2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

# Splitting APIs: An Exploratory Study of Software Unbundling

Anderson S. Matos
*GREat Research Group*
*Federal Universisty of Ceará*
Fortaleza, Brazil
severo@alu.ufc.br

João B. Ferreira Filho
*GREat Research Group*
*Federal Universisty of Ceará*
Fortaleza, Brazil
bosco@dc.ufc.br

Lincoln S. Rocha
*GREat Research Group*
*Federal Universisty of Ceará*
Fortaleza, Brazil
lincoln@dc.ufc.br

***Abstract*—Software unbundling consists of dividing an existing software artifact into smaller ones. Unbundling can be useful for removing clutter from the original 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 paper proposes a first empirical study on unbundling to understand its effects on popular APIs. We explore the possibilities of splitting libraries into $2$ or more bundles based on the use that their client projects make of them. We mine over than $71,000$ client projects of 10 open source APIs and automatically generate $2,090$ sub-APIs to then study their properties. We find 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**

## I. 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* [1]), and from the point of view of development it can negatively affect maintainability [2], [3]. Recently, it has been observed that mobile apps go through an evolutionary phenomenon of unbundling [4]. 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 complex ones; on the other hand, decomposing is also becoming important as we reach 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, the main goal of these approaches is to enhance non-functional properties of a program, such as flexibility, comprehensibility, maintainability, etc. [5]–[9]; 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 [4]. Commercially, changing a software risks its user acceptance, therefore its popularity (downloads, sales, etc.). 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 paper, we explore the unbundling of APIs. We analyze how APIs are used by their client projects and we propose to split these APIs according to the different ways that groups of projects use them structurally – **unbundling based on structural 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 [10], so this scenario favours large-scale studies and makes APIs the best fit for analyzing software and its usage by the community.

First, we select $10$ APIs and mine projects on GitHub that use each API. We collect over $71,000$ client projects of these 10 APIs and we identify which classes of each API they are using. Second, for a given API, we try to discover clusters of usages, each cluster being a group of projects that use the API in a similar fashion (i.e., use the same classes from the subject API). Once we identify these 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 generate $2,090$ sub-APIs and then study 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

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00062

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 paper.

The remainder of this paper is structured as follows. Section II gives a background about unbundling based on usage and its main concepts, such as uniqueness and representativeness of bundles. Section III describes our empirical exploratory study. Section IV presents results from the empirical study. We then discuss the results in Section V. In Section VI we touch threats to the validity of this study. Section VII lists and explains related work. Finally, we present our conclusions and main ideas for further investigations in Section VIII.

## II. 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.

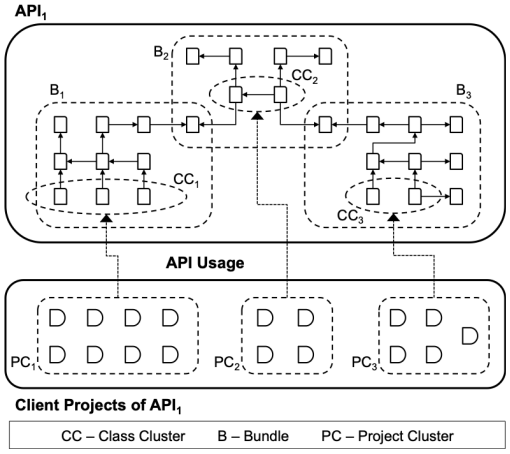


Fig. 1. API usage by client projects.

Following, we explain the concepts that will help us guide this exploratory study, our main objective being unbundling (i.e., dividing an API into bundles) and studying these parts of an API. 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 understand the 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 understand and formalize an algorithm for the unbundle.

### A. Similarity based on API Usage

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

*1) API usage:* Let us first define what is usage in our context. 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 $API_1$. In this paper, we are 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) 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 $s = 1$. We want $s$ to be 0 if two projects do not share any references to the same classes of an API. Meanwhile, we 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 [11], which denotes the intersection over the union of two sets. Therefore we define the similarity of usage $s$ between a project $P_a$ and a project $P_b$ as being:

$$s = \frac{U(API_n, P_a) \cap U(API_n, P_b)}{U(API_n, P_a) \cup U(API_n, P_b)} \quad (1)$$

*3) 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 [12] of Dijkstra's paper "The structure of the "THE" multiprogramming system" [13], 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 of this API.

### B. Representativeness and Uniqueness of Bundles

To better evaluate the quality of the bundles taken from an API, we 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 be sufficient for many clients. The bundle representativeness

$R_{Bi}$ (2) is the number of clients covered ($C_c$) by a given bundle $B_i$ divided by the total of clients ($T_c$) an API has.

$$R_{Bi} = \frac{C_c(B_i)}{T_c} \tag{2}$$

The uniqueness of a bundle $B_n$, $U_{Bn}$, is the bundle size in number of classes, divided by the size of the intersection among all bundles from the same split, see (3). In this study, we 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 we are able to group similar bundles and study the most specific ones, in other words, those that do not share much code with their siblings, therefore serving to specialized group of clients. To decide if these unique bundles are good choices for API splitting, we should also consider other metrics, such as its representativeness.

$$U_{Bi} = \frac{size(B_i)}{size(B_1 \cap B_2 ... \cap B_i \cap ... \cap B_n)} \tag{3}$$

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

### C. Unbundling Algorithm

Given one API to be unbundled, our algorithm (Algorithm 1) receives as input the set of all classes and interfaces ($\mathcal{S}$) of the API version under consideration; a set ($\mathcal{C}$) initially composed of all classes and interfaces of a cluster derived from the API client projects' usage, and a set of classes and interfaces ($\mathcal{B}$) belonging to the derived sub-API (i.e., a bundle).

**Algorithm 1** Unbundling Algorithm.

```
1: //S  is a given set of all API classes and interfaces
2: //C  is a given set of classes and interfaces such that  S ⊇ C
3: //B  is a set of bundle classes and interfaces, initially empty, such that  S ⊇ B
4: function UNBUNDLING(S, C, B)
5:     for all c ∈ C do
6:         B = B ∪ {c}
7:         for all b ∈ DEPENDENCIES(c, S) do
8:             if b ∉ B then
9:                 B = B ∪ UNBUNDLING(S, {b}, B)
10:            end if
11:        end for
12:    end for
13:    return B
14: end function
15:
16: //c  is a given class or interface
17: //C  is a given set of classes and interfaces
18: function DEPENDENCIES(c, C)
19:    return {x | x ∈ C ∧ c requires, needs or depends on x}
20: end function
```

The set $\mathcal{B}$ is initially empty. Each element (class or interface) in the set $\mathcal{C}$ is added to the set $\mathcal{B}$ along with its dependencies. Those dependencies may induce other dependencies of their own, which will also compose the set $\mathcal{B}$. 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.

## III. Empirical Study

In this section, we explore unbundling a number of popular APIs following an empirical method, defining research questions, a protocol, experiment variables and subject APIs. Using the unbundling algorithm, we slice the subject 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.

### A. Research Questions

**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.

**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 describe the usage silhouette, which in turn should help us understand if 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.

**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, we may find that the final bundles are equal, in comparison to each other or even to the original API. 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.

**RQ1c.** *How representative are the bundles?*

We want to investigate how useful are the bundles we generate when splitting an API. 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 even though they are much smaller than the original API.

**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.

### B. Subject APIs

We select 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 I 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 [14].

TABLE I
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 |

### C. Protocol

Fig. 2 shows the protocol of our study. 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 projects' code; this allows us to then calculate a similarity matrix of the projects according to the classes they use from the given API. Third, with the similarity matrix ready, we cluster the projects' usages; we take the classes that cover 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 and splits the API accordingly.

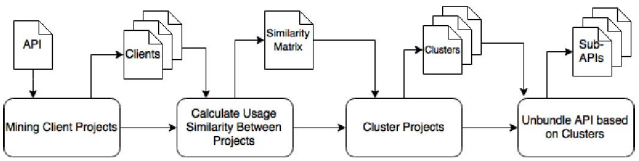


Fig. 2. Unbundling process protocol.

### D. 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 the 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 matrix filled with the Jaccard index for each pair of client projects the API has taken as a similarity measure. We then proceed to build a tree structure using Ward hierarchical clustering algorithm [15]. 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 allow us to cut the result tree in $k$ levels, being $k$ the tree height.

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.

### E. Experiment Variables

Table III-E lists all the variables that are used in this experiment and also guides the analysis of the results. All bundle variables (size, uniqueness, and representativeness) relate directly to the quality of the unbundling process output, whereas the cluster data helps us understand the usage and the source code evolution from the beginning to the moment all the dependencies are included.

## IV. RESULTS

In this section, 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 at https://github.com/severoufc/junbundler.

Before showing these results, we illustrate in Fig. 3, 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

TABLE II
EXPERIMENT VARIABLES.

| Name | Type | Scale Type | Unit | Range | Description |
|---|---|---|---|---|---|
| Split Value | Controlled | Number | Integer | $[2, 20]$ | Number of parts to divide a given API |
| Cluster Size | Independent | Ratio | % | $[0, 100]$ | Proportion of classes or interfaces imported from an API over the API size |
| 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 |
| Number of Clients | Independent | Number | Integer | $[0, \infty]$ | Quantity of API clients |
| Average Cluster Uniqueness | Dependent | Ratio | % | $[0, 100]$ | Overall difference in structure among a set of clusters generated from the same split |
| 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 |

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.

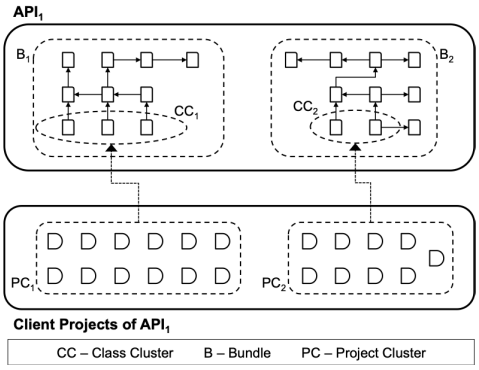


Fig. 3. Ideal API unbundling scenario.

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 V. This section is devoted to answer the research questions presented in Section III.

### *A. 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, we are able to identify, using our unbundling process, a set of clients that will be fully served by a smaller subset of the current API. For instance, dividing the 10 APIs into 2 bundles, we were always able to reduce the size of the sub-APIs, which is shown in Table III.

TABLE III
BUNDLES SIZE FOR ALL APIS (SPLIT INTO 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% |

### *B. 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 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. 4, 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. 4 allows us to conclude that it is possible to cluster clients of APIs according to the usage they make of them and, most importantly, that these clusters of projects are reasonably different with respect to the classes they use from the subject API. However, as we are not able to ship modified APIs with the set of classes described by the usage only (their dependencies must be included to make this sub-API usable), in the next subsection, we discuss this same finding under a bundle perspective.

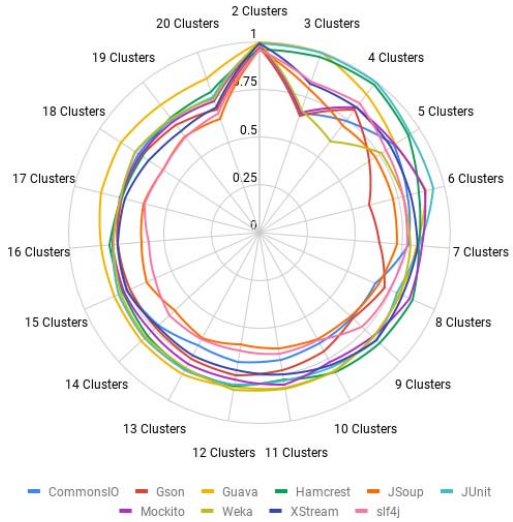


Fig. 4. Average uniqueness among clusters.

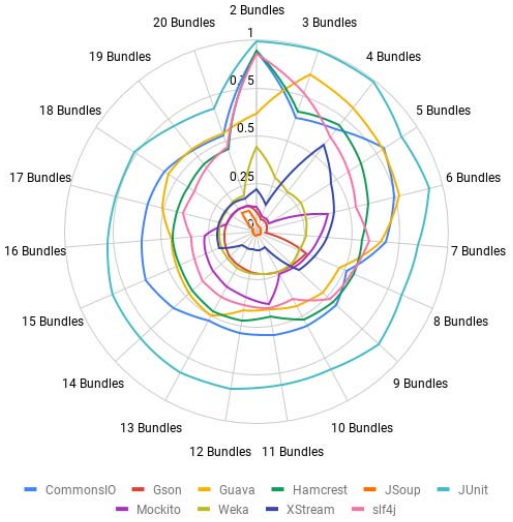


Fig. 5. Average uniqueness among bundles.

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

In Fig. 5, 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 and 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 increases the difference between them considerably, especially for some APIs like Gson and JSoup. To better visualize this progression, we plotted the uniqueness per project, shown in Fig. 6. We can interpret the plots from Fig. 6 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 in 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.

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**.

### D. 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 the API.

Fig. 7 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. 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 (Fig. 3), but clearly serves well the unbundling goal.

Inspecting the API bundles we found that the aforementioned expected assumptions were not all true for them. Our findings are summarized in Fig. 8. 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 not the only ones from medium to high representativeness bundles, 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.

To check a representativeness middle case, we studied Hamcrest bundles. Its average representativeness varied only between $48\%$ to $59\%$. When looking into the real bundles,

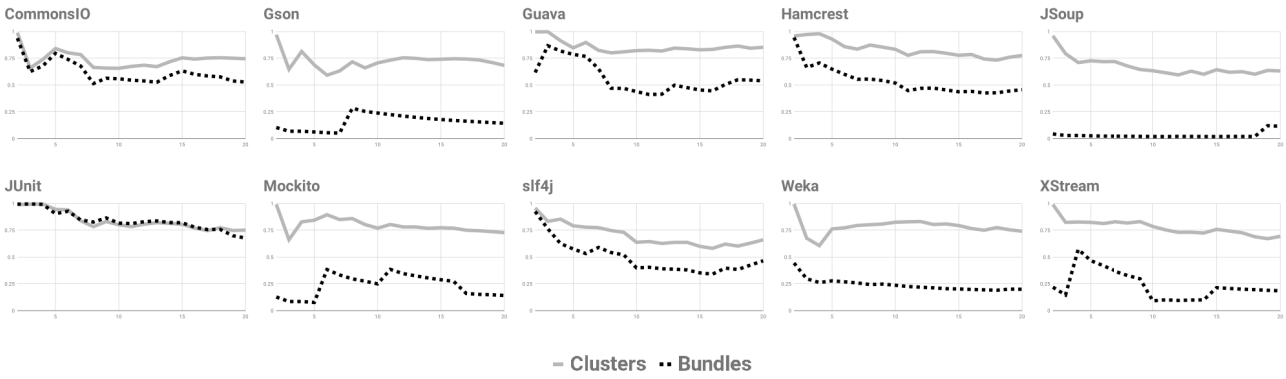


Fig. 6. Average cluster vs bundle uniqueness per API.

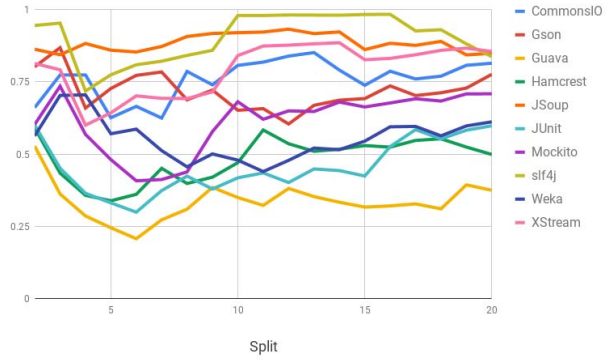


Fig. 7. Average representativeness per API over the splitting process.

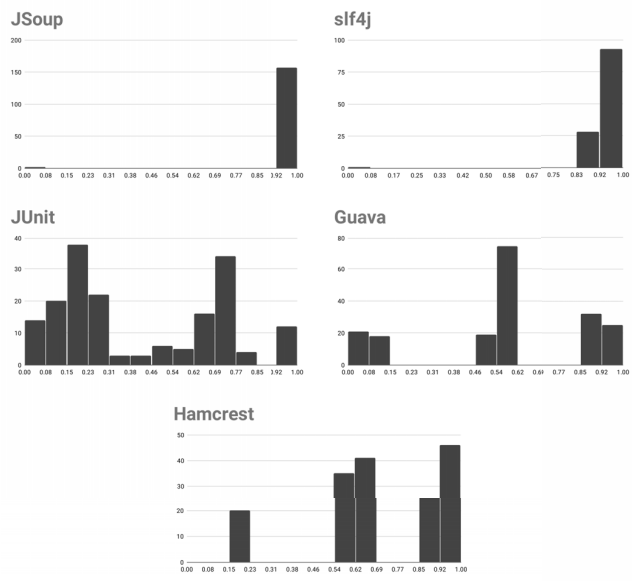


Fig. 8. N. of bundles vs representativeness.

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.

### *E. 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%, as shown in Fig 9. 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 Section VI.

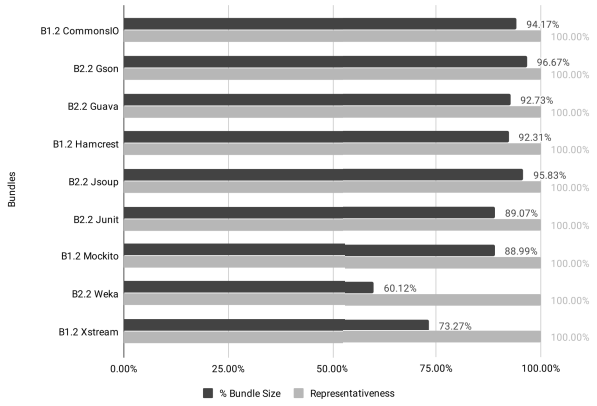


Fig. 9. Bundles sizes vs representativeness.

If we set a lower representativeness bar, to 95% coverage,

the API reduction results go even further. As shown in Fig. 10, 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.

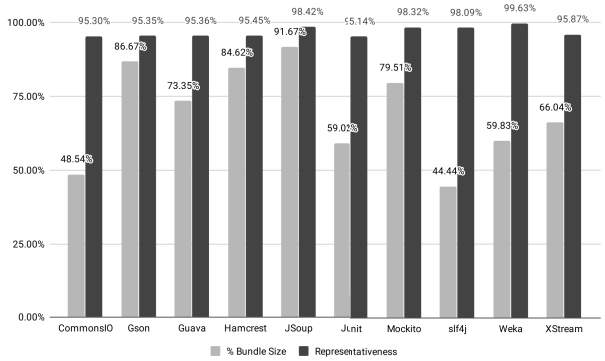


Fig. 10. Smallest bundles with over 95% of representativeness.

## V. Discussion

### A. On finding completely disjoint bundles

We found some similar results to the ideal split (see Fig. 3), but not exactly the same scenario. The best result in terms of the 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 if 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.

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. All selected bundle groupings are described in Table IV.

The mandatory aspect in the search of disjoint bundles is the uniqueness. Fig. 11 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.

TABLE IV
Selected bundle groups for studied APIs.

| API | Bundle | | | 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 | 95.00% | 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% | 99.50% |
| | 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% | 100.00% |
| | B12.20 | 72.33% | 99.73% | |

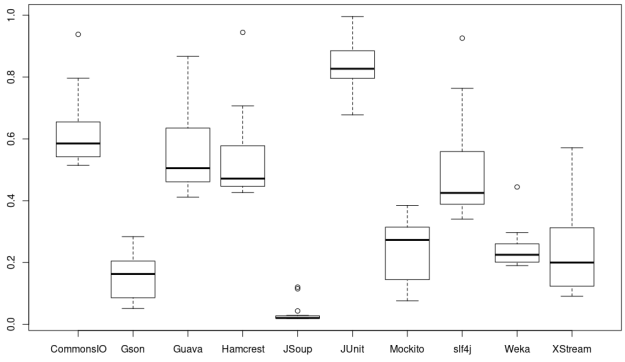


Fig. 11. Average uniqueness of bundles per API.

### B. 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, 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.

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. This thresh-

old 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. 12.

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 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.

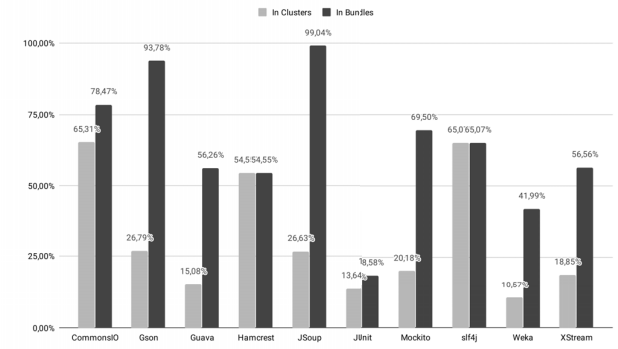


Fig. 12. Presence of classes with high fan-out in APIs clusters and bundles.

### C. 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 usage 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 package. 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.

For Weka API, which serves a big number of purposes in math and machine learning, usage shows clients adopt classifiers and GUI features the most if we take aside features from the core package. Clusterization comes next and 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 free resources after tests run. 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.

## VI. Threats to Validity

The threats to validity associated with our investigation are discussed using the four threats classification (construct, internal, external, and reliability validity) presented in [16].

**Construct Validity**. 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.

**Internal Validity**. As discussed in Section V, 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. But the study lacks a deeper analysis taking into account other coupling measures or API design guideline. This may be the subject to a follow-up study.

**External Validity**. 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 draw a generic picture for unbundling to the whole extent of existing APIs.

**Reliability**. The most significant reliability threat of this study is the fact that it uses a dataset of public client projects hosted on GitHub from September/2015. This dataset was

mined using the Boa[1] language and infrastructure. Most of the studied APIs have already evolved in architecture and features, and some of them even may have been unbundled.

Some data preparation was applied to the mined projects, 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 version prior to September/2015). Clients that fit these scenarios were excluded from the dataset and so their usages were not taken into account.

Finally, due to CPU and memory limitations of machines available, we randomly reduced the set of JUnit clients in this study to a quarter of its original size. The quantity of 19.636 projects listed on Table I corresponds to the actual number of clients used through the clustering process.

## VII. Related Work

Back in 1972, Parnas [17] 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 [18], [19]. We investigated the API unbundling process based on clients usage, witch 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 [20], [21]. 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 [22], [23]. The usage oriented unbundling process has the potential to play important role in this task, marking clients usage as structural API changes candidates.

To assist developers in the understanding API usage, Leuenberger *et al.* [24] developed a tool that finds the API clients by exploiting the Maven dependency management system, drawing a representative picture of the API usage. Härtel *et al.* [25] provides a way to classify API clients according to programming domains (e.g., databases, collections, or security), which can improve the search in repositories such as Maven and the understanding the technology stack used in software projects. Sawant and Bacchelli [26] and Leuenberger [23] propose a model for usage analysis exploiting components that rather than included through import statements are used on method calls. This fine-grained inspection improves the quality of usage information and expands feature use characteristics from class to method level.

Rama and Kak [27] 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. Murphy-Hill *et al.* [28] studied usability problems in API when they scale-up and identified users struggle when switching from methods that fail to convey their essence by their names (e.g., from `of()` to `copyOf()` in the `ImmutableList` class). Some proposed tools are meant to help developers identify and change their code to conformity [29] after API changes or even identify if they are duplicating code already available in the API [30].

## VIII. Conclusion and Future Work

We explored in an empirical way the possibilities of unbundling well-known APIs based on the use that client projects make. We could find that it is possible to generate smaller APIs that still attend to most or all the clients. 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.

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.

Following this study, the library maintainers and users might be 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 are 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 using the tools we have by this date.

Another derived study we want to perform involves API evolution. 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 tendencies found with the real split applied to the API. This may lead to some understanding of the criteria for unbundling in real-world situations.

## Acknowledgement

This research was partially funded by INES 2.0[2], FACEPE grant APQ-0399-1.03/17, and CNPq grant 465614/2014-0.

[1] http://boa.cs.iastate.edu/

[2] www.ines.org.br

## REFERENCES


[1] D. V. Thompson, R. W. Hamilton, and R. T. Rust, "Feature fatigue: When product capabilities become too much of a good thing," *Journal of marketing research*, vol. 42, no. 4, pp. 431–442, 2005.

[2] B. Anda, "Assessing software system maintainability using structural measures and expert assessments," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 204–213.

[3] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Empirical study on refactoring large-scale industrial systems and its effects on maintainability," *Journal of Systems and Software*, vol. 129, pp. 107–126, 2017.

[4] J. B. Ferreira Filho, M. Acher, and O. Barais, "Software unbundling: Challenges and perspectives," *Transactions on Modularity and Composition*, 2016.

[5] M. Mortensen, "Improving software maintainability through aspectualization," Ph.D. dissertation, Department of Computer Science, Colorado State University, 2009.

[6] M. Mortensen, S. Ghosh, and J. M. Bieman, "A test driven approach for aspectualizing legacy software using mock systems," *Information and Software Technology*, vol. 50, no. 7, pp. 621–640, 2008.

[7] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm, "Modularizing crosscutting contracts with aspectjml," in *Proceedings of the of the 13th international conference on Modularity*. ACM, 2014, pp. 21–24.

[8] H. Washizaki and Y. Fukazawa, "A technique for automatic component extraction from object-oriented programs by refactoring," *Science of Computer programming*, vol. 56, no. 1, pp. 99–116, 2005.

[9] S. A. Ajila, A. S. Gakhar, and C.-H. Lung, "Aspectualization of code clonesan algorithmic approach," *Information Systems Frontiers*, pp. 1–17, 2013.

[10] P. Leitner and C.-P. Bezemer, "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*, ser. ICPE '17. New York, NY, USA: ACM, 2017, pp. 373–384.

[11] P. Jaccard, "Nouvelles researches sur la distribution florale," *Bull Soc Vaud Sci Nat*, vol. 44, pp. 223–270, 1908.

[12] D. L. Parnas, "Software structures: A careful look," *IEEE Software*, vol. 35, no. 6, pp. 68–71, 2018.

[13] E. W. Dijkstra, "The structure of the the multiprogramming system," in *The origin of concurrent programming*. Springer, 1968, pp. 139–152.

[14] Y. Poirier. (2018) What are the most popular libraries java developers use? based on githubs top projects. [Online]. Available: https://blogs.oracle.com/java/top-java-libraries-on-github

[15] J. H. Ward, "Hierarchical grouping to optimize an objective function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01621459.1963.10500845

[16] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley Publishing, 2012.

[17] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972. [Online]. Available: http://doi.acm.org/10.1145/361598.361623

[18] A. van der Hoek and N. Lopez, "A design perspective on modularity," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 265–280. [Online]. Available: http://doi.acm.org/10.1145/1960275.1960307

[19] F. Beck and S. Diehl, "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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 354–364. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025162

[20] W. Granli, J. Burchell, I. Hammouda, and E. Knauss, "The driving forces of api evolution," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*, ser. IWPSE 2015. New York, NY, USA: ACM, 2015, pp. 28–37. [Online]. Available: http://doi.acm.org/10.1145/2804360.2804364

[21] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, "How do developers react to api evolution? a large-scale empirical study," *Software Quality Journal*, vol. 26, no. 1, pp. 161–191, Mar. 2018. [Online]. Available: https://doi.org/10.1007/s11219-016-9344-4

[22] K. Jezek and J. Dietrich, "Api evolution and compatibility: A data corpus and tool evaluation," *Journal of Object Technology*, vol. 16, no. 4, pp. 2:1–23, aug 2017.

[23] A. M. Eilertsen and A. H. Bagge, "Exploring api: Client co-evolution," in *Proceedings of the 2Nd International Workshop on API Usage and Evolution*, ser. WAPI '18. New York, NY, USA: ACM, 2018, pp. 10–13. [Online]. Available: http://doi.acm.org/10.1145/3194793.3194799

[24] M. Leuenberger, H. Osman, M. Ghafari, and O. Nierstrasz, "Kowalski: Collecting api clients in easy mode," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 653–657.

[25] J. Härtel, H. Aksu, and R. Lämmel, "Classification of apis by hierarchical clustering," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 233–243. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196344

[26] A. A. Sawant and A. Bacchelli, "A dataset for api usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 506–509. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820599

[27] G. M. Rama and A. Kak, "Some structural measures of api usability," *Softw. Pract. Exper.*, vol. 45, no. 1, pp. 75–110, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1002/spe.2215

[28] E. Murphy-Hill, C. Sadowski, A. Head, J. Daughtry, A. Macvean, C. Jaspan, and C. Winter, "Discovering api usability problems at scale," in *Proceedings of the 2Nd International Workshop on API Usage and Evolution*, ser. WAPI '18. New York, NY, USA: ACM, 2018, pp. 14–17. [Online]. Available: http://doi.acm.org/10.1145/3194793.3194795

[29] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869486

[30] D. Kawrykow and M. P. Robillard, "Improving api usage through automatic detection of redundant code," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–122. [Online]. Available: https://doi.org/10.1109/ASE.2009.62