time. As a conclusion, the authors provide insights for the design of a tool to support client developers by recommending missing deprecation messages.

To support client applications migrating between library versions, some tools and techniques are proposed in the literature. For example, Kim et al. [2007] present a solu-

[1] `https://eclipse.org`
[2] `https://cran.r-project.org`
[3] `https://www.npmjs.com`

tion to automatically infer rules from structural changes, computed from modifications on method signatures. Kim and Notkin [2009] propose LSDiff, a tool to compute differences between two library versions. Nguyen et al. [2010] propose LibSync, a tool that uses graph-based techniques to support developers migrating between library versions. Henkel and Diwan [2005] present CatchUp, an approach to capture and replay refactoring. Hora and Valente [2015] present apiwave, an approach to keep track of API evolution by mining import statement changes.

Finally, Dagenais and Robillard [2008] present an approach that recommends API replacements based on library changes. In contrast, Schäfer et al. [2008] propose to mine API usage change rules based on client changes. In the same context, Wu et al. [2010] present an approach to infer evolution rules based on call dependency as well as on text similarity analyses. Meng et al. [2012] propose a history-based matching approach to support library evolution. In all these works the authors are concerned in providing ways to help clients to deal with breaking changes. However, in none they investigate the real extension of this problem. In this dissertation, we advanced the knowledge on library evolution by providing two empirical studies on API breaking changes, and discussing their results to provide insights on tools and techniques that may support library developers and clients on these evolutionary activities.

5.2 Breaking Changes Impact

Besides analyzing the evolution of libraries, and measuring the frequency of occurrence of breaking and non-breaking changes, there are many studies that also take into consideration the impact of such changes on client applications. In fact, they go deeper in their analysis by observing the usage of API elements (*i.e.,* the more used an element is, the more critical would it be a modification that breaks its contract).

In this context, Raemaekers et al. [2012] evaluated the stability of frequently used APIs in terms of four defined metrics based on method removals, implementation change, the ratio of changes in old methods to changes in new ones, and the percentage of new methods. Therefore, the authors defined the following metrics: *WRM* (number of removed methods weighted by usage frequency and age), *CEM* (changes observed in existing methods), *RCNO* (percentage of changes in new to old methods), and *PNM* (ratio of new methods observed). Next, the authors extracted their metrics by performing a historical analysis of stability and impact on 140 clients of the Apache Commons Library. They focused on the clients history, observing the usage frequency and updates in their Maven build files. As a result, they discuss three major scenarios on which the metrics would be useful for software developers and project managers: (i) deciding about depending on a certain library, (ii) deciding whether to encapsulate or not the dependencies on a project, and (iii) determining the state of maintenance of a library. However, the real size of the phenomenon, and the impact of such instability on client applications were questions left open. In this work, we study the evolution of a larger set of libraries and compute the impact of breaking changes in an ultra-large dataset of client applications. Additionally, we identify a set of characteristics related to development and social coding that are associated with API breaking changes.

Another important work was performed by Jezek et al. [2015] in the context of analyzing binary compatibility of OSGi-based systems.[4] Nowadays, these are representative systems due to the increasing necessity of high availability ("24/7 systems") and the consequent necessity of features to swap libraries and components at runtime. In this process, some incompatibilities may arise due to subtle differences between Java compilers and the Java Virtual Machine–JVM (*i.e.,* some API changes that may be backward compatible according to the rules used by the compiler, but incompatible from the JVM's point of view). Therefore, the authors defined a set of specific changes that may lead to unexpected runtime behavior during "hot upgrades", dividing them in three main groups: (i) binary and source code incompatible changes; (ii) binary incompatible and source code compatible changes; and (iii) constant inlining. To evaluate these changes, and advance the understanding about API evolution, they used a dataset of 109 Java programs and 564 program versions obtained in the Qualita Corpus [Tempero et al., 2010]. As a result, the authors observed that API instability is a common phenomenon, and also that only in a few cases it affects clients. However, the study bases their conclusions on a small and specific set of libraries and client applications, focusing on the analysis of particular changes in this kind of systems. In this work, besides the larger dataset of libraries and client applications, we focus on source code compatibility, analyzing syntactic changes on API code (which represents the majority of breaking change scenarios).

In the Android context, McDonnell et al. [2013] investigate API stability and adoption. The authors state that Android APIs evolve faster than client migration. Linares-Vásquez et al. [2014] analyze how the number of questions in Stack Overflow increases when APIs are changed. They show that Android developers are more active when they face API modifications.

Finally, Robbes et al. [2012] investigate the impact of deprecation in a Smalltalk ecosystem. They find that some API deprecation have large impact on client applica-

[4]`http://www.osgi.org`

tions and that deprecation messages usually have low quality. In a recent work, Sawant et al. [2016] perform a partial replication of this study in Smalltalk, but analyzing Java APIs hosted on GitHub. They compare and contrast the results of both studies, providing insights on update practices and similarities in reaction behavior between both languages. Still in the Smalltalk ecosystem, Hora et al. [2015] study the impact of API replacement and improvement messages. The results show that a large amount of clients are affected by API changes but most of them do not react.

5.3 Final Remarks

In this chapter, we discussed the state of the art in the subjects related to this dissertation: Library Evolution and Breaking Changes Impact. We briefly analyzed some of these related work, and highlighted their limitations, which motivated the development of our studies. To the best of our knowledge, this dissertation is the largest empirical study investigating API breaking changes and their impact on client applications. It also reveals development and social coding characteristics that impact on the frequency of breaking changes. Moreover, this is the first study investigating the motivations behind API breaking changes based on the actual explanations of developers on specific breaking changes they have recently applied.

In the next chapter, we present our conclusions, discussing the major contributions of this dissertation and analyzing the implications of our results. In addition, we also prospect future work on API breaking changes.

Chapter 6

Conclusion

In this chapter we present our conclusions for this master dissertation. First, we begin by providing a summary of our two empirical studies and their results, as well as discussing our main contributions (Section 6.1). Then, we conclude by prospecting future work on API *breaking changes* (Section 6.2).

6.1 Summary and Contributions

In this dissertation, we investigated API *breaking changes* at a large-scale level. To accomplish that, we proposed and implemented an APIDiff tool, which purpose is to analyze two versions of a Java library and identify both *breaking* and *non-breaking changes* between them. A catalog of 21 modifications was defined and implemented, including 12 breaking and 9 non-breaking changes. We used this tool to empirically study (i) the frequency and the impact of API breaking changes, and (ii) the motivations that drive API developers to introduce such changes in their libraries. In this section, we summarize each study, highlighting their respective contributions.

Historical and Impact Analysis. In our first study, we measured the amount of breaking changes on real-world libraries and its impact on client applications. For that, we conducted a large-scale study with the top *317* Java libraries, their 9K releases, and 260K possible clients. We selected popular and mature GitHub repositories by filtering characteristics such as *number of stars*, *number of releases*, and *age*. Four research questions were investigated to support the analysis on (i) the frequency of API breaking changes, (ii) the behavior of these changes over time, (iii) the impact on client applications, and (iv) the characteristics of libraries with high frequency of such changes. Therefore, the lessons learned from our results are:

- ***Libraries often break backward compatibility.*** We found that 27.99% of all API changes break backward compatibility. On the median, 14.78% of the changes, per library, are breaking changes.

- ***Breaking changes frequency increase over time.*** Comparing the first and fifth years of the studied libraries, the percentage of breaking changes increases 20% (from 29.02% to 49.14%).

- ***Most breaking changes do not have a massive impact on clients.*** Despite the high number of breaking changes verified, only 2.54% of clients are potentially impacted (on the median). However, this ratio reaches 100% for outlier values.

- ***Development and social coding measures are associated with API breaking changes.*** We found that libraries with high frequency of breaking changes are larger, more popular, and more active.

API Breaking Changes Motivations. In our second investigation, we surveyed the major contributors of libraries with more than *50 breaking changes* observed in our first study. The goal was to (i) elicit the reasons why developers implement breaking changes, and (ii) check whether they are aware about the risks of these changes. From the initial dataset of 317 libraries, 90 registered more than 50 breaking changes, from which we retrieved 49 developers contact. Then, we sent an email for each of them, and received 14 answers (response rate of 28%). *Seven* were selected for analysis. From this study, we learned that:

- ***Library developers break contracts with specific motivations.*** We elicited a list of *five* reasons that motivate developers to break API contracts: LIBRARY SIMPLIFICATION, REFACTORING, BUG FIX, DEPENDENCY CHANGES, and PROJECT POLICY;

- ***Most developers are aware of the risks of such changes.*** We found that developers are usually aware of the impact on clients and, in some cases, adopt strategies to alleviate them.

6.2 Future Work

Based on the previous results, we observe the opportunity of developing techniques to assist both API developers and clients to deal with breaking changes, mitigating their impact. In addition, we also prospect some future work to strengthen the knowledge on the motivations of API *breaking changes*. Therefore, we suggest:

1. Plotting historical curves to analyze library stability. First, we suggest the development of tools that may apply the techniques used in our quantitative study, improving API evolution and generating important feedback in real-world scenarios. In this context, historical analysis (*i.e.,* information retrieved from releases history) may be adopted to measure library stability and pressure developers to avoid compatibility faults. For example, a historical curve may be plotted with the rates of breaking changes along libraries life cycle, revealing how stable they are, and highlighting for developers the necessity of taking especial attention (see Figure 3.12 for an example of three important libraries curves). Also, this analysis would provide useful information for client developers when reasoning whether to depend or not on a library.

2. Using impact analysis to reason about performing API breaking changes. We also observe that techniques applying impact analysis may be helpful for library developers, supporting their decisions before changing highly used APIs. In this case, we may use information provided by JAVALI to calculate the impact of breaking changes in specific API elements. Therefore, before performing a *breaking change*, developers would be able to analyze the extent of their impact (*i.e.,* the amount of affected clients), and then decide whether to perform it or not.

3. Applying firehouse interviews to enlarge our list of API breaking changes motivations. Finally, we suggest an in-depth study based on *firehouse interviews* [Murphy-Hill et al., 2015] with the contributors of popular Java libraries hosted on GitHub. The idea is to replicate a methodology used in a previous study about refactoring motivations [Silva et al., 2016]. During several months, we would monitor a large dataset of libraries, fetching commits from each remote repository to a local copy. Next, we would use our APIDiff tool to iterate through each commit and identify changes that break compatibility. Finally, an email would be sent to the author of the commit asking two main questions:

- Could you describe why did you perform the listed breaking changes?
- Are you aware of the possible impact of them in case they are released to clients?

In this case, our goal is to contact API developers as soon as they introduce a breaking change, while the modification is still fresh. In this way, we would receive more answers, which is important to increase confidence on the initial list of reasons for breaking changes elicited in this dissertation.

Bibliography

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 Symposium on the Foundations of Software Engineering (FSE)*, pages 109--120.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016). 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.

Businge, J., Serebrenik, A., and van den Brand, M. G. (2013). Eclipse API usage: the good and the bad. *Software Quality Journal*, 23(1):107--141.

Couto, C., Pires, P., Valente, M. T., Bigonha, R., and Anquetil, N. (2014). Predicting software defects with causality tests. *Journal of Systems and Software*, 93:24--41.

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. (2006). How do APIs evolve? A story of refactoring. In *22nd International Conference on Software Maintenance (ICSM)*, pages 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.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2015). Boa: ultra-large-scale software repository and source-code mining. *Transactions on Software Engineering and Methodology*, 25(1):1--34.

Grissom, R. and Kim, J. (2005). Effect sizes for research: a broad practical approach. *Lawrence Erlbaum Associates Publishers.*

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., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2014). APIEvolutionMiner: keeping API evolution under control. In *21th Working Conference on Reverse Engineering (WCRE)*, pages 420--424.

Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. (2015). How do developers react to API evolution? The Pharo ecosystem case. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251--260.

Hora, A. and Valente, M. T. (2015). apiwave: keeping track of API popularity and migration. In *31st 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 278--289.

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

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 92--101.

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

Kim, M., Notkin, D., and Grossman, D. (2007). Automatic inference of structural changes for matching across program versions. In *29th International Conference on Software Engineering (ICSE)*, pages 333--343.

Kingsum, C. and Notkin, D. (1996). Semi-automatic update of applications in response to library changes. In *12th International Conference on Software Maintenance (ICSM)*, pages 359--379.

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.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M. D., Oliveto, R., and Poshyvanyk, D. (2013). API change and fault proneness: a threat to the success of Android apps. In *9th International Symposium on the Foundations of Software Engineering (FSE)*, pages 477--487.

Linares-Vásquez, M., Bavota, G., Penta, M. D., Oliveto, R., and Poshyvanyk, D. (2014). How do API changes trigger Stack Overflow discussions? A study on the Android SDK. In *22nd International Conference on Program Comprehension (ICPC)*, pages 83--94.

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.

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. *IEEE Computer*, 29(9):45--51.

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.

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

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.

Raemaekers, S., van Deursen, A., and Visser, J. (2014). Semantic versioning versus breaking changes: A study of the Maven repository. In *14th Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 215--224.

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 1--11.

Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T. (2013). Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613--637.

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.

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.

Tempero, E. D., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas Corpus: a curated collection of java code for empirical studies. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 336--345.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015). What are the characteristics of high-rated apps? A case study on free Android applications. In *31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 301--310.

Tourwé, T. and Mens, T. (2003). Automated support for framework-based software evolution. In *19th International Conference on Software Maintenance (ICSM)*, page 148.

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