UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

ANDERSON SEVERO DE MATOS

SPLITTING APIS: AN EXPLORATORY STUDY OF SOFTWARE UNBUNDLING

FORTALEZA
2019

ANDERSON SEVERO DE MATOS

SPLITTING APIS: AN EXPLORATORY STUDY OF SOFTWARE UNBUNDLING

Dissertação apresentada ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Lincoln Souza Rocha.
Coorientador: Prof. Dr. João Bosco Ferreira Filho.

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M381s Matos, Anderson Severo de.
Splitting APIs : An Exploratory Study of Software Unbundling / Anderson Severo de Matos. – 2019.
55 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2019.
Orientação: Prof. Dr. Lincoln Souza Rocha.
Coorientação: Prof. Dr. João Bosco Ferreira Filho.

1. Desmembramento de software. 2. Modularidade. 3. Uso de APIs. 4. Mineração de repositórios de software. 5. Estudo exploratório. I. Título.

CDD 005

ANDERSON SEVERO DE MATOS

SPLITTING APIS: AN EXPLORATORY STUDY OF SOFTWARE UNBUNDLING

Dissertação apresentada ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestreem Ciências da Computação. Área de concentração: Engenharia de Software.

Aprovada em: ___/___/______.

Aos meus pais, por me ensinar o valor do trabalho. À minha noiva, pelo exemplo e apoio em seguir na carreira acadêmica. À universidade pública, por dar a muitos jovens a chance de brilhar e tornar este país um lugar melhor.

AGRADECIMENTOS

Ao Prof. Dr. Lincoln Souza Rocha por me orientar em minha dissertação de mestrado e em vários aspectos da produção acadêmica e pesquisa na área de Engenharia de Software. Sobretudo, pelo compartilhamento de experiências sobre a união entre ensino, pesquisa e prática.

Ao Prof. Dr. João Bosco Ferreira Filho por fundar a base deste trabalho e gentilmente guiar o seu desenvolvimento, de forma a produzirmos uma contribuição para pesquisas futuras no campo de Software Unbundling.

Ao Prof. Dr. Gustavo Henrique Lima Pinto por nos representar no 16o Mining Software Repositories, realizado em Montreal - CA, garantindo a apresentação e publicação do artigo baseado nessa dissertação.

Aos meus pais, Liduina e Matos, por me dar a liberdade de perseguir o desejo de estudar Ciências da Computação, sem exigir uma direção de formação acadêmica ou profissional. Meus mais sinceros agradecimentos a todo o carinho e dedicação em ensinar a todos na família o valor dos estudos, do trabalho e da empatia.

À minha futura esposa, Eudenia Magalhães, pela compreensão com as minhas ausências durante o curso e a escrita dessa dissertação. Obrigado pelas alegrias, os desafios e o companheirismo em todos os momentos da nossa vida.

A todos os professores do Centro de Ciências com os quais tive a oportunidade de estudar, e ao corpo administrativo do MDCC (Mestrado e Doutorado em Ciências da Computação) pelo suporte em diversas questões no decorrer do curso.

Aos colegas de turma e de profissão, que me acompanharam em atividades e debates fundamentais para o complemento da formação acadêmica.

E à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), instituição que resiste a tempos difícies para a pesquisa e educação superior no país, na pessoa do Presidente Anderson Ribeiro Correia, pelo financiamento da pesquisa de mestrado via bolsa de estudos.

Esta pesquisa foi parcialmente financiada pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código 001, e pelo INES 2.0, FACEPE sob concessão APQ-0399-1.03/17, e CNPq sob concessão 465614/2014-0.

“Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.”

(Antoine de Saint-Exupery)

RESUMO

O desmembramento de software consiste em dividir um artefato de software existente em outros menores. Desmembramento pode ser útil para remover código em desuso do software, para separar funcionalidades que podem não estar compartilhando o mesmo propósito da aplicação como um todo, ou simplesmente para isolar uma funcionalidade emergente que merece ser uma aplicação por conta própria. Esse fenômeno é frequente em aplicativos para dispositivos móveis e também está se propagando para APIs. Esta pesquisa propõe um primeiro estudo empírico sobre o desmembramento para entender seus efeitos em APIs populares. Exploramos as possibilidades de dividir bibliotecas em pacotes de 2 ou mais blocos baseados no uso que os projetos de clientes fazem deles. Nós mineramos mais de 71.000 projetos clientes de 10 APIs de código aberto e geramos automaticamente 2.090 sub-APIs para então estudar suas propriedades. Descobrimos que é possível desmemebrar APIs automaticamente em bundles menores a partir de conjuntos de uso disitintos formados por grupos de clientes; os bundles de código gerados podem variar em termos de representatividade e singularidade, o que é analisado minuciosamente neste estudo.

Palavras-chave: desmembramento de software; modularidade; uso de APIs; mineração de repositórios de software; estudo exploratório.

ABSTRACT

Software unbundling consists of dividing an existing software artifact into smaller ones. Unbundling can be useful for removing clutter from an application or separating different features that may not share the same purpose, or simply for isolating an emergent functionality that merits to be an application on its own. This phenomenon is frequent with mobile apps and it is also propagating to APIs. This research proposes a first empirical study on unbundling to understand its effects on popular APIs. The study explores the possibilities of splitting libraries into 2 or more bundles based on the use that their client projects make of them. I mine over than 71,000 client projects of 10 open source APIs and automatically generate 2,090 sub-APIs to then study their properties. Results show that it is possible to have sets of different ways of using a given API and to unbundle it accordingly; the bundles can vary their representativeness and uniqueness, which is analyzed thoroughly in this study.

Keywords: software unbundling; modularity; API usage; mining software repositories; exploratory study.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Software tends to evolve, often extending existing features or offering new ones. This evolution is therefore important to maintain the users' acceptance of the software. However, it can also lead to an uncontrolled growth. As the software absorbs new features, it may lose focus of its original purpose or simply may incorporate unnecessary code and interface features, which in one way yields to lack of usability (*feature fatigue* (THOMPSON *et al.*, 2005)), and from the point of view of development, it can negatively affect maintainability (ANDA, 2007; SZŐKE *et al.*, 2017).

Recently, it has been observed that mobile apps go through an evolutionary phenomenon of unbundling (FILHO *et al.*, 2016). Unbundling consists of dividing a given software into two or more bundles. This happens for reasons such as: there is an emergent functionality that merits to be a software on its own; there is a collection of features that are often used separately; or there is a need for simplifying user experience, easing maintainability and removing clutter.

For decades, software engineers have been studying ways to compose software artifacts into more capable and complex ones; on the other hand, decomposing is also becoming important as software reaches a maturity level in which many applications, systems and APIs are too large to be efficiently maintained and used. However, dividing a software artifact is a challenging task from many perspectives. Conceptually, unbundling goes beyond modularizing, aspectualizing or simply componentizing a software, as the main goal of these approaches is to enhance non-functional properties of a program, such as flexibility, comprehensibility and maintainability. (MORTENSEN, 2009; MORTENSEN *et al.*, 2008; REBÊLO *et al.*, 2014; WASHIZAKI; FUKAZAWA, 2005; AJILA *et al.*, 2013); unbundling can benefit from all these good properties, but for the goal of simply dividing and having 2 or more different applications (sometimes only to cope with market trends), serving to different clients, maintained by different teams and used by different users (FILHO *et al.*, 2016). Commercially, changing a software risks its user acceptance, therefore its popularity (e.g., downloads, sales). From the engineering perspective, how can we efficiently identify, extract and isolate in a different program a given number of features that may be coupled to others? In other words, **should we and how can we unbundle software?**

In this study, I explore the unbundling of APIs. I analyze how APIs are used by their client projects and propose to split these APIs according to the different ways that groups of projects use them structurally – unbundling based on usage. Open source APIs/libraries (e.g., Apache Commons IO, JUnit, and Google Guava) are used almost ubiquitously across a large span of other open source and industrial applications (LEITNER; BEZEMER, 2017), so this scenario favours large-scale studies and makes APIs the best fit for analyzing software and its usage by the community.

First, I selected 10 APIs and mined projects on GitHub that use each API. We collected over 71,000 client projects of those 10 APIs and we identified which classes of each API they are using. Second, for a given API, we tried to discover clusters of usages, each cluster being a group of client projects that use the API in a similar fashion (i.e., use the same classes from the subject API). Once we identify those clusters, we get the classes that they use and try to automatically separate them from the original API, together with all their dependencies. By following this protocol and exploring the granularity of the unbundling (we divide each API from 2 to 20 bundles), we generated 2,090 sub-APIs and then studied some properties, such as their uniqueness and representativeness in regards to the other generated sub-APIs (i.e., bundles) and the client projects they attend, respectively.

Our study allows to understand how usage can drive unbundling in APIs, identifying if a given API may have different groups of clients with respect to the usage and allowing for an automatic division. Our exploration on the number of bundles also gives an idea about how granular the division should be. For the sake of simplicity, we have used the words API and library interchangeably in this research.

The remainder of this research is structured as follows. Chapter 2 gives a background about how software evolves, how does unbundling occur in real world applications and focuses on unbundling based on usage and its main concepts, such as uniqueness and representativeness of bundles. Chapter 3 describes our empirical exploratory study and the methodology that supports it, along with a deep dive into the study protocol and the bundles clustering process. Chapter 4 presents results from the empirical study visiting all research questions and their main outcomes. We then discuss the results in details throughout the Chapter 4, and touch the threats to validity of this research in Chapter 5. Chapter 6 consists of a comprehensive examination of related work. Finally, we present our conclusions and main ideas for further investigations in Chapter 7.

2 THEORETICAL BACKGROUND

In this Chapter, I discuss the software engineering concepts that supports this unbundling research, as well as the narrowed vision on unbundling based on usage. Through an overview of each component that composes the unbundling process and its outcomes, this contextualization prepares the field for more technical details over the APIs selection, client mining and source code splitting we drill down through the next chapters.

2.1 Software Evolution

There is not a widely accepted definition for Software Evolution (BENNETT; RAJLICH, 2000), but to understand it and how it relates to this study we can start from one of its simplest concepts, which is the tendency of software to change over time (PENNY *et al.*, 2003).

For many years authors have been discussing the difference between software evolution and maintenance, given that both address changes in the software; some people even argue that both terms can be used interchangeably. However, let's draw a line between the terms for the sake of clarity of their usage in this research.

Following (GODFREY; GERMAN, 2008) perspective', for software maintenance I will assume all the changes applied in a production version of the software in order to keep it running according to the original requirements, be it a series of bug fixes or performance adjustments. On the other hand, we will take software evolution as the changes in the software to reach new features that were not prospected by the time of its conception or to serve new users demands over time.

Lehman, Belady and colleagues (BELADY; LEHMAN, 1976; LEHMAN *et al.*, 1997) observed large-scale proprietary software evolution behaviour for over 20 years, proposing and updating what they call laws of evolution. The eight laws regard many aspects of software evolution, from which I can highlight the software nature of continuously increase in size and complexity, to change in order to meet users needs, and to decline in quality unless the proper design revision is not performed through its lifetime.

The concept of reengineering, which includes both reverse and forward engineering activities (TRIPATHY; NAIK, 2014), also captures the inner idea behind software evolution. In the reverse engineering phase, one must draw an abstract view of the target piece of software in order to comprehend its architecture and modules. From that picture, an assessment can be performed to indicate a path the forward engineering tasks will follow to produce an improved version of this software. This process is defined by Jacobson and Lindstörm (JACOBSON; LINDSTRÖM, 1991) with the expression below:

$$\textit{Reengineering} = \textit{Reverse engineering} + \Delta + \textit{Forward engineering} \tag{2.1}$$

This study moves towards the discovery of metrics that support software evolution, in such way one can run the unbundling process here proposed and understand to what extent the target software is prone to be unbundled, as well as the feasibility of this process. This approach matches Jacobson and Lindstörm proposition described in the equation 2.1.

2.2 Software Unbundling

The software unbundling, sometimes called the unbundling phenomenon, might be seen as a branch of software evolution studies, and consists on dividing software artifacts in smaller pieces. It does not happen only for the sake of modularization, but rather motivated by the presence of emergent functionalities in the software or due to business decisions, such as a company/department split.

According to Filho *et al.* (2015), a software project qualifies to be unbundled only when it is a mature product, i.e., it has passed its conceptual phase and consists in a dense piece of software that conveys a well defined class of features. The other characteristic is that this software already has its client corpus, so applying changes into it might have serious consequences to those clients, which in turn can leverage adoption issues to the software.

Once a unbundling scenario is identified, its execution will face some challenges (FILHO *et al.*, 2016), which permeate subjective decisions and technical maintainability tasks. From the business perspective, unbundling a mature software means changing it to pursue a new purpose or at least giving it space and resources to fully attack an identified niche. This risks the original software overall image and clients trust. Therefore the identification of causes and objectives for the emergent pieces of software must be clear. This then touches the existing user acceptance criteria.

When it comes to software engineering challenges, like decomposing software and administration of code replication, teams can rely on the few decades of knowledge this field has on improving maintainability, understandability, flexibility and testability. These factors act like unbundling facilitators instead of end goals, so the easier it is for a software to be divided, for instance due to its weak coupling parts, the more successful will be its unbundling from a code management point of view.

Notorious real cases of unbundling are the creation of Swarm app[1] as a feature extracted from Foursquare[2] , and the split of Linkedin[3] into half dozen purpose oriented apps. But the phenomenon has been propagated on to API domain as well, like in the Machine Learning field, where MLib[4] emerged from Apache Spark platform and Mahout came out from Apache Hadoop[5]. This research also investigate a change on JUnit, which released 3 sub-projects as their new suite for testing.

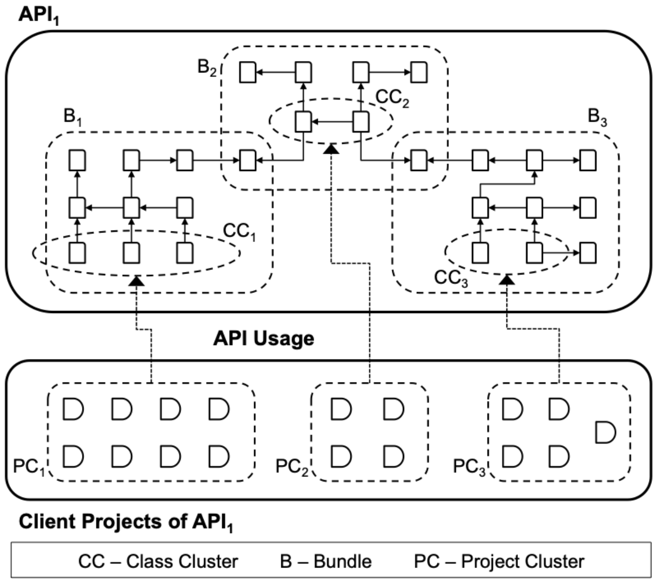
2.3 Unbundling based on Usage

In Fig. 1, we illustrate an API with many client projects using its classes. API_1 has groups of classes that are commonly used together for different reasons, they may have complementary functions, structural dependencies or may just represent a set of unrelated features required by their users. Eventually, as depicted in Fig. 1, it is possible to identify (almost)-distinct usages of the same API, forming clusters of client projects and therefore different purposes of use; PC_1 , PC_2 and PC_3 are clusters of projects that use three different regions of API_1 by calling its classes and interfaces in direct (CC_1, CC_2 and CC_3) and indirect (B_1, B_2 and B_3) ways.

Following, we explain the concepts that will help us guide this exploratory study, our main objective being unbundling an API (i.e., dividing the API into smaller bundles) and studying the properties of the parts resulted from this process. Therefore we need to first understand A) if there are groups of clients of an API that use it in a similar fashion. We will use this information to unbundle the API accordingly. B) When unbundling, we need to seek ways of evaluating if we generate bundles that are different of one another and how so. Also, if these bundles are useful to a set of clients of the original API and how many. C) We need to understand and formalize an algorithm for the unbundling process.

1 https://www.swarmapp.com/
2 https://foursquare.com/
3 https://www.linkedin.com/
4 https://spark.apache.org/mllib/
5 https://hadoop.apache.org/

Figure 1 – API usage by client projects.



Source: Produced by the author.

2.3.1 Similarity based on API Usage

We define the following concepts that allow us to express how similar one client project is to another based on their usage of a given API.

2.3.1.1 API usage

Let's define what is usage in the context of this research. We say a project P_1 uses API_1 if there exists at least one reference from P_1 to some class in API_1 , being a reference an invocation from any class in P_1 to any code in API_1. Here, I am most interested in the granularity level of classes; therefore, the usage of API_1 by a project P_1, named $U(API_1 , P_1)$ is the set of classes of API_1 that P_1 imports.

2.3.1.2 Clusters

Let's define a cluster as a simply subset of API source code, but it the context of this research we have certain rules for building this cluster. When we select a target API and at least one of its client projects', a cluster of this API can be built based on the usage this client or group of clients make of it. By collecting only the classes and interfaces which are directly demanded by the API client (or group of clients), without taking in consideration any compilation constraints, we a cluster of this API that, theoretically, serves well the client usage that dictated its shape.

2.3.1.3 Similarity of Usages

Defining how similar is the usage that a project makes of an API to another project can be complicated in some cases. The easiest situation is when their usage is equal, then we can state that the similarity *s* = 1. I expect s to be 0 if two projects do not share any references to the same classes of an API. Meanwhile, I want to have a ratio between the number of intersecting usages and the total of usages of the two projects. A reasonable similarity measure with the aforementioned characteristics is the Jaccard index (JACCARD, 1908), which denotes the intersection over the union of two sets. Therefore we define the similarity of usage *s* (2.2) between a project P_a and a project P_b as being:

$$s=\frac{U\left(\mathrm{API}_n,\mathrm{P}_a\right)\cap U\left(\mathrm{API}_n,\mathrm{P}_b\right)}{U\left(\mathrm{API}_n,\mathrm{P}_a\right)\cup U\left(\mathrm{API}_n,\mathrm{P}_b\right)} \tag{2.2}$$

2.3.1.4 Bundles

Having a cluster of classes CC_1 from a given API_1, composed only of classes and interfaces that are directly used by a set of client projects of this API, we name bundle the union of CC_1 and the complementary group of class and interfaces on which CC_1 depends (identified as B_1, B_2 and B_3 in Fig.1). This definition guarantees that a client project can compile and run with one or more bundles from API_1, instead of the whole API, but on the other hand might lead to the definition of bundles virtually equal to each other.

Our concept of bundle is aligned with Parnas' analysis (PARNAS, 2018) of Dijkstra's paper "The structure of the "THE" multiprogramming system" (DIJKSTRA, 1968), where the use structure is defined as a relation in which for a program *A* that depends on program *B* to work, the specification of *B* must be satisfied. In our case, *B* is a bundle of an API and *A* is a client project of this API.

2.3.2 Similarity based on API Usage

To better evaluate the quality of the bundles taken from an API, I define two metrics related to the coverage these bundles offer over client projects of an API and the relationship of intersection among other bundles from the same API, named: representativeness and uniqueness.

Representativeness is the percentage of clients a bundle would serve. In the search for an optimal quantity of bundles to divide an API, a bundle that covers the needs of a large number of clients would emerge as a good option, meaning that there exists a smaller set of classes from the API that can support those clients. For a given API_n , the *i* bundle representativeness R_{Bi} (2.3) is the number of clients covered (C_c) by a given bundle B_i divided by the total of (covered) clients *(C_c (API_n))* its API has.

$$R_{\mathrm{Bi}}=\frac{C_c(B_i)}{C_c(\mathrm{API}_n)} \tag{2.3}$$

The uniqueness of a bundle B_n, U_{Bn} , is the bundle size, divided by the size of the intersection among all bundles from the same split(2.3). In this study, I vary the number of bundles from 2 to 20. This metric, which is equivalent to the inverse of the similarity, tells us how unique a bundle is with respect to the other bundles generated after a given API split. By looking at uniqueness someone is able to group similar bundles and study the odd ones, in other words, those that do not share much code with their siblings, therefore serving to specialized group of clients. In the unbundling perspective, understanding the small usage niches might also turn out to reveal an important aspect of how clients consume a given API. To decide if these unique bundles are good choices for API splitting, I should also consider other metrics, such as its representativeness.

$$U_{\mathrm{Bi}}=\frac{size(B_i)}{size(B_1 \cap B_2 \cap \ldots \cap B_i \cap \ldots \cap B_n)} \tag{2.4}$$

Ideally, I am interested in bundles that have high representativeness and are (almost-)unique among the possible bundles.

2.3.3 Unbundling Algorithm

As part of this study, we should define a reproducible process for unbundling an API. Given one API to be unbundled, our algorithm (Fig. 2) receives as input the set of all classes and interfaces (*S*) of the API version under consideration; a set (*C*) initially composed of all classes and interfaces of a cluster, and a set of classes and interfaces (*B*) belonging to the derived sub-API (i.e., a bundle).

The set B is initially empty. Each element (class or interface) in the set *C* is added to the set *B* along with its dependencies. Those dependencies may induce other dependencies of their own, which will also compose the set *B*. This process guarantees a given bundle is compilable and therefore might be a useful part of the API for those clients who only need the main set of classes and interfaces that originated the bundle.

Figure 2 – Unbundling algorithm.

```
//$\mathscr{S}$ is a given set of all API classes and interfaces
//$\mathscr{C}$ is a given set of classes and interfaces such that $\mathscr{S} \supseteq \mathscr{C}$
//$\mathscr{B}$ is a set of bundle classes and interfaces, initially empty, such that $\mathscr{S} \supseteq \mathscr{B}$
Function UNBUNDLING($\mathscr{S}$,$\mathscr{C}$,$\mathscr{B}$):
    forall $c \in \mathscr{C}$ do
        $\mathscr{B} = \mathscr{B} \cup \{c\}$
        forall $b \in$ DEPENDENCIES($c, \mathscr{S}$) do
            if $b \notin \mathscr{B}$ then
                $\mathscr{B} = \mathscr{B} \cup$ UNBUNDLING($\mathscr{S}, \{b\}, \mathscr{B}$)
            end
        end
    end
    return $\mathscr{B}$

//c is a given class or interface
//$\mathscr{C}$ is a given set of classes and interfaces
Function DEPENDENCIES($c$,$\mathscr{C}$):
    return $\{x \mid x \in \mathscr{C} \wedge c$ requires, needs or depends on $x\}$
```

Source: Produced by the author.

In other words, the algorithm initiates from a cluster, which is equivalent to the usage of a set of clients, and recursively includes all of its dependencies, until the cycle is closed. It is very important to understand the unbundling algorithm depends on the clustering process, and it’s only responsible for building viable alternative sub-APIs for at least the same group of clients a cluster should support. From the result bundles of each cluster, we may proceed into the analysis of their properties and came up with the best fit for API split.

3 EMPIRICAL STUDY

In this chapter, I explore unbundling a number of popular APIs following an empirical method, defining research questions, a protocol, experiment variables and selecting subject APIs. Applying the unbundling algorithm, we slice an APIs into smaller bundles capable of meeting many of its clients' needs. We want to measure some characteristics of these bundles, such as size and similarities to one another, and how many clients they cover. Furthermore, how we should proceed in order to better split a given API to maximize client coverage without producing identical bundles. Producing identical bundles would mean that an API could not be divided to attend different clients and therefore that all clients use the API in the same way structurally.

3.1 Methodology

This research is based on a exploratory design, divided into two phases: a software repository mining process, from which we collect several information of Java open source projects hosted on GitHub, followed by an extensive series of data analysis on the usage these client projects make of well known Java APIs, and experimentation on rearranging these APIs source code into smaller parts.

The underlying goal I strive to achieve by experimenting with API source code splitting is to understand whether client usage is capable of dictating feasible real world scenarios for API division (unbundling), by either promoting sections of the code related to emergent features, isolating core functionalities that majority of clients depend on or even identifying API areas which that not in use to justify its presence in the code base.

3.2 Research Questions

The following questions were chosen to guide the analysis of the unbundling results, as they emerged as natural inquiring about the feasibility and advantages of unbundling an API.

RQ1. *Can we automatically synthesize smaller APIs based on their usage by client projects?*

This research question assesses the applicability of the unbundling process, if it is actually viable to divide well-established APIs of the market. The ability to generate a smaller

API automatically allows development and management teams to evolve their product based on an observable and measurable process. To answer this question, we rely on the API clients' usage to build a new API that contains a set of features used by a group of clients. To be considered successfully built, these new sub-APIs should compile and serve the group of clients it is derived from.

Quick peek: For each studied API, we were able to identify a set of clients that will be fully served by a smaller subset of the original API source code.

RQ1a. *Can we find (almost-)disjoint usages of APIs by client projects?*

To answer this question, we analyze the code structure of the clusters generated from the clients' usage, by looking at the ratio between cluster intersections and union sizes. Both peer-to-peer and group analysis are welcome to help describing the usage silhouette, which in turn should help us understand whether there are client's niches from a particular API. Disjoint usages can be seen as a trace of feasible API unbundling towards an ideal splitting scenario, where each bundle serves a group of clients and those groups share minimum or even no code at all.

Quick peek: Fully disjoint usages were not found, but average cluster uniqueness (from 2 to 20) is higher than 50%.

RQ1b. *Do different usages result in disjoint APIs after unbundling?*

Even if we start from disjoint usages described by API clients, by the moment we include their dependencies to make them compilable and usable bundles, we may find that these final bundles are equal, in comparison to each other or even to the original API, which is not a favorable result. To answer this question, once the bundles are built, we analyze how their code structure compare to the clusters that originated them, and also compare their final composition to their siblings and the original API.

Quick peek: As expected, bundle uniqueness always drops compared to their original clusters.

RQ1c. *How representative are the bundles?*

We want to investigate how useful are the bundles we generate when splitting an API, where by useful we mean how many clients a bundle can serve. We assume this relates mostly to how granular is the division. For instance, if we divide an API into 2 bundles, supposedly these bundles cover near half of client projects each one, however it can also happen that one bundle covers more than 90% and the other one less than 10%. Thus, if this division is more granular, 10 or 20 bundles for example, we study if there are bundles that emerge as very representative, meaning they cover a huge set of clients, even though they are much smaller than the original API.

Quick peek: Bundles tend to inflate in size, depicting high representativeness.

RQ2. *Can we reduce an API but keep it representative to the majority of its client projects?*

This question relates to the fact that an API may contain code that is never actually used by the set of clients. We could consider it similar to dead code, but in this case, this code can still be reached during execution, however, there is no client project that invokes part of the API's code that will run that other region of code. In summary, the original API offers code that was never used by any client, that may or may never be used. Considering this, we want to investigate if we could remove such "nearly"-dead code from these API's but keep it still fully useful for the client projects.

Quick peek: Except for slf4j, all APIs were reduced in size with
no orphan clients afterwards

3.2.1 Subject APIs

We have selected some of the most used APIs available and hosted on GitHub, such as web frameworks, test engines, data parsers and machine learning tools. These projects have at least half a decade of development and are used by numerous public open-source projects also hosted on GitHub. Table 1 lists the 10 APIs analysed in our study. Some of these

APIs are also listed in a recent study about the most popular Java libraries based on GitHub's most popular projects (POIRIER, 2018).

Table 1 – Summary of selected APIs.

Name	Version	#KLoC	#Classes	#Years	#Clients
CommonsIO	2.4	25	103	11	11396
Gson	2.3.1	12	60	10	775
Guava	18.0	128	454	9	12171
Hamcrest	1.3	2	39	10	220
JSoup	1.8.2	15	48	9	2792
JUnit	4.12	17	183	20	19636
Mockito	1.10.19	23	327	10	4894
slf4j	1.7.12	4	27	13	17212
Weka	3.7.12	450	1033	24	543
Xstream	1.4.7	33	318	13	2202

Source: Produced by the author.

The remaining of this section is a brief overview about these APIs, so we can better understand their structure when applying the unbundling process on them. The Apache CommonsIO offers a solid set of I/O (Input/Output) features such as file reading, writing, copy and event monitoring, implementations of filters and comparators, and other Java file and directory utilities. Gson is a Google library for JSON manipulation and parsing. Guava is another library from Google devoted to a variety of utilities like specialized types of collections, including immutables, hashing functions and also I/O. Hamcrest is an original Java library for string matching that was ported to several languages like Ruby, Pyhton and PHP. Hamcrest is adopted in a large set of projects, including JUnit (by the time of 4.12 version release).

JSoup is a HTML parser library capable of manipulate and extract data from HTML content, offering an API that supports DOM, CSS and jQuery operations. JUnit is the de facto API for Java unit test, having dozen of thousands of open-source clients. The library was recently submitted to a major change, from version 4 to 5, where it was divided into 3 new sub projects. Mockito is a mocking framework, which serves as a complement for unit testing by helping developers create mocks, blocks of information that simulates real world data for test execution.

The math/machine learning API we have in the studied group is Weka, produced by the Waikato University, offering a comprehensive set of algorithms for data preparation, classification, regression and clustering. The last API is XStream, which performs a key functionality for desktop apps, but mostly for web applications: the serialization of objects to XML format. In the era of web services, a variety of Java applications rely on the conversion to and from XML format, in order to process both client data and configuration files of themselves or other libraries.

An important note to make clear at this point is which version of each API we took for analysis. Our dataset is based on a snapshot provided by the Boa (DYER *et al.*, 2013) framework, which dates from late 2015. As by this study I do not perform an exact match between the client target version of the API and the one we took for analysis, the version listed on Table 1 corresponds to the very last stable version of them made available before September 2015.

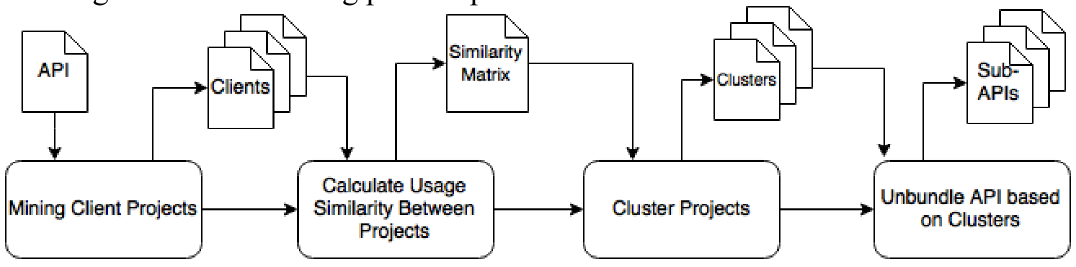
3.2.2 Protocol

Fig. 3 shows the protocol of our study. In the figure, the bottom rounded corner boxes represent the process we apply on the inputs, while the straight shaped upper boxes are an input for a starting process, an output from a finalized process or both at same time.

For each API of the data set, we mine a public open-source dataset of Java projects that use the subject API. Second, we calculate the usage that each project makes of the current API, listing all the classes used in the API code; this allows us to then calculate a similarity matrix of the projects according to the classes they use from the given API. The similarity matrix is detailed defined in the next section (3.2.3).

When the similarity matrix is ready, we cluster the client projects' usages as follow: we take the classes that compose those usages and pass them as seeds to the unbundling algorithm. Finally, the implementation of our unbundling algorithm checks and includes the dependencies of the seeds, creating the bundles, and splits the API accordingly. Those bundles are the last artifact on Fig. 3 process chain, which we call sub-APIs.

Figure 3 – Unbundling process protocol.



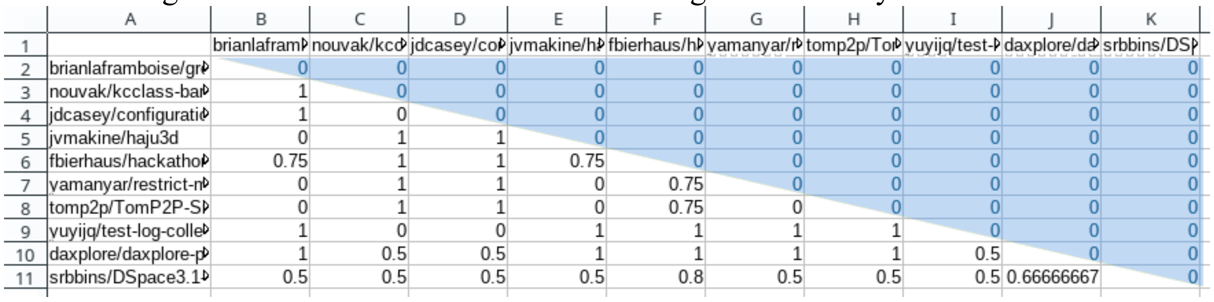
Source: Produced by the author.

3.2.3 Clustering

This study depends on a method to organize clients based on their usage, so then we can build the set of classes (seeds) that cover each usage and proceed with further analysis, such as package distribution of those classes or the relation between final bundle size and initial seeds' set size. The adopted method is hierarchical clustering following an agglomerative approach.

To start, we calculate the Triangular Similarity Matrix for each API, which consists of a triangular matrix filled with a similarity measure, in our case the Jaccard index, for each pair of client projects the API has. Fig. 4 exemplifies how does these matrix look like for our analysis over CommonsIO clients. We then proceed to build a tree structure using Ward hierarchical clustering algorithm (WARD, 1963). Each client is assigned to its own cluster and the algorithm works iteratively joining the two most similar clusters at each step. This process ends up with a unique cluster, which is the root node of the tree, and allows us to cut the result tree in *k* levels, being *k* the tree height.

Figure 4 – Section from CommonsIO Triangular Similarity Matrix.



	A	B	C	D	E	F	G	H	I	J	K
1		brianlafram	nouvak/kcc	jdcasey/co	jvmakine/h	fbierhaus/h	yamanyar/r	tomp2p/To	yuyijq/test-	daxplore/da	srbbins/DS
2	brianlaframboise/gr	0	0	0	0	0	0	0	0	0	0
3	nouvak/kcclass-ba	1	0	0	0	0	0	0	0	0	0
4	jdcasey/configurati	1	0	0	0	0	0	0	0	0	0
5	jvmakine/haju3d	0	1	1	0	0	0	0	0	0	0
6	fbierhaus/hackatho	0.75	1	1	0.75	0	0	0	0	0	0
7	yamanyar/restrict-n	0	1	1	0	0.75	0	0	0	0	0
8	tomp2p/TomP2P-S	0	1	1	0	0.75	0	0	0	0	0
9	yuyijq/test-log-colle	1	0	0	1	1	1	1	0	0	0
10	daxplore/daxplore-p	1	0.5	0.5	1	1	1	1	0.5	0	0
11	srbbins/DSpace3.1	0.5	0.5	0.5	0.5	0.8	0.5	0.5	0.5	0.66666667	0

Source: Produced by the author.

At each level, the tree provides a set of all clients organized in clusters by their similarity. By selecting these clusters at a given level, we can collect the classes their clients depends on, called seeds, include their internal dependencies (classes the seeds depend on) and build the bundles. Those bundles are compilable chunks of the original API, like sub-APIs, that can serve a particular group of clients and, therefore, may represent a viable outcome of the API unbundling process.

3.2.4 Experiment Variables

Table 2 lists all the variables that are used in this experiment and also guides the analysis of the results. *Split Value* is simply the amount of division we apply to an API based on clustering their clients usage. We decided to vary the splitting from 2 to 20, so we can study how different the sub-APIs are to each other when we make them smaller and smaller.

Cluster related variables helps us to understand the client usage and the cluster structure evolution since its automatic generation until the moment it becomes a bundle by the inclusion of all code dependencies. On the other hand, the bundle related variables (*bundle size*, *bundle uniqueness*, and *bundle representativeness*) relate directly to the quality of the unbundling process output. They also can give some hints about API design and how it impacts bundle characteristic's, through the comparison to the values previously acquired for clusters (e.g., how much a bundle has grown in comparison to its original cluster).

The *Number of Clients* served the criteria for selecting APIs at the beginning of the research, but here it composes the bundle analysis along with *Orphan Clients* and *Bundle Representativeness* to asses how the bundles are performing on client coverage.

Table 2 – Experiment variables.

Name	**Type**	**Scale Type**	**Unit**	**Range**	**Description**
Split Value	Controlled	Number	Integer	[2, 20]	Number of parts to divide a given API
Number of Clients	Independent	Number	Integer	[0, ∞]	Quantity of API clients
Cluster Size	Independent	Ratio	%	[0, 100]	Proportion of classes or interfaces imported from an API over the API size
Average Cluster Uniqueness	Dependent	Ratio	%	[0, 100]	Overall difference in structure among a set of clusters generated from the same split
Bundle Size	Independent	Ratio	%	[0, 100]	Proportion of classes or interfaces imported from an API by a set of clients in addition to their dependencies, over the API size
Average Bundle Uniqueness	Dependent	Ratio	%	[0, 100]	Overall difference in structure among a set of bundles generated from the same split
Bundle Representativeness	Dependent	Ratio	%	[0, 100]	Proportion of clients covered by the bundle over the total of API clients
Orhpan Clients	Dependent	Ratio	%	[0, 100]	Quantity of clients uncovered by a bundle or a group of bundles

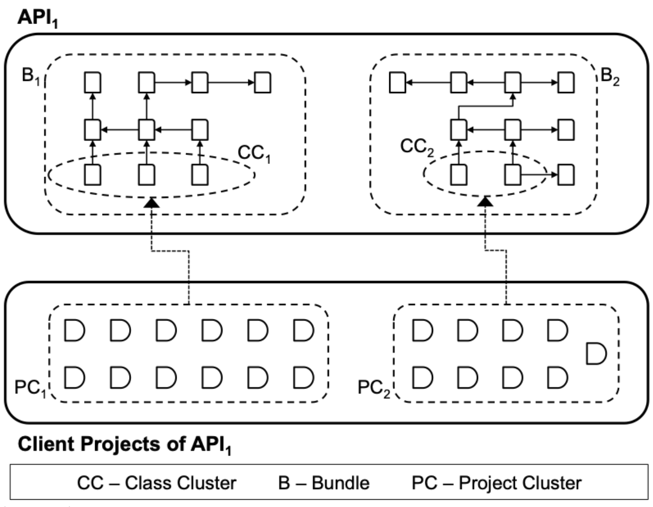
Source: Produced by the author.

4 RESULTS

In this chapter, we present the results obtained after applying our unbundling strategy, from identifying clusters to the actual splitting. The steps to reproduce and the source code of the application are available online[6].

Before showing the results, we illustrate in Fig. 5, anecdotally, what an ideal splitting point for an API would be: a bundle B_1 derived from a cluster CC_1 containing half the API classes and interfaces, covering (almost)-half of the API client projects PC_1, and a second bundle, B_2, derived from a cluster CC_2, containing the complement of the API classes and interfaces covers the remaining client projects, represented by PC_2 . In an API such as API_1, there would be no orphan clients after the unbundling process, and the two generated bundles have no overlapping dependencies.

Figure 5 – Ideal API unbundling scenario.



Source: Produced by the author.

As expected, this ideal scenario is utopical and is not translated into the actual results of this exploratory study. In quantitative terms, an ideal scenario would imply that for each API, we could find a splitting point in which we could create 2 or more clusters with zero intersection and that would generate bundles with also no intersection between them. Therefore, **we aim at reporting how similar/different are the clusters and the bundles we can generate for each API, and how representative they are in terms of usage**. We further discuss the reasons for this differentiation from the ideal to the actual scenario in Section 4.2. This section is devoted to answer the research questions presented in Section 3.

6 https://github.com/severoufc/junbundler

4.1 Research Questions

4.1.1 RQ1. Can we automatically synthesize smaller APIs based on their usage by client projects?

The answer to this question is yes. For each API of the dataset, it was possible to identify, using our unbundling process, a set of clients that will be fully served by a smaller subset of the target API. For instance, dividing the 10 APIs into 2 parts, we were always able to reduce the size of at least one of the sub-APIs (bundles), which is shown in Table 3.

Table 3 – Bundle size and representativeness for split in 2.

API	Bundle 1		Bundle 2	
	Size	**Rep.**	**Size**	**Rep.**
CommonsIO	94.2%	100.0%	5.8%	33.8%
Gson	86.7%	95.3%	96.7%	100.0%
Guava	35.5%	50.7%	92.7%	100.0%
Hamcrest	92.3%	100.0%	5.1%	18.6%
JSoup	91.7%	98.4%	95.8%	100.0%
JUnit	0.5%	20.1%	89.0%	100.0%
Mockito	89.0%	100.0%	77.7%	93.0%
slf4j	7.4%	89.3%	100.0%	100.0%
Weka	33.4%	62.4%	60.1%	100.0%
XStream	73.3%	100.0%	57.2%	76.0%

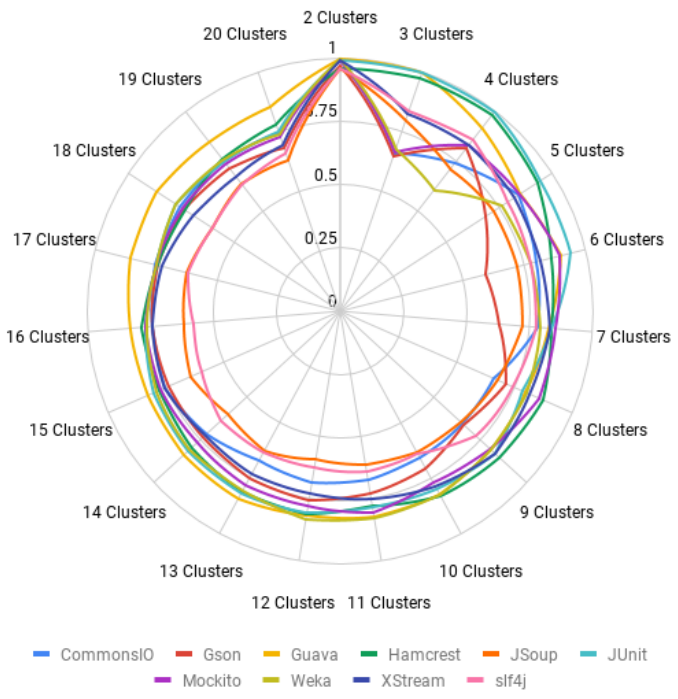
Source: Produced by the author.

4.1.2 RQ1a. Can we find (almost-)disjoint usages of APIs by projects?

We were not able to find a fully-disjoint usage for any studied API, but all of them have at least one split value where the average uniqueness among the usages, clusters, is higher than 95% and all of them show more than 50% of cluster uniqueness through the clustering process (that goes from 2 to 20 clusters). This result, shown in Fig. 6, exposes that almost-disjoint usages are possible to acquire for all the studied APIs. In fact, the average uniqueness for all projects is 75.5%.

Fig. 6 allows us to conclude that it is possible to group clients of APIs according to the usage they make of the API and, most importantly, that these group of clients are reasonably different with respect to the classes they use from the target API. However, as we are not able to ship modified APIs containing only the set of classes described by the usage (their dependencies must be included to make this sub-API usable), in the next subsection, RQ1b, we discuss this same finding under a bundle perspective.

Figure 6 – Average uniqueness among clusters.



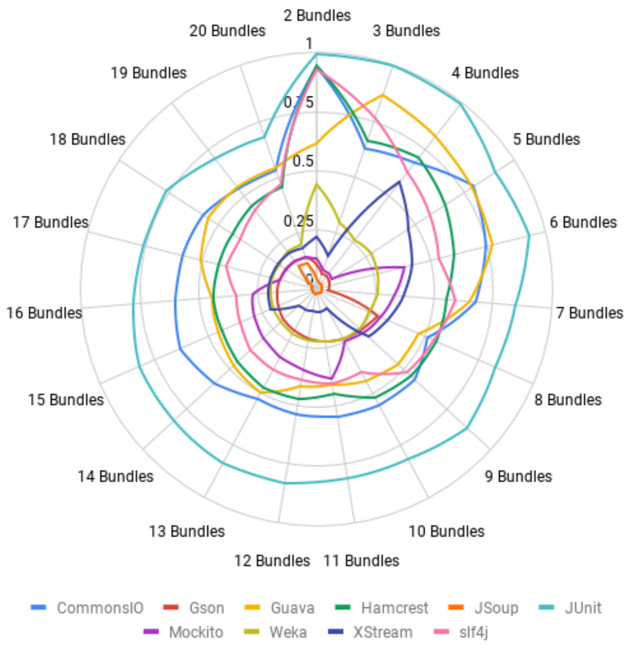
Source: Produced by the author.

4.1.3 RQ1b. Do different usages result in disjoint APIs after unbundling?

In Fig. 7, we plot the average uniqueness among bundles to identify, for each analyzed API, if there is an optimum number of bundles for splitting (*Split Value*). This optimal point yields, on average, the most dissimilar set of bundles among themselves. For example, if we would divide the XStream API with the goal of having the most distinct bundles, the best would be to divide it into four bundles, as their average uniqueness is close to the maximum. On the other hand, Guava may result in more different sub-APIs if split into between 3 to 6 parts. Some other APIs have their uniqueness peek when they get split further, like Mockito, which should be divided into 11 bundles, if we only take uniqueness into account.

If we compare the average uniqueness found through the cluster analysis with the bundle analysis, it is noticeable that the inclusion of dependencies erodes the bundle uniqueness considerably, especially for some APIs like Gson and JSoup. To better visualize the progression from cluster to bundle, we plotted the uniqueness per API, shown in Fig. 8. We can interpret these results as follows. There are APIs that have their classes structured in a way that it is easier to extract bundles according to some usages, while for others, their design does not facilitate unbundling by usage. This is rather realistic as we can assume that an architect may not always divide the classes and their dependencies of a project according to the possible uses that a client will make.

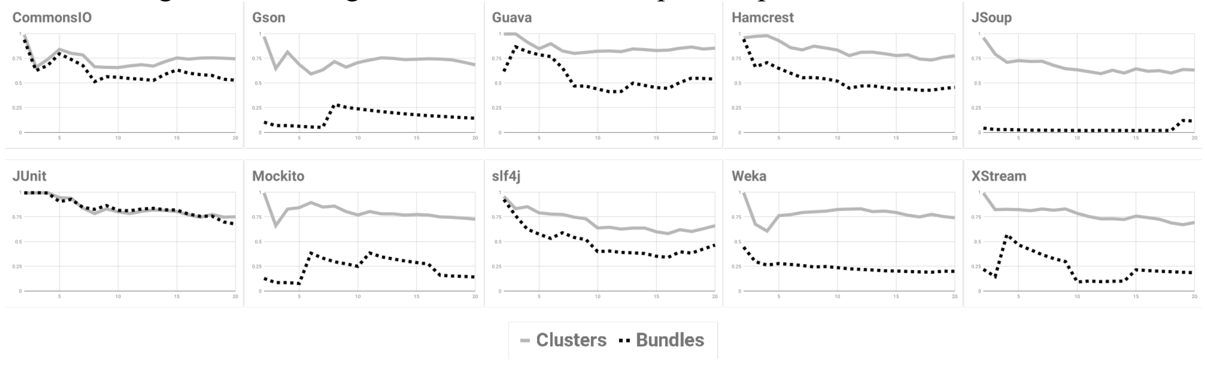
Figure 7 – Average uniqueness among bundles.



Source: Produced by the author.

Thus, these numbers gives us a first insight about the possible granularity of bundles; we go further with a qualitative analysis of each sub-API after we gather the representativeness values for each bundle, which is discussed in the next subsection, **RQ1c**.

Figure 8 – Average cluster vs bundle uniqueness per API.



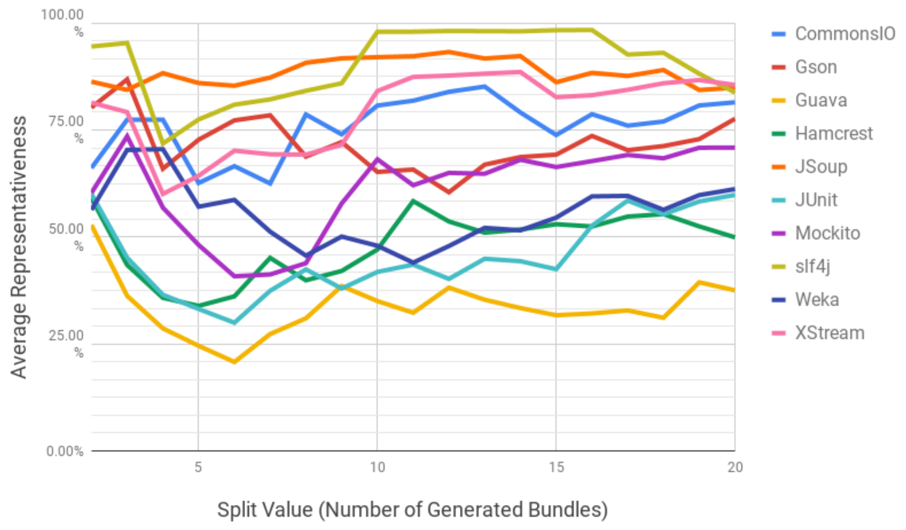
Source: Produced by the author.

4.1.4 RQ1c. How representative are the bundles?

Looking at the big picture, the average representativeness for the set of all bundles generated in this study is 81.21%, which is a high value. But inspecting this value for each API, we see they have their particularities, in which we should rely on to understand the easiness or even the feasibility of unbundling an API.

Fig. 9 shows the average representativeness for all bundles, through the splitting process. Some APIs representativeness' never comes below 70% (e.g., JSoup, slf4j), regardless of how many parts we try to divide it in. Their bundles, which cover more than 3/4 of the clients set, will contain a large number of classes, which implies large intersection areas with their siblings. On the other hand, we should inspect more closely to verify if there are some small bundles in both size and coverage that may fill up the gap for just a specific group of clients. This last scenario diverges from the ideal one we described at the beginning of this section (Fig. 5), but clearly serves well the unbundling goal.

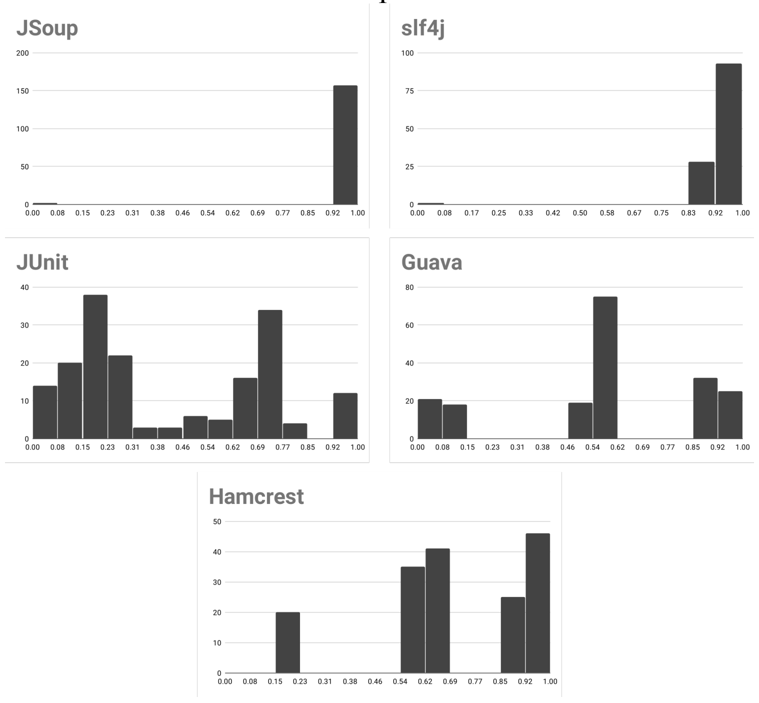
Figure 9 – Average representativeness per API over the splitting process.



Source: Produced by the author.

Inspecting the API bundles we found that the aforementioned expected assumptions were not all true for them. Our findings are summarized in three groups represented in Fig. 10. For the first group, whose representativeness is the highest with JSoup and slf4j, we see the bundles extremely condensed on the right hand of the distribution, meaning that these high average values are isolated cases, but almost all the bundles have their representativeness above 80%. On the other hand, the second group represented by JUnit and Guava reported well-distributed bundles in terms of representativeness, which makes them more prone to offer a set of bundles that complement each other in terms of coverage.

Figure 10 – Number of bundles vs representativeness.



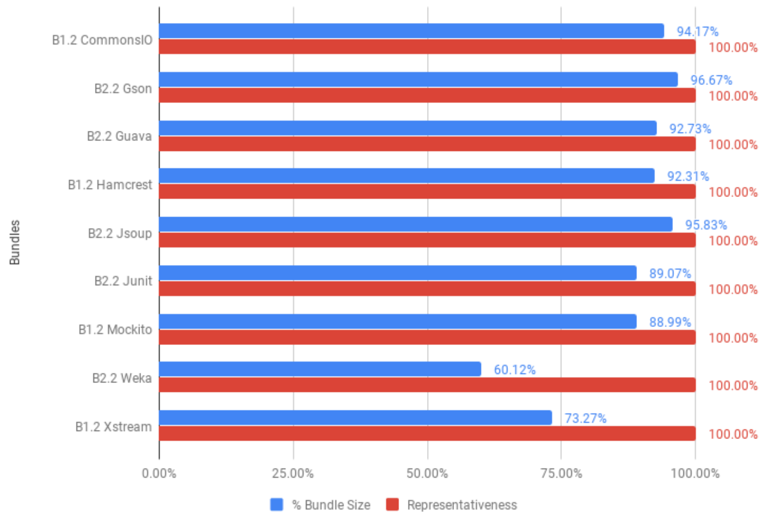
Source: Produced by the author.

To check a representativeness middle case, we studied Hamcrest bundles. Its average representativeness varied only between 48% to 59%. When looking more closely into the Hamcrest bundles, we found a stronger concentration on the upper half of the scale, but also some bundles in the region of 15% to 20%. This helped us understand that the average must not be taken as a good indicator for the representativeness, although higher averages in general indicate unbalanced values for representativeness at the top of the scale.

4.1.5 RQ2. Can we reduce an API but keeping it representative to the majority of its client projects?

After running the unbundling process for all APIs, we successfully generated at least one bundle per split that reaches 100% of representativeness with some reduction in size, when compared to the original API, except for slf4j. This reduction varies between just half a dozen classes to over 30% of the API size, as shown in Fig 10. This result leads us to confirm that it is possible to produce smaller APIs based on their client's usage without compromising the client coverage. But this type of reduction would only be client safe for real-world situations if all clients usage was known by the API development team. In the case of this study, some clients were not taken into account, as described in Chapter 5.

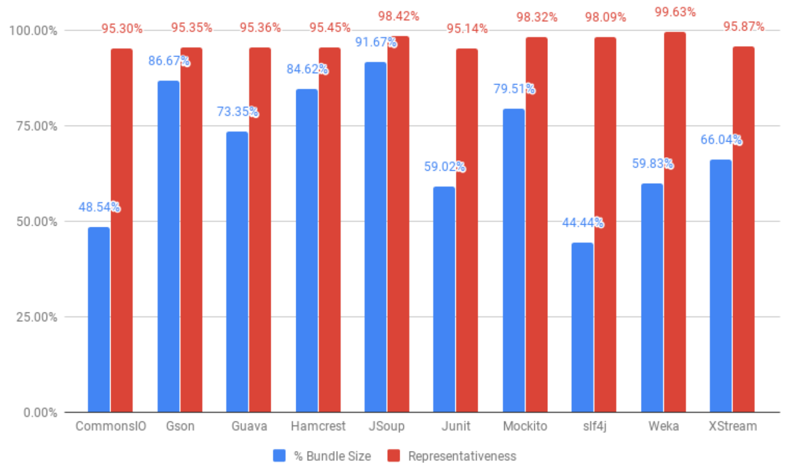
Figure 11 – Reduced bundles sizes with high representativeness.



Source: Produced by the author.

If we set a lower representativeness bar, to 95% coverage, the API reduction results go even further. As shown in Fig. 12, except for JSoup, the unbundling process produced bundles at least 10% smaller than the original API, with some of them, like Commons IO and slf4j, weighing less than half the classes their API have from start. This may derive a whole different study, on the presence and relevance of the source code that serves no client of these APIs.

Figure 12 – Smallest bundles with over 95% of representativeness.



Source: Produced by the author.

4.2 Discussion

This section is devoted to discuss in deeper way the findings from each research question and their implications on this study and future work. To do so, we organised the discussion in groups that cover three keys aspects of this class of problem. Because we are analyzing APIs and the outcomes of their division on smaller parts oriented to client usage, it is fundamental to assess our result in comparison to the ideal API splitting scenario, to understand what is possible preventing us to unbundle these APIs as desired and which types of clients do compose the audience of the studied APIs. We will now address these topics.

4.2.1 On finding completely disjoint bundles

We found some similar results to the ideal split (see Fig. 5), but not exactly the same scenario. The best one in terms of division between bundles was found for JUnit, where the combination of two bundles from the same split, weighting 68, 31% and 7.65% of the original API size, covered 98.13% of clients. Even though almost 2% of clients were not covered, this bundle combination was the closest to the ideal scenario since its bundles are considerably smaller than the original API. This result came from a 20 part split, which is the maximum number of division we apply in the unbundling process.

It was also possible to build a combination of two bundles from Hamcrest weighting 89.74% and 30.77% of the original API size, covering 99.55% of clients. Although this case has better coverage (less than a half percent of uncovered clients), its bundles are bigger than those found for JUnit. For both Commons IO and XStream, the unbundling algorithm was capable of producing a combination of two bundles which covers all the clients, but the bundles' size varies between 62% to 91% of the API size.

Other results are also important as a proof of concept, like the group of 3 bundles for slf4j, and 5 bundles for Weka, which cover more than 99.5% of clients but have a very low uniqueness among themselves. All selected bundle groupings are listed in Table 4 along with their names (which is formed by the letter 'B' followed by the bundle index and the split index), size in comparison to the original API size and client coverage. The 'Final Coverage' corresponds to the combination of coverage from each bundle selected for a given API. Note that every bundle for a API came from the same split.

Table 4 – Selected bundle groups for studied APIs.

API	Bundle 1			Final Coverage
	Name	Size	Coverage	
CommonsIO	B8.20	91.26%	99.70%	100.00%
	B17.20	89.32%	99.92%	
Gson	B5.20	91.67%	97.29%	99.98%
	B15.20	85.00%	98.97%	
Guava	B5.20	73.35%	95.36%	99.57%
	B19.20	87.67%	99.55%	
Hamcrest	B2.20	89.74%	96.82%	99.55%
	B18.20	30.77%	61.36%	
JSoup	-	-	-	-
JUnit	B8.20	68.31%	98.00%	98.13%
	B15.20	7.65%	80.42%	
Mockito	-	-	-	-
slf4j	B2.20	48.15%	98.10%	95.50%
	B14.20	66.67%	99.31%	
	B17.20	55.56%	98.26%	
Weka	B4.20	54.40%	91.16%	99.64%
	B5.20	41.05%	79.37%	
	B6.20	50.15%	90.42%	
	B11.20	49.08%	84.53%	
	B14.20	44.53%	89.13%	
XStream	B2.20	62.58%	96.41%	100.0%
	B12.20	72.33%	99.73%	

Source: Produced by the author.

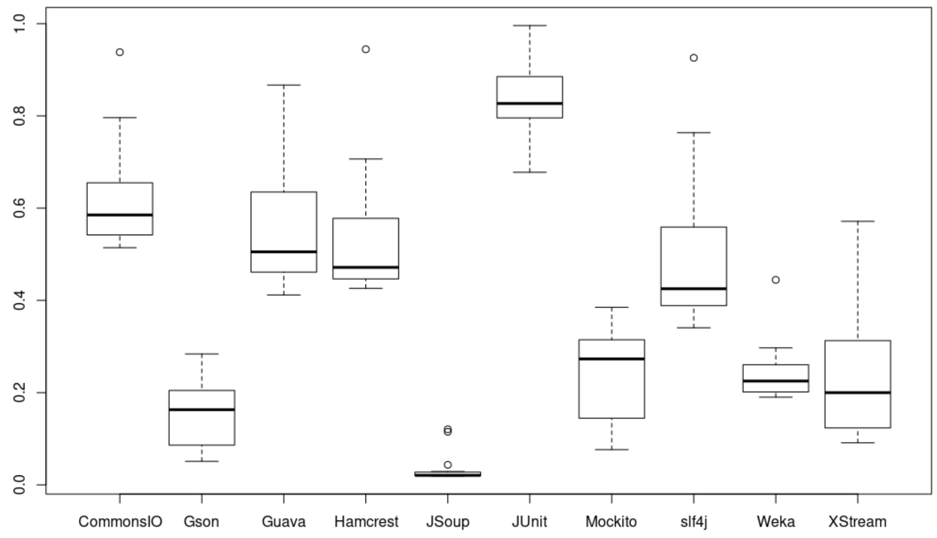
The mandatory aspect in the search of disjoint bundles is the uniqueness. Fig. 13 represents the APIs in terms of average uniqueness. This chart shows the boxplot of average uniqueness between bundles for all the 20 splits of each API. We can highlight two APIs from this picture: JUnit stays up on the chart alone, with the highest median and the third shortest variability, which means it has the better bundle uniqueness ratio. At the bottom, JSoup is also isolated with extremely low variance but also the lowest median and upper uniqueness. This complies with the fact we were not able to join a set of at least 2 distinct bundles to work as new JSoup sub-APIs - its bundles are too big, usually weighing more than 90% of the original API size, which makes them virtually the same.

4.2.2 On API coupling

From the uniqueness boxplot analysis, we decided to investigate why some APIs presented a poor result, like JSoup, or a more split-prone result, such as JUnit. Considering the possible design flaws that would affect coupling, and therefore the unbundling result, we selected the fan-out measure as it tells about dependency; high fan-out values for a class indicates it requires many other classes to accomplish its duties, so it has many dependencies.

From the perspective of software unbundling, if a class with high fan-out is present in a given cluster, this cluster size will inflate by the time its bundle is built. Moreover, this affects the uniqueness because the bigger the bundle, the smaller are its chances to be unique.

Figure 13 – Average uniqueness of bundles per API.



Source: Produced by the author.

After running a fan-out calculation for all classes of the studied APIs, we sorted them and selected only those which weight more than 3 times the standard deviation fan-out. This threshold is intended to help us pay attention only to those classes that are outside the API mean dependency level. Then, we identified which clusters and bundles contained those classes and plotted results, as shown in Fig. 14.

Some of the API configuration drawn in the boxplot was reinforced by this analysis. The most interesting one shows more than 99% of JSoup bundles containing the classes with the highest fan-out in the API. The 3 classes that fit in the criteria represent 18.23% of the total fan-out in the whole API source code. This symptom clarifies why we were not able to unbundle JSoup at all. On the other hand, JUnit has the lowest presence of high fan-out classes, with only 18.58% of the bundles containing some of the 6 classes that fit in the criteria (out of 160 classes). With this distribution of classes, JUnit has emerged as the best API for unbundling in our study.

Other APIs unbundling results also comply with the panorama we have got from the fan-out analysis as well, but they yield more mixed output values, which makes their interpretation fuzzy. For example, slf4j and Mockito have similar profiles of high fan-out classes in their bundles, but they have only a small overlap in terms of uniqueness. We found that Mockito, which has lower uniqueness than slf4j, has a smaller proportion of high fan-out classes, but more than 60% of those classes weighs at least twice as much as the slf4j ones (going up to 4.5 times).

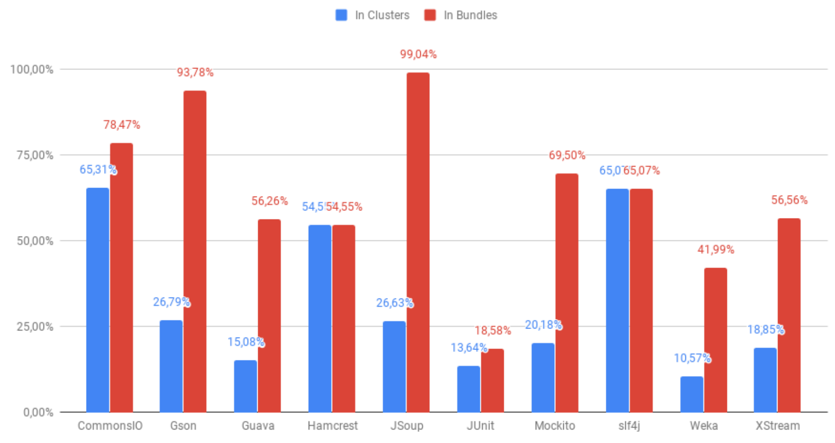
That difference might help to explain why slf4j shows a better uniqueness than Mockito, but the trend does not apply to Hamcrest and Guava, which have virtually the same uniqueness and high fan-out classes distribution in bundles, but a very distinct fan-out weight profile among each other. So we consider that a deeper analysis with the support of other coupling indicators should be applied to highlight small specificities in such cases.

4.2.3 On categories of clients

We were also interested in analyzing the relationship between API features and how clients make use of them. So we ranked the classes from the raw usage mined data to identify which features better represent a given API, or if there is some smaller group of features deserving attention. For most of the studied APIs, client usage relies on root or core packages and presents a considerable difference in volume to the closer packages. This scenario enforces the idea of clients making use of the APIs main classes as an entry point to modularized features.

There are favorable cases to highlight usage categories, such as for Guava, which does not offer a root package or mandatory entry point class, forcing API clients to explicitly reference classes they will use. The List class stands out as the most used, followed by Maps and ImmutableList. Guava clients usage shows that the collections, IO, and concurrent features are the most adopted, whereas hashing and reflection functions are less popular.

Figure 14 – Presence of classes with high fan-out in APIs clusters and bundles.



Source: Produced by the author.

We also noticed the ’annotations’ package usage in Guava. The adoption of `@VisibleForTesting` annotation, intended to indicate relaxed visibility for a type or member in order to facilitate tests, is as expressive as the use of the String class from the ’base’ package. This annotation helps developers to markdown their code for later inspection about visibility choices, which relates more with code documentation and quality than the collections and IO features Guava are known for. The adoption of this annotation (along with `@Beta` annotation, designed to be internal to Guava but also adopted in some clients to denote their own public API may be changed or removed) may indicate a path for API expansion or split to cover specific clients needs.

For Weka API, which serves a big number of purposes in math and machine learning, usage shows that clients adopt classifiers and GUI features the most if we take aside the core package. Clusterization comes next along with experiment tools and filtering algorithms. But the classifier usage alone is bigger than all other feature groups usage combined. From an unbundling perspective, the classification section of the API could be a suitable product on its own.

The JUnit usage tells us that clients follow the unit tests main rules: initialize resources, run the tests, use assertions to make sure things are going as expected and, finally, free resources after tests run. The classes related to these mentioned features are at the top of usage ranking, followed by the `RunWith` that allows client classes to run the tests outside a JUnit-based runner. The first 6 classes on the usage ranking actually correspond to more than 77% of total usage, meaning JUnit has a solid core for most of the clients.

This feature-oriented analysis is important as a post unbundling process must be carried out in a real-world unbundling scenario to help managers and development team understand where their software is most used by the clients. Or, in an opposite approach, where lies the code that may become a candidate to split due to feature specificity.

4.3 Subject APIs Evolution

This study analyses 10 well known APIs from Java open source ecosystem, as their last stable versions of September 2015. We believe that since this research touches the Software Evolution field, we should also conduct an analysis on the 10 APIs source code evolution to identify if some of the scenarios we saw or foresee through the unbundling process are present on these APIs current versions. The following sections discuss a back and then comparison for each API, along with some related insights from the unbundling perspective. The API jars, source codes and few usage metrics were collected in the Maven Repository[7] website.

4.3.1 CommonsIO

The Apache CommonsIO remained almost unchanged from our reference version, 2.4 (June 2012), to the 2.6 (October 2017), in terms of structure, keeping the same distribution of files and packages, with the inclusion of a `serialization` package and some new classes in `input`, `output` packages, as well as in the root level. We did not find any change in code towards feature separation in this API, despite the 5 years of distance between them. These two versions of CommonsIO are the API most popular, according to the usage metrics provided in Maven Repository. Because CommonsIO is a heavily used library in the Java ecosystem and also has a focused niche, we believe this behaviour should last for many versions to come.

4.3.2 Gson

GSon at 2.8.6 (October, 2019) doesn't differ a lot from 2.3.1 (November 2014), from the unbundling perspective. New packages were included for reflection and binding utilities. Like what we have seen in the CommonsIO scenario, Gson did not derived from its core functionalities, but it also might be a much less prone candidate for unbundling, due to its poor level of bundle uniqueness. All three classes that fit the high fan-out criteria discussed in Section 4.2.2 are present in over 93% of the Gson bundles we generated.

4.3.3 Guava

Guava has started to ship two separate versions since May 2017, when 22.0 was released along with 22.0-android. Since 23.1 the version naming changed to X-jre and X-android, as it still is by the version 28.1. Although it seems like a feature isolation, we found the android version of Guava rather started as a sibling API in its first release, which was incrementally evolving to better attend the Android development requirements, than emerged from a natural unbundling process.

7 https://mvnrepository.com/

4.3.4 Hamcrest

The Hamcrest repository we took for analysis (version 1.3, July 2012) moved from `hamcrest-library`, which is now deprecated, to just `hamcrest`. In fact, Hamcrest has merged its side projects into a single one, bringing packages like `core`, `comparator` and `io` into the single supported release. So, in comparison to the last release, 2.2 (October 2019), there are only additions to the code base. Seems that for Hamcrest the unbundling process is not valuable, even though we found promising results (see Section 4.2.1), since the project is moving in the opposite direction, maybe because of its size and number of maintainers. This case illustrates that the unbundling process is ultimately a project decision based on many factors, not just a source code analysis trigger.

4.3.5 JSoup

JSoup is one of the special cases in this study, due to its extremely low bundle uniqueness levels. From 1.8.2 (April 2015) to 1.12.1 (May 2019) the API did not change in structure or parallel distributions. The three classes ranked on high fan-out criteria that appeared in more than 99% of generated bundles are still present in the API, and they are a requirement for at least 35% of the code base to run, especially the `org.jsoup.nodes.Element` class.

4.3.6 JUnit

By the same measure JSoup is the strongest case against unbundling viability of this study, JUnit has emerged as the most unbundling prone API, as we could find a combination of two bundles with 68.3% and 7.6% of the original API size that, together, cover over 98% of the clients. More than that, we found JUnit has changed very much in structure since its 4.12 release from December 2014.

Since 2016, JUnit was divided into three distinct sub-projects: Platform, Jupiter and Vintage. The Vintage sub-project allows the clients to run their JUnit 3 or 4 tests on a compatible `TestEngine`, assuring they can migrate to the new API version, whereas new JUnit 5 based tests can run over the Jupiter sub-project. On the other hand, the Platform sub-project enables clients to build their own testing platform, and the official documentation states that its purpose "*... is to decouple the internals of discovering and executing tests from all the filtering and configuration that's necessary from the outside.*".

In our research we could not identify this shift to a environment that gives the client control over the test engine implementation as a source code feature that emerged from JUnit 4 and below. The hypothesis then is about this decision coming from community and development team expertise. To confirm this information we should proceed to a deeper JUnit usage analysis, which is out of scope for this research.

4.3.7 Mockito

Mockito is currently in version 3.1.0 (October 2019), and as Guava and Hamcrest, is now divided into purposes distributions (such as *scala*, *testng*, *android*, etc), including a version aligned with the new junit-jupiter test format. Because Mockito is a mock library, it's expected it provides features for multiple testing environments, and since 2017 Mockito started publishing separate versions to accomplish this. Back in 1.10.19 version, which we took for reference in this study, everything Mockito offered was packed into a single jar file. On that version the support for Android platform was just briefly mentioned in the `Mockito` and `MockMaker` classes.

We understand a client usage analysis could be useful in this case, for instance, to understand which side library (e.g.: JUnit, TestNG) is more associated with Mockito use; also, to which environments Mockito could dedicate a distribution such as it does for Android applications.

4.3.8 slf4j

The slf4j last stable version is 1.7.29 (October 2019), and comparing it to the one took as reference for this study, we saw a particular behaviour: nothing was deleted in slf4j in over than four years of development. All files either kept their structure, or got new lines of code. Some new files were also added to the code base.

But this API also publishes independent distributions since 2005, such as NOP (no-operations) and JCL (Job Control Language) binding, or the Android flavor, supported since 2010. So considering the good uniqueness level seen in the results for slf4j, like the set of three bundles weighting between 50% to 66% of the original code base (see Table 4), it is possible that new sub-projects that deserves to be distributed on their own, even still under the slf4j environment, will follow the path of these distributions we see for very long time in the project repository.

There are also some alpha and beta versions of slf4j default API published since 2017, but it’s not clear to state about which change in the API they relate to, so we can not link them with evidence of unbundling without a much deeper analysis.

4.3.9 Weka

Weka is the largest API we studied, with more than a thousand classes in the reference version (3.6.12). Currently in version 3.8.3, there is a massive number of changes inside Weka, including another thousand of new files and over 200 files removed. It is virtually impossible to keep track of Weka changes without some guidance from the community or the development team, which is outside the scope of this research.

On the other hand, we can affirm that this API also has some parallel releases of specialized machine learning and math features, like Rotation Forest algorithm, Partial Least Squares filter and Prefuse Graph, among others. These seem not be very active projects, probably due to the strict range of the feature scope (once the concept is correctly implemented, only performance improvements should be added to the code base). They can be seen as plugins for Weka toolset, rather than inner features that arise to be a new project.

4.3.10 XStream

XStream is the last studied API that also has separated modules, such as Core, Hibernate, Benchmark, etc. We noticed that the Hibernate sub-project was first published as 1.4 version in 2011, along with the Core 1.4. The Hibernate related classes were present in the 1.3.1 version, as well as in the 1.4 version of the Core package, but does not appear in the current 1.4.11.1, released in October 2018. This evidence show that Hibernate capabilities that were once part of XStream Core are now fully detached to its own sub-project, which describes some simillarity with the unbundling characteristics.

We also noticed that XStream Core have been evolved in the same slf4j fashion, without any exclusion in the code base since the version we took for analysis in this study. This points to a concise evolution of the core library in the path of its purposes.

5 THREATS TO VALIDITY

Following the best practices of most software engineering studies, we dedicate this chapter to discuss some threats to this research validity, as we understand the experimental investigation may suffer from various inconsistencies that varies from misalignment between research questions to faults on the experiment reproducibility. The threats to validity associated with our investigation are discussed using the four threats classification (construct, internal, external, and reliability validity) presented in (RUNESON *et al.*, 2012).

Construct Validity. The construct validity relates to the adherence of researchers ideas and the intent of research questions. To avoid interpretation inconsistencies about the results and research question divergences, a *peer debriefing* approach was adopted for both research design validation and document review. These steps were performed multiple times during the study life span, especially to avoid misunderstandings between the unbundling and software evolution purposes.

Internal Validity. This validity criteria refers to the possibility of a unknown factor interference over a identified causal relation, and therefore the risk of misleading results and conclusions. As discussed in Section 4.2, this investigation has found a possible causal relation between the higher fan-out level classes in an API and its unbundling results, having uniqueness as the third variable for validation. A micro study was performed on this matter and we found evidences to support our hypothesis that high fan-out values lead to a worse unbundling scenario for APIs. However, I also agree the study lacks a deeper analysis taking into account other coupling measures or API design guidelines. This may be the subject to a follow-up study, and is briefly discussed on Chapter 7.

External Validity. The external validity concerns about the extension of a particular research results for a bigger context or for a group outside the subject limits. In other words, by thinking about external validity we are addressing the generalization this research can leverage in the field of software engineering.

Our study meant to draw a picture of API unbundling, which can be applied to any open API project for feature acquisition and evolution. However, we acknowledge the limitation that we should not suggest a generic scenario for unbundling to the whole extent of existing APIs.

Reliability. This factor is mostly related to the reproducibility of the experiment and the data analysis, making possible for a completely different researcher to reach the same results found by the original authors. To offer a decent level of reliability, a research needs to state very clearly the process and sources for data collection, processing and analysis. The authors should also acknowledge any interference made in the data that evades the expected workflow.

Some data preparation was applied to the mined projects in this study, such as the removal of clients that import "*" (which makes us unsure of which classes were actually used from a given package) and clients that import classes that do not exist in the API version we choose for analysis (e.g, a client imports a class or interface that was present in earlier versions of the API but was removed in the last stable version prior to September/2015). Clients that fit these scenarios were excluded from the dataset and so their usages were not taken into account.

We also acknowledge that due to CPU and memory limitations of the machines available for experimentation, we randomly reduced the set of JUnit clients to a quarter of its original size. The quantity of 19.636 projects listed on Table 1 corresponds to the actual number of clients used through the clustering process. We did not experimented multiple versions of this random client filtering, so the idea is briefly discussed in Chapter 7.

Like any exploratory research, this study had its boundaries and was limited to handle 10 well known APIs from the Java ecosystem. This can be interpreted as threat from the perspective of the variety of APIs niche, usage and maturity. Also, because only open source APIs and clients are analyzed, this study might be missing a relevant number of interesting scenarios in proprietary APIs.

Finally, another significant reliability threat of this study is the fact that it uses a dataset from September/2015 of public client projects hosted on GitHub. This dataset was mined using the Boa[8] language and infrastructure, and from the results extracted in the form of text files, all the unbundling process was conducted as described in this document. One can reproduce this study by the same patterns as long as the source code for JUnbundler (see Chapter 4) is still available, along with the availability of the Boa infrastructure and dataset.

8 http://boa.cs.iastate.edu/

6 RELATED WORK

This chapter describes previous contributions in API design, modularity, software evolution, usage and usability, that connect to this research by looking at the same source code splitting problem, design issues that lead to high levels of coupling or even propose tools and process for better results in repository mining.

Back in 1972, Parnas (PARNAS, 1972) was one of the very first to investigate software modularization as a mechanism to improve software flexibility and understandability. From those days to now, a lot of work has been done towards a better understanding of software modularity and how such concern must be taken into account during the software lifecycle (HOEK; LOPEZ, 2011; BECK; DIEHL, 2011). We investigated the API unbundling process based on clients usage, which can be seen as a dimension of software modularity. In this section, we describe selected related work.

Evolution is a core concern for any software project intended to follow up the industry changes. This is even stronger for APIs as they serve a number of client projects, which have themselves a particular evolution cycle (GRANLI *et al.*, 2015; HORA *et al.*, 2018). So as far as clients are expected to accommodate API changes, we also see co-evolution where the API change to meet client needs based on their usage (JEZEK; DIETRICH, 2017; EILERTSEN; BAGGE, 2018). The usage oriented unbundling process has the potential to play important role in this task, marking clients usage as structural API changes candidates.

Haenni *et al.* (HAENNI *et al.*, 2014) distinguish open source API developers view' as upstream and downstream, based on their relation of authors or clients of the code, respectively. Through their survey they identified the main needs from both categories, but we are more interested in the upstream one. From the upstream perspective, of developers who maintain an API that is used by others, the research highlighted "API usage details" as a valuable information, mainly about the API usability level, which methods are most called and which are not used. They also highlight an respondent note that indicates his interest in understanding which parts of the API are used in conjunction and which are used independently. This result aligns very well to our effort to collect information about the APIs usage from their real clients, as a input for structural changes.

Departing from the assumption of client usage being relevant for API evolution, many authors work on tools and process to collect and asses this information. To assist developers in the understanding API usage, Leuenberger *et al.* (LEUENBERGER *et al.*, 2017) developed a tool, KOWALSKI, that finds the API clients by exploiting the Maven dependency management system, drawing a representative picture of the API usage. It builds a tree of client based on the usage indexation provided by the Maven Central website, which serves a huge number of artifacts for Maven based projects. This factor limits the KOWALSKI reach to only Maven Central hosted clients, like JUnbundler limitation to GitHub hosted projects. From the set of downloaded client JAR files, the application process the method calls from these clients, which can help developers identify usage hotspots on the target API.

Härtel *et al.* (HäRTEL *et al.*, 2018) explored a method for clustering APIs according to programming domains (e.g., databases, collections, parsers, security). That work aims to improve the search and the understanding of technology stacks used in open source projects based on Maven repositories (e.g., Maven Central, JCenter) and hosted on GitHub. The authors proposed a curated API suite, built from a combination of selection based on GitHub activity of popular Java APIs, and the clustering of these using the categories and tags provided by Maven Central respository.

Sawant and Bacchelli (SAWANT; BACCHELLI, 2015) and Leuenberger (EILERTSEN; BAGGE, 2018) propose models for usage analysis exploiting components that rather than simply included through import statements are actually used through method calls. Their approach relies on bytecode analysis; Leuenberger work is based on the ASM analysis framework, which visits the entry points in the source code and validates it with the final bytecode instructions. This method allows the authors to build a call tree with the full qualified names of the objects involved, which is a more trustworthy call validation approach than relying on class imports on source code. This fine-grained inspection improves the quality of usage information and expands feature use characteristics from class to method level.

Rama and Kak (RAMA; KAK, 2015) developed a set of general-purpose metrics for quantitative assessment of API usability. Such metrics examine the API method declarations from the perspective of several commonly held beliefs regarding what makes APIs difficult to use. These identified imply in how clients make use of the target API, so it might not just relate directly with refinements for the API design, but also engrave trends on the unbundling results, which is an aspect we could analyze in a near future research extension for our study.

Murphy-Hill| *et al.* (MURPHY-HILL *et al.*, 2018) studied usability problems in API when they scale-up. They identified that users tend to struggle when switching from methods that fail to convey their essence by their names (e.g., from `of()` to `copyOf()` in the

`ImmutableList` available in both Google API and Java default collections library). Their results might not be of interest from an unbundling perspective in its preparation phase, but draws attention to post-unbundling, when API developers should guarantee their clients can accomplish things with the same or better level of quality and comprehension the original API offered them.

Some proposed tools are meant to help developers to identify and change their code to conformity after API changes (NGUYEN *et al.*, 2010) or even identify if they are duplicating code already available in the API (KAWRYKOW; ROBILLARD, 2009). These also fall into the post-unbundling scenario, when the divided sub-APIs should be maintained with an extra effort to avoid duplication, when possible, or completely remove occurrences of features that were extracted in their old place.

We should also leave a note about the lack of related work on the same track this study follows: unbundling based on usage. The unbundling phenomenon itself is not yet a subject stressed in the community, being the contribution of Filho *et al.* (FILHO *et al.*, 2015) one of the very few in the field, moreover, the usage based branch has not been discussed by the time this study was released. Our expectation is to motivate new investigations on how to interpret and benefit from the client usage role over software unbundling. Chapter 7 present some hints about where to go from here.

7 CONCLUSION AND FUTURE WORK

This research explored in an empirical way the possibilities of unbundling well-known APIs based on the use that their client projects make. We could find that it is possible to generate smaller APIs that still attend to most or all the clients, and that it can be done automatically. These sub-APIs were thoroughly studied to evidence how unique they were and how this uniqueness relate to their sizes and their capabilities of matching clients' needs.

As expected and aligned with preliminary analysis, we did confirm that when assembling bundles from their clusters seeds, the uniqueness level of the later compared to the first drops due to size increase, which implies that bundles are much alike each other than clusters. From the same inflation aspect, bundles show higher representativeness than their original clusters. Finally, even if a best scenario was not found for any of the studied APIs (where we expected to see distinct and complementary bundles in which an API could be divided), this study was able to produce reduced profile APIs with no orphan clients for almost all the studied set, being slf4j the only exception. This result relates with the slf4j status of most difficult API to unbundle. JUnit has also emerged as the most unbundling prone API in our research.

This study may be used as a starting point for further investigation about API architecture and unbundling strategies. As mentioned before, the split measure for an API is strictly particular to the very API, in such a way that generic guidelines and formulas would not rise as silver bullets in this situation. Also, unbundling an API should take into account other engineering perspectives (e.g., feature distribution among bundles, refactoring effort and project goals), not only the measurable aspects of where to divide the code. So this work intends to support other researches moving towards the construction of tools and workflows to help in the execution of the unbundling, and reinforces the value of a specific analysis for each unbundling candidate.

The main contributions of this work are the experiment protocol, the unbundling algorithm and some metrics. The proposed protocol can be applied for similar studies or extended to reach more detailed aspects of the unbundling process. The algorithm defines the core of the bundle construction process, from which we guarantee some characteristics as the capacity to serve a group of clients, due to its bottom-up inclusion of dependencies approach. The uniqueness and representativeness metrics are the preliminary measures with which we can analyse the bundles, and help to decide which ones should compose the unbundled API.

Before extending and surpassing this study model, we see at least some small efforts that benefit from the existing infrastructure. Firstly, an October/2019 Boa dataset has been released, allowing a whole series of related studies, such as the evolution of clients and their adherence to new versions of the studied APIs. Also, by focusing on JUnit and its huge client base, it is possible to perform multiple sessions of randomly selecting a quarter of clients, and then compare how similar are the results for JUnit unbundling when varying these clients. The idea is to analyse the impacts of diverse clients usage on splitting the same API.

Beyond this study limits, we comprehend a natural and enriching step would be to get API maintainers and users involved to validate some of the results found, and further discuss the applicability of this unbundling process. We can also imagine efforts for analyzing web API usages from the set of endpoints a group of clients consumes. From that information, we would able to extract the same dependency tree as we did in this study and build sub-APIs. This approach demands a more powerful scheme for repository mining, but most of the data processing tools we have by this date can be applied.

Another study we want to perform involves API evolution, and it’s an extension on the analysis made in section 4.3. We aim to analyze how the aspects discussed here, such as bundles uniqueness and representativeness, evolved in time on APIs that were submitted to the unbundling process. To do so, we should choose a fewer number of APIs and watch their unbundling characteristics through a timeline of released versions, grouping the clients by the version they were using, as well as compare the trends found with the real split applied to the API. This may lead to some understanding of the criteria for unbundling in real-world situations.

A paper derived from this study was published on the 16th International Conference on Mining Software Repositories (2019), and later published on the event proceedings under the DOI 10.1109MSR.2019.00062.

REFERENCES

AJILA, S. A.; GAKHAR, A. S.; LUNG, C.-H. Aspectualization of code clonesÑan algorithmic approach. **Information Systems Frontiers**, Springer, p. 1–17, 2013.

ANDA, B. Assessing software system maintainability using structural measures and expert assessments. In: IEEE. **Software Maintenance, 2007. ICSM 2007. IEEE International Conference on**. [*S.l.*], 2007. p. 204–213.

BECK, F.; DIEHL, S. On the congruence of modularity and code coupling. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 354–364. ISBN 978-1-4503-0443-6. Disponível em: http://doi.acm.org/10.1145/2025113.2025162. Acesso em: 07 maio 2019.

BELADY, L. A.; LEHMAN, M. M. A model of large program development. **IBM Systems journal**, IBM, v. 15, n. 3, p. 225–252, 1976.

BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: a roadmap. In: CITESEER. **Proceedings of the Conference on the Future of Software Engineering**. [*S.l.*],2000. p. 73–87.

DIJKSTRA, E. W. The structure of the "the" multiprogramming system. In: **The origin of concurrent programming**. [*S.l.*]: Springer, 1968. p. 139–152.

DYER, R.; NGUYEN, H. A.; RAJAN, H.; NGUYEN, T. N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: **Proceedings of the 35th International Conference on Software Engineering**. [*S.l.: s.n.*], 2013. (ICSE'13), p. 422–431.

EILERTSEN, A. M.; BAGGE, A. H. Exploring api: Client co-evolution. In: **Proceedings of the 2Nd International Workshop on API Usage and Evolution**. New York, NY, USA: ACM, 2018. (WAPI '18), p. 10–13. ISBN 978-1-4503-5754-8. Disponível em: http://doi.acm.org/10.1145/3194793.3194799. Acesso em: 04 junho 2019.

FILHO, J. B. F.; ACHER, M.; BARAIS, O. Challenges on software unbundling: growing and letting go. In: ACM. **Companion Proceedings of the 14th International Conference on Modularity**. [*S.l.*], 2015. p. 43–46.

FILHO, J. B. F.; ACHER, M.; BARAIS, O. Software unbundling: Challenges and perspectives. **Transactions on Modularity and Composition**, 2016.

GODFREY, M. W.; GERMAN, D. M. The past, present, and future of software evolution. In: IEEE. **2008 Frontiers of Software Maintenance**. [*S.l.*], 2008. p. 129–138.

GRANLI, W.; BURCHELL, J.; HAMMOUDA, I.; KNAUSS, E. The driving forces of api evolution. In: **Proceedings of the 14th International Workshop on Principles of Software Evolution**. New York, NY, USA: ACM, 2015. (IWPSE 2015), p. 28–37. ISBN 978-1-4503-

3816-5. Disponível em: http://doi.acm.org/10.1145/2804360.2804364. Acesso em: 12 dezembro 2018.

HAENNI, N.; LUNGU, M.; SCHWARZ, N.; NIERSTRASZ, O. A quantitative analysis of developer information needs in software ecosystems. In: ACM. **Proceedings of the 2014 European Conference on Software Architecture Workshops**. [*S.l.*], 2014. p. 12.

HäRTEL, J.; AKSU, H.; LäMMEL, R. Classification of apis by hierarchical clustering. In: **Proceedings of the 26th Conference on Program Comprehension**. New York, NY, USA: ACM, 2018. (ICPC '18), p. 233–243. ISBN 978-1-4503-5714-2. Disponível em: http://doi.acm.org/10.1145/3196321.3196344. Acesso em: 22 maio 2019.

HOEK, A. van der; LOPEZ, N. A design perspective on modularity. In: **Proceedings of the Tenth International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2011. (AOSD '11), p. 265–280. ISBN 978-1-4503-0605-8. Disponível em: http://doi.acm.org/10.1145/1960275.1960307. Acesso em: 30 outubro 2018.

HORA, A.; ROBBES, R.; VALENTE, M. T.; ANQUETIL, N.; ETIEN, A.; DUCASSE, S. How do developers react to api evolution? a large-scale empirical study. **Software Quality Journal**, Kluwer Academic Publishers, Hingham, MA, USA, v. 26, n. 1, p. 161–191, mar. 2018. ISSN 0963-9314. Disponível em: https://doi.org/10.1007/s11219-016-9344-4. Acesso em: 15 junho 2018.

JACCARD, P. Nouvelles researches sur la distribution florale. **Bull Soc Vaud Sci Nat**, v. 44, p. 223–270, 1908.

JACOBSON, I.; LINDSTRÖM, F. Reengineering of old systems to an object-oriented architecture. In: ACM. **ACM Sigplan Notices**. [*S.l.*], 1991. v. 26, n. 11, p. 340–350.

JEZEK, K.; DIETRICH, J. Api evolution and compatibility: A data corpus and tool evaluation. **Journal of Object Technology**, v. 16, n. 4, p. 2:1–23, aug 2017. ISSN 1660-1769.

KAWRYKOW, D.; ROBILLARD, M. P. Improving api usage through automatic detection of redundant code. In: **Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 111–122. ISBN 978-0-7695-3891-4. Disponível em: https://doi.org/10.1109/ASE.2009.62. Acesso em: 16 junho 2018.

LEHMAN, M. M.; RAMIL, J. F.; WERNICK, P. D.; PERRY, D. E.; TURSKI, W. M. Metrics and laws of software evolution-the nineties view. In: IEEE. **Proceedings Fourth International Software Metrics Symposium**. [*S.l.*], 1997. p. 20–32.

LEITNER, P.; BEZEMER, C.-P. An exploratory study of the state of practice of performance testing in java-based open source projects. In: **Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering**. New York, NY, USA: ACM, 2017. (ICPE '17), p. 373–384. ISBN 978-1-4503-4404-3.

LEUENBERGER, M.; OSMAN, H.; GHAFARI, M.; NIERSTRASZ, O. Kowalski: Collecting api clients in easy mode. In: **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S.l.: s.n.*], 2017. p. 653–657.

MORTENSEN, M. **Improving software maintainability through aspectualization**. Tese (Doutorado) — Department of Computer Science, Colorado State University, 2009.

MORTENSEN, M.; GHOSH, S.; BIEMAN, J. M. A test driven approach for aspectualizing legacy software using mock systems. **Information and Software Technology**, Elsevier, v. 50, n. 7, p. 621–640, 2008.

MURPHY-HILL, E.; SADOWSKI, C.; HEAD, A.; DAUGHTRY, J.; MACVEAN, A.; JASPAN, C.; WINTER, C. Discovering api usability problems at scale. In: **Proceedings of the 2Nd International Workshop on API Usage and Evolution**. New York, NY, USA: ACM, 2018. (WAPI ’18), p. 14–17. ISBN 978-1-4503-5754-8. Disponível em: http://doi.acm.org/10.1145/3194793.3194795. Acesso em: 25 fevereiro 2019.

NGUYEN, H. A.; NGUYEN, T. T.; WILSON JR., G.; NGUYEN, A. T.; KIM, M.; NGUYEN, T. N. A graph-based approach to api usage adaptation. In: **Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2010. (OOPSLA ’10), p. 302–321. ISBN 978-1-4503-0203-6. Disponível em: http://doi.acm.org/10.1145/1869459.1869486. Acesso em: 28 março 2019.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. **Commun. ACM**, ACM, New York, NY, USA, v. 15, n. 12, p. 1053–1058, dez. 1972. ISSN 0001-0782. Disponível em: http://doi.acm.org/10.1145/361598.361623. Acesso em: 20 novembro 2018.

PARNAS, D. L. Software structures: A careful look. **IEEE Software**, IEEE, v. 35, n. 6, p. 68–71, 2018.

PENNY, G. *et al.* **Software maintenance: concepts and practice**. [*S.l.*]: World Scientific, 2003.

POIRIER, Y. **What are the Most Popular Libraries Java Developers Use? Based on Github’s Top Projects**. 2018. Disponível em: https://blogs.oracle.com/java/top-java-libraries-on-github. Acesso em: 22 setembro 2018.

RAMA, G. M.; KAK, A. Some structural measures of api usability. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 45, n. 1, p. 75–110, jan. 2015. ISSN 0038-0644. Disponível em: http://dx.doi.org/10.1002/spe.2215. Acesso em: 20 janeiro 2019.

REBÊLO, H.; LEAVENS, G. T.; BAGHERZADEH, M.; RAJAN, H.; LIMA, R.; ZIMMERMAN, D. M.; CORNÉLIO, M.; THÜM, T. Modularizing crosscutting contracts with aspectjml. In: ACM. **Proceedings of the of the 13th international conference on Modularity**. [*S.l.*], 2014. p. 21–24.

RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. **Case Study Research in Software Engineering: Guidelines and Examples**. 1st. ed. [*S.l.*]: Wiley Publishing, 2012. ISBN 1118104358, 9781118104354.

SAWANT, A. A.; BACCHELLI, A. A dataset for api usage. In: **Proceedings of the 12th Working Conference on Mining Software Repositories**. Piscataway, NJ, USA: IEEE Press,

2015. (MSR ’15), p. 506–509. ISBN 978-0-7695-5594-2. Disponível em: http://dl.acm.org/citation.cfm?id=2820518.2820599. Acesso em: 27 setembro 2018.

SZŐKE, G.; ANTAL, G.; NAGY, C.; FERENC, R.; GYIMÓTHY, T. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. **Journal of Systems and Software**, Elsevier, v. 129, p. 107–126, 2017.

THOMPSON, D. V.; HAMILTON, R. W.; RUST, R. T. Feature fatigue: When product capabilities become too much of a good thing. **Journal of marketing research**, American Marketing Association, v. 42, n. 4, p. 431–442, 2005.

TRIPATHY, P.; NAIK, K. **Software Evolution and Maintenance: A Practitioner’s Approach**. [*S.l.*]: John Wiley & Sons, 2014.

WARD, J. H. Hierarchical grouping to optimize an objective function. **Journal of the American Statistical Association**, v. 58, n. 301, p. 236–244, 1963. Disponível em: http://www.tandfonline.com/doi/abs/10.1080/01621459.1963.10500845. Acesso em: 07 outubro 2018.

WASHIZAKI, H.; FUKAZAWA, Y. A technique for automatic component extraction from object-oriented programs by refactoring. **Science of Computer programming**, Elsevier, v. 56, n. 1, p. 99–116, 2005.

APENDIX A – BOA SCRIPT FOR GSON CLIENTS MINING

The source code below shows an example of how to mine the Boa repository for client projects of Google Gson API.

```
# mining how many projects use Google GSON
importsInProjects: output collection[string] of string;

p: Project = input;

visit(p, visitor {
      # only look at the latest snapshot of Java files
      before n: CodeRepository -> {
             snapshot := getsnapshot(n, "SOURCE_JAVA_JLS");
             foreach (i: int; def(snapshot[i]))
                    visit(snapshot[i]);
             stop;
      }
      # look for imports
      before node: ASTRoot ->
          exists(j: int; match("^com\\.google\\.gson\\.", node.imports[j])) {
            importsInProjects[node.imports[j]] << p.name;
                    stop;
          }
      # look for FQN
      before node: Type ->
          if (match("^com\\.google\\.gson\\.", node.name)) {
            importsInProjects[node.name] << p.name;
                    stop;
          }
});
```