API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages

Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, Benoit Baudry

The Journal of Systems & Software 184 (2022) 111134

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages ☆

Nicolas Harrand [a,*], Amine Benelallam [b], César Soto-Valero [a], François Bettega [c], Olivier Barais [b], Benoit Baudry [a]

[a] KTH Royal Institute of Technology, Stockholm, Sweden
[b] Univ Rennes, Inria, CNRS, IRISA, Rennes, France
[c] Univ. Grenoble Alpes, Inserm, CHU Grenoble Alpes, HP2, Grenoble, France

ARTICLE INFO

Article history:
Received 18 December 2020
Received in revised form 8 September 2021
Accepted 18 October 2021
Available online 9 November 2021

Keywords:
Mining software repositories
Bytecode analysis
Software reuse
Java
Maven Central Repository

ABSTRACT

Hyrum's law states a common observation in the software industry: "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody". Meanwhile, recent research results seem to contradict this observation when they state that "for most APIs, there is a small number of features that are actually used". In this work, we perform a large scale empirical study of client–API relationships in the Maven ecosystem, in order to investigate this seeming paradox between the observations in industry and the research literature.

We study the 94 most popular libraries in Maven Central, as well as the 829,410 client artifacts that declare a dependency to these libraries and that are available in Maven Central, summing up to 2.2M dependencies. Our analysis indicates the existence of a wide spectrum of API usages, with enough clients, most API types end up being used at least once. Our second key observation is that, for all libraries, there is a small set of API types that are used by the vast majority of its clients. The practical consequences of this study are two-fold: (i) it is possible for API maintainers to find an essential part of their API on which they can focus their efforts; (ii) API developers should limit the public API elements to the set of features for which they are ready to have users.

1. Introduction

Software libraries provide reusable functionalities via Application Programming Interfaces (APIs). Software applications that reuse these functions in their code declare the list of APIs they wish to use. This declaration creates a *dependency* between the *client* application and the *library* API. Our study focuses on two well-documented intuitions about such client–library dependencies. On one hand, Hyrum's law captures a common observation in the software industry: "*With a sufficient number of users of an API, …all observable behaviors of your system will be depended on by somebody*" (Hyrum Wright, 2019). Applied to API usages, this would suggest that with enough clients, even the most exotic API elements would eventually be used by at least a client. On the other hand, recent research results concur to consolidate the intuition that APIs are unnecessarily large and that client dependencies actually focus on a small part of common APIs (Qiu et al., 2016; Eghan et al., 2019; Sawant and Bacchelli, 2017; Mastrangelo et al., 2015).

We are intrigued by the seeming contradiction between these observations: the first one suggests that every API member is eventually used, while the second one suggests that only a small part of APIs is really necessary. Our analysis of millions of dependencies reveals a continuum between these two extremes, rather than a contradiction. In other words, we confirm that libraries contain a portion of API types that are used by a vast majority of clients. Meanwhile, in the presence of a sufficient number of clients, we observe that the rare or exotic API types would eventually fit at least one adventurous client. These results provide evidence that API developers can make trade-offs between the share of API elements they consider in maintenance, documentation, and automated migration tools and the share of clients that they support. To support all clients, developers need to invest effort that is proportional to the total size of APIs. Yet, accepting to support only the majority of clients, which use the core API types, allows for significant effort savings. Considering the typical API maintenance task of migration (Bartolomei et al.,

☆ Editor: W.K. Chan.

* Corresponding author.
E-mail addresses: harrand@kth.se (N. Harrand), amine.benelallam@inria.fr (A. Benelallam), cesarsv@kth.se (C. Soto-Valero), francois.bettega@univ-grenoble-alpes.fr (F. Bettega), barais@irisa.fr (O. Barais), baudry@kth.se (B. Baudry).

https://doi.org/10.1016/j.jss.2021.111134

2009), the migration of the 6317 clients of library `gson:2.3.1` to `jackson-databind` requires supporting migration rules for 162 API types, but this number of types can be decreased to 20 (12%) if only 5669 (90%) clients are to be supported.

Our work explores this spectrum of dependency relations, focusing on the Maven Central ecosystem. This choice is motivated by two factors: it is the most popular repository to distribute code artifacts that run on the Java Virtual Machine; it contains both APIs and clients that depend on these APIs. The Maven Dependency Graph (Benelallam et al., 2019) provides a snapshot of Maven Central as of September 6, 2018. From this graph, we determine the 94 most used libraries and all the client artifacts in the repository, that depend on any version of one of these libraries. This forms the dataset for our study: 5225 libraries (union of all versions of the 94 most popular libraries), 901,876 clients, summing up to 2,169,273 dependencies.

We study how Maven artifacts depend on each other, around three dimensions. First, we analyze the client-side, to determine to what extent each declared dependency is actually used at least once in their code, i.e., there is at least one API member used by the client's code. Second, we analyze the API side of dependencies to determine how different API types are used by the clients. We split this analysis into two steps: we start by investigating the usages of an API, cumulating all its versions; later, we analyze the most popular version of each API. Finally, to put our findings to use, we propose a new actionable way to explore the continuum of dependencies and assess the impact of focusing on a small subset of APIs, e.g., for maintenance activities through extinction sequences. For tasks where costs and effort increase with the size of APIs such as API migration (Bartolomei et al., 2009), a tradeoff can be made between cost and number of clients supported.

The key findings of our study are as follows: (i) on the clients-side, we found that 41,13% of declared dependencies do not translate into API usages at the bytecode-level; (ii) on the libraries-side, we observe the following: when considering the most used version of each library, it is very likely that every public member is used; (iii) meanwhile, we notice that every API can be reduced to a small fraction and still fulfill the needs of a majority of the clients. The size of this fraction varies from one API to another, as library API purpose, size, and usage differ. Our dataset is large enough to include some of the most extreme cases that occur in the extraordinary practice of software development, e.g., a very small API with only annotations, some giant APIs which clients use in a very focused way, or even some artifacts that are massively used even if they have no public types.

The contributions of this paper are as follows:

- A systematic large-scale analysis of 2,169,273 Maven client–API relations.
- A public dataset of 5225 libraries (union of all versions of the 94 most popular libraries) and 901,876 clients drawn from Maven Central (Harrand, 2019b) along with an open reproduction package (Harrand, 2019a). This large dataset can fuel the ongoing research initiatives in the areas of dependency management and release engineering.
- Novel empirical evidence about Maven dependencies: all APIs include a small set of types that is used by the majority of their clients, while most API types are used by at least one client, when considering the most popular version of an API. These findings open new directions to improve Maven's build process and to focus effort on the relevant subset of APIs for maintenance and migration tasks.

This paper is organized as follows. Section 2 introduces the key concepts of Maven. In Section 3 we present our research methodology, analysis infrastructure and the dataset for this study. In Section 4 we discuss the empirical observations about the actual usage of client–library Maven dependencies. In Section 5 we discuss how our results could generalize in other ecosystems.

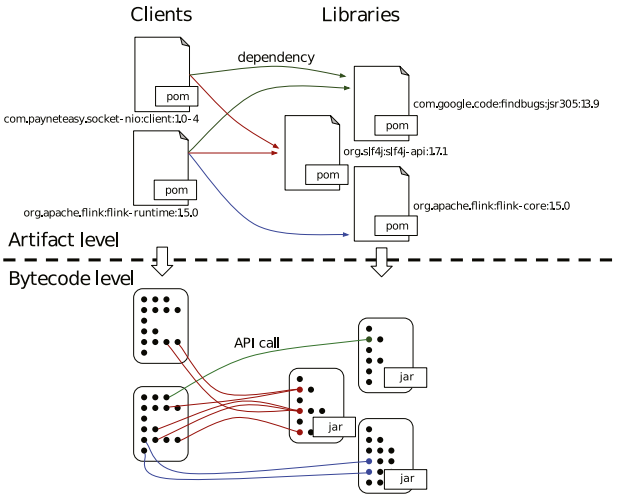


Fig. 1. Software reuse principles in JVM-based projects.

2. API usage in the Maven ecosystem

Maven is a software project management tool for Java and other languages targeting the JVM (e.g., Groovy, Kotlin, Clojure, Scala). It automates most phases of a software development lifecycle, from build to deployment. Maven relies on a specification file, named *pom.xml*, where developers explicitly declare what should happen at each building phase. Dependency management is one important phase where Maven automatically fetches software artifacts on which a project depends. Those artifacts are hosted on remote repositories, either public or private. Currently, Maven Central is the most popular public repository. It hosts millions of software artifacts coming in the form of binary sources (e.g., *jar*). These artifacts are uniquely identified by GAV coordinates, referring to groupId (G), artifactId (A), and version (V). Artifacts in Maven Central cannot be modified or updated, meaning that all the releases of each artifact are stored in the repository.

Maven lets developers the possibility to specify the scope of a dependency declaration. This scope determines when the *jar* of the dependency is added to the classpath of a project (Apache, 2020). *Compile* is the default scope, which implies that the dependency is added to the compilation, test, and runtime classpaths. A dependency with *Compile* scope is also added transitively to the classpath of the artifact's clients (and their clients and so forth), if they do not explicitly exclude it. The dependencies declared with a *Test* are meant to be required only to test the project: they are added only to the compilation and test classpaths; they are not added as transitive dependencies for the clients. The scopes *Provided* and *System* indicate that the dependency will be provided directly by the user if needed at runtime. Such dependencies are resolved by Maven only for the compilation and test classpath, and are not transitive. The scope *Provided* can be used for a dependency that is required only to build a project but not necessary for its execution. Finally, a *Runtime* dependency is only added to the runtime classpath and is transitive.

Fig. 1 illustrates a simplified example of API usages within the Maven ecosystem. API usages happen at two levels, the artifact-level, and the code-level. At the artifact level, a project declares a list of libraries that have to be added to the project's classpath in order to build correctly. At the code-level, the members of the API (e.g., types, methods, etc.) are called, e.g., via object instantiation.

```
<dependency>
    <groupId>com.google.code.findbugs</groupId>
    <artifactId>jsr305</artifactId>
    <version>1.3.9</version>
    <scope>compile</scope>
</dependency>
```

Listing 1: Excerpt of the *pom.xml* file of flink-runtime:1.5.0

```
// API members of slf4j-API
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
// API members of findbugs
import javax.annotation.Nonnull;

public abstract class ClusterEntrypoint implements
     AutoCloseableAsync, FatalErrorHandler {

   protected static final Logger LOG =
        LoggerFactory.getLogger(ClusterEntrypoint.class);
   private final Configuration conf;
   private final Thread hook;
   ...
   protected ClusterEntrypoint(Configuration conf){
     ...
     hook = SHU.addShutdownHook(...);
   }
   public void startCluster() throws
        ClusterEntrypointException {
     LOG.info("Starting {}.",
          getClass().getSimpleName());
     try { sContext.runSecured((Callable<Void>) ()
          -> { runCluster(configuration); ... });
       }
   }
   @Nonnull
   private Configuration
        generateClusterConfiguration
        (Configuration conf) {
      final Configuration result = new
           Configuration();
      ...
      return result;
   }
   ...}
```

Listing 2: Code snippet of ClusterEntrypoint class in flink-runtime:1.5.1

2.1. Artifact-level API dependency

Fig. 1 represents the dependency relationships between five artifacts. The *com.payneteasy.socket-nio:client:1.0-4* artifact declares dependencies towards two libraries: *com.google.code:findbugs:jsr305:13.9* and *org.slf4j:slf4j-api:1.7.1*. The *org.apache.flink:flink-runtime:1.5.0* artifact declares a dependency towards three libraries: *com.google.code:findbugs:jsr305:13.9*, *org.slf4j:slf4j-api:1.7.1* and *org.apache.flink:flink-core:1.5.0*.

The dependencies declared in the *pom.xml* file of each artifact are identified by their exact coordinates. For example, in Listing 1, the artifact *org.apache.flink:flinkruntime:1.5.0* declares a dependency towards *com.google.code.findbugs:jsr305:1.3.9* to reuse the javax annotations defined in this library. Consequently, when building the *flink-runtime* project, Maven will fetch the resource *jar* corresponding to *jsr305:1.3.9*, together with all its transitive dependencies, and add them to the project's classpath.

2.2. Code-level API dependency

There are many ways to use external APIs at the code level through inheritance, implementation, composition, genericity, static method invocation etc.. Listing 2 shows a snippet from the class *org.apache.flink.runtime.entrypoint.ClusterEntrypoint* of the library *org.apache.flink:flinkruntime:1.5.0* (Apache, 2019). It illustrates different ways in which *org.apache.flink:flinkruntime:1.5.0* uses some dependencies that are declared in its *pom.xml*. The class *ClusterEntrypoint* implements the *AutoCloseableAsync* class exposed by the *org.apache.flink:flinkcore:1.5.1* dependency (line 7), while lines 9 and 10 are examples of field declarations. On line 9 the dependency is used through a call to the static method *getLogger()*. Lines 22–24 illustrate reuse examples of API members such as annotations, methods call, or in methods signatures.

3. Methodology

In this section we present our research questions and the metrics we use to answer these questions. Then, we introduce the dataset for this study and the methodology we have used to collect API usages.

3.1. Research questions

This study is structured around the following research questions:

RQ1: How are the APIs used in the code of clients that declare a dependency towards them? In this work, we study how the APIs are used in the code of clients that declare a dependency towards them. Previous studies hint on the fact that some declared dependencies are actually never used, for example, because API users do not systematically maintain their build file (Constantinou et al., 2014; Zaimi et al., 2015). In this research question, we investigate to what extent there is a gap between what clients declare in their *pom.xml* and what they actually use in their code. We also investigate what causes these discrepancies.

RQ2: How is the usage frequency of API types distributed and how does that depend on the number of clients? API developers aim at providing reusable functionalities to a large number of clients. This desire to satisfy many users can become a double-edged sword from the users' perspective, which can be overwhelmingly loaded by a large number of API types that they do not need (Piccioni et al., 2013; Myers and Stylos, 2016). This research question focuses on the distribution of usage frequency of API types. We measure the share of API types that are never used by any clients, rarely used or used by most clients. We discuss how these shares vary depending on the number of clients. For this question we only consider Library Client relationship where the client uses at least one type of the API.

RQ3: How is the usage frequency of API types distributed when focusing on the popular version of an API? In this question, we focus on the most popular version of each library in our dataset to determine the effect of a "sufficient number of users" on API usage. This sheds new light on usages ratios, compared to RQ2 that considers all versions of the libraries. This question is at the core of our analysis of API usage with respect to Hyrum's law, which requires a large number of clients to study.

RQ4: Can inter-package calls explain the existence of API types that are unused by the clients? In Java, developers need to set the visibility of classes or methods to public if they want to allow inter-package usages. In other words, some parts of a library's API might be public only because they are intended to be used by other package of the library, and might not be meant to be reused by the library's clients. In this research question, we analyze whether inter-package usages indeed contribute to explain the existence of API types that are not used by the clients.

RQ5: How many API classes are essential for most of the clients? The usage of API types is demonstrated to be strongly

related to the needs of the clients (Sawant et al., 2018). In the long term, these needs determine what constitutes the essential part of an API. Here, we address the key intuition of this work: the existence of a *reuse-core* for the APIs, i.e. a set of highly used elements according to the clients' state of practice. In this research question, we investigate what proportion of the API is essential for the clients and how this reuse-core varies according to various API usages.

3.2. Metrics and definitions

For further references, we introduce the following notations:

- *library*: an artifact declared as a dependency by a *client*
- *types*(*library*): the set of distinct types that are visible for *client* elements, i.e. classes, interfaces, or annotations
- $types_{obs}(library)$ is the subset of *types*(*library*) used by at least one client
- `LIB` a set of libraries sharing the same groupId and artifactId, regardless of the version.
- *clients*(*library*): the set of clients that declare a dependency towards a *library*
- $clients_{obs}(library)$ is the set of *client* that call at least one element of $types_{obs}(library)$
- $clients_{obs}(type)$ is the set of *clients* that call at least one member, i.e. fields and public and protected methods, including constructors, of a given *type*

To answer RQ1, we measure the possible gap between clients that declare a dependency towards an API in their *pom.xml* and the ones that actually call this API at least once in their bytecode.

Metric 1. *The dependency usage rate (DUR) of a `LIB` is the proportion of clients that call at least one API member of a library ∈ `LIB`, (observed through static analysis), among all the clients that declare a dependency towards any version of `LIB`:*

$$DUR(\texttt{LIB}) = \frac{|\bigcup_{l \in \texttt{LIB}} clients_{obs}(l)|}{|\bigcup_{l \in \texttt{LIB}} clients(l)|}$$

RQs 2, 3 and 4 study what proportion of the clients of a library use each type of its API. We consider that a client uses a type if it uses at least one member of this type, i.e. $client \in clients_{obs}(type)$. We name this proportion type usage rate (TUR), and define it as follows.

Metric 2. *The type usage rate (TUR) of a given type ∈ library corresponds to the proportion of clients that reference at least one member of said type (observed through static analysis), i.e. $clients_{obs}(type)$, among the clients that actually use library, i.e. $clients_{obs}(library)$:*

$$TUR(type) = \frac{|clients_{obs}(type)|}{|clients_{obs}(library)|}$$

RQ5 investigates how necessary is each type of the API of the most popular version of each `LIB`. To assess this necessity, we adapt the concept of extinction sequence (Albert et al., 2000) to simulate the hiding of each type ∈ *types*(*library*) from the least used to the most used. We call $LU(library, n)$ the set of *n*% least used types in $types_{obs}(library)$.

Metric 3. *We measure the surviving client share (SCS) unaffected by the hiding of $LU(library, n)$.*

$$SCS(library, n) = \frac{\left|\left\{ c \;\middle|\; \begin{array}{l} \forall type \in LU(library, n), \\ c \notin clients_{obs}(type) \end{array} \right\}\right|}{|clients_{obs}(library)|}$$

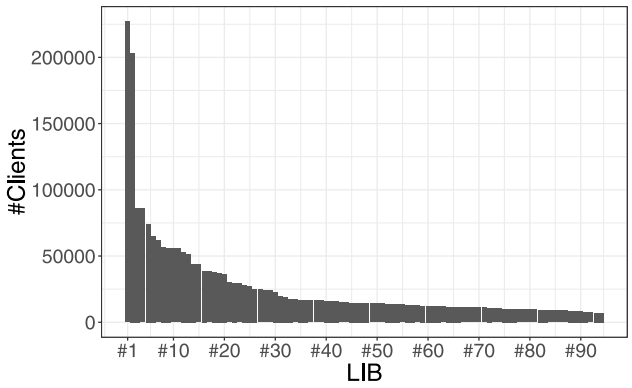


Fig. 2. Distribution of LIB number of clients.

To answer RQ5, we compute the variation of $SCS(library, n)$, where we vary *n* from 0% to 100%.

3.3. Dataset

In this work, we analyze software dependencies both at the artifact and at the code levels. At the artifact level, we analyze *pom.xml* files of client projects to determine the list of direct dependencies they declare. At the code level, we analyze the bytecode of both the clients and the libraries. On the client-side, we determine what parts of the libraries' API they actually use. On the library-side, we evaluate the extent to which an API is actually used by its clients. Hence, our dataset is composed of bytecode and pom file of both libraries and clients.

We leverage the Maven Dependency Graph (MDG) (Benelallam et al., 2019) to identify the most popular APIs in Maven Central, as well as their client artifacts. Then we extract usage information through static analysis of the *jar* artifacts. This section details these two steps.

The MDG captures all artifacts in Maven Central as nodes and their dependencies as directed edges. Every node has a *coordinates* property referring to the artifact's coordinates (GAV) and a packaging referring to the format of the artifacts binaries. Furthermore, every edge has a property *scope* identifying the dependency scope. We extract the 100 most popular libraries. We exclude 6 `LIBs` from these 100 libraries. One of them, `appcompat-v7`, is not packaged as a jar but as an apk. The 5 other ignored `LIBs` contain no type and so, no API usage can be observed in their clients. Three of these 5 libraries are written in Clojure, and two others, `spring-boot-starter` and `spring-boot-starter-web`, are packaged as jar files that contain no bytecode, i.e., there are no API types. In fact, these `LIBs` serve as an alias for a group of commonly used dependencies that are transitively inherited through a single entry-point. Hence, we study a set of 94 libraries. We compute the popularity of a `LIB` based on the number of distinct clients that declare a dependency towards a version of the `LIB` with a *Compile* scope.

The raw dataset for our study includes all dependency relationships from any *client* artifact, in Maven Central, towards any version of one of our `LIBs`. This represents 2,376,526 dependency relationships between 901,876 clients (belonging to 99,949 unique pairs *(GroupID, ArtifactID)*) and 94 `LIBs`. The `LIBs` are in a total of 5225 versions in the dataset.

Fig. 2 shows the distribution of the number of clients for each LIB. The two most popular libraries are the standard `scala-library` and `slf4j-api` with respectively 227, 014 and 203,366 clients. The number of clients per library decreases quickly in this ranking to reach 7007 clients for the least popular library

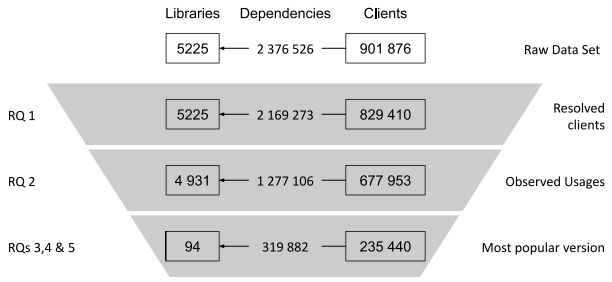


Fig. 3. Progressive data set filtering.

Table 1
Description of 6 illustrative library examples.

Library[a]	#Types	#Clients	#Clients$_{obs}$	Category
javax.inject:1	6	23,211	14,442	Extension
commons-cli:1.3.1	24	2557	2042	Utility
slf4j-api:1.7.21	38	31,752	21,398	Logging
junit:4.12	281	24,454	15,583	Testing
hibernate-core:4.3.11.Final	2746	539	453	ORM
commons-io:2.4	112	35,000	21,959	Utility

[a]For readability, we refer to a library using only its artifactId and version.

of our dataset, `xercesImpl`. These libraries cover a broad range of application domains, from logging, networking, language extensions, to collections and bytecode manipulation. Our dataset includes libraries from 15 of the 20 most popular categories of libraries from *mvnrepository.com*.[1] The only categories not covered are, two related to Android applications (since we exclude apk), two related to testing (as we exclude *test* dependencies), and a category related to web assets which do not contain bytecode.

As illustrated in Fig. 3, we filter our dataset through the research questions. For RQ1, we focus on the 2,169,273 dependencies concerning the 829,410 client artifacts that we could resolve (those for which we could download the jar). For RQ2, we focus on the dependencies for which we could observe an actual usage in the bytecode of the client. At this stage we exclude 2 LIB that do not contain public types. This represents 4931 libraries, 1,277,106 dependencies and 677,953 clients. In RQs 3, 4 and 5 we analyze the client–API dependencies for the most popular version of each library. This corresponds to 94 libraries, 319,882 dependency relationship and 235,440 unique clients. This latest version of the dataset supports our investigations of API usage with "a sufficient number of users", a key condition to study long tail distributions.

Given the large number of libraries and clients, the plots displayed in the section represent a lot of information, and it is sometimes difficult to keep the intuition between the data and the software engineering phenomena that are at stake. To keep the discussion concrete, we select 6 libraries that we use to illustrate all the research questions. Table 1 summarizes the name, the number of types, the number of clients, the number of clients that actually use the library and the application domain for these 6 libraries. We select these libraries because they represent a diverse set of domains, sizes, API types, and number of clients. We select the most used version of each `LIB`.

3.4. API usages collection

We collect the *jar* file of each version of each of our 94 `LIBs` from Maven Central and statically analyze it to extract all its API members. Then, we store this list of members in a relational database. Table 2 shows descriptive statistics about the APIs and clients for our study. The Libraries Overview part shows the number of API members (types, methods, fields) in our set of libraries, the number of dependencies declared towards these libraries and the number of distinct clients that declare these dependencies. The Libraries Members part details the distribution of the number of type definitions and the total number of methods and fields across the library APIs. The Libraries Types part distinguishes between different kinds of type definitions (classes interfaces and annotations) that we found in APIs. We provide a detailed description of these types since they will form the main granularity at which we analyze API usages. The smallest API in our dataset is the *javax.inject:javax.inject:1* library, which contains 1 interface and 5 annotations, of which, only one defines a default method.

In a second step, we collect, from Maven Central, the *jar* file of every single artifact that declares a dependency to at least one of the libraries in our dataset. The artifacts are resolved with Eclipse Aether (Eclipse, 2019), a Java library that fetches artifacts from remote repositories for local consumption. We analyze the bytecode of each of these clients, looking for local variables, fields, parameters, return types, annotations, type extensions or implementations that are referencing library types, including in lambda expressions. We also analyze invocations that target any element of the resolved API members. The bytecode analysis is implemented on top of ASM (OW2, 2019), a popular Java library for bytecode manipulation and analysis. The source code is available on GitHub (Harrand, 2019a). For each API usage, we count the number of times an element is referenced. The Clients Overview part of Table 2 gives the distribution of the number of API types used, as well as the number of dependencies declared by each client.

Table 3 is an excerpt of the database of API usages that we collected. This excerpt corresponds to the usages collected in the bytecode corresponding to the example in Listing 2. For example, this excerpt shows that the class *ClusterEntrypoint* references the class *Logger* of *slf4j-api:1.7.21* one time, and calls the method *Logger.info* 6 times.

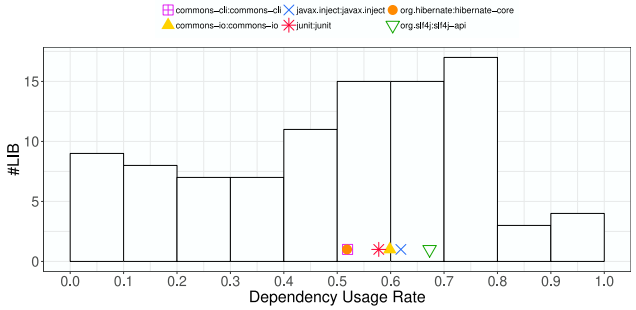


Fig. 4. Distribution of dependency usage rate (DUR) among the 94 `LIBs`. Each bin represents to the number of `LIBs` (y-axis) with a DUR belonging to range of the bin (x-axis).

4. Results

4.1. RQ1 How are the APIs used in the code of clients that declare a dependency towards them?

In this research question, we examine the cases where a client declares a dependency towards a library in its *pom.xml* file, but its bytecode does include any usage of the library's API. We measure the extent of the phenomenon and investigates its causes.

[1] https://mvnrepository.com/open-source.

Table 2
Descriptive statistics of libraries (GAV) and clients (GAV).

	Libraries Overview			Libraries Members		
	#Membs	#In. Dep.	#Dist-Clis.	#Types	#Meths	#Fields
Min.	8	0	0	6	1	0
1st Qu.	1102.00	11	6	101	889.50	46
Median	2333	56	21	221	1922	158
Mean	7895.59	479.13	89.64	662.07	6617.91	615.61
3rd Qu.	4813	261	67	458	4101.50	360
Max.	118 690	47 819	5375	10 256	108 117	13 682
Total	41,085,887	2,169,273	475,928	3,453,949	34,120,704	3,511,234
	Libraries Types			Clients Overview		
	#Classes	#Intfcs.	#Anns.	#Type Usgs.	#Out. Dep.	
Min.	0	0	0	0	1	
1st Qu.	58	8	0	1	1	
Median	145	29	0	5	2	
Mean	477.09	80.41	8.72	25.14	2.70	
3rd Qu.	310	66	9	23	3	
Max.	9200	930	98	15,379	45	
Total	2,964,707	442,157	47,085	21,268,765	2,169,273	

Table 3
The API usages collected in the code of Listing 2.

Library	Class	Member signature	#Calls
slf4j-api	org.slf4j.LoggerFactory	getLogger(Class;)Logger;	1
	org.slf4j.Logger	TYPE	1
		info(String;)V	6
		error(String;Throwable;)V	2
jsr305	javax.annotation.Nonnull	TYPE	1
	javax.annotation.Nullable	TYPE	2
	javax.annotation.concurrent.GuardedBy	TYPE	9

Fig. 4 shows a histogram with the distribution of dependency usage rate *DUR*(LIB) among our 94 LIBs. We compute the *DUR* for every single *library* in LIB. The leftmost bin includes nine LIBs for which less than 10% of their clients include at least one usage of the LIB's API, i.e. with *DUR* in [0, 0.1[. *org.apache.maven:maven-plugin-api* has the maximum rate, with 96.9% of its clients that use at least one element of its API. The median rate is 52.4%. No LIB is actually called by 100% of its clients.

`spring-boot-configuration-processor` is an example of extremely low *DUR* (0.2%). This LIB contains a set of annotations, as well as an annotation processor that can be used by IDEs to assist with the development of spring-boot applications.[2] According to the official documentation, in order to avoid shipping this dependency at runtime, it is recommended to declare it as `optional`. We suspect that most of the clients that do not mark it as *optional*, do so accidentally. There is however one single client, `spring-boot-security-saml`,[3] (across its 14 versions), which does use its API[4] to generate Markdown documentation based on the annotations provided by `spring-boot-configuration-processor`.

The group of LIBs with a *DUR* below 20%, is composed of two types of libraries. First, we find libraries that are meant to assist users at development time. For example, `spring-boot-configuration-processor` enable a developer to generate customized metadata based on annotations[5] to provide auto-completion and documentation. Other examples are `jaxb-core`, which is used to generate Java source code from XML files, and `lombok` that enriches the Java language with annotations that are used to generate boilerplate code. When these libraries are erroneously declared with the default scope (`compile` instead of `provided`), they are needlessly considered as a runtime dependency. Second, we distinguish libraries that are not supposed to be called directly by the client. Instead, they are used by other existing dependencies. For example, a client declares `slf4j` or `mysql-connector-java` as dependencies in order to let its other dependencies use different logging facades or database connectors.

The seven LIBs with the highest usage rate among their clients (*DUR*(LIB) > 80%), include the standard libraries for Scala and Kotlin, as well as other frameworks used as domain-specific languages. This latter category includes the `maven-plugin-api`, which provides a way for developers to create Maven plugins, and the Apache `camel-core` library, an integration framework for systems producing and consuming data.

The majority of LIBs, 58 out of the 94, have a *DUR* between 40% and 80%. The median *DUR* of the population is 53.1%. This indicates that, for common libraries of the ecosystem, slightly less than half of clients declare a dependency towards a library and do not make any direct static call to it. For instance, among the 79,364 clients that declare a dependency towards a version of *commons-io*, only 47,495 (59.8%) refer to an element of its API in their bytecode. Similarly, the *DUR* of *slf4j-api* is 67.3%. This corresponds to 117,692 clients of 174,895 containing calls to *slf4j-api* in their bytecode.

Discussing the root cause of unused dependencies:

We distinguish two common situations where a dependency is declared but not used by a client. First are declared dependencies that ended up in the *pom.xml*, most likely, by accident; either through an ingenuous copy-and-paste or inherited from an earlier version of the client's *pom.xml* where it was actually used. This hypothesis is consistent with the observations of McIntosh and colleagues (McIntosh et al., 2014) who found that build files are more prone to clones than other software artifacts. Take the *javax.inject* for example, which has a dependency usage rate of 61.9%, slightly above the median. Since this library contains only 5 annotations and one interface, it is unlikely that any client has used it through reflection. Moreover, the fact that it has only one

[2] https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/reference/htmlsingle/#configuration-metadata-annotation-processor.

[3] https://github.com/ulisesbocchio/spring-boot-security-saml.

[4] https://github.com/ulisesbocchio/spring-boot-security-saml/blob/master/spring-boot-security-saml/src/main/java/com/github/ulisesbocchio/spring/boot/security/saml/util/ConfigPropertiesMarkdownGenerator.java.

[5] https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/reference/htmlsingle/#configuration-metadata-annotation-processor.

version (*javax.inject:1*) excludes the hypothesis that this dependency is used to prevent versions conflict. This leaves us with two plausible explanations for the 38.1% of unused dependencies: (1) forgetting to update the *pom.xml* and removing unused dependencies during maintenance, or (2) a simple copy-and-paste of an existing *pom.xml*. A living example of the latter hypothesis is the multi-module Maven project *com.eurodyn.qlack2.fuse* where all the modules that declare a dependency to *javax.inject* use at least one API member, except *qlack2-fuse-file-upload-rest*. This module contains only one type (Eurodyn, 2019) that does not import nor use any member of *javax.inject* API. In this case, it is safe to suggest that this dependency was copied and pasted from another module at the time it was created.

Another common reason for clients to declare a dependency U without actually using it is to expose it in the classpath so that another one of its dependencies D can use U. Two mechanisms support this: either U *shadows* another dependency of D and consequently, when the client is built, D uses U instead of the other dependency; or D declares U as an *optional* dependency, which is enforced as soon as the client of D declares U as a dependency. Classpath shadowing is commonly used by the clients of the `slf4j` logging framework. *netty*, a framework for building asynchronous network applications, declares *javassist* as an *optional* dependency, to accelerate encoding/decoding methods. *async-http-client*[6] depend on *netty* and declares *javassist* as dependency to improve *netty*'s performances[7] but no call to *javassist*'s API are present in *asynch-http-client*'s bytecode.

We replicated the study for this research question with another static analysis tool (based on the Apache `maven-dependency-analyzer`[8]), on a subset of the client–library dependencies (Soto-Valero et al., 2021). This study unveiled a similar ratio of unused dependencies: 44.2% (19,673 out of 44,488) of direct dependencies declared in *pom.xml* files were not followed by any static usage of the API of the dependency. This result is consistent with the observation of our current work, i.e., 41,13% of declared dependencies do not translate in an actual usage of the dependency.

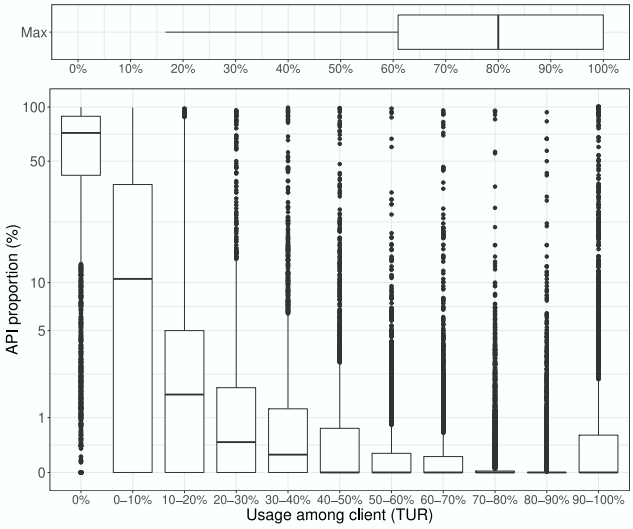
Answer to RQ1: 892,167 out of 2,169,273 dependencies declared in the clients *pom.xml* files are not used (41,13%). We observe three main causes: some libraries are not meant to be used directly; developers mistakenly define the scope of the dependency; a *pom.xml* file is hardly maintained and cleaned. These observations indicate the need for better support to analyze and maintain build files.

4.2. RQ2: How is the usage frequency of API types distributed and how does that depend on the number of clients?

In this research question, we study how client usages of APIs are distributed across its types. We focus on the client–library relationships for which we are able to observe at least one usage of the library on the client's bytecode. This represents 1,277,106 dependency relationships between 677,953 unique clients, and 4931 unique libraries (58.87% of the dependencies in our dataset).

Fig. 5b shows the distribution of usage rates of API types for of all 4931 libraries used by at least a client. The *x*-axis represents the Type Usage Rates (*TUR*) grouped in 11 categories. The first category is for types having a *TUR* that is equal to zero, while the remaining categories are grouped by 10% ranges, the lowest

(a) Distribution of the share of client using the most used type in the API (Maximum TUR).



(b) Distribution of type usage rates (TUR) of API types

Fig. 5. Distribution of type usage rates of API types of all 4931 libraries used by at least a client.

bound excluded. The *y*-axis represents the proportion of types in each library that falls into each category, from 0% to 100%, on a logarithmic scale. This figure is read as follows: The first column on the left represents the distribution of the share of API being used by none of its clients, the second column to the left shows the distribution of the share of API being used by more than 0% but less than 10% of its clients, and so forth.

First, let us analyze the share of API types for each library being used by no client (leftmost boxplot). The first quartile of this distribution is 41.7%, its median is 71.8%, and its third quartile is 88.9%. This means that for 50% of libraries, more than 71.8% of the API types are used by no client in our dataset.

On the opposite side of the figure, the rightmost column shows that the proportion of API types used by more than 90% of clients is greater than 0.6% for 25% of libraries. This hints the existence of a handful number of API types in each library that are used by a vast majority of the clients. We investigate extreme usages further with Fig. 5a by showing the distribution of the share of clients using the most used type of each API. The first quartile of this distribution is 60.9%, it is median 80.0%, and its third quartile is 100%. This means that for more than 25% of the libraries, there is at least one type used by all clients. Whilst, for more than 75% of them, there is at least one API type used by more than 60.9% of the clients.

These two observations, a small number of API types used by many clients and a large portion of types used by no clients, are consistent with observations in previous work (Sawant and Bacchelli, 2017; Qiu et al., 2016).

Now, let us analyze the second leftmost boxplot. It captures the proportion of API types that are rarely used. These types are used by one client at least, but no more than 10% of the clients. We observe that this category, $TUR \in]0, 10\%[$ exhibits a large variability: the first quartile is 0%, the third quartile is 37%, the maximum is 99.4% and the median is 10.5%. This large variability, similar to the leftmost boxplot, suggests two phenomena. First, some libraries have a very large portion of types that are rarely used, a phenomenon that has not been observed previously. Second, these large variabilities might come from large variations in the number of clients for each library.

6 https://github.com/AsyncHttpClient/async-http-client/blob/77714b5215afd670d7ca6cd698de21596a0606de/providers/netty4/pom.xml.

7 https://github.com/AsyncHttpClient/async-http-client/issues/430.

8 http://maven.apache.org/shared/maven-dependency-analyzer.

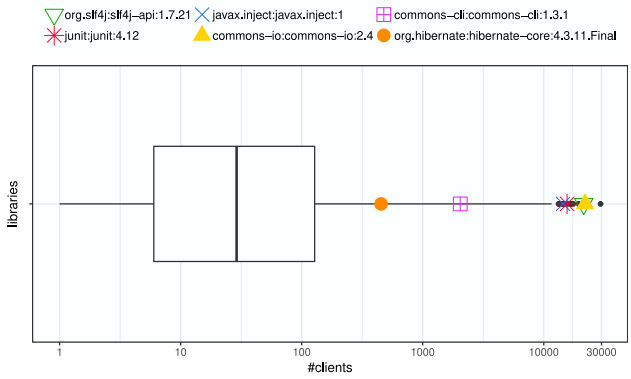


Fig. 6. Distribution of the number of clients per library.

Fig. 6 is a boxplot representing the distribution of the number of clients of each library. The *x*-axis is the number of clients on a logarithm scale. The plot indicates the quartiles of the distribution. For instance, 365 of the 4805 libraries have exactly 1 client. 75% of libraries have more than 6 clients, 50% have more than 29 and 25% have more than 268. The maximum number of clients observed is 29,466 for `scala-library:2.11.8`.

We observe a strong negative correlation (-0.26, p-value $< 2.2e-16$) between the number of clients that use a library and the share of the API types that are unused. Furthermore, we observe an even stronger positive correlation (0.34%, p-value $< 2.2e-16$) between the number of clients using an API and the share of its types that is rarely used (<10%). This means that libraries with few clients tend to have a large ratio of unused API types. When the number of clients increases, the ratio of unused API types decreases in favor of rarely used types. These observations hint that the number of library clients matters when studying the API usage. The next research question investigates this phenomenon further, with a focus on the most used version of each `LIB`.

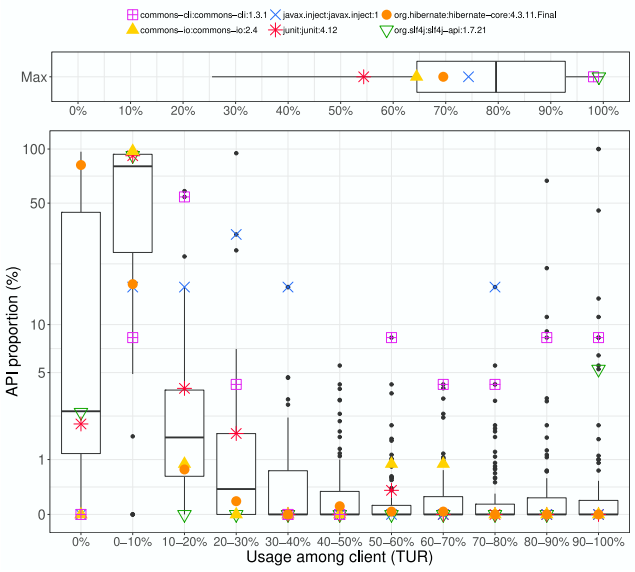
Answer to RQ2: All libraries include a handful of API types that are used by a vast majority of their clients. Meanwhile, 50% of libraries include more than 71.8% of types that are used by no client. These observations confirm previous studies observing that most API clients focus their usage on a small part of the public types. The rate of unused API types varies significantly depending on the number of clients, which motivates a detailed analysis of the most popular library versions.

4.3. RQ3: How is the usage frequency of API types distributed when focusing on the popular version of an API?

In this section we focus on the most popular version of each of the 94 `LIBs` and their 235,440 clients. The goal is to analyze the distribution of API usages in the presence of a sufficient number of clients. We investigate in particular how the share of API types used by no clients, and few clients, change compared with library with fewer clients.

Fig. 7b shows the distribution of the *TUR* of API types, focusing on the most used version of each `LIB` (instead of aggregating all versions as in Fig. 5b). The *x*-axis shows categories based on the percentage of clients using each part of an API. The *y*-axis represents the share of API types used by a certain ratio of the API clients, on a logarithm scale. The first column shows the distribution of the share of API types of our 94 libraries used by no client. The second column shows the distribution of the share of API types used by at least one client but less than 10%.

(a) Distribution of the share of client using the most used type in the most used version of each `LIB`.



(b) Distribution of usage rates of API types of the most used version of each `LIB`.

Fig. 7. Distribution of usage rates of API types of the most used version of each `LIB`.

Overall, we notice that both distributions in Figs. 7b and 5b share the same general characteristic: the proportion of library types decreases while increasing the *TUR*. Meanwhile, we notice some key differences. First, we remark that the proportion of types used by absolutely no client drastically decreases when focusing on the most popular versions of the `LIB`, with a median value at 2.6% (while it was 71.8% when considering all the versions of the `LIB`). Second, we observe that the proportion of API types used by less than 10% of the clients has increased, with a median value of 80.2%. We observe that with a *sufficient number of users*, for most libraries, the share of API types used by no client falls drastically. With a sufficient number of clients, the share of unused API types decreases to the profit of the share of API types that are rarely used.

The third interesting difference while considering the most popular version is about the usage distribution of the most popular type (box plot on top of Fig. 7b): the median does not change, but the quartile values do. Precisely, 64% to 93%, instead of 61% to ~100%. This is consistent with the increase of the quartile values in the categories [70%, 80%[and [80%, 90%[and the decrease of the quartile values [90%, 100%]. The maximum is usage rate is not 100% anymore, since, with a sufficient number of clients, it is less likely to have all them use the same single type.

Overall, the distribution of the share of clients using the most used type of an API, as well as the share of API types used by more than 50% of clients, indicates the existence, in most libraries, of a small compact subset of APIs being used by most client. This is consistent with previous work (Qiu et al., 2016; Lämmel et al., 2011; Thummalapenta and Xie, 2008).

Here we analyze our illustrative examples in details, and check if our assumption that with enough client all API types end being used by at least one. *commons-io:2.4* is a good example: it exposes 112 API types, it is used by 21,959 clients and there is no type that is used by no client. We can observe on Fig. 7a, that the most used type of *commons-io:2.4* is `IOUtils` used by only 64.5% of its clients. This is lower than 75% of libraries in our dataset.

Our assumption holds for regular APIs such as *commons-cli:1.3.1*, *javax.inject:1* and *slf4j-api:1.7.21*, which can be partly explained by the small number of types they offer for reuse (resp. 6, 24 and 38). While our assumption holds, *commons-io:2.4* and *slf4j-api:1.7.21* also have a large share of types that are rarely used (in the]0, 10%] category), indicating a large diversity of usage profiles.

The case of *junit:4.12* is distinct from the other examples. Our assumption globally holds for this library, since only 6 of the 281 public types are not used. The distinctive feature with respect to API usage appears in the boxplot at the top of Fig. 7a: the most used type of *junit:4.12*, (*org.junit. Assert*), is used by only 54.4% of its clients. This singular case can be explained by the fact that version 4.x of junit contains both a new API (including the type *org.junit. Assert*), and the API of version 3.x for backward compatibility reasons (including a type *junit.framework. Assert*).

hibernate-core:4.3.11.Final, our 6th example, is a counter-example. It exposes 2746 types, the version we analyze is used by 453 clients, but 81.8% of its types are never used.

Answer to RQ3: Focusing on the most popular version of each LIB, we confirm that, in the Maven ecosystem, with a sufficient number of clients, only a very small share of the API types are never used. Meanwhile, we observe a new phenomenon: a large part of API types are used by less than 10% of clients (median proportion of types used by less than 10% of clients is 95,00%).

4.4. RQ4: Can inter-package calls explain the existence of API types that are unused by the clients?

Java imposes a design constraint on multi-package libraries: a class member must be publicly visible in order to be used by another class, from another package, inside the library. Yet, once a class is public, it is not possible to limit the visibility boundaries to only the packages of the library. Once a class is visible beyond its package boundaries, it is accessible to the rest of the world. Even though, several different conventions can inform a library user that a public type is meant for internal usage, such as naming the package `internal` or annotating the type as such, non is enforced. Therefore, one could argue that some types are public only to be used internally by the library itself, which could explain a part of the API types that are not used by the clients. If this was true, we would observe that the types that are not used by external clients are actually used through internal calls. Here we investigate this hypothesis and its consequences on the results presented above.

We consider the most popular version of each LIB. For each library, we distinguish between the types that are used by one client at least and the types that are used by no client. For each category of type, we measure the share that is used through inter-package calls inside the library.

The boxplot at the top of Fig. 8 is the distribution of intra-library usages for types that are used by at least one external client. One point on this plot corresponds to the proportion of types of one library that are used by at least one client of this library and that are also used inside the library.

The boxplot at the bottom of Fig. 8 is the distribution of intra-library usages for types that are used by no client. One point here is the proportion of types of library used by no client but used internally. In this boxplot, 12 libraries, that have no public type that is used by no clients, do not appear. Among our examples, *commons-cli:1.3.1*, and *javax.inject:1* are not on the lower line of the plot since they do not have any unused public type and are single package library.

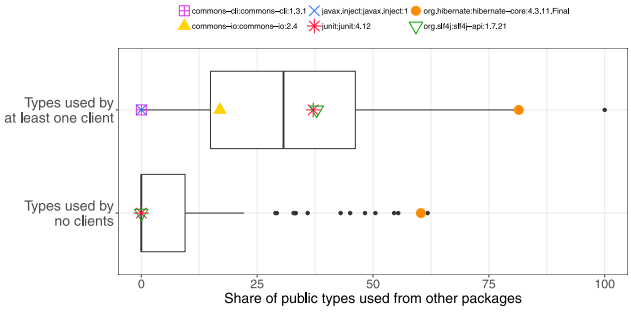


Fig. 8. Distribution of inter-package usage rates of API types of the most used version of each LIB.

Furthermore, 9 libraries have only 1 package which makes their share of types used from other packages equal to 0% (for both lines).

The comparison of these two distributions informs us that, not only types used by no clients are not more likely to be used by another package of the same library, but in fact the opposite is true. The median inter-package usage rate for types used by at least a client is 30.7% while it is 0% for types used by no clients at all. Furthermore, a $t-test$ rejects that their mean is the same with $p\text{-}value < 0.001$. The bottom plot reveals that for more than half of the libraries, no unused type is used by other packages of the library. For all these libraries, the types that are publicly visible are not public to allow internal usages, but most probably to be used by the clients of these libraries. We also observe that 90% of the libraries have less than 23% of unused types that are used by another package of the library. This consolidates the observation that API members are not made public for inter-package usages. In other words, types used internally by a library, are not less popular among its clients. Consequently, Fig. 8 shows that the declaration of some types as public to allow their internal usage, cannot explain the majority of the unused public types.

Answer to RQ4: The existence of public types that are not used by the clients of a library is not explained by the Java constraint of setting a type as public for internal usages. Based on this new observation and on the results of RQ3, we can conclude that, as soon as a developer sets an element as public, it will likely be used by some client, given a sufficient number of clients, regardless of the developer's initial intentions.

4.5. RQ5: How many API classes are essential for most of the clients?

In this last research question, we explore how the long tail distribution of API usages can be navigated. We demonstrate that, if developers are willing to satisfy a majority of their API clients (and not all of them), then they can still identify a small core of API types on which they can focus their efforts and eventually apply the good practices from the literature. For example, testing, and documentation efforts can be focused on the small core of API types that are the most used without alienating a large amount of clients. Similarly, automated library manipulation such as specialization or automated migration can support only this core while supporting the majority of the population of clients.

The dataset for this question includes the most popular version of each LIB and their clients. The API of each library is reduced to API types that are used by at least one client.

Fig. 9 represents the distribution of extinction sequences for the 94 libraries in our dataset. We simulate these extinction

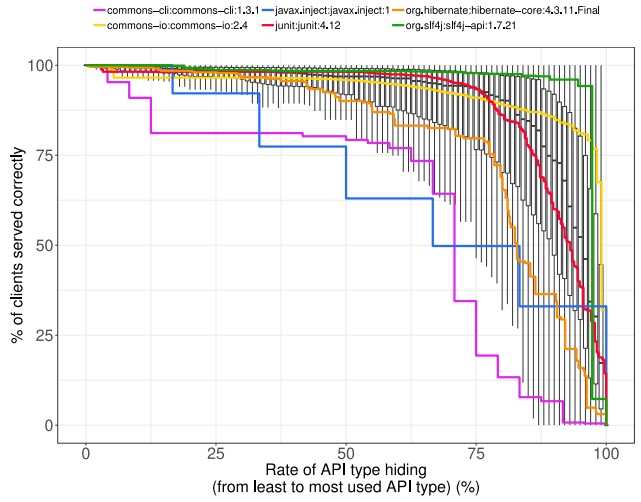


Fig. 9. Distribution of extinction sequences for API types. Each sequence simulates the hiding of API types from the least to the most used type. Colored lines show the extinction of 6 libraries, while the boxplot represents the distribution of all 94 most used version. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

sequences by hiding types of their API, from the least popular one to the most popular one. At every step of this process, we observe the share of client that are no longer able to use the API. The *x*-axis shows the share of hidden API types (from 0% to 100%). Given a share of hidden API types, the *y*-axis indicates the share of clients that can still access all the types they need. In the rest of this section, we say that these clients are correctly served. The colored lines represent the extinction sequences of our 6 case studies, while the boxplots represent the distributions for the population of 94 artifacts. All extinction sequences start with 0% of API types being hidden, and 100% of clients correctly served. They all end with 100% of API types hidden and 0% of clients served.

The yellow line represents the extinction sequence for *commons-io:2.4*. We observe that it is possible to hide a large part of the types while correctly serving the vast majority of the clients. The big drop occurs when simulating hiding the 3 last types of the API: `FilenameUtils`, `FileUtils`,`IOUtils`. 14 819 out of the 21 959 clients (67.5%) only use either `FileUtils`, `IOUtils` or both, and 7039 (32%) use only `IOUtils`, the most used types of the API.

The blue line represents the extinction sequence for *javax.inject:1*. This API has only comports 6 types (5 annotations and an interface) and all of them are used by the clients. Consequently, each simulation of type hiding corresponds to a sharp decrease of correctly served clients. The least popular type is `Scope`, which is still is used by 1129 of the 14 442 clients (7.8%). About half of 14 442 clients (49.8%) use only one or both of the most popular types of the API: `Singleton` and `inject`.

commons-cli:1.3.1's extinction sequence is represented in purple. The API of *commons-cli:1.3.1* is another example of a rather small API (24 types). 64.3% of its client (1313 out of 2042) use the 9 most used types: we observe a steep drop when removing the ninth most used type (`OptionGroup`). The rest of the sequence presents an unusual shape because the most popular type is mostly used in conjunction with one other of the popular types. With only 10 clients using **only** the most used type `Options`.

slf4j-api:1.7.21's extinction sequence is represented in green. It is one of the most extreme cases in our dataset. The API contains 37 types, but 20 164 of the 21 398 clients (94.2%) use only two types: `LoggerFactory` and `Logger`. As only 1560 clients use only `Logger`, the sharpest drop in correctly served clients is caused by the removal of `LoggerFactory`. This occurs because most clients use these two types in conjunction. The rest of the API provides more advanced logging options that only few clients use. Fig. 10 illustrates this singular situation. This figure shows a chord diagram representing types of the API of *org.slf4j:slf4j-api:1.7.21* and its clients. Nodes on the upper par represents API Types with a size proportional to the number of clients using them. The lower part represents three groups of clients (with a proportional size): in red, clients only using the two most popular types (`Logger` and `LoggerFactory`), in blue, clients that do not use any of these two types, and in yellow other clients.

The orange line represents *hibernate-core:4.3.11.Final*'s extinction sequence. It simulates the progressive hiding of the 507 types of its API used by at least one client. This sequence informs us that the 397 least used types of the API may be hidden without affecting more than 75% of the clients.

The boxplots of Fig. 9 represent the distributions of extinction sequences for all 94 libraries in our dataset. The median values show that for half of libraries, 88% of the API types or more can be hidden, while leaving more than 75% of clients unaffected. This illustrates the implication of the long tail distribution of API usages. There is a small set of features used by most client, and the rest is used but rarely. Similarly, the series first quartiles of these distributions show that for three quarters of libraries 77% or more of the API types may be hidden without affecting at least 75% of clients.

All APIs include a small set of features that serve a vast majority of the clients. This confirms that API developers who are willing to ignore a minority of clients can indeed focus their maintenance, documentation and development efforts on the small subset of the API that is the most used.

> **Answer to RQ5:** With enough clients, most API types are used, while most clients can be served successfully with a small subset of API types. In particular, for more than half of the libraries in this study, 88% of the API types or more can be hidden while leaving more than 75% of clients unaffected. If API developers are willing to ignore a minority of clients, they can focus their maintenance, documentation and development effort on the small subset of the API that is the most used. It also opens opportunities to automate library migration, only supporting a limited part of API targeted while supporting a large majority of clients using it.

5. Discussion

In this section, we reflect on how our observations could hold in other dependency ecosystems. We articulate this reflection around four characteristics of our dataset and how they influence our observations.

5.1. Source code language for clients

All artifacts in our dataset are Java bytecode. Yet, the source code may vary (Java, Scala, Kotlin, Groovy). We analyzed the clients implemented mostly in Scala, to check if a client's source code language affects the usage of external APIs. We choose Scala because it is the most popular language of the ecosystem, aside from Java, according to the popularity of its standard library.

To assess if clients implemented in Scala use APIs differently than those written purely in Java, we select all the clients that declare a dependency (excluding test scope) towards any version of the Scala standard library (*org.scala-lang:scala-library*. Let us note that these clients might also use other languages, e.g., Java.

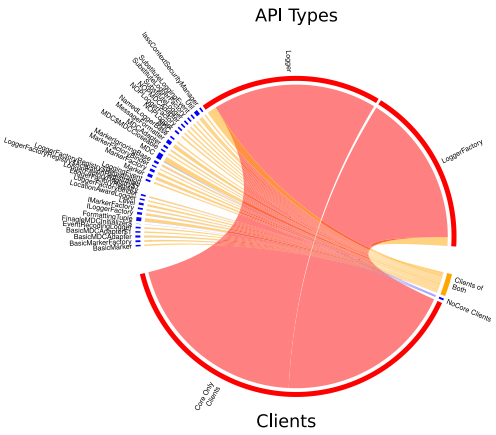


Fig. 10. Chord diagram representing the bipartite graph of the *org.slf4j:slf4j-api:1.7.21* API types and its clients. Nodes on the upper part represent API Types, with a size proportional to the number of clients using them. The lower part represents three groups of clients (with a proportional size): in red, clients only using the two most popular types (`Logger` and `LoggerFactory`), in blue, clients that do not use any of these two types, and in yellow other clients. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

This subset of clients represents 31.4% of the dataset used for RQ2 (212 844 artifacts). Then, we select libraries which API is called both by clients using Scala and clients that do not include Scala. This represents 1322 out of the 4931 clients of our dataset.

We perform a Welch test to determine if the average number of types of the API called by Scala clients is significantly different from the one called by clients not using Scala. While the average number of types used by clients using Scala (12.6) is slightly higher than the one of client not using Scala (11.8), we cannot reject the hypothesis that these means are equal (p-value 0.45). In other words, clients using Scala do not use significantly more types of an API than other clients.

5.2. Build tool

Our observations about unused dependencies (Section 4.1) are affected by the build system used to compile and test a client. Indeed, different languages and associated build tools implement different policies regarding external API usages. In RQ1 and in our subsequent studies (Soto-Valero et al., 2021) we observed that Maven can build and package projects that include libraries that are not used. Meanwhile, the Go compiler does not compile programs which declare unused imports,[9] and widely used tools such as goimports[10] can refactor imports of a go program to remove unused ones.

The ecosystem of Go libraries likely behaves differently than the Java ecosystem, regarding our observations in RQ1. We can therefore speculate that the different practices enforced by the build tools impact the distribution of API usages, as studied in this work.

5.3. API size

Our dataset shows a loose correlation (0.21, p-value 0.04) between the *core-index* of a library and the size of its API (number of public types). Consequently, variations in the range of sizes for a given set of APIs are likely to change the range of core sizes and the core-index.

In our dataset, the 3 libraries that are written in Scala (`scala-library`, `scala-compiler`, and `scala-reflect`), have very large APIs in terms of number of bytecode types, in part because the Scala compiler generates types to implement various Scala language constructs. This tends to exacerbate the importance of unused or rarely used types.

The range of API sizes varies in other ecosystems. For example Abdalkareem et al. (2017) observe the increasing popularity of *trivial packages* in the npm ecosystem, i.e. packages that contain less than 250 LOC. These packages represent 16.8% of the npm population in 2017. Meanwhile, 90% of our artifacts have more than 147 accessible elements. This difference in API sizes between npm and Maven influences the observations about the core of APIs: it is very likely that the core of npm libraries is proportionally larger than in Maven.

In summary, the trends in size of APIs vary in different ecosystems. Yet, these variations should not affect the existence of a core set of API members that are used by a majority of clients.

5.4. Update frequency and interconnection

The number of clients for a given library in a specific version influences the shape of the extinction sequence and the number of used API members. The frequency of updates impacts this number of clients: a popular and stable library attracts more and more clients over time, while libraries that update very frequently have a scattered population of clients over multiple versions (Soto-Valero et al., 2019). Indeed, it is common for client projects to keep outdated dependencies (Kula et al., 2018; Bogart et al., 2016). Consequently, variations in update frequency have an impact on the shape of the Core.

Decan et al. (2018) study 7 software ecosystems (Cargo, CPAN, CRAN, npm, NuGet, Packagist and RubyGems), describing the variations in update frequencies, as well as how interconnected the ecosystems are. They observe that, while all ecosystems grow over time, some also increase in ratio of dependency over artifact. This indicates that in these ecosystems, dependencies are more and more interconnected. They also observe that, across all the studied ecosystems, a small number of artifacts concentrate most of the usages by others. Furthermore, this inequality seems to increase over time. In RQ2 we show how a large number of clients tends to imply no or few unused parts of APIs (confirming Hyrum's law), but does not change the fact that in general a small part of APIs concentrate most usages. Based on the observations of Decan about various update frequencies in different ecosystems, it is likely that the relative size of APIs with no observed client, will likely vary in from one ecosystem to another.

5.5. The notion of Core outside Maven

While our dataset focuses on artifacts from Maven Central, we are confident that the results would be similar on other Java corpora. We refer to previous works with other dataset to elaborate on the generalizability of our findings. Qiu et al. (2016) analyzed 5000 projects mined from GitHub and observed that API usages follow Zipf's law. This is consistent with our results, and implies that extinction sequences would produce similar results on their corpus.

Lämmel et al. (2011) perform API usages analyses similar to ours, based on the SourceForge ecosystem. Their dataset includes 6286 clients for which they mine API usages. Several of their findings align with ours and comfort our results. (i) The 4 most used external APIs of their dataset are libraries providing XML parsing, Logging and Testing functionalities, and are also in our dataset. (ii) The number of clients using popular libraries follow a

[9] https://golang.org/doc/faq#unused_variables_and_imports.
[10] https://godoc.org/golang.org/x/tools/cmd/goimports.

similar decreasing exponential (See Fig. 2). (iii) Based on the number of clients in their dataset, Lammel and colleagues conclude that most APIs are not well covered by usages, which does not contradict our observations on libraries with a limited number of clients. (iv) In their case studies, they observe rare projects that largely cover APIs and many projects using only a limited subset of APIs. This is consistent with our observations. Furthermore, the low cumulative coverage they observe is consistent with our observation that most clients focus their API usages on a small subset of APIs.

When it comes to other software ecosystems, they vary in their custom values and policies (Bogart et al., 2016), which may change the exact values obtained. Our study focuses on the Maven ecosystem and JVM-based code artifacts. We acknowledge that key variations in other ecosystems require replications with other data to understand to what extent our observations about the existence of a reuse-core generalize.

6. Related work

Several existing works have investigated the usage of APIs in open-source projects and industrial applications. In this section we discuss the related work along the following aspects.

API usage in practice. Several studies have focused on understanding how developers actually make use of APIs on a daily basis (Roover et al., 2013; Blom et al., 2013; Bauer et al., 2014). Some of the motivations include improving API design (Myers and Stylos, 2016) and increasing developers' productivity (Lim, 1994). Qiu et al. (2016) present empirical evidence showing that a considerable proportion of API members are not widely used, i.e., many classes, methods, and fields of popular libraries have never been used. They have found that, on a corpus of 5000 projects, API usage distribution follows a power law, which is consistent with our findings. Sawant and Bacchelli (2017) propose a tool to mine API usages and evaluate it on a dataset of project mined on GitHub using 5 popular Java APIs. They study how the small set features truly used is often introduced in early version of an API. Pham et al. (2016) implement a bytecode based analysis tool to learn API usages of Android frameworks. Their approach is intended to automatically generate recommendations for incomplete API usages, and thus reducing API usage errors and improving code quality. While their dataset covers one application domain, in our paper, we analyze clients of libraries serving different domains. Kula et al. (2018) observe that even if dependency usage is common, maintenance operations on dependencies such as keeping them up to date is often not prioritized. To our knowledge, none of the previous studies has performed on a population as significant as ours, nor proposed the concept of extinction sequences in this context.

Lämmel et al. (2011) perform a large-scale study on API usage analysis based on AST elements migration. This is the work that is the most closely related to ours. Yet it differs in several important aspects. The size and origin of the dataset: we studied a corpus of more than 800,000 clients from Maven Central, i.e. compiled projects. They studied the sources of 6286 projects from SourceForge. We build our dataset by choosing the most popular libraries and then resolve all the clients of those libraries in the ecosystem. We discuss other topics such as bloated dependencies and propose the use of extinction sequences to describe API usages.

API recommendation and comprehension. As open-source software projects continuously grow both in quantity and complexity, recent research has paid special attention to understanding these large systems by studying API properties (Zheng et al., 2011). In particular, API recommendation systems based on usability (Stylos and Myers, 2008), diversity (Mendez et al., 2013), and stability (Raemaekers et al., 2012) have been proposed. Steidl et al. (2012) present an approach based on network metrics to retrieve central classes on large software systems. While this approach relies on internal usages (i.e. classes within the same project) to determine the central classes, in our approach, we rely on external usages. Thummalapenta and Xie (2008) present a tool that detects hotspots and coldspots of eight widely used open-source frameworks. Their tool is integrated as an Eclipse plugin and aim at helping users of APIs to discover their relevant parts. Duala-Ekoko and Robillard (2012) conducted a study about the common questions that programmers ask when working with unfamiliar APIs. Horvath et al. (2019) mine client usage Apache Beam to study how developers discover functionalities of the API. They observe a long tail distribution of API usages, which is consistent to our observations. Our work expands the existing knowledge in the area by characterizing the essential API elements based on the clients' usages, which becomes a valuable criterion to reuse functionalities, i.e., following the wisdom of the crowd.

Software dependency ecosystems. During the last decade, researchers have investigated the dependency relationships in software packaging ecosystems (Mancinelli et al., 2006; Pashchenko et al., 2018; Soto-Valero et al., 2019). In particular, research efforts focus on the study of library evolution (Decan et al., 2018), updating behaviors (Raemaekers et al., 2017) and the security risks (Zapata et al., 2018). Bogart et al. (2016) highlight the different values and customs of different software ecosystems. Raemaekers et al. (2013) constructed a Maven dataset of 148,253 *jar* files for analyzing the evolution of API members based on code metrics. Gabel and Su (2010) perform a study on the uniqueness of source code showing that most existing code is reused code. Unlike previous work, our study focuses on the analysis of API usages to characterize the reuse-core of API types.

7. Threats to validity

We report about internal, external, and construct threats to the validity of our study.

Internal validity. This study relies on a very rich and complex network of software artifacts. The complexity is such that we could not completely resolve the artifacts captured in the MDG (Benelallam et al., 2019). Indeed, the MDG contains a minority of artifacts, hosted on other repositories than Maven Central. For network reasons, e.g. download limitations, some artifacts could not be resolved. In total, we resolved 829,410 of the 901,876 artifact (91.84%), which corresponds to 2,169,273 dependency relationships (91.78%). Our analysis covered 87,207,807 usages of 5,076,307 different API elements. We believe that the results obtained with this large set of APIs and clients represent a good approximation of how clients use popular libraries.

External validity. Our findings might not generalize to all Java APIs. We selected the 94 LIBs based on their popularity and on the popularity of Maven Central. We also noticed that these APIs cover a variety of usage domains (e.g., collections, logging, XML parsing). As Maven Central is a collection of opensource components,[11] they may not behave as pure end-applications. All the client artifacts in our dataset are artifacts from this repository. This means that the exact sets of types included in the $Core_n$ of the libraries studied in this work could be different when observing a different set of clients. Yet, both Qiu et al. (2016), and Lämmel et al. (2011), observe usages distributions that consistent with ours, on a dataset of Java applications mined respectively on GitHub and SourceForge. Consequently, we are

[11] https://central.sonatype.org/pages/about.html#what-is-the-central-repository.

confident about the relevance of our study subjects and the scale of their dependency relationships.

Construct validity. The main threat to construct validity is related to the limitations of static analysis, which may fail to capture dynamic calls from the users to some API members. Reflection and libraries handling dependency injection such as spring-boot, or OSGI plugins allow clients to use API members through dynamic calls. Reif and colleagues recently studied the impact of Java dynamic features, which are not soundly handled by static analysis, in the context of call-graphs construction (Reif et al., 2019). While the empirical evidence show that many projects do use reflection, the prevalence of reflection (proportion of methods that do use it) in their call graph is limited. None of the forms of reflection (namely Trivial Reflection, Locally Resolvable Reflection, and Context-sensitive Reflection) is present in more than 0.47% of the methods in their Top50Maven Corpus. Consequently, the presence of reflection constructs among client libraries has a minimal impact on our observations.

Reliability. The code to query the Maven Dependency Graph, collect both libraries and client artifacts, and analyze the usages, developed for this study may contain bugs. To limit this threat, 3 researchers were involved in the iteration of development, analysis of the results, and manual review of data points. We also made our infrastructure publicly available for further replication (Harrand, 2019a). Finally, in order to advocate for open-science, we made all the data used in this study publicly available online (Harrand, 2019b).

8. Conclusion

In this paper we study the long tail nature of client–API usages. We perform a systematic empirical analysis of 2,169,273 dependencies that are declared by 829,410 client artifacts towards the 94 most popular libraries available in Maven Central.

A novel result is the observation that most of the API types are used by one client at least, when considering the most popular version of an API. For more than half the top 94 in Maven Central, less than 2% of types are used by no clients of the repository at all. This original result sets an antecedent to further explore Hyrum's law about behavior usage. It is interesting to note that this new observation does not contradict with the state of the art: our analyses also confirms that most APIs have a small number of types that are used by the vast majority of their clients. For more than half of the API, only 12% of types are necessary to serve 75% of the clients. This means that API developers can focus their effort on maintenance and documentation in order to best serve the majority of their clients.

We envision two main threads of future works. First, we wish to explore novel ways of designing public Java APIs in order to reduce the number of types exposed to clients. This may be addressed by the feature of *modules* introduced in Java 9. Second, we wish to leverage the existence of a core set of API types as an instrument to build adapters between APIs that provide similar features, focused on the subset of most used API elements. This is motivated by the growing challenges of dependency management and the need to abstract dependencies from their concrete implementation in order to address these challenges (Cox, 2019).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program, by the TrustFull project financed by the Swedish Foundation for Strategic Research.

References

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E., 2017. Why do developers use trivial packages? An empirical case study on npm. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, pp. 385–395. http://dx.doi.org/10.1145/3106237.3106267.

Albert, R., Jeong, H., Barabási, A.-L., 2000. Error and attack tolerance of complex networks. Nature 406 (6794), 378–382.

Apache, 2019. Flink. https://github.com/apache/flink/releases/tag/release-1.5.1 (Accessed: 2019-06-30).

Apache, 2020. Maven scope documentation. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope (Accessed: 2020-07-13).

Bartolomei, T.T., Czarnecki, K., Lämmel, R., Van Der Storm, T., 2009. Study of an API migration for two XML apis. In: International Conference on Software Language Engineering. Springer, pp. 42–61.

Bauer, V., Eckhardt, J., Hauptmann, B., Klimek, M., 2014. An exploratory study on reuse at google. In: Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices. In: SERIP 2014, ACM, New York, NY, USA, pp. 14–23. http://dx.doi.org/10.1145/2593850.2593854, URL: http://doi.acm.org/10.1145/2593850.2593854.

Benelallam, A., Harrand, N., Soto-Valero, C., Baudry, B., Barais, O., 2019. The maven dependency graph: a temporal graph-based representation of maven central. In: 16th International Conference on Mining Software Repositories. In: MSR 2019, ACM, New York, NY, USA, pp. 1–4. http://dx.doi.org/10.1145/2597073.2597097, URL: http://doi.acm.org/10.1145/2597073.2597097.

Blom, S., Kiniry, J., Huisman, M., 2013. How do developers use apis? A case study in concurrency. In: Proceedings of the 2013 18th International Conference on Engineering of Complex Computer Systems. pp. 212–221. http://dx.doi.org/10.1109/ICECCS.2013.39.

Bogart, C., Kästner, C., Herbsleb, J., Thung, F., 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 109–120. http://dx.doi.org/10.1145/2950290.2950325.

Constantinou, E., Ampatzoglou, A., Stamelos, I., 2014. Quantifying reuse in OSS: A large-scale empirical study. Int. J. Open Source Softw. Process. 5 (3), 1–19. http://dx.doi.org/10.4018/IJOSSP.2014070101.

Cox, R., 2019. Surviving software dependencies. Commun. ACM 62 (9), 36–43.

Decan, A., Mens, T., Grosjean, P., 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir. Softw. Eng. 1–36. http://dx.doi.org/10.1007/s10664-017-9589-y.

Duala-Ekoko, E., Robillard, M.P., 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 266–276.

Eclipse, 2019. Aether. https://projects.eclipse.org/projects/technology.aether (Accessed: 2019-06-30).

Eghan, E.E., Alqahtani, S.S., Forbes, C., Rilling, J., 2019. API Trustworthiness: an ontological approach for software library adoption. Softw. Qual. J. http://dx.doi.org/10.1007/s11219-018-9428-4.

Eurodyn, 2019. FileUploadRestTemplate Class of Qlack2. https://github.com/eurodyn/Qlack2/blob/340c3874eeba6b433b5b612b06f1ab7911857156/Fuse/qlack2-fuse-faile-upload/qlack2-fuse-file-upload-rest/src/main/java/com/eurodyn/qlack2/fuse/fileupload/rest/FileUploadRestTemplate.java (Accessed: 2019-06-30).

Gabel, M., Su, Z., 2010. A study of the uniqueness of source code. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE '10, ACM, New York, NY, USA, pp. 147–156. http://dx.doi.org/10.1145/1882291.1882315, URL: http://doi.acm.org/10.1145/1882291.1882315.

Harrand, N., 2019a. Replication package for the empirical investigation of an API core. https://github.com/castor-software/core-83 (Accessed: 2019-06-30).

Harrand, N., 2019b. Zenodo. https://zenodo.org/record/2567268 (Accessed: 2019-06-30).

Horvath, A., Grover, S., Dong, S., Zhou, E., Voichick, F., Kery, M.B., Shinju, S., Nam, D., Nagy, M., Myers, B., 2019. The long tail: Understanding the discoverability of api functionality. In: 2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, pp. 157–161.

Hyrum Wright, 2019. The hyrums law. http://www.hyrumslaw.com (Accessed: 2019-06-30).

Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K., 2018. Do developers update their library dependencies? Empir. Softw. Eng. 23 (1), 384–417. http://dx.doi.org/10.1007/s10664-017-9521-5.

Lämmel, R., Pek, E., Starek, J., 2011. Large-scale, AST-based API-usage analysis of open-source java projects. In: Proceedings of the 2011 ACM Symposium on Applied Computing. In: SAC '11, ACM, New York, NY, USA, pp. 1317–1324. http://dx.doi.org/10.1145/1982185.1982471, URL: http://doi.acm.org/10.1145/1982185.1982471.

Lim, W.C., 1994. Effects of reuse on quality, productivity, and economics. IEEE Softw. 11 (5), 23–30. http://dx.doi.org/10.1109/52.311048.

Mancinelli, F., Boender, J., Cosmo, R.D., Vouillon, J., Durak, B., Leroy, X., Treinen, R., 2006. Managing the complexity of large free and open source package-based software distributions. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). pp. 199–208. http://dx.doi.org/10.1109/ASE.2006.49.

Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., Nystrom, N., 2015. Use at your own risk: The java unsafe API in the wild. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. In: OOPSLA, ACM, New York, NY, USA, pp. 695–710. http://dx.doi.org/10.1145/2814270.2814313, URL: http://doi.acm.org/10.1145/2814270.2814313.

McIntosh, S., Poehlmann, M., Juergens, E., Mockus, A., Adams, B., Hassan, A.E., Haupt, B., Wagner, C., 2014. Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In: Companion Proceedings of the 36th International Conference on Software Engineering. In: ICSE Companion 2014, ACM, New York, NY, USA, pp. 145–154. http://dx.doi.org/10.1145/2591062.2591181, URL: http://doi.acm.org/10.1145/2591062.2591181.

Mendez, D., Baudry, B., Monperrus, M., 2013. Empirical evidence of large-scale diversity in API usage of object-oriented software. In: Proceedings of the 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 43–52. http://dx.doi.org/10.1109/SCAM.2013.6648183.

Myers, B.A., Stylos, J., 2016. Improving API usability. Commun. ACM 59 (6), 62–69.

OW2, 2019. ASM Bytecode manipulation. https://asm.ow2.io (Accessed: 2019-06-30).

Pashchenko, I., Plate, H., Ponta, S.E., Sabetta, A., Massacci, F., 2018. Vulnerable open source dependencies: Counting those that matter. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. In: ESEM '18, ACM, New York, NY, USA, pp. 42:1–42:10. http://dx.doi.org/10.1145/3239235.3268920, URL: http://doi.acm.org/10.1145/3239235.3268920.

Pham, H.V., Vu, P.M., Nguyen, T.T., et al., 2016. Learning API usages from bytecode: a statistical approach. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 416–427.

Piccioni, M., Furia, C.A., Meyer, B., 2013. An empirical study of API usability. In: Proceedings of the 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 5–14. http://dx.doi.org/10.1109/ESEM.2013.14.

Qiu, D., Li, B., Leung, H., 2016. Understanding the API usage in java. Inf. Softw. Technol. 73, 81–100.

Raemaekers, S., van Deursen, A., Visser, J., 2012. Measuring software library stability through historical version analysis. In: Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM). pp. 378–387. http://dx.doi.org/10.1109/ICSM.2012.6405296.

Raemaekers, S., van Deursen, A., Visser, J., 2013. The maven repository dataset of metrics, changes, and dependencies. In: Proceedings of the 10th IEEE Working Conference on Mining Software Repositories. In: MSR 2013, ACM, IEEE, San Francisco, CA, USA, pp. 221–224.

Raemaekers, S., van Deursen, A., Visser, J., 2017. Semantic versioning and impact of breaking changes in the maven repository. J. Syst. Softw. 129, 140–158. http://dx.doi.org/10.1016/j.jss.2016.04.008, URL: http://www.sciencedirect.com/science/article/pii/S0164121216300243.

Reif, M., Kübler, F., Eichberg, M., Helm, D., Mezini, M., 2019. Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 251–261. http://dx.doi.org/10.1145/3293882.3330555.

Roover, C.D., Lämmel, R., Pek, E., 2013. Multi-dimensional exploration of API usage. In: 2013 21st International Conference on Program Comprehension (ICPC). pp. 152–161. http://dx.doi.org/10.1109/ICPC.2013.6613843.

Sawant, A.A., Bacchelli, A., 2017. Fine-GRAPE: fine-grained API usage extractor – an approach and dataset to investigate API usage. Empir. Softw. Eng. 22 (3), 1348–1371. http://dx.doi.org/10.1007/s10664-016-9444-6.

Sawant, A.A., Robbes, R., Bacchelli, A., 2018. On the reaction to deprecation of clients of 4 + 1 popular java apis and the JDK. Empir. Softw. Eng. 23 (4), 2158–2197. http://dx.doi.org/10.1007/s10664-017-9554-9.

Soto-Valero, C., Benelallam, A., Harrand, N., Barais, O., Baudry, B., 2019. The emergence of software diversity in maven central. In: 16th International Conference on Mining Software Repositories. In: MSR 2019, ACM, New York, NY, USA, pp. 1–10. http://dx.doi.org/10.1145/2597073.2597097, URL: http://doi.acm.org/10.1145/2597073.2597097.

Soto-Valero, C., Harrand, N., Monperrus, M., Baudry, B., 2021. A comprehensive study of bloated dependencies in the maven ecosystem. Empir. Softw. Eng. 26 (3), 45. http://dx.doi.org/10.1007/s10664-020-09914-8.

Steidl, D., Hummel, B., Juergens, E., 2012. Using network analysis for recommendation of central software classes. In: Proceedings of the 19th Working Conference on Reverse Engineering. In: WCRE 2012, ACM, IEEE, Kingston, Canada, pp. 93–102.

Stylos, J., Myers, B.A., 2008. The implications of method placement on API learnability. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: SIGSOFT '08/FSE-16, ACM, New York, NY, USA, pp. 105–112. http://dx.doi.org/10.1145/1453101.1453117, URL: http://doi.acm.org/10.1145/1453101.1453117.

Thummalapenta, S., Xie, T., 2008. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 327–336. http://dx.doi.org/10.1109/ASE.2008.43.

Zaimi, A., Ampatzoglou, A., Triantafyllidou, N., Chatzigeorgiou, A., Mavridis, A., Chaikalis, T., Deligiannis, I., Sfetsos, P., Stamelos, I., 2015. An empirical study on the reuse of third-party libraries in open-source software development. In: Proceedings of the 7th Balkan Conference on Informatics Conference. In: BCI '15, ACM, New York, NY, USA, pp. 4:1–4:8. http://dx.doi.org/10.1145/2801081.2801087, URL: http://doi.acm.org/10.1145/2801081.2801087.

Zapata, R.E., Kula, R.G., Chinthanet, B., Ishio, T., Matsumoto, K., Ihara, A., 2018. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 559–563.

Zheng, W., Zhang, Q., Lyu, M., 2011. Cross-library API recommendation using web search engines. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. In: ESEC/FSE '11, ACM, New York, NY, USA, pp. 480–483. http://dx.doi.org/10.1145/2025113.2025197, URL: http://doi.acm.org/10.1145/2025113.2025197.