Information and Software Technology 73 (2016) 81–100

Contents lists available at ScienceDirect

Information and Software Technology

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

Understanding the API usage in Java

Dong Qiu[a], Bixin Li[a,*], Hareton Leung[b]

[a] School of Computer Science and Engineering, Southeast University, Nanjing, China
[b] Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong

ARTICLE INFO

Article history:
Received 7 September 2015
Revised 25 January 2016
Accepted 25 January 2016
Available online 3 February 2016

Keywords:
API usage
Empirical study
Java

ABSTRACT

Context: Application Programming Interfaces (APIs) facilitate the use of programming languages. They define sets of rules and specifications for software programs to interact with. The design of language API is usually artistic, driven by aesthetic concerns and the intuitions of language architects. Despite recent studies on limited scope of API usage, there is a lack of comprehensive, quantitative analyses that explore and seek to understand how real-world source code uses language APIs.

Objective: This study aims to understand how APIs are employed in practical development and explore their potential applications based on the results of API usage analysis.

Method: We conduct a large-scale, comprehensive, empirical analysis of the actual usage of APIs on Java, a modern, mature, and widely-used programming language. Our corpus contains over 5000 open-source Java projects, totaling 150 million source lines of code (SLoC). We study the usage of both core (official) API library and third-party (unofficial) API libraries. We resolve project dependencies automatically, generate *accurate resolved* abstract syntax trees (ASTs), capture used API entities from over 1.5 million ASTs, and measure the usage based on our defined metrics: *frequency, popularity* and *coverage*.

Results: Our study provides detailed quantitative information and yield insight, particularly, (1) confirms the conventional wisdom that the usage of APIs obeys Zipf distribution; (2) demonstrates that core API is not fully used (many classes, methods and fields have never been used); (3) discovers that deprecated API entities (in which some were deprecated long ago) are still widely used; (4) evaluates that the use of current compact profiles is under-utilized; (5) identifies API library coldspots and hotspots.

Conclusions: Our findings are suggestive of potential applications across language API design, optimization and restriction, API education, library recommendation and compact profile construction.

1. Introduction

Syntax and semantics define a programming language. Application Programming Interfaces (APIs) facilitate its use. Most of today's software projects heavily depend on the use of API libraries [1]. They improve code reuse, reduce development cost and promote programmers' productivity. However, API design has been artistic and biased, driven by aesthetic concerns and the intuitions of API designers. They usually have limited knowledge on how programmers actually use the API, which leads to many unnatural and rarely used API features being introduced, while not some expected ones [2,3]. Meanwhile, the ever-growing APIs (increasing features have been introduced) remain a significant barrier to novice programmers [4]. In addition, API libraries have become one of the most influential factors for the choice of programming languages [5]. Poor design of the APIs increases the learning curve for developers and greatly influence their productivity. Therefore, it is significant to understand the actual usage of the current API libraries, and optimize the designs to promote API *usability* for programmers.

Studying how a large number of real-world programs use APIs can help validate or disprove the many popular "theories" concerning what APIs are most adopted, most useful, easiest to use; whether APIs have been fully used by the programmers, *etc.* that abound concerning programming in popular literature and on the Internet. For language education, the gap between APIs and their actual usage may guide pedagogy, giving teachers insight into what is common (and perhaps should be) and rare (and perhaps should not be). It also guides novice programmers to select a proportionally smaller fraction, *i.e.* most essence of the entire APIs to reduce the cost of learning. Language API designers may leverage data on

* Corresponding author. Tel.: +86 25 52090877; fax: +86 25 52090879.
E-mail addresses: dongqiu@seu.edu.cn (D. Qiu), bx.li@seu.edu.cn (B. Li), hareton.leung@polyu.edu.hk (H. Leung).

http://dx.doi.org/10.1016/j.infsof.2016.01.011
0950-5849/

actual API usage to optimize the design of API libraries, *e.g.* simplifying unpopular APIs and identifying unused APIs that could be eliminated. In addition, API usage analysis is crucial in mining API usage patterns [6–9], and offers supports for API migration [10,11]. It also produces a positive effect in software maintenance [12].

To this end, we perform a large-scale empirical study on a diverse corpus of over 5000 real-world Java projects to gain insight into how APIs are used in practice. We retrieve project dependencies with the aid of Maven [13], generate *accurate resolved* abstract syntax trees (ASTs) for approximately 150 million SLoC, capture used API entities (*i.e.* packages, classes, methods and fields) from over 1.5 million ASTs, and measure the usage based on our defined metrics: *frequency* (whether an API has been frequently used), *popularity* (whether an API has been widely used) and *coverage* (whether an API has been fully used). We analyze almost all the API libraries that are adopted by practical projects, including both core API and third-party APIs. Besides, we investigate some extra issues, *e.g.* construction of API subsets and selection of the versions of the third-party APIs. In summary, this paper makes the following contributions:

- It presents a large-scale, comprehensive, empirical analysis of the use of APIs in a modern programming language, namely Java;
- This is the first work to deeply study both core API and third-party APIs, including the use of deprecated API entities. It is also the first to study how API usage guide the design of the compact profiles (*i.e.* subset of APIs);
- Some interesting results are demonstrated: (1) 1% of the most-used packages account for 80% of all API usage, while 70% least-used packages are used $<$ 0.5% of all API usage and 50% only $<$ 0.1%; (2) 15.3% of the classes, 41.2% of the methods and 41.6% of the fields from the core API are never used; (3) 9.5% of the packages have all subordinative methods never used and 29.2% of the classes have all subordinative methods never used; (4) 51.1% of deprecated classes, 43.5% of the deprecated methods and 18.1% of the deprecated fields from the core API have been adopted.

Taken together, our results permit API designers to empirically consider whether the design of the API facilitates programmers' development based on their actual usage. Our study also identifies both *hotspots* (*i.e.* frequently and widely used APIs) and *coldspots* (*i.e.* rarely and narrowly used APIs) to inform programmers to selectively learn and adopt the APIs. For example, if the APIs are never used, alerting programmers to use them cautiously in practical development is indispensable. In addition, the results assist to construct appropriate subsets of the APIs, that can be employed in either resource-constrained devices or high security environment. We believe that our work enables data-driven language API design, optimization and simplification, analogous to how Cocke's study at IBM in the 1970s on the actual usage of CISC instructions eventually led to the RISC architectures [14].

2. Methodology

This section first discusses the research questions studied, presents the basic information of the corpus used in this study then, and illustrates the process of how we set up and perform the experiments.

2.1. Research questions

The goal of this study is to answer the key research question: *How programming language APIs are used in real open-source projects*. To better investigate the question, we focus on the following dimensions:

Table 1
An overview of the Java corpus.

Corpus summary	
Repository	Github
No. of projects	5185
No. of files	1,595,600
Source lines of code	152,341,840
No. of imports	12,518,834
No. of class use	75,076,400
No. of field use	34,149,616
No. of method use	59,225,800
No. of unique class use	2,034,177
No. of unique field use	5,031,510
No. of unique method use	5,403,540

Global view of API usage. Most of current software projects heavily depend on the use of API libraries. Understanding the API usage *provenance* can provide an overview of API use distribution, *i.e.* how much of the API entities are reused from existing APIs (core APIs or third-party APIs) and how much of them are designed and created specific to projects. We are also interested in investigating how much of the API entities are adopted to construct a project in general and further validate whether the scale of the software is correlated with the API usage. In addition, we desire to confirm the conventional wisdom that the use of API entities obeys Zipf distribution.

Core API usage. Core API library is essential API that facilitates the use of the programming language, which is ordinarily developed by official organizations which maintain such programming language (*e.g.* Java SE Development Kit, *i.e.* JDK from Oracle [15]). However, as new features have been introduced increasingly, the scale of the core API library is growing rapidly, consuming more resources for devices and increasing the learning curve for novice programmers. It is significant to understand the *utilization* of the core API, *i.e.* whether all API entities from the core library have been fully used. The introduction of a new concept, *compact profiles*, which are subsets of the entire core API, motivates us to inspect the *utilization* of compact profiles analogously. In addition, identifying *hotspots* and *coldspots* can be suggestive of optimizing the design of current core APIs and guiding novices to learn the essence preferentially. We also investigate the use of *deprecated* API entities.

Third-party API usage. Third-party API libraries are supplements to the core API library, providing extra functionalities that are not supported by the core API or analogous functionalities with preferable implementations. Many of them are developed and maintained by reputable commercial companies (*e.g.* `guava` from Google) or open-source communities (*e.g.* `commons-*` from Apache). We are interested in investigating how heavily a project depends on third-party APIs, *i.e.* how much of third-party API libraries are required to construct a project in general. In addition, a library is available in multiple versions. It would be interesting to figure out how many distinct versions a typical library has in general. It is also significant to investigate how programmers select and adopt concrete versions.

2.2. Gathering the corpus

Our large-scale corpus consists of 5185 (including over 1.5M Java files and 15M non-comment lines of code) open-source and real-world Java projects whose source code is available from Github, one of the most popular repository hosting services. We rigorously select applications based on the *popularity* by synthetically considering their size of *watchers, stars* and *forks* provided by Github. Table 1 lists the corpus summary information. The corpus is diverse, covering various application domains and size. It

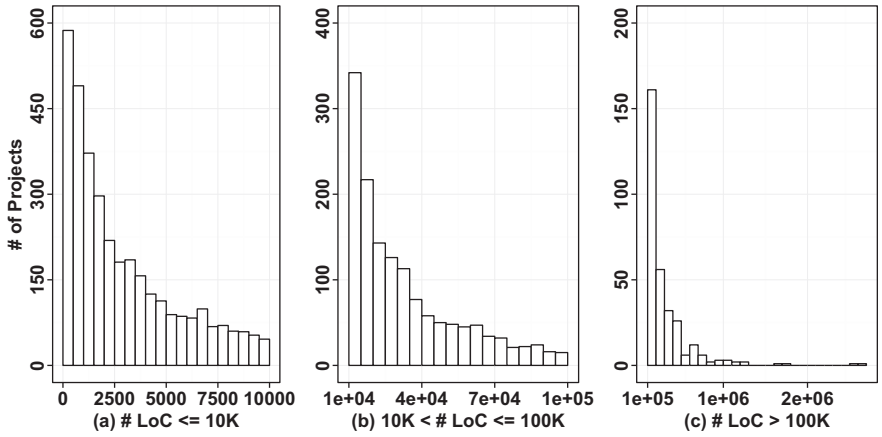


Fig. 1. Distribution of project size in the corpus.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Fig. 2. An XML snippet shows how Maven manage dependencies.

contains not only widely-used Java projects maintained by the reputable open-source communities (*e.g.* Tomcat, Hadoop, Derby from the Apache Software Foundation and JDT, PDT, EGIT from the Eclipse Foundation), but also relatively small projects developed by indie programmers. Fig. 1 gives an indication of the distribution of the application sizes, measured in terms of non-comment SLoC. All applications are managed by *Maven* [13], one of the most commonly-used build management systems in the open-source communities. Our corpus was constructed at the end of 2014 where all applications were checked out from Github between 2014/12/29 and 2014/12/31.

2.3. Resolving dependencies

Most software systems heavily depend on API libraries [1]. The process of compiling and building a project will fail if its dependent API libraries are missing, which probably results in the failure of correctly resolving API entities employed by this project. Distinct from the strategy adopted by Lämmel et al. that manually resolved the required dependencies through web search and download [16], we employ one of the most popular and widely-used software project management and comprehension tool, *Maven*, to handle the dependency management automatically [13]. It supports transitive dependency resolution *i.e.* calculating the closure of relevant dependent libraries that current dependency requires, and retrieve all of them automatically. Employed dependencies within the projects are usually specified in the maven configuration file *pom.xml* (POM), associated with a collection of remote repositories where dependencies can be retrieved. The dependency is uniquely identified by its coordinate `groupId:artifactId:version`, analogous to an integrated address and timestamp. The coordinate is determined when the dependency is created as a project. Fig. 2 shows an example that involves dependency `junit:junit:4.0` in projects. The property `scope` within the definition refers to the classpaths of the task under distinct environments (*e.g.* compile, test, runtime and *etc.*). In this study, we concentrate on the extraction and analysis of the dependencies in POM. In total, 103,256 dependencies are retrieved.

2.4. Collecting API usage

An API, in programming languages, usually refers to a set of exposed features or functions that facilitate the software reuse for programmers. In the context of Java, an API is usually expressed as a collection of pre-written *classes*,[1] associated with their respective *fields* and *methods*. Interrelated classes are customarily organized by *packages* that can provide access protection and namespace management. In practice, an API is normally related to a reusable software library, thus facilitating code reuse. In this study, we directly regard a Java API as a collection of Java API entities, including *packages, classes, fields* and *methods*. The term *API usage* refers to how programmers adopt given API libraries, *i.e.* applying the concrete API entities into their source code. Hence, we link API usage to the actual use of packages, classes, methods and fields. Table 2 lists all typical scenarios of API usage that we capture in this study, associated with corresponding examples. The use of a package is aggregated by the use of its declared classes, methods and fields. One remarkable thing is that API entities may be homonyms: identical lexemes with distinct effects on behavior. As an example, the method named `get()` can be located in the class `java.util.List` or `java.util.Set`. We employ the resulting type information and obtain their *fully qualified* name, *i.e.* `java.util.List.get()` and `java.util.Set.get()` in this case, to distinguish them. In another case (see the code below),

```
Object o = new Object(); o.toString().toString();
```

the first method invocation `toString()` on object *o* is resolved to `java.lang.Object.toString()` while the second `toString()` is resolved to `java.lang.String.toString()`. Based on the resolved dependencies in Section 2.3, we can generate *resolved* ASTs from source code, attached with *accurate* type bindings for all API entities.

In general, the employed API entities originate from two sources: external API libraries and internal APIs designed specific to an individual project. The former refers to the reuse of the existing libraries, consisting of the core API library and third-party

[1] The concept of *class* is generalized. In Java, it refers to the reference types, including normal classes, interfaces, enumerations and array types.

Table 2
The scenarios of API usage captured in this study.

	Usage scenarios	Examples	Resolved API entities
Class use	Class instance creation[a]	`List<String> i =newArrayList<>();`	*java.util.List*
	Variable declaration	`String s = ''abc'';`	*java.lang.String*
	Static class use	`intmax = Math.max(1,2);`	*java.lang.Math*
	Inheritances (extends/implements)	`public classmyThread extendsThread {...}`	*java.lang.Thread*
	Parameters	`public booleanmatches(String regex){...}`	*java.lang.String*
	Return types	`publicDate getTime(String time){...}`	*java.util.Date*
	Annotations	`@Override`	*java.lang.Override*
	Exceptions	`try{...}catch(IOException e) {...}`	*java.io.IOException*
	Generics	`List<String> i =newArrayList<>();`	*java.lang.String*
Method use	Instance method invocation[b]	`intlen =newString(''a'').length();`	*java.lang.String.length()*
	Static method invocation	`intmax = Math.max(1,2);`	*java.lang.Math.max(int, int)*
	Constructor invocation	`String s =newString();`	*java.lang.String.String()*
Field use	Instance field access[c]	`Point p =newPoint(); p.x=1;`	*java.awt.Point.x*
	Static field access	`System.out.println(''Hello World!'');`	*java.lang.System.out*

[a] Only explicitly declared superclasses are considered. Any inherited class, which is not shown in the declaration, is not captured, *e.g.* `java.util.Collection` in this case.
[b] It also includes the method invocation within its declaring class, *e.g.* `public void m1(){...} public void m2(){m1();...}`.
[c] It also includes the field access within its declaring class. *e.g.* `this.age = 10;`.

libraries; the latter refers to the implementation of extra project-specific functions. In this paper, we stipulate that: (1) the official JDK [15] is set as the core API; (2) the other API libraries externally imported by a project are set as the third-party APIs; (3) API entities that are defined within a project belong to the project-specific API. In the discussion of API usage *provenance*, we capture the use of all API entities that programmers employ in their development, including project-specific ones. Our aim is to figure out: How much of the API are reused and how much are written by programmers for specific purposes? When we make a further investigation on the usage of the core and third-party API libraries, only *accessible* API entities (*i.e.* specified by the modifier *public* in Java) are under our consideration.

2.5. Metrics

Three categories of metrics are introduced to measure the API usage: *popularity*, *frequency* and *coverage*. First, we formalize these measures. Considering an API library x, $P_l(x)$ denotes the set of packages defined in x. We also use $C_l(x)$, $M_l(x)$ and $F_l(x)$ to denote the set of classes, methods and fields in x respectively. For any package $y \in P_l(x)$, $C_p(y)$, $M_p(y)$ and $F_p(y)$ are used to represent the set of classes, methods and fields defined in y respectively. For any class $z \in C_l(x)$, $M_c(z)$ and $F_c(z)$ are used to represent the set of methods and fields defined in z respectively. Hence, we have

$$C_l(x) = \bigcup_{\forall y \in P_l(x)} C_p(y)$$

$$M_l(x) = \bigcup_{\forall y \in P_l(x)} M_p(y) = \bigcup_{\forall z \in C_l(x)} M_c(z)$$

$$F_l(x) = \bigcup_{\forall y \in P_l(x)} F_p(y) = \bigcup_{\forall z \in C_l(x)} F_c(z)$$

Our corpus is a set of software projects: $C = \{S_1, S_2, S_3, \ldots\}$. When c_i is a class, $U_c = \{c_1, c_2, c_3, \ldots\}$ is the set of classes adopted in C. We use $O_c(S)$ to denote the multiset of classes employed in software project S. We let m_X denote the multiplicity function of the multiset X; the multiplicity $m_{O_c(S)}(c)$ returns the multiplicity, *i.e.* the occurrence of class c. We elide X, when its binding is clear from context. $U_c(S)$ denotes the set that underlies $O_c(S)$ whose indicator function returns 1 for every class in $O_c(S)$ with multiplicity > 0, *i.e.* the set of unique classes used by S. Likewise, we use $O_p(S)$, $O_m(S)$ and $O_f(S)$ to represent the usage of packages, methods and fields respectively, and $U_p(S)$, $U_m(S)$ and $U_f(S)$ to represent their underlying unique set, correspondingly. Then, we illustrate the three categories of metrics in detail, which are also shown in Table 3.

Popularity. Calculating the popularity of an API library (or entity[2]) is to evaluate whether it is widely-used across the community, *i.e.* adopted by as many projects as possible. Within our corpus, we employ the metrics $NP(x)/RP(x)$, *i.e.* the Number/Ratio of the Projects that use a particular API library (or entity) to measure its popularity.

Frequency. Calculating the frequency of an API entity is to evaluate whether it is used frequently in practice. Regarding an API entity x, we count its Simple Occurrence ($SO(x)$, which does not include the occurrence of x's subordinative entities) and Cumulative Occurrence ($CO(x)$, which includes the $SO(x)$ values of x's subordinative entities). When calculating the $CO(x)$ value of a class, both the $SO(x)$ values of itself and its methods and fields defined in this class should be counted. Regarding the follow code,

```
System.out.println(''Hello World!'');
```

SO_c(`java.lang.System`) = 1 while CO_c(`java.lang.System`) = 2, as the use of its declared field `java.lang.System.out` should also be counted. Besides, we use $PSO(x)$ or $PCO(x)$, *i.e.* average $SO(x)$ or $CO(x)$ value per Project, to measure the "local" frequency of x within a single project that employs it.

Coverage. Calculating the coverage of an API library (or entity[3]) is to evaluate whether it is fully used by programmers. Regarding the Class Coverage $Cov_c(x)$ (*i.e.* the coverage of a single class), we calculate the ratio of covered *public* methods (or fields) that are used at least once *w.r.t.* the total *public* methods (or fields) defined in class x. Similarly, we use the ratio of covered *public* classes (or methods and fields) to estimate the Package Coverage $Cov_p(x)$ (*i.e.* the coverage of a single package). Regarding the Library Coverage $Cov_l(x)$ (*i.e.* the coverage of an API library), the ratio of covered *public* packages (or classes, methods and fields) can be adopted. Higher coverage of an API indicates a higher API utilization.

2.6. Tool support

We have developed a tool called, Java API Usage Extractor (*JAPIExtractor*), that collects, manages and analyzes the API usage

[2] The entity can be a package, class, method or field.
[3] The entity can be a package or class.

Table 3
Predefined metrics of API usage.

Category	Metric	Description	Calculation
Popularity	$NP_l(x)/RP_l(x)$	Number/Ratio of the projects that use library x	$NP_l(x) = \lvert C'\rvert$ where $C' = \{S_i \lvert x \in U_l(Si)\}$, $RP_l(x) = NP_l(x)/\lvert C\rvert$
	$NP_p(x)/RP_p(x)$	Number/Ratio of the projects that use package x	$NP_p(x) = \lvert C'\rvert$ where $C' = \{S_i \lvert x \in U_p(Si)\}$, $RP_p(x) = NP_p(x)/\lvert C\rvert$
	$NP_c(x)/RP_c(x)$	Number/Ratio of the projects that use class x	$NP_c(x) = \lvert C'\rvert$ where $C' = \{S_i \lvert x \in U_c(Si)\}$, $RP_c(x) = NP_c(x)/\lvert C\rvert$
	$NP_m(x)/RP_m(x)$	Number/Ratio of the projects that use method x	$NP_m(x) = \lvert C'\rvert$ where $C' = \{S_i \lvert x \in U_m(Si)\}$, $RP_m(x) = NP_m(x)/\lvert C\rvert$
	$NP_f(x)/RP_f(x)$	Number/Ratio of the projects that use field x	$NP_f(x) = \lvert C'\rvert$ where $C' = \{S_i \lvert x \in U_f(Si)\}$, $RP_f(x) = NP_f(x)/\lvert C\rvert$
Frequency	$SO_c(x)/PSO_c(x)$	Occurrence of class x in total/per project	$SO_c(x) = \sum_{S_i \in C} m_{O_c(S_i)}(x)$, $PSO_c(x) = SO_c(x)/NP_c(x)$
	$SO_m(x)/PSO_m(x)$	Occurrence of method x in total/per project	$SO_m(x) = \sum_{S_i \in C} m_{O_m(S_i)}(x)$, $PSO_m(x) = SO_m(x)/NP_m(x)$
	$SO_f(x)/PSO_f(x)$	Occurrence of field x in total/per project	$SO_f(x) = \sum_{S_i \in C} m_{O_f(S_i)}(x)$, $PSO_f(x) = SO_f(x)/NP_f(x)$
	$CO_c(x)/PCO_c(x)$	Cumulative occurrence of class x in total/per project	$CO_c(x) = SO_c(x) + \sum_{y \in M_c(x)} SO_m(y) + \sum_{z \in F_c(x)} SO_f(z)$, $PCO_c(x) = CO_c(x)/NP_c(x)$
	$CO_p(x)/PCO_p(x)$	Cumulative occurrence of package x in total/per project	$CO_p(x) = \sum_{y \in C_p(x)} CO_c(y)$, $PCO_p(x) = CO_p(x)/NP_p(x)$
	$CO_l(x)/PCO_l(x)$	Cumulative occurrence of library x in total/per project	$CO_l(x) = \sum_{y \in P_l(x)} CO_p(y)$, $PCO_l(x) = CO_l(x)/NP_l(x)$
Coverage	$Cov_l^p(x)$	Ratio of used *public* packages within library x	$Cov_l^p(x) = \lvert P_l(x) \cap (\bigcup_{S_i \in C} U_p(S_i))\rvert / \lvert P_l(x)\rvert$
	$Cov_l^c(x)$	Ratio of used *public* classes within library x	$Cov_l^c(x) = \lvert C_l(x) \cap (\bigcup_{S_i \in C} U_c(S_i))\rvert / \lvert C_l(x)\rvert$
	$Cov_l^m(x)$	Ratio of used *public* methods within the library x	$Cov_l^m(x) = \lvert M_l(x) \cap (\bigcup_{S_i \in C} U_m(S_i))\rvert / \lvert M_l(x)\rvert$
	$Cov_l^f(x)$	Ratio of used *public* fields within the library x	$Cov_l^f(x) = \lvert F_l(x) \cap (\bigcup_{S_i \in C} U_f(S_i))\rvert / \lvert F_l(x)\rvert$
	$Cov_p^c(x)$	Ratio of used *public* classes within the package x	$Cov_p^c(x) = \lvert C_p(x) \cap (\bigcup_{S_i \in C} U_c(S_i))\rvert / \lvert C_p(x)\rvert$
	$Cov_p^m(x)$	Ratio of used *public* methods within the package x	$Cov_p^m(x) = \lvert M_p(x) \cap (\bigcup_{S_i \in C} U_m(S_i))\rvert / \lvert M_p(x)\rvert$
	$Cov_p^f(x)$	Ratio of used *public* fields within the package x	$Cov_p^f(x) = \lvert F_p(x) \cap (\bigcup_{S_i \in C} U_f(S_i))\rvert / \lvert F_p(x)\rvert$
	$Cov_c^m(x)$	Ratio of used *public* methods within the class x	$Cov_c^m(x) = \lvert M_c(x) \cap (\bigcup_{S_i \in C} U_m(S_i))\rvert / \lvert M_c(x)\rvert$
	$Cov_c^f(x)$	Ratio of used *public* fields within the class x	$Cov_c^f(x) = \lvert F_c(x) \cap (\bigcup_{S_i \in C} U_f(S_i))\rvert / \lvert F_c(x)\rvert$

from our corpus. *JAPIExtractor* uses Eclipse EGit [17] to interact with git-based project repositories programmatically. It integrates Eclipse Aether [18] and Maven API [19] libraries to extract dependencies from *pom.xml* and automatically retrieve them from given remote repositories. It also leverages Eclipse JDT [20] parser to parse Java code and build its abstract syntax tree (AST), attached with accurate bindings of API entities. All generated ASTs are stored in a database. Our tool can quickly traverse ASTs and obtain the use of API entities. It also provides statistical functions that assist with frequency, popularity and coverage analysis. In addition, to calculate the coverage of a given API library, our tool can load its corresponding *jar* file(s) and capture all public API entities with the aid of Java *reflection* technique. As the deprecated API entities exist, our tool adopts a two-step strategy to identify them precisely. It first applies Java reflection to identify the API entities deprecated by annotations. It then leverages the Apache Commons BCEL [21] to analyze class files within the jar and capture the API entities deprecated by Javadoc tag. All the data of API usage are available online.[4]

3. Global analysis of API usage

3.1. API usage provenance

Research in software engineering has shown that reuse can promote the productivity of the development team, reduce the time-to-market and improve the overall quality of software products [22]. Adopting API libraries is one of effective and efficient reuse approaches [23]. We are interested in the API *provenance* to figure out how much of the code is reused from existing API libraries and how much are newly added. To this end, we collect the use of all APIs, including project-specific API entities, which are probably protected or private for the internal use.

Table 4 shows the distribution of the API provenance from the global perspective, where the use of API entities is counted by their *SO* values. From the perspective of the classes, we find over 40% of used classes originate from the core API library and 15% originate from the third-party libraries. The remaining 45% are implemented internally specific to projects. From the perspective of the methods, the usage from the third-party libraries accounts for a larger proportion (23.27%, among which 2.27% of them are not resolved correctly,[5] also originate from third-party libraries) and correspondingly the use of methods from the core libraries decreases to 15%. Significantly different from above two distributions, the use of the fields from existing libraries (containing both core API and third-party libraries) accounts for a relatively small proportion (12%). A possible reason is that most fields are designed to record status of objects, which are usually retrieved by programmers through corresponding methods, *e.g. getters*. The tasks of accessing or manipulating these fields are privately executed by their declaring classes. The remaining static and public fields that can be accessed with no restrictions only account for a tiny proportion.

We also present the distribution of API usage provenance by projects in Fig. 3. From the perspective of classes, the use of core API library accounts for a relatively large proportion (35–53%) of all API usage, while the use of third-party libraries accounts for a lower proportion (8–32%). From the perspective of methods, the provenance distribution looks similar, in which the use of project-specific methods accounts for a higher proportion. Analogous to the results in Table 4, the use of project-specific fields is still dominant.

In summary, current software projects do depend on the use of API libraries. Approximately 45% of the employed API entities are taken from existing API libraries (28.27% from the core API and 15.53% from the third-party libraries). In other words, if we simplify the software development as a task of composing API entities, 45% of the coding work (in terms of SLoC) should be accomplished through reusing existing API libraries, and the remaining 55% of the work is required to conduct from scratch. Both of the core API and third-party libraries are fundamental to software development.

3.2. Project size vs. API usage

From the data in Table 1, we find that, on average, one class is employed per two SLoC; one method is employed per three SLoC and one field is employed per five SLoC. Likewise, one Java file employs 47 classes, 37 methods and 21 fields on average. It is reasonable to speculate that project size is tightly correlated with API usage. To validate our intuitive conjecture, we generate a log–log

[4] http://dong-qiu.github.io/papers/lang_api_study/lang_api_study.html.

[5] When resolving bindings of API usage, *not all* of the required third-party libraries are available on the Internet. We discuss more in the construct validity.

Table 4
Distribution of the API provenance.

	Core		Third-party		Project-specific		Unresolved	
	SO	SO (%)	SO	SO (%)	SO	SO (%)	SO	SO (%)
Classes	30.2M	40.27	11.4M	15.25	33.3M	44.41	48K	0.06
Methods	15.5M	26.19	12.5M	21.16	29.8M	50.38	1.3M	2.27
Fields	1.9M	5.49	2.2M	6.36	30.0M	87.97	60K	0.18
Total	47.6M	28.27	26.1M	15.53	93.1M	55.34	1.4M	0.86

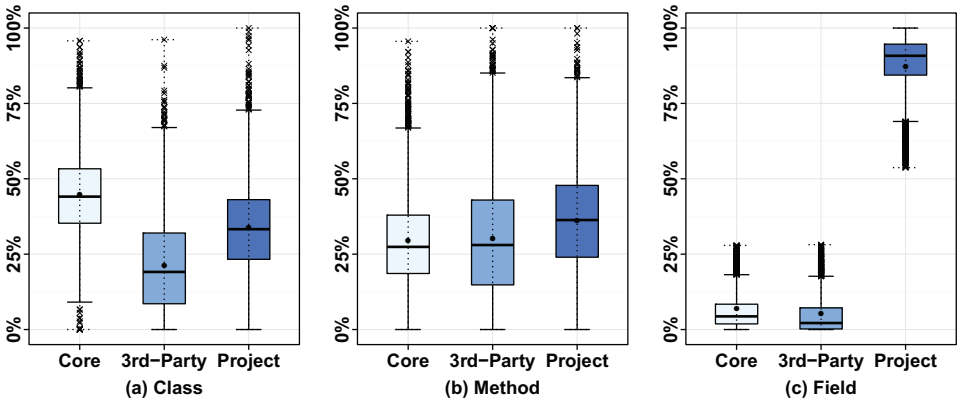


Fig. 3. The provenance distribution of the API occurrences by projects.

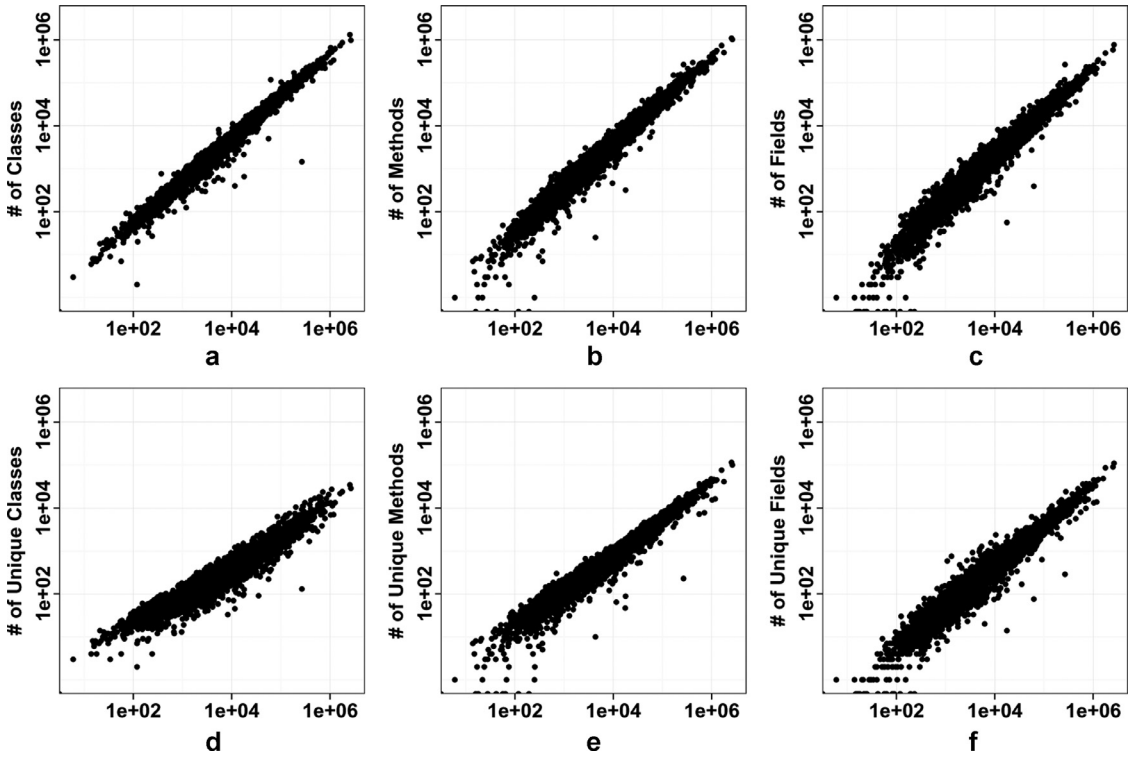


Fig. 4. The *SO* of API entities per project *w.r.t.* its size. The *x-axes* represent the project size (measured in the count of Java files); the *y-axes* represent the size of classes, methods, fields, *unique* classes, *unique* methods and *unique* fields, respectively. Both axes are logarithmic.

plot for the count of API entities (including classes, methods and fields separately) used per project *w.r.t.* its project size. Fig. 4(a)–(c) demonstrate the results. Project sizes grow as quickly as the counts of the employed API entities, indicating an approximately linear trend. Basically every project uses less than 100,000 API classes (or methods, fields). We also generate a log–log plot for the number of distinct API entities (including classes, methods and fields separately) adopted per project *w.r.t.* its project size. Fig. 4(d)–(f) demonstrate the results, which clearly indicate that project sizes grow much more quickly than the size of uniquely-used API entities. Most projects adopt less than 10,000 unique API classes (or methods, fields).

3.3. API usage follows power-laws

A power law indicates that a small fraction of elements is extremely common, whereas a large fraction is extremely rare [24].

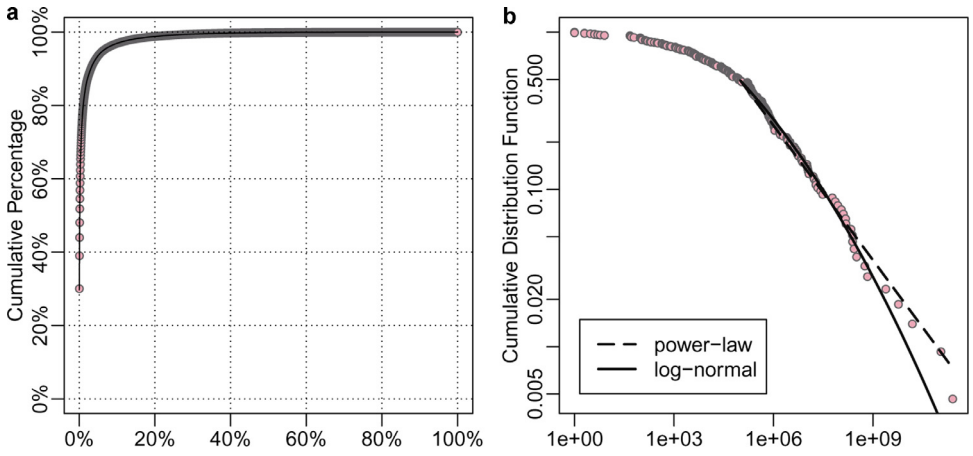


Fig. 5. API usage distribution based on the CO_p values. (a) demonstrates the cumulative percentage of the CO_p values; (b) demonstrates the cumulative distribution function of the CO_p values where the points are data, solid line is best-fit log-normal and the dashed line is best-fit power-law.

A number of metrics on software systems have been observed to obey the power-law [24–26]. Identification of these laws is conducive to capture potential software's characteristics; their existence is important to software engineering [26]. In this study, we are interested in investigating whether the API usage also obeys the power law. We employ the metric package frequency (CO_p) to validate this hypothesis.

Fig. 5(a) shows that a small number of packages account for most core API usage. The top 1% of the packages account for 80% of all API usage and top 3% account for 90%. The heavy tail covers many rarely used packages, 70% of the least frequently used packages account for less than 0.5% of all API usage; 50% for less than 0.1%. We also fit the API usage data to some candidate distributions. Fig. 5(b) demonstrates that the power-law distribution has a much better fit than the log-normal distribution.

At present, most APIs continuously introduce new API entities that implement additional features and functionalities, making their footprint gigantic. API designers seldom take actions to prune and simplify APIs over time. We have discovered that a large amount of API entities are rarely adopted by programmers. The heavy tail directly identifies the *coldspots* of the APIs, which are possible targets to simplify and optimize. It is significant for API designers to reconsider their designs of the API based on its actual use before they plan to develop a new release, *e.g.* moving the extremely unpopular API entities from the kernel to the optional feature set. Certainly, it is inequitable to apply only one metric *frequency* to determine the *coldspots*, because some of the API entities, which are employed only once (*e.g.* configuring global settings) in most applications, are used far fewer than others in use frequency. We take some other metrics, *e.g. coverage* and *popularity*, into consideration in the next few sections.

4. API usage of core library

4.1. Coverage analysis

API coverage analysis is considered as a principal way to assist API migration [16] and increase API usability [27]. It can also be applied to inspect whether the API library has been sufficiently utilized. New features have been introduced ceaselessly while few existing features that are rarely used have been removed from the core APIs. It is expected to result in the rapid growth of the core library and more resources consumption for devices. We desire to identify those *coldspots* of the core API that are rarely or never touched by programmers and provide suggestions on simplifying API libraries.

Coverage is inevitably affected by the version of the core API as new API entities are introduced. We first discuss the coverage of core API from Java 8. From the perspective of library coverage (measured by Cov_l^c, Cov_l^m and Cov_l^f), we discover that 15.3% of the classes, 41.2% of the methods and 41.6% of the fields are never adopted by any project in our corpus. To further analyze the package coverage (*i.e.* the ratio of the classes or methods that have been covered in a package) and class coverage (*i.e.* the ratio of the methods or fields that have been covered in a class) in detail, we adopt the metrics Cov_p^c and Cov_p^m to assess package coverage, and Cov_c^m and Cov_c^f to assess class coverage. Fig. 6(a)–(d) show the results. We discuss them separately.

Package coverage. Regarding the Cov_p^c values in Fig. 6(a), 50% (109/217) of the packages have all classes been used at least once. Only two packages (`org.omg.CORBA.DynAnyPackage` and `org.omg.stub.java.rmi`) whose subordinative classes have never been employed. The former package only provides four exception classes related to the interface `DynAny` and the latter package contains only one single class in total. Besides, 7.4% (16/217) of the packages have less than 50% of their classes used. Considering the Cov_p^m values in Fig. 6(b), only 9% (18/209) of the packages have all subordinative methods adopted (eight packages are excluded as they have no classes). 9.5% (20/209) of the packages have none of the methods adopted in which most package names start with `org.omg.*`. Besides, 28.2% of the packages have less than 50% of the methods used.

Class coverage. Considering the Cov_c^m values in Fig. 6(c), 34.4% (1179/3429) of the classes have all methods been used at least once (814 classes are excluded as they have no methods). 29.2% (1001/3429) of classes have none of the methods adopted. Besides, 14.1% (483/3429) of the classes have less than 50% of their methods used. Considering the Cov_c^f values in Fig. 6(d), 45.9% (492/1073) of the classes have all fields been adopted (3170 classes are excluded as they have no fields). 33.1% (355/1073) of the classes have none of the fields adopted. Besides, 8.1% (87/1073) of the classes have less than 50% of the fields used. The coverage data indicate that the utilization of the class from the core API is not high, in which about 40% of the classes have not been sufficiently used.

Normally, as new API entities are introduced into the core API, the coverage of the core API of newer version inevitably decreases. To validate this intuitive conjecture, we calculate the coverage on the core APIs from other Java versions. Table 5 lists the results of

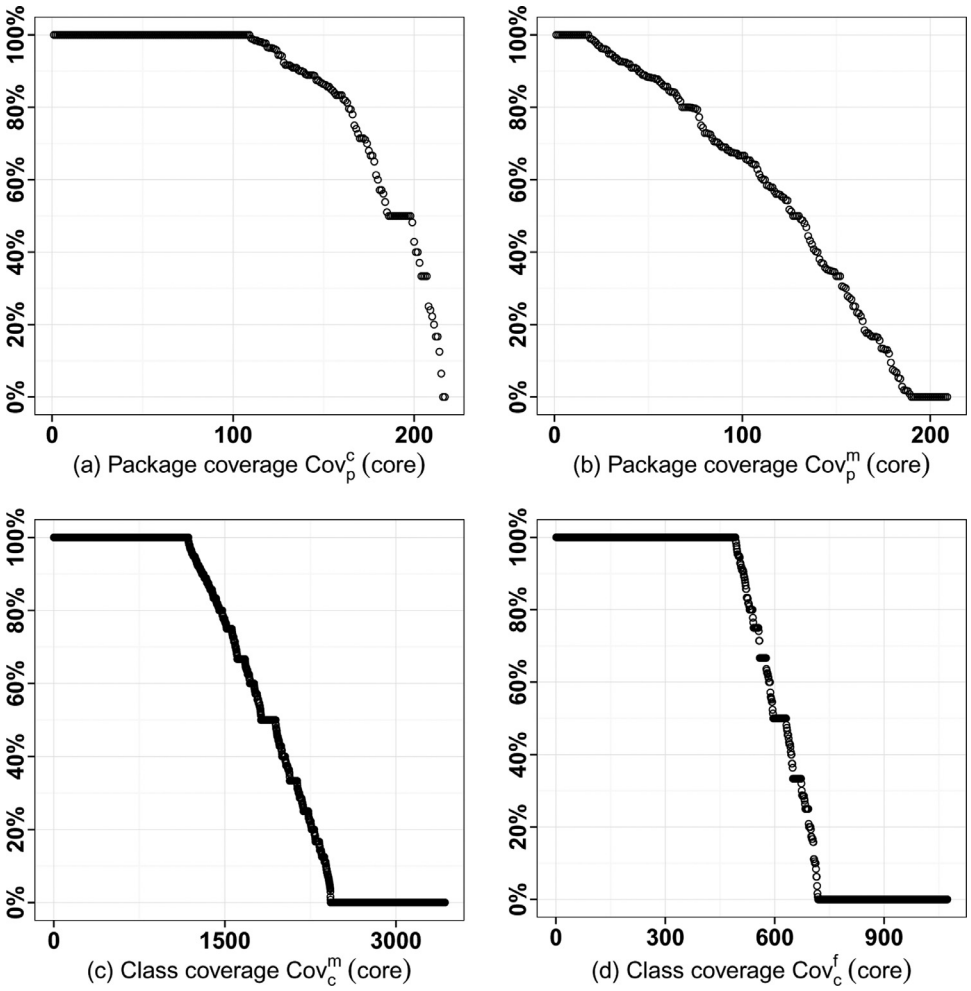


Fig. 6. Package and class coverage of the core API. The *x-axes* list all packages (in (a) and (b)) and classes (in (c) and (d)), sorted in the descending order by their coverage metrics: (a) Cov_p^c; (b) Cov_p^m; (c) Cov_c^m and (d) Cov_c^f.

Table 5
Library coverage of core APIs from all Java versions. The core API with and without deprecated API entities are both considered.

Versions	With deprecated API entities				Without deprecated API entities			
	Cov_l^p (%)	Cov_l^c (%)	Cov_l^m (%)	Cov_l^f (%)	Cov_l^p (%)	Cov_l^c (%)	Cov_l^m (%)	Cov_l^f (%)
jdk1.8	99.1	84.8	58.9	58.4	99.1	85.2	59.2	58.9
jdk1.7	99.0	85.4	60.6	58.8	99.0	85.8	61.0	59.3
jdk1.6	99.0	85.5	61.3	60.1	99.0	86.0	61.6	60.6
jdk1.5	98.8	83.8	60.0	58.4	98.8	84.2	60.3	58.9
jdk1.4	98.5	83.4	61.0	59.1	98.5	83.8	61.3	59.6
jdk1.3	97.4	84.7	61.8	59.1	97.4	85.4	62.4	59.8
jdk1.2	98.3	85.9	63.8	62.5	98.3	86.8	64.4	63.2
jdk1.1	100.0	97.1	81.1	85.6	100.0	98.7	82.6	87.0

library coverage from `jdk1.1` to `jdk1.8`.[6] Four types of the library coverages (*i.e.* Cov_l^p, Cov_l^c, Cov_l^m and Cov_l^f) are all taken into account. We also separate the results by including and excluding deprecated API entities as deprecation is also an influential factor for coverages. Contrary to our expectation, the coverages remain basically stable over most releases. Take Cov_l^m as an example, its values keep within a small range between 58.9% and 63.8% except `jdk1.1`. They decrease with some minor fluctuations (*e.g.* in `jdk1.6`) as the core API evolves. Likewise, the Cov_l^c values also remain in a minor range between 83.8% and 85.9%, and the Cov_l^f values fall within 58.4% and 62.5%. The factor of deprecated API entities does not influence the coverage much, because the size of deprecated API entities is tiny *w.r.t.* the size of entire API entities.

We further inspect the package coverage by Cov_p^c and Cov_p^m values (in Table 6), and class coverage by Cov_c^m values (in Table 7) over multiple versions of the core API. In Table 6, we list the number of packages defined in each version in the second column. Since it is impossible to present the entire set of Cov_p^c (or Cov_p^m) values, we aggregate them by four ranges, and calculate their distribution. Analogous to the library coverage, except `jdk1.1`, the distribution of Cov_p^c (or Cov_p^m) values remain generally stable. The

[6] The version string is used here. More details can be found at http://www.oracle.com/technetwork/java/javase/jdk8-naming-2157130.html .

Table 6
The distribution of package coverage of core API from all Java versions.

Versions	No. of packages	Cov_p^c values					Cov_p^m values				
		0	(0, 0.5)	[0.5, 1)	1	0/0[a]	0	(0, 0.5)	[0.5, 1)	1	0/0[b]
jdk1.8	217	0.9%	7.4%	41.5%	50.2%	0.0%	9.2%	27.2%	51.6%	8.3%	3.7%
jdk1.7	209	1.0%	7.2%	34.4%	57.4%	0.0%	9.1%	25.8%	52.2%	9.1%	3.8%
jdk1.6	203	1.0%	7.4%	30.5%	61.1%	0.0%	9.4%	25.6%	51.7%	9.4%	3.9%
jdk1.5	166	1.2%	9.0%	33.7%	56.0%	0.0%	9.6%	28.3%	47.0%	10.2%	4.8%
jdk1.4	135	1.5%	9.6%	34.8%	54.1%	0.0%	11.1%	25.2%	46.7%	12.6%	4.4%
jdk1.3	76	2.6%	13.2%	39.5%	44.7%	0.0%	7.9%	25.0%	56.6%	3.9%	6.6%
jdk1.2	59	1.7%	11.9%	39.0%	47.5%	0.0%	3.4%	30.5%	57.6%	3.4%	5.1%
jdk1.1	22	0.0%	4.5%	18.2%	77.3%	0.0%	0.0%	18.2%	77.3%	4.5%	0.0%

[a] 0/0 refers to those packages that do not have any subordinative classes.
[b] 0/0 refers to those packages that do not have any subordinative methods.

Table 7
The distribution of class coverage of core API from all Java versions.

Versions	No. of classes	Cov_c^m values				
		0	(0, 0.5)	[0.5, 1)	1	NaN[a]
jdk1.8	4240	23.5%	11.4%	18.0%	27.8%	19.2%
jdk1.7	4024	23.3%	10.2%	18.5%	28.3%	19.8%
jdk1.6	3793	22.6%	10.5%	18.6%	28.6%	19.8%
jdk1.5	3279	23.4%	10.9%	18.3%	27.0%	20.5%
jdk1.4	2723	23.8%	11.4%	18.9%	25.2%	20.7%
jdk1.3	1840	22.6%	12.4%	22.8%	22.2%	20.0%
jdk1.2	1524	20.5%	12.5%	23.8%	23.0%	20.1%
jdk1.1	477	6.9%	7.8%	25.6%	35.0%	24.7%

[a] NaN refers to those classes that do not have any subordinative methods.

packages whose Cov_p^c values equal to 1 account for 45–60% across all `jdk` versions. The packages with low rate of covered classes ($Cov_p^c \in (0, 0.5)$) account for 7–13%. On the contrary, the packages whose Cov_p^m values equal to 1 account for only 3–12%. Most Cov_p^m values fall in the range [0.5, 1). The packages with low rate of covered methods ($Cov_p^m \in (0, 0.5)$) account for a relatively high proportion (25–30%).

Switching to the class coverage, we discover that the Cov_c^m values are equally distributed comparatively. The fully-used classes ($Cov_c^m = 1$) account for 22–29% of all classes across most `jdk` versions. The underutilized classes ($Cov_c^m \in (0, 0.5)$) account for 10–12% as well. It is astonishing that 20–24% of the classes have their subordinative methods never been adopted by any project within our corpus on almost all `jdk` versions.

We are also concerned with how much of the core API library is employed by project. Hence, we calculate the library coverage of core API for each project. Fig. 7 shows the results. Most (over 90%) of the projects adopt less than 20% of packages, 10% of classes, 5% of the methods and 5% of the fields, which represent a tiny fraction against the entire core library. The scale of current core API is too enormous for most projects. We will discuss more in Section 4.4.

In summary, the utilization of the core API is not high, no matter in the old or new releases. Loading the entire core API to run applications wastes some resources.

4.2. Hotspots of API entities

API hotspots are API entities that are widely and frequently used in practice, which can serve as instructive starting points for both developers and novices to understand, learn and reuse a given library [28]. We identify the most popular and frequently used API entities from the core API. Tables 8–11 demonstrate the results, sharing the similar structure. The first column shows the qualified name of each API entity; the second column lists its popularity (RP_p, RP_c, RP_m and RP_f values for packages, classes, methods and fields respectively) and corresponding ranking[7]; the third column lists its frequency (CO_p, SO_c, SO_m and SO_f values for packages, classes, methods and fields respectively) and corresponding ranking; the last column shows its average frequency within projects that employ it (PCO_p, PSO_c, PSO_m and PSO_f values for packages, classes, methods and fields respectively). Regarding the use of packages, it is as expected that `java.lang`, `java.util` and `java.io` are top three prevalent packages, where both of their popularity (over 90%) and frequency (over 3.0E+06) far exceed the remaining packages. Some language features that are implemented through the core API library, *e.g.* reflection (`java.lang.reflect`), regular expression (`java.util.regex`), concurrency (`java.util.concurrent`) and annotation (`java.lang.annotation`), are also widely-used. Regarding the use of classes, the classes within the three most used packages also dominate the most popular classes, which are mainly relate to exceptions (5/20), collections (5/20), string (2/20) and build-in annotations (2/20). The use of annotations, especially `java.lang.Override`, is far beyond our expectations. We randomly check some projects and discover that most usage of annotation `@Override` and `@SuppressWarnings` are automatically generated by modern IDEs (*e.g.* Eclipse), whenever a class does override a method or the named compiler warnings should be suppressed in the annotated element. Programmers seldom instantiate build-in annotations proactively. We also find that the adoption of exceptions is frequent, which is probably caused by the mechanism of exception handling. Any invoked method that might throw certain exceptions must be enclosed by either a `try` statement that can catch this exception or a method specifying that it can throw the exception [29]. Such language constraints consequentially lead to the increased use of exception-related classes. We discuss more in Section 7. The most popular methods are mainly originated from three top-used class `java.lang.String`, `java.lang.List` and `java.lang.Map`. They concentrate on the manipulation of collections and string. The use of fields is more widely distributed; they primarily deal with the threshold values (*e.g.* `java.lang.Integer.MAX_VALUE`), system's I/O (*e.g.* `java.lang.System.out`) and constant enums for settings (*e.g.* `java.util.Locale.ENGLISH`). We also include a special field named *length, i.e.* a build-in member of an array type, because arrays are special objects in Java [29]. Its popularity is much higher as it is not a field for a particular class.

The frequency and popularity of an API entity are not invariably consistent. Take package `java.lang.reflect` as an example, its RP_p value ranks 6th, while CO_p value only ranks 18th. This is probably because packages differ in scale (*i.e.* the size of

[7] The ranking system takes all API entities from both of the core API and third-party libraries into consideration and ranks them uniformly.

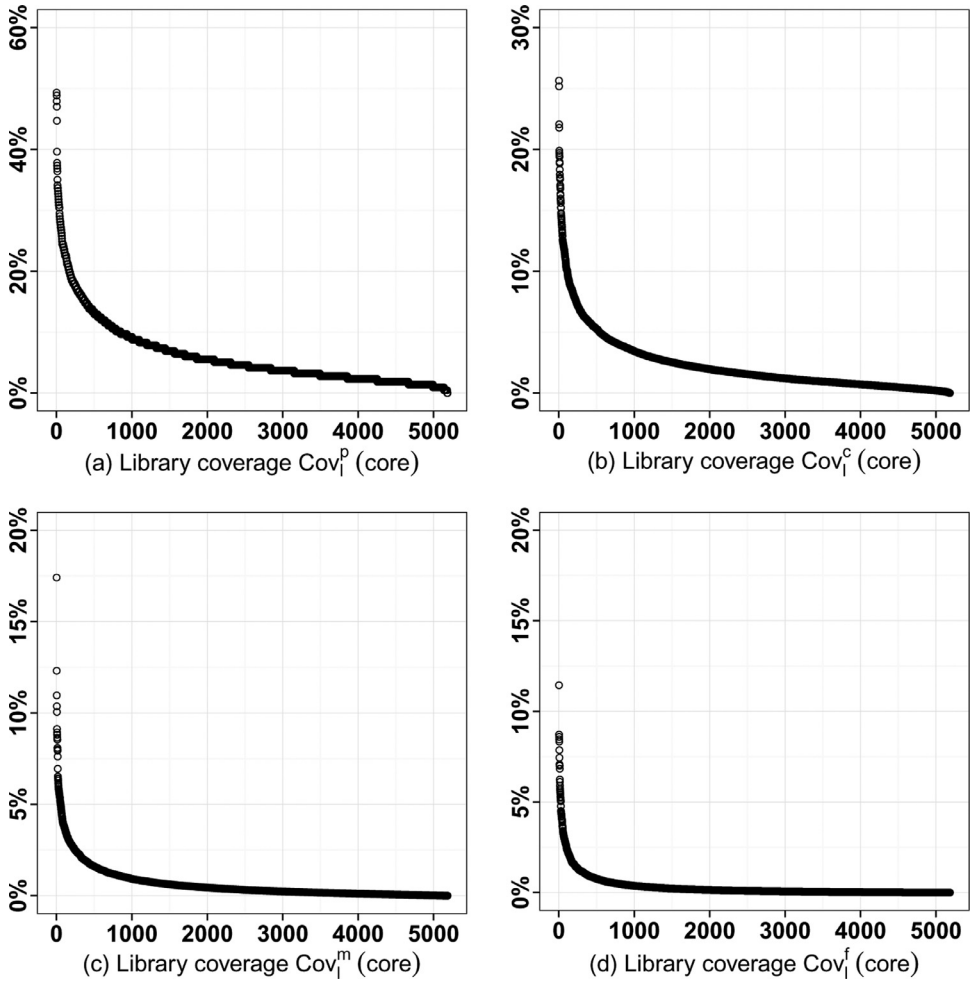


Fig. 7. Library coverage of the core API library *w.r.t.* projects. The *x-axes* list all projects, sorted in the descending order by their coverage metrics: (a) Cov_l^p; (b) Cov_l^c; (c) Cov_l^m and (d) Cov_l^m values, respectively.

Table 8
Top 20 popular packages from the core API.

Packages	RP_p/Rank	CO_p/Rank	PCO_p
java.lang	99.9%/1	2.66E+07/1	5145
java.util	97.1%/2	1.05E+07/2	2093
java.io	93.1%/ 3	3.30E+06/3	684
java.net	64.5%/4	5.05E+05/12	151
java.lang.reflect	53.5%/6	3.49E+05/20	126
java.util.concurrent	53.4%/7	4.31E+05/15	92
java.util.regex	44.0%/8	2.17E+05/38	95
java.text	43.9%/9	1.49E+05/69	65
java.security	34.8%/11	1.71E+05/61	95
java.util.concurrent.atomic	30.8%/12	1.42E+05/72	89
java.lang.annotation	28.4%/14	1.76E+05/57	119
java.math	27.6%/15	2.50E+05/35	175
java.util.zip	26.2%/17	4.95E+04/215	36
java.nio.charset	26.1%/18	4.31E+04/255	32
java.util.logging	25.6%/19	2.71E+05/31	204
java.nio	23.8%/20	2.61E+05/33	211
org.xml.sax	20.2%/24	8.38E+04/122	80
java.sql	19.8%/25	6.45E+05/9	627
org.w3c.dom	18.6%/29	3.22E+05/21	333
javax.xml.parsers	18.5%/30	4.61E+04/235	48

Table 9
Top 20 popular methods from the core API.

Methods	RP_m/Rank	SO_m/Rank	PSO_m
java.lang.String.*equals(java.lang.Object)*	81.7%/1	5.70E+05/1	135
java.util.List.*add(java.lang.Object)*	81.3%/2	5.26E+05/3	125
java.util.Map.*put(java.lang.Object, java.lang.Object)*	75.4%/3	3.99E+05/6	143
java.util.Map.*get(java.lang.Object)*	74.6%/4	3.43E+05/9	89
java.util.List.*size()*	74.3%/5	3.58E+05/8	93
java.lang.Object.*getClass()*	73.5%/6	2.47E+05/14	65
java.lang.String.*length()*	71.7%/7	2.47E+05/13	66
java.util.List.*get(int)*	68.6%/8	3.86E+05/7	109
java.lang.StringBuilder.*toString()*	66.4%/9	1.08E+05/29	31
java.io.PrintStream.*println(java.lang.String)*	64.6%/10	2.70E+05/12	81
java.lang.StringBuilder.*append(java.lang.String)*	64.5%/11	5.07E+05/4	151
java.lang.Throwable.*getMessage()*	61.8%/12	1.38E+05/21	43
java.lang.Object.*toString()*	61.0%/13	9.88E+04/31	31
java.lang.String.*startsWith(java.lang.String)*	59.9%/14	8.55E+04/38	28
java.lang.String.*substring(int, int)*	59.6%/15	8.63E+04/37	28
java.util.Iterator.*next()*	59.6%/16	1.51E+05/19	49
java.util.Arrays.*asList(java.lang.Object[])*	59.2%/17	1.19E+05/23	39
java.lang.Class.*getName()*	58.8%/18	1.68E+05/17	55
java.lang.String.*substring(int)*	56.3%/19	6.63E+04/49	23
java.lang.String.*split(java.lang.String)*	56.1%/20	4.29E+04/83	15

classes a package contains). Regarding packages `java.awt`[8] and `java.lang.reflect`, although the former package is employed by less projects, the number of its subordinative classes far exceeds the number of those in the latter package, which easily forms a cumulative increase in use frequency. Besides, the average frequency (*i.e. PCO* or *PSO* values) of UI-related API entities (`javax.swing` is an example) is much higher than other entities.

[8] The data of package `java.awt` are not shown in Table 8. Its CO_p ranking is 10th ($CO_p = 2.42E+08$) while RP_p ranking is 32nd ($RP_p = 16.9\%$). Its $PCO_p = 818$, is higher than most packages.

Table 10
Top 20 popular classes from the core API.

Classes	RP_c/Rank	SO_c/Rank	PSO_c
java.lang.String	99.4%/1	9.21E+06/1	1787
java.lang.Override	94.2%/2	2.75E+06/2	563
java.util.List	89.2%/3	1.25E+06/4	271
java.lang.Exception	88.8%/4	1.15E+06/5	249
java.lang.Object	87.0%/5	1.52E+06/3	337
java.io.IOException	83.8%/6	8.47E+05/6	195
java.util.ArrayList	83.7%/7	5.06E+05/11	117
java.lang.System	82.8%/8	5.67E+05/10	132
java.lang.Integer	82.3%/9	7.07E+05/8	166
java.util.Map	80.4%/10	5.93E+05/9	142
java.util.HashMap	74.4%/11	2.55E+05/17	66
java.lang.Class	68.8%/12	4.49E+05/13	126
java.lang.IllegalArgumentException	68.6%/13	2.54E+05/18	71
java.lang.RuntimeException	68.3%/14	1.46E+05/31	41
java.lang.StringBuilder	67.4%/15	2.33E+05/21	67
java.util.Arrays	67.2%/16	1.69E+05/29	48
java.lang.SuppressWarnings	66.6%/17	1.72E+05/26	50
java.util.Set	65.0%/18	2.81E+05/16	83
java.io.File	63.4%/19	4.91E+05/12	150
java.lang.Throwable	63.3%/20	2.07E+05/23	63

Table 11
Top 20 popular fields from the core API.

Fields	RP_f/Rank	SO_f/Rank	PSO_f
X[].*length* (*X*[] is an *array type*)	80.9%/1	6.83E+05/1	163
java.lang.System.*out*	64.6%/2	2.58E+05/3	77
java.lang.System.*err*	36.9%/3	6.82E+04/9	36
java.lang.Integer.*MAX_VALUE*	36.2%/4	2.35E+04/18	12
java.lang.Boolean.*TRUE*	27.2%/5	2.91E+04/13	21
java.lang.annotation.RetentionPolicy.*RUNTIME*	26.0%/6	2.73E+04/14	20
java.util.concurrent.TimeUnit.*SECONDS*	24.0%/7	1.91E+04/24	15
java.lang.Boolean.*FALSE*	22.1%/8	2.11E+04/21	18
java.io.File.*separator*	20.8%/9	2.15E+04/20	20
java.util.concurrent.TimeUnit.*MILLISECONDS*	20.6%/10	1.59E+04/26	15
java.lang.Long.*MAX_VALUE*	19.4%/11	9.68E+03/53	10
java.lang.annotation.ElementType.*TYPE*	19.1%/12	7.06E+03/77	7
java.lang.annotation.ElementType.*METHOD*	17.6%/13	8.20E+03/62	9
java.lang.Integer.*MIN_VALUE*	16.8%/14	6.90E+03/81	8
java.lang.annotation.ElementType.*FIELD*	14.3%/15	5.47E+03/119	7
java.util.logging.Level.*SEVERE*	13.6%/16	1.19E+04/36	17
java.util.Locale.*ENGLISH*	13.5%/17	7.17E+03/75	10
java.lang.System.*in*	13.3%/18	2.64E+03/288	4
java.util.Calendar.*YEAR*	13.1%/19	6.20E+03/96	9
java.util.Locale.*US*	12.7%/20	4.47E+03/157	7

Table 12
Ten selected popular ($RP_c \geq 20.0\%$) classes with lowest frequency.

Classes	PSO_c	RP_c (%)	SO_c
java.lang.Runtime	6.0	25.6	7881
java.io.FileReader	6.0	23.1	7267
java.util.concurrent.Executors	6.1	27.0	8502
java.io.OutputStreamWriter	6.6	23.0	7844
java.io.BufferedInputStream	7.2	22.3	8304
java.io.FileWriter	7.4	21.9	8400
java.lang.InstantiationException	7.5	26.9	10437
java.io.InputStreamReader	8.5	44.8	19797
java.lang.reflect.Constructor	8.8	24.4	11138
java.util.concurrent.ExecutorService	8.9	23.9	10985

The above findings lead us to investigate one special group of API entities, with high popularity and low frequency. Tables 12 and 13 demonstrate the results of API classes and methods, which meet the required popularity ($RP \geq 20.0\%$). All entities in the table are ranked by their *PSO* values. We find these classes are related to I/O, initialization and services, which usually require a single use in writing code. Similarly, methods in the table are mainly invoked for loading properties, creating URL connections and performing read–write operations in I/O.

4.3. Usage of deprecated API entities

An API can declare parts of itself as *deprecated*. In general, programmers are discouraged from using a deprecated API entity because it may be dangerous, or a better alternative API entity is provided by the new release. Nevertheless, we find deprecated API entities are still applied in practice. To investigate how programmers use them, we capture all adopted deprecated entities from our corpus. Java provides two mechanisms to deprecate an API entity by: (1) using build-in annotation `@Deprecated` preceding the API entity declaration; (2) using `@deprecated` tag to make Javadoc show an API entity as deprecated. We capture the use of deprecation for both cases.

From the perspective of projects, 48.5% (2513/5185) of them adopt at least one deprecated API entity, in which 30.3% (1575/5185) employ deprecated classes, 38.8% (2012/5185) employ deprecated methods, and 15.1% (785/5185) employ deprecated fields. Only 51.5% of the projects within our corpus have none of the employed API entities marked as deprecated in the API documentation. The data indicate that deprecated API entities are still heavily used in many projects.

We further investigate the usage of deprecated entities from the core API. In `jdk1.8`, we find 51.1% (24/47) of the deprecated classes, 43.5% (240/552) of the deprecated methods, and 18.1% (13/72) of the deprecated fields are adopted by programmers. Table 14 shows the distribution of the usage of deprecated API entities from distinct core API versions where both size of unique deprecated entities in use and their simple occurrences summations are listed separately. Deprecated entities, especially methods, from `jdk1.1`, `jdk1.2` and `jdk1.4` are heavily used. We also list the top 10 used deprecated API entities in Tables 15–17. The *getters* methods in `java.util.Date` are adopted by various projects. Some heavily-used deprecated API entities, *e.g.* method `java.lang.Thread.stop()`, are inherently *unsafe* and may cause arbitrary behaviors [30].

The phenomenon—nearly half of the projects employ deprecated API entities; half of the deprecated API classes and methods are in use—strongly indicates widespread and heavy use of the deprecated API entities. Identifying how they are used, especially those that can lead to erroneous behaviors, is beneficial to discover and locate potential vulnerabilities of source code.

4.4. Usage of compact profiles

As discussed in Section 4.1 that projects use a small subset of the core API library in general, the loading of the entire library to run applications consumes unnecessary resources. Java 8 API introduced a new concept called *compact profile*, enabling applications to be compiled and run under a subset of the core API. It allows programmers to select a proper profile that closely matches an application's functional requirements. Three supported profiles are provided [29]: compact1 (*CP1*, containing 50 packages), compact2 (*CP2*, containing 82 packages) and compact3 (*CP3*, containing 118 packages). Each successive profile is a superset of its predecessor. For example, *CP2* contains all the packages defined in *CP1*. *Full* profile (*i.e.* the entire core API) is a superset of *CP3*. As current design of the compact profiles is package aggregation in fact, we are interested in investigating whether the selection of packages for a certain compact profile has close tie with their package popularity (*i.e.* ratio of the projects that adopted this package). To simplify our description, we use *CP2(A)* to represent the group of packages specified in *CP2* while not in *CP1*. Likewise, we use *CP3(A)* to represent packages specified in *CP3* while not in *CP2*, and use

Table 13
Ten selected popular ($RP_m \geq 20.0\%$) methods with lowest frequency.

Methods	PSO_m	RP_m (%)	SO_m
java.util.Properties.*load(java.io.InputStream)*	4.1	24.8	5339
java.lang.Class.*getConstructor(java.lang.Class[])*	4.6	21.5	5170
java.net.URL.*openConnection()*	4.7	23.1	5635
java.lang.reflect.Constructor.*newInstance(java.lang.Object[])*	4.8	26.3	6507
java.io.FileOutputStream.*close()*	4.9	21.8	5484
java.net.URL.*openStream()*	5.0	21.3	5466
java.io.File.*listFiles()*	5.0	25.3	6611
java.io.OutputStream.*write(byte[], int, int)*	5.2	21.3	5708
java.io.InputStream.*read(byte[])*	5.2	27.1	7292
java.nio.charset.Charset.*forName(java.lang.String)*	5.3	20.3	5600

Table 14
The distribution of used deprecated API entities by core API versions.

Versions	Deprecated classes		Deprecated methods		Deprecated fields	
	No. of unique	ΣSO_c	No. of unique	ΣSO_m	No. of unique	ΣSO_f
jdk1.8	3	35	13	94	3	16
jdk1.7	1	1	2	153	4	63
jdk1.6	1	20	11	1044	1	120
jdk1.5	1	1	9	217	0	0
jdk1.4	7	442	61	1060	0	0
jdk1.3	0	0	3	16	0	0
jdk1.2	9	404	86	979	2	5
jdk1.1	2	150	54	4187	3	69

Table 15
Top 10 used deprecated classes from the core API.

Classes	NP_l/Rank	SO_c/Rank	Since
java.io.StringBufferInputStream	46/1	126/2	jdk1.1
java.rmi.RMISecurityManager	16/2	24/12	jdk1.8
org.xml.sax.Parser	15/3	74/7	jdk1.4
org.xml.sax.DocumentHandler	11/4	95/4	jdk1.4
org.xml.sax.HandlerBase	11/5	80/6	jdk1.4
org.xml.sax.AttributeList	10/6	91/5	jdk1.4
java.security.Identity	9/7	200/1	jdk1.2
javax.xml.bind.Validator	9/8	20/14	jdk1.6
org.xml.sax.helpers.AttributeListImpl	8/9	57/8	jdk1.4
java.security.IdentityScope	5/10	117/3	jdk1.2

Table 16
Top 10 used deprecated methods from the core API.

Methods	NP_l/Rank	SO_m/Rank	Since
java.io.File.*toURL()*	186/1	983/1	jdk1.6
java.net.URLEncoder.*encode(java.lang.String)*	119/2	286/7	jdk1.4
java.util.Date.*getYear()*	91/3	538/2	jdk1.1
java.lang.Thread.*stop()*	86/4	186/9	jdk1.2
java.net.URLDecoder.*decode(java.lang.String)*	84/5	211/8	jdk1.4
java.util.Date.*getHours()*	81/6	480/4	jdk1.1
java.util.Date.*getMonth()*	78/7	509/3	jdk1.1
java.util.Date.*getMinutes()*	77/8	375/6	jdk1.1
java.util.Date.*getDate()*	74/9	460/5	jdk1.1
java.io.DataInputStream.*readLine()*	67/10	170/11	jdk1.1

Full(A) to represent packages specified in *Full* profile while not in *CP3*.

Fig. 8 shows the results. Packages from *CP1* have relatively higher package popularities among which 17 of them with RP_p values exceed 20%. The RP_p values of almost all packages from *CP2(A)*, *CP3(A)* and *Full(A)* are less than 20%. As expected, *CP1* also involves a considerable amount of packages that are not widely adopted ($RP_p \leq 20\%$). The distributions of RP_p values of packages from *CP2(A)*, *CP3(A)* and *Full(A)* look similar.

From the view of utilization, we are also interested in figuring out, under current definition of compact profiles, how much of the projects can be correctly compiled and run by an appointed subset profile instead of the entire core API. Fig. 9(a) demonstrates the results. Over 40% (2256/5185) of the projects can be compiled and run under *CP1*. 11.7% and 6.2% of the projects require *CP2* and *CP3*, respectively. The remaining 38.4% of the projects still need to employ the *Full* profile. The data indicate that most projects focus on using *CP1* and *Full* profiles. The adoption of *CP2* and *CP3* is less favored.

We further investigate the coverage of compact profiles by corresponding groups of projects. Fig. 9(b)–(e) demonstrate the results. For projects that can be correctly compiled and run under *CP1*, 8–16% of the packages from *CP1* are employed by most projects. In other words, *CP1* is still superfluous for most projects to some extent. Regarding the projects that require *CP2*, 10–22% of the packages from *CP1* are adopted. Only 3–9% of the extension packages in *CP2* (*i.e. CP2(A)*) are adopted. Similar behaviors are shown in another two groups: projects that require more functional profiles (*e.g. CP2, CP3*) in compilation and running actually adopt an extremely small subset relative to the entire extensions. Current designs of compact profiles inevitably consume unnecessary resources none the less. We believe *leaner* compact profiles can be calculated through observing the patterns of how projects employ packages in practice. More discussions are presented in Section 6.

5. API usage of third-party library

5.1. Library popularity analysis

Apart from using the core API library, programmers usually select appropriate third-party libraries to maximize code reuse and improve the efficiency of the development process. Globally, projects in our corpus employ 103,256 external third-party dependencies. Suppose we ignore the possibility that a library has multiple versions (discussed in Section 5.3), 16,329 distinct third-party libraries are adopted. However, most usage is concentrated on a limited range. Only 15 libraries are adopted by over 10% of

Table 17
Top 10 used deprecated fields from the core API.

Fields	NP_f/Rank	SO_f/Rank	Since
java.util.logging.Logger.*global*	13/1	120/1	jdk1.6
javax.imageio.spi.ImageWriterSpi.*STANDARD_OUTPUT_TYPE*	8/2	36/3	jdk1.7
java.util.jar.Attributes.Name.*IMPLEMENTATION_VENDOR_ID*	5/3	8/7	jdk1.8
javax.imageio.spi.ImageReaderSpi.*STANDARD_INPUT_TYPE*	4/4	22/5	jdk1.7
java.util.jar.Attributes.Name.*IMPLEMENTATION_URL*	4/5	7/8	jdk1.8
java.io.StringBufferInputStream.*pos*	3/6	36/2	jdk1.1
java.io.StringBufferInputStream.*count*	3/7	24/4	jdk1.1
java.io.StringBufferInputStream.*buffer*	3/8	9/6	jdk1.1
java.awt.datatransfer.DataFlavor.*plainTextFlavor*	3/9	4/9	jdk1.7
java.lang.SecurityManager.*inCheck*	2/10	2/11	jdk1.2

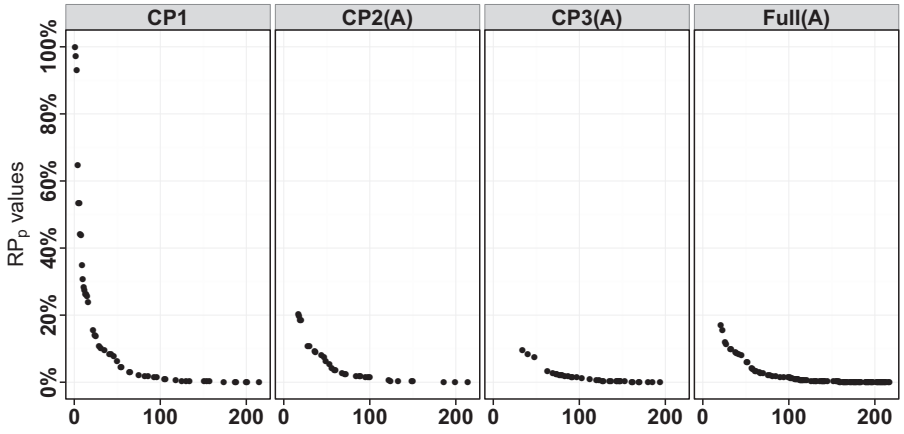


Fig. 8. Package popularity *w.r.t.* compact profiles. The *x-axes* list all packages, ordered by their corresponding RP_p values. We separate the packages into four groups: (a) *CP1*; (b) *CP2(A)*; (c) *CP3(A)* and (d) *Full(A)*.

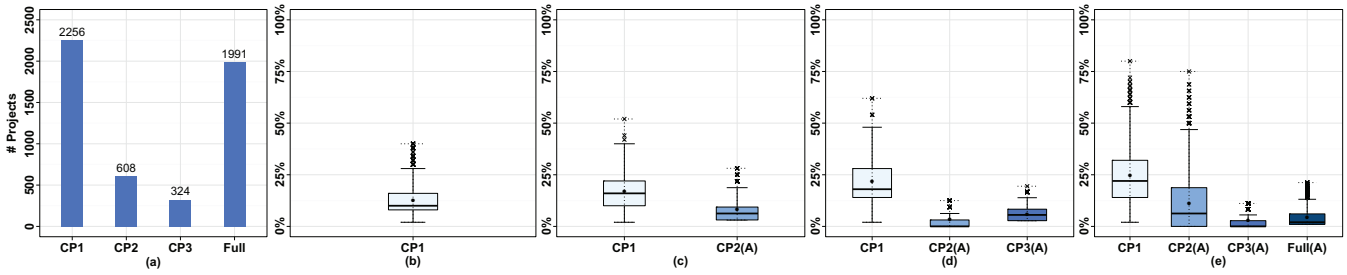


Fig. 9. Compact profiles usage. (a) shows the size of projects that can be correctly compiled and run under four profiles. Based on this classification, (b), (c), (d), (e) show the distribution of coverage on certain compact profile by corresponding group of projects. (b) shows the boxplot of $Cov_l^p(CP1)$ of projects that require *CP1*; (c) shows the boxplot of $Cov_l^p(CP1)$ and $Cov_l^p(CP2(A))$ of projects that require *CP2*; (d) shows the boxplot of $Cov_l^p(CP1)$, $Cov_l^p(CP2(A))$ and $Cov_l^p(CP3(A))$ of projects that require *CP3*; (e) shows the boxplot of $Cov_l^p(CP1)$, $Cov_l^p(CP2(A))$, $Cov_l^p(CP3(A))$ and $Cov_l^p(Full(A))$ of projects that require *Full* profile.

all projects within the corpus, and 265 libraries are adopted by over 1% projects. 9830 libraries are employed by only one project, in which many of them are project-specific, or submodules cross-referenced among their parent projects. Hence, the size of widely-used third-party libraries is not large.

We also rank the *popularity* of the third-party libraries by the size of the projects that depend on them. As most libraries contain multiple versions in general, we omit the *version* in the statistics and group them by the *groupId* and *artifactId*. Table 18 lists the top 20 popular libraries used in our corpus. Majority of them are managed and contributed by the well-known, reputable open-source communities or companies (*e.g.* `commons-*` libraries by Apache Software Foundation, `guava` and `android` by Google and `springframework-*` by Pivotal). The application domains of these third-party libraries are mainly centralized in the core utilities (*e.g.* `guava` and `springframework`), logging framework (*e.g.* `slf4j` and `log4j`) and testing framework (*e.g.* `junit` and `testng`), which are all valuable supplement to Java core libraries. Among them, the most-used library `junit` is adopted by over 80% of all projects, as it has become the de-facto unit testing standard framework for Java. `slf4j` and `log4j` are employed in over 30% and 20% of the projects respectively, and they are the first two choices when using the logging framework.

From the perspective of projects, we are interested in how much of third-party libraries are adopted to construct a project in general. To this end, for each project, we compute the size of dependent third-party libraries in different development phases. As discussed in Section 2.3 that Maven provides mechanism to specify the use of dependencies under given scopes (phases), we simply classify the libraries into three groups: *compile, test* and *runtime*. Fig. 10 shows the results. Most projects adopt 2–20 third-party libraries in general. Projects at the *test* phase require more libraries (4 or 5) than *compile* and *runtime* phases. Projects at runtime depend on fewer libraries. Among our corpus, 170 (3.3%) projects do not employ any third-party library. In addition, many outliers (projects depending on numerous libraries) are not shown in this

Table 18
Top 20 adopted libraries across the corpus.

Rank	Library (GroupId)	Library (ArtifactId)	NP_l	RP_l (%)
1	junit	junit	4270	82.4
2	org.slf4j	slf4j-api	1654	31.9
3	log4j	log4j	1189	22.9
4	com.google.guava	guava	1099	21.2
5	commons-io	commons-io	1038	20.0
6	org.slf4j	slf4j-log4j12	1009	19.5
7	commons-lang	commons-lang	782	15.1
8	javax.servlet	servlet-api	752	14.5
9	org.mockito	mockito-all	750	14.5
10	org.springframework	spring-context	667	12.9
11	org.apache.httpcomponents	httpclient	602	11.6
12	ch.qos.logback	logback-classic	58	11.2
13	commons-codec	commons-codec	580	11.1
14	com.google.android	android	539	10.4
15	commons-logging	commons-logging	513	9.9
16	org.springframework	spring-core	512	9.9
17	joda-time	joda-time	511	9.9
18	org.springframework	spring-webmvc	493	9.5
19	org.codehaus.jackson	jackson-mapper-asl	488	9.4
20	org.testng	testng	473	9.1

Table 19
Top 20 popular packages from the third-party APIs.

Packages	Library	RP_p/Rank	CO_p/Rank
org.junit	junit4	61.9%/5	2.80E+06/44
junit.framework	junit3	35.8%/10	1.43E+06/52
org.slf4j	slf4j	29.3%/13	4.47E+05/80
org.junit.runner	junit4	26.8%/16	3.50E+04/127
com.google.common.collect	guava	20.9%/23	5.09E+05/75
org.apache.commons.io	commons	20.0%/25	5.12E+04/123
com.google.common.base	guava	19.4%/27	2.85E+05/88
org.mockito	mockito	19.3%/28	4.11E+05/81
org.hamcrest	hamcrest	18.0%/31	2.57E+05/86
org.apache.commons.lang	commons	16.8%/33	7.85E+04/121
org.mockito.stubbing	mockito	16.6%/34	1.07E+05/95
org.apache.log4j	log4j	14.9%/37	1.86E+05/47
org.apache.http	httpcomponents	13.0%/40	7.22E+04/137
org.junit.runners	junit4	12.8%/41	1.31E+04/1013
org.apache.http.impl.client	httpclient	11.5%/43	1.31E+04/1008
org.apache.http.client.methods	httpclient	11.5%/44	2.12E+04/550
org.apache.commons.logging	commons	11.2%/46	2.44E+05/37
org.apache.http.client	httpclient	10.7%/49	9.29E+03/1537
javax.inject	javaee	10.7%/51	6.40E+04/150
org.junit.rules	junit4	10.7%/52	2.01E+04/591

boxplot. Take project *fabric8*[9] as an example of outliers, which is an integration platform for management of Java containers, totally requires 745 libraries in the *compile* phase.

5.2. Hotspots of API entities

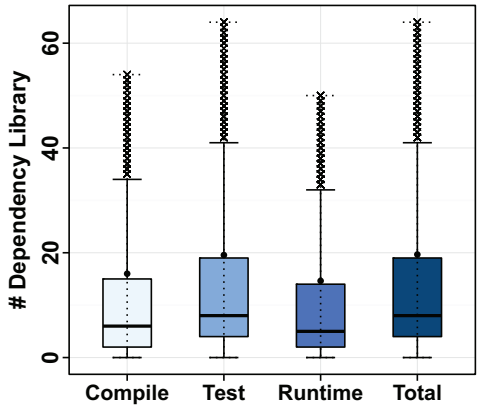
We also identify the hotspots of the third-party libraries. Tables 19–22 show the results, which share the similar structure with Table 8. Compared with the top 20 adopted third-party libraries in Table 18, the results of top 20 popular packages show no exceptions; they mainly come from the API libraries of `junit`, `guava` and `apache-*` series. The only distinctive package `org.hamcrest` from `hamcrest` is not shown in most popular list since `hamcrest` has been pulled into `junit4`. It is curious that popularity of some package exceeds the popularity of the library it belongs to, *e.g.* RP_p(`org.mockito`)=19.3% while RP_l(`org.mockito:mockito-all`)=14.5%. This is because one API entity might be packaged into libraries of multiple versions for distinct usage scenarios. In this example, `org.mockito` is involved in both `org.mockito:mockito-all` and `org.mockito:mockito-core` where they correspond to the entire version and the core version, respectively. Regarding the top popular classes and methods, most are covered by the libraries of `junit4` and `slf4j`. Instead, the most popular fields are from more libraries, but still within the top 20 libraries. As a more detailed representation, for each of the top 10 packages in Table 19, Table 23 further lists the most popular subordinative API entities (classes, method and fields) if exist.

We indeed discover many third-party libraries, which expand or facilitate the use of the core API, are also beloved by programmers, *e.g.* `guava`, `commons-io` and `joda-time`. The popularities of various packages from the third-party libraries even exceed those from the core API. It reflects the essentiality, on one hand, to introduce supplementary functionalities that are urgently requested by programmers into the core API; on the other hand, to revise the design of the core API to enhance its usability. More discussions are presented in Section 6.

5.3. Multiple versions analysis

A library is available in multiple versions as it evolves. Library designers usually encourage programmers to apply the newest release since it add new features, fix bugs or improve quality. However, programmers are cautious to select the most appropriate release for projects. We are interested in the distribution of the versions of the libraries that are adopted by projects.

[9] https://github.com/fabric8io/fabric8.



	Compile	Test	Runtime	Total
Min	0	0	0	0
Max	667	739	643	745
Mean	15.7	19.3	14.3	19.4
Median	6	8	5	8
Q1	2	4	2	4
Q3	15	19	14	19

Fig. 10. The distribution of dependency library usage by projects. The left figure shows the boxplot of size of the libraries *w.r.t.* projects. The statistics in the right table provides details of the boxplot.

Table 20
Top 20 popular fields from the third-party APIs.

Fields	Library	RP_f/Rank	SO_f/Rank
com.google.common.base.Charsets.*UTF_8*	guava	6.0%/51	4.94E+03/138
javax.ws.rs.core.MediaType.*APPLICATION_JSON*	javaee	5.7%/57	2.37E+03/345
javax.servlet.http.HttpServletResponse.*SC_OK*	tomcat	5.4%/65	3.19E+03/225
android.view.View.*VISIBLE*	android	5.3%/68	1.08E+04/45
org.springframework.web.bind.annotation.RequestMethod.*GET*	spring	5.2%/69	3.75E+03/195
android.view.View.*GONE*	android	5.1%/70	1.10E+04/44
android.os.Build.*VERSION.SDK_INT*	android	5.1%/73	3.43E+03/214
javax.servlet.http.HttpServletResponse.*SC_NOT_FOUND*	tomcat	5.0%/75	1.88E+03/458
org.springframework.web.bind.annotation.RequestMethod.*POST*	spring	4.7%/83	2.62E+03/293
android.widget.Toast.*LENGTH_SHORT*	android	4.5%/90	2.23E+03/377
javax.servlet.http.HttpServletResponse.*SC_INTERNAL_SERVER_ERROR*	javaee	4.4%/93	1.33E+03/670
android.view.ViewGroup.LayoutParams.*WRAP_CONTENT*	android	4.1%/103	3.85E+03/188
android.view.MotionEvent.*ACTION_DOWN*	android	4.0%/110	1.01E+03/938
android.view.MotionEvent.*ACTION_UP*	android	4.0%/110	9.28E+02/1027
android.view.View.*INVISIBLE*	android	3.7%/118	1.92E+03/442
android.view.ViewGroup.LayoutParams.*MATCH_PARENT*	android	3.7%/122	4.43E+03/158
javax.servlet.http.HttpServletResponse.*SC_BAD_REQUEST*	javaee	3.7%/122	2.06E+03/413
javax.mail.Message.RecipientType.*TO*	javaee	3.7%/122	5.60E+02/1884
android.widget.Toast.*LENGTH_LONG*	android	3.5%/131	1.55E+03/570
org.apache.http.HttpStatus.*SC_OK*	httpcore	3.5%/132	1.62E+03/546

Table 21
Top 20 popular classes from the third-party APIs.

Classes	Library	RP_c/Rank	SO_c/Rank
org.junit.Test	junit4	60.6%/25	7.81E+05/7
org.junit.Before	junit4	42.5%/43	4.66E+04/83
org.slf4j.Logger	slf4j	29.1%/74	5.68E+04/68
org.slf4j.LoggerFactory	slf4j	28.8%/75	5.25E+04/75
org.junit.After	junit4	28.4%/81	1.82E+04/231
org.junit.Assert	junit4	26.2%/89	2.86E+05/15
org.junit.runner.RunWith	junit4	25.9%/91	2.51E+04/167
javax.servlet.http.HttpServletRequest	servlet	21.5%/118	5.43E+04/72
javax.servlet.http.HttpServletResponse	servlet	20.1%/123	4.56E+04/85
org.junit.BeforeClass	junit4	19.8%/125	1.23E+04/336
junit.framework.TestCase	junit3	19.8%/128	4.02E+04/98
org.junit.Ignore	junit4	18.4%/142	1.04E+04/397
junit.framework.Assert	junit3	16.8%/153	1.05E+05/40
org.junit.AfterClass	junit4	14.1%/177	7.27E+03/356
org.apache.commons.io.IOUtils	commons	14.0%/178	9.52E+03/437
com.google.common.collect.Lists	guava	13.9%/181	3.16E+04/124
org.apache.commons.io.FileUtils	commons	13.8%/182	1.03E+04/399
org.apache.commons.lang.StringUtils	commons	13.4%/187	2.49E+04/169
javax.servlet.http.HttpServlet	javaee	13.1%/193	5.24E+03/734
javax.servlet.ServletRequest	javaee	10.7%/219	5.13E+03/744

Fig. 11(a) demonstrates the results. Only libraries containing more than one version adopted are shown in the figure. Among them, 2.9% (149/5196) of the libraries have over 20 different versions employed across the corpus. The library with maximum number of versions, `org.eclipse.jetty:jetty-server`, contains 76 distinct versions that are adopted by various software projects. We carefully check 149 libraries and find that most of them are hot third-party libraries. This stems from the fact that these libraries are popular and the lifecycle of releasing a new version is much shorter than unpopular libraries. Most libraries contain 2–20 versions in general.

We also select some representative libraries to demonstrate how distinct versions are adopted. Fig. 11(b)–(d) show the distributions of multiple versions usage from three widely-used libraries: `junit`, `guava` and `android`. Regarding the `junit`, 25 distinct versions are adopted where over 1/3 of projects select version 4.11. Versions of 4.10, 4.9 and 4.8.∗ were also popular among the library consumers. It is interesting that many projects still insist on the use of old `junit3`, especially 3.8.1. This may be due to `junit4` introduced many new features and provided an entirely new approach to define test cases. Migrating large projects from `junit3`

Table 22
Top 20 popular methods from the third-party APIs.

Methods	Library	RP_m/Rank	SO_m/Rank
org.junit.Assert.*assertEquals(java.lang.Object, java.lang.Object)*	junit4	41.5%/49	4.26E+05/5
org.junit.Assert.*assertTrue(boolean)*	junit4	38.6%/62	2.36E+05/15
org.junit.Assert.*assertEquals(long, long)*	junit4	36.2%/73	2.75E+05/11
org.junit.Assert.*assertNotNull(java.lang.Object)*	junit4	29.3%/109	9.97E+04/30
org.junit.Assert.*assertFalse(boolean)*	junit4	28.4%/120	8.33E+04/40
org.slf4j.LoggerFactory.*getLogger(java.lang.Class)*	slf4j	27.2%/132	4.99E+04/70
org.junit.Assert.*fail(java.lang.String)*	junit4	23.8%/163	5.06E+04/69
org.junit.Assert.*assertNull(java.lang.Object)*	junit4	23.1%/166	3.89E+04/94
org.slf4j.Logger.*info(java.lang.String)*	junit4	21.7%/176	4.42E+04/80
junit.framework.Assert.*assertTrue(boolean)*	junit3	21.5%/180	1.14E+05/27
org.junit.Assert.*assertTrue(java.lang.String, boolean)*	junit4	21.2%/187	5.17E+04/67
junit.framework.Assert.*assertEquals(java.lang.String, java.lang.String)*	junit3	20.5%/193	1.45E+05/20
junit.framework.Assert.*assertEquals(int, int)*	junit3	19.2%/208	1.35E+05/22
org.slf4j.Logger.*error(java.lang.String, java.lang.Throwable)*	slf4j	19.1%/214	2.81E+04/133
org.slf4j.Logger.*debug(java.lang.String)*	slf4j	19.0%/215	4.80E+04/76
org.mockito.Mockito.*mock(java.lang.Class)*	mockito	17.1%/243	5.50E+04/61
junit.framework.Assert.*assertEquals(java.lang.Object, java.lang.Object)*	junit3	16.9%/250	1.12E+05/28
org.slf4j.Logger.*warn(java.lang.String)*	slf4j	16.4%/262	1.44E+04/260
org.mockito.Mockito.*when(java.lang.Object)*	mockito	16.1%/271	9.82E+04/32
org.slf4j.Logger.*error(java.lang.String)*	slf4j	15.6%/286	1.55E+04/244

Table 23
Top popular subordinative API entities correspond to top 10 packages.

Packages [Rank(RP_p)]		Top five subordinative API entities [Rank(RP)]
org.junit[a] [5]	Classes:	*Test* [25], *Before* [43], *After* [81], *Assert* [89], *BeforeClass* [125]
	Methods:	Assert.*assertEquals(java.lang.Object, java.lang.Object)* [49], Assert.*assertTrue(boolean)* [62], Assert.*assertEquals(long,long)* [73], *Assert.assertNotNull(java.lang.Object)* [109], Assert.*assertFalse(boolean)* [120]
junit.framework [10]	Classes:	*TestCase* [128], *Assert* [153], *Test* [276], *TestSuite* [280], *AssertionFailedError* [551]
	Methods:	Assert.*assertTrue(boolean)[180]*, Assert.*assertEquals(java.lang.String, java.lang.String)* [193], Assert.*assertEquals(int, int)* [208], Assert.*assertEquals(java.lang.Object, java.lang.Object)* [250], Assert.*assertNotNull(java.lang.Object)* [302]
	Fields:	TestResult.*fErrors* [7364], TestResult.*fFailures* [8291], TestSuite.*fTests* [11052], ComparisonFailure.*fActual* [11052], ComparisonFailure.*fExpected* [11052]
org.slf4j [13]	Classes:	*Logger* [74], *LoggerFactory* [75], *MDC* [2796], *Marker* [2987], *ILoggerFactory* [3406]
	Methods:	LoggerFactory.*getLogger(java.lang.Class)* [132], Logger.*info(java.lang.String)* [176], Logger.*error(java.lang.String,java.lang.Throwable)* [214], Logger.*debug(java.lang.String)* [215], Logger.*warn(java.lang.String)* [262]
org.junit.runner [16]	Classes:	*RunWith* [91], Parameterized.*Parameters* [361], *Parameterized* [367], *Suite* [472], *Description* [489]
	Methods:	ParentRunner.*getTestClass()* [3555], Description.*getMethodName()* [3631], Description.*getDisplayName()* [6812], ParentRunner.*run(org.junit.runner.notification.RunNotifier)*[7567], Description.*getTestClass()* [7618]
	Fields:	MethodSorters.*NAME_ASCENDING* [1742], MethodSorters.*JVM* [11052], Description.*EMPTY* [14189], Description.*TEST_MECHANISM* [22882]
com.google.common.collect [23]	Classes:	*Lists* [181], *Maps* [229], *Sets* [253], *ImmutableList* [279], *Iterables* [292]
	Methods:	Lists.*newArrayList()* [531], Maps.*newHashMap()* [615], List.*newArrayList(java.lang.Object[])* [640], Lists.*newArrayList(java.lang.Iterable)* [713], Sets.*newHashSet()* [811]
	Fields:	BoundType.*CLOSED* [5625], BoundType.*OPEN* [5625], MapMaker.*useCustomMap* [11052], MapMaker.*keyStrength* [11052], MapMaker.*valueStrength* [11052]
org.apache.commons.io [25]	Classes:	*IOUtils* [178], *FileUtils* [182], *FilenameUtils* [588], *LineIterator* [3298], *Charsets* [5190]
	Methods:	FileUtils.*deleteDirectory(java.io.File)* [940], IOUtils.*toString(java.io.InputStream)* [959], IOUtils.*copy(java.io.InputStream, java.io.OutputStream) [977], IOUtils.closeQuietly(java.io.InputStream) [992], FileUtils.readFileToString(java.io.File) [1309]*
	Fields:	Charsets.*UTF_8* [3106], IOCase.*INSENSITIVE* [4572], IOUtils.*LINE_SEPARATOR* [5181], IOUtils.*LINE_SEPARATOR_UNIX* [8291], FileUtils.*ONE_MB* [14189]
com.google.common.base [27]	Classes:	*Function* [246], *Joiner* [285], *Predicate* [312], *Preconditions* [319], *Charsets* [393]
	Methods:	Joiner.*on(java.lang.String)* [760], Joiner.*join(java.lang.Iterable)* [795], Preconditions.*checkNotNull(java.lang.Object)* [827], Preconditions.*checkArgument(boolean, java.lang.Object)* [935], Preconditions.*checkNotNull(java.lang.Object, java.lang.Object)* [1059]
	Fields:	Charsets.*UTF_8* [51], CharMatcher.*WHITESPACE* [2034], CaseFormat.*UPPER_CAMEL* [2177], Charsets.*US_ASCII* [2274], CaseFormat.*LOWER_CAMEL* [2470]
org.mockito [28]	Classes:	*Mockito* [274], *Mock* [314], *ArgumentCaptor* [509], *MockitoAnnotations* [585], *Matchers* [855]
	Methods:	Mockito.*mock(java.lang.Class)* [243], Mockito.*when(java.lang.Object)* [271], Mockito.*verify(java.lang.Object)* [396], Mockito.*verify(java.lang.Object, org.mockito.verification.VerificationMode)* [541], Matchers.*any(java.lang.Class)* [544]
	Fields:	Mockito.*RETURNS_DEEP_STUBS* [2367], Mockito.*CALLS_REAL_METHODS* [3962], Mockito.*RETURNS_MOCKS* [6093], Mockito.*RETURNS_SMART_NULLS*[116736], Answers.*RETURNS_SMART_NULLS* [116736]
org.hamcrest [31]	Classes:	*Description* [465], *Matcher* [497], *BaseMatcher* [919], *Matchers* [933], *TypeSafeMatcher* [1264]
	Methods:	CoreMatchers.*is(java.lang.Object)* [706], CoreMatchers.*equalTo(java.lang.Object)* [920], MatcherAssert.*assertThat(java.lang.Object, org.hamcrest.Matcher) [1070], Matchers.is(java.lang.Object) [1099], Matchers.equalTo(java.lang.Object) [1247]*
org.apache.commons.lang [33]	Classes:	*StringUtils* [187], *ArrayUtils* [574], *StringEscapeUtils* [629], *RandomStringUtils* [1307], *NotImplementedException* [1595]
	Methods:	StringUtils.*isEmpty(java.lang.String)* [949], StringUtils.*isBlank(java.lang.String)* [1044], StringUtils.*isNotBlank(java.lang.String)* [1216], StringUtils.*join(java.util.Collection, java.lang.String)* [1602], StringUtils.*join(java.lang.Object[], java.lang.String)* [1647]
	Fields:	StringUtils.*EMPTY* [510], SystemUtils.*IS_OS_WINDOWS* [2232], SystemUtils.*LINE_SEPARATOR* [3470], ArrayUtils.*EMPTY_STRING_ARRAY* [3808], SystemUtils.*IS_OS_LINUX* [4572]

[a] No fields are defined in this package. The same situation occurs in package *org.slf4j* and *org.hamcrest*.

to `junit4` requires massive code rewriting and refactoring. Regarding the `guava`, 32 versions in total are adopted across our corpus. Distinguished from `junit`, its usage scatters in more distinct versions. Most projects select stable releases (*e.g.* 15.0 and 18.0) instead of release candidates (*e.g.* 14.0_*rc*3 and 18.0_*rc*1). It is unexpected that the usage of 14.0.1 is more frequent than its subsequent versions (*e.g.* 15.0–18.0). Regarding the `android`, 31 versions are adopted where its version adoptions are more concentrated, especially on 4.0.1.2.

The above cases indicate that programmers are cautious about selecting concrete versions. Projects do not employ the newest version of the library in most cases because of the concern on stability, robustness and security. Various projects conservatively select previous stable release instead. Higher popularity of a specific library version indicates higher maturity to some extent. It is significant to remind programmers to switch to the proper version at an appropriate time by monitoring the usage distribution of the library of various versions over time. For projects starting from scratch, it is also vital to recommend the proper version to programmers.

6. Applications

Our work and results offer a number of insights, inspiring some potential applications:

API design and restriction. Recent studies have shown that open-source API libraries have been one of the most influential factors that affect the selection of programming languages [5]. Well-designed, easy-to-use APIs definitely induce higher acceptances

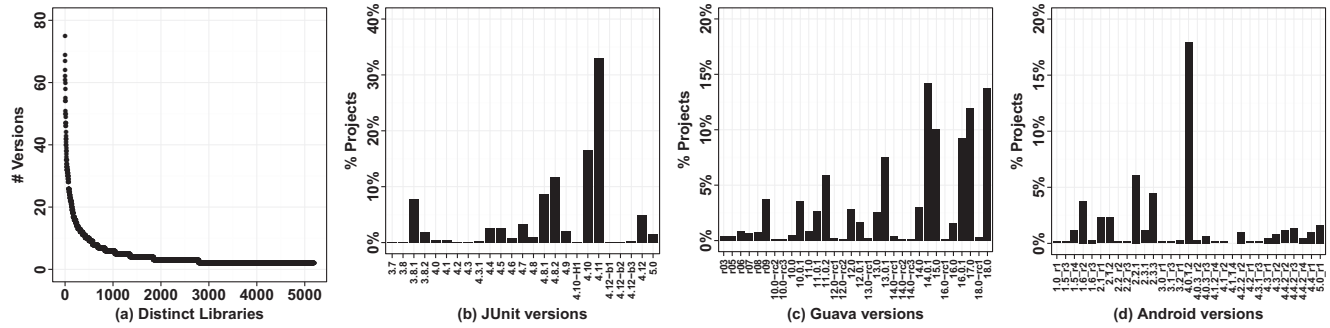


Fig. 11. (a) shows the third-party libraries usage distribution *w.r.t.* their versions. (b) `junit`, (c) `guava` and (d) `android` are three cases that demonstrate how libraries are used across multiple versions.

from programmers. However, current API design is usually artistic, driven by the aesthetic concerns and intuitions of language architects. Designers usually have limited knowledge on how programmers actually use an API. As more open-source repositories have been publicly available, understanding how APIs are employed is critical. We investigate the API usage from the perspectives of *frequency, popularity* and *coverage*, and expect API designers to improve the designs via a *data-driven* approach. Based on our empirical results, we make an attempt to provide some recommendations to the API designers: (1) be cautious to modify the interfaces of those API entities with high popularity, as such updates may cause a wider influence to current software systems that depend on them; (2) give more emphasis to API entities with high frequency and optimize their implementations, as they may be the key elements that affect the system's performance; (3) do not neglect API entities with low coverage; reconstruct them, *e.g.* generate more efficient compact profiles to exclude rarely covered packages, or re-layout the packages with many uncovered classes and move them to the optional sections, to keep the API succinct; for API entities that are rarely used (with both low frequency and popularity) in practice, identify the possible reason and consider how to simplify/optimize/re-design them. In addition, monitoring API usage from core library *vs.* functionality substitutable third-party libraries (*e.g.* Java `Date` *vs.* `Joda` time library[10]) can also help designers of the core API identify the weak spots precisely. Introducing and integrating well-designed and widely-used third-party libraries into the core API through the API usage data can cater for most of the programmers' needs.

API libraries contain not only *good* features, but also *bad* features. Poorly-designed API entities inevitably induce programmers to write bad code, which probably impact the quality of the software. Take method `java.lang.String.substring(int,int)` as an example, its usage may lead to memory leak[11] prior to Java 6. Normally, these buggy API entities are not encouraged to adopt before they have been fixed. In order to prevent programmers from abusing these features, restricting the use of certain API entities that contain bad features is indispensable. We can learn and construct such subset (without restricted API entities) for given requirements (e.g. performance, reliability, security) from high-quality practical code and enforce their proper use.

API compact profile construction. As features and functionalities have been introduced continuously, the footprint of the core API is becoming gigantic and complex, gradually raising the minimum requirements for devices. The usage of *compact profiles* indeed reduces the consumption of resources for some applications. However, we discover that about two fifths of the projects within our corpus still require to load the entire core API. In addition, among the projects that require *CP2* to correctly compile and run, the use of the API entities from *CP2(A)* is tiny. Similar behaviors are found in projects that require *CP3* or *Full* profiles. The design of current compact profiles is more driven by the functionality partitions on packages. However, from the perspective of reducing resource consumption, the utilization of the compact profiles is low in practice.

As we have obtained the data on how projects employ API entities, better compact profiles can be computed via a data-driven approach. By analyzing the behaviors of co-used API entities (especially packages), we can generate leaner compact profiles to ensure fewer resources are consumed by most projects.

Library recommendation. Software projects highly depend on third-party libraries, and the size of the dependencies is large. Manually selecting appropriate libraries is a burdensome work. Our data involve rich co-use information on API libraries (*e.g.* Projects that adopt `commons-io` often adopt `commons-lang`). Through mining association rules of library usage from a large corpus, it is not difficult to recommend best-fit candidates based on other project dependencies in use. In addition, Milena et al. proposed an approach, based on the *popular vote of the majority*, that recommends a most suitable version (*i.e.* with a higher usage) for a given library by monitoring the usage trend of all versions [1]. Different from their solution, our approach is more straightforward. Based on the library version co-use information (*e.g.* `commons-io:2.4` and `commons-lang:3.1` are often adopted simultaneously by projects), we can suggest a library with its proper version, which can also avert version conflicts among multiple dependencies.

Language API education. Novice programmers are disorientated by the large (and ever-growing) API libraries easily. API entities with frequent usage and high coverage in practice usually indicate their fundamentality and importance to the entire API library. Identifying the *essence* of the APIs is essential as it suggests a direct starting point for API learners. Hence, to improve the understandability of an API, it is significant to embed the statistics of API *frequency, coverage* and *popularity* into its corresponding API documentation. To some extent, it guides novice programmers to quickly identify *hotspots* of the APIs, and also alert programmers to use *coldspots* cautiously in practical development.

Besides, novice programmers are more likely to make mistakes when they learn to employ new API libraries, *e.g.* choose the right methods from the alternatives within an API library [4].

[10] The design of standard date classes from the core API prior to Java 8 is poor. On-line discussions suggest that `Joda` API is the priority for programmers. Then, the core API integrates `Joda` and create a new `Time` API since Java 8.

[11] http://bugs.java.com/view_bug.do?bug_id=6294060 .

Understanding such potential *correlation* (*i.e.* the use of some API entity tends to make novices write buggy code) and identifying these error-prone API entities are valuable to API learners. Hence, it would be interesting to link the usage of API entities with bugs that have been fixed and recorded in bug repositories to statistically investigate how the usage of API entities correlates with code quality. Timely alert to both novices and experienced developers to these "risky" entities before employing them, or recommending them alternative, easy-to-use candidates are both effective measures to avoid introducing unnecessary defects. The mined correlations are also strong evidence for API restriction.

7. Threats to validity

Construct validity. The construct validity of our study rests on the measurements performed, in particular related to the corpus construction, dependency resolution and API entity resolution.

Regarding the corpus construction, we select and download over 5000 projects with diverse characteristics, such as project sizes and domains. All projects are obtained from GitHub, one of the most popular and widely-used project hosting services, that hosts massive amounts of git-based projects. The reason we select Git is that it offers great convenience on checking out multiple snapshots. Thus, there is no indication that the projects we select are biased toward any specific project type. In addition, all projects within the corpus are managed by Maven, one of the most widely-used Java build tool. The reason we select Maven is that it facilitates the acquisition of all dependencies by projects automatically. Similarly, projects in the corpus are selected based on given criteria. The selection process does not involve our own human bias.

Regarding the dependency resolution, as Maven manages every project in our corpus, developers are supposed to list all required dependencies and associated remote repositories that can retrieve these dependencies in the projects' Maven configuration files. However, *not all* the dependencies can be retrieved from given repositories since some of them are not available or accessible on the Internet. As a reparative strategy, we manually supplement some popular and recognized repositories to mitigate such issue. In our effort, only 4% of dependencies are not resolved correctly. The percentage is insignificant and it does not affect the results of dependency analysis.

The failure of obtaining all required dependencies inevitably results in the failure of correctly resolving some of the API entities adopted by the project. However, based on the above strategy, we only find 0.06% of the classes, 2.27% of the methods and 0.18% of the field in use are not resolved correctly. Compared to the study by Lämmel et al. [16], the percentage of failed resolution in our study has been reduced significantly, which is an acceptable rate and does not impact the results.

Internal validity. We identify some uncontrollable factors that may have affected the results of our API usage analysis. One possible factor is language and API constraints. On one hand, some language conventions potentially expand the use of some API entities. For example, if one project contains a `main` method, it implies that the class `java.lang.String` is being employed, even if there is no manipulation on this string object. We check all 5185 projects and discover that, 38.9% of them contain at least one `main` method, and the total size of the `main` methods is small. It has little influence on the popularity and frequency of `java.lang.String`, since this class is mainly used in non-main methods. Exception handling is another language constraint that might affect the use of exception-related classes. For example, when a programmer uses class `java.io.FileReader` to read data from a file, he is forced to catch the checked exceptions, *e.g.* `java.io.IOException` in this case, even if there is no operation on this object. Such statically declared exceptions demanded by exception handle mechanism indeed increase the use of exception classes, like `IOException` which is highly ranked in Table 10. On the other hand, the architectural constraints might decrease or eliminate the use of some "special classes" that are not intended to be explicitly declared in the code. As an example, JDBC-drivers are not instantiated by programmers by declaring the driver classes, but to pass a name to the classloader which then instantiates the class through the reflection mechanism. In the case of a statement `Class.forName(''com.mysql.jdbc.Driver'');`, the class `com.mysql.jdbc.Driver` never appear directly in the code, which is also ignored by our analysis process. Since reflection-related API entities are widely used, this might have some effect on the statistics of these "special classes".

Another factor is related to the use of IDEs. Modern IDEs, such as Eclipse directly generate build-in annotations in the code, *e.g.* `@Override`, whenever a class does override a method. This is one of the most important reasons that annotations like `@Override` and `@SuppressWarnings` are among top 20 popular classes. However, it is difficult to distinguish whether such annotations are written by programmers or auto generated by IDEs. Involving the generated annotations, which are not real programmers' behaviors of using API entities, does have some impact on the actual usage of annotation-related classes.

External validity. Threats to external validity are concerned with whether the results are applicable in general. In this study, we select over 5000 most popular Java projects to obtain the general findings of language API usage. However, projects under analysis are all from open-source community, developed in Java, and managed by Maven. It would be desirable to analyze more varieties of applications (*e.g.* enterprise applications), developed in more diverse programming languages (*e.g.* C++) and managed by distinct build tools (*e.g.* Ivy and Gradle) to confirm our general conclusions.

8. Related work

API usage analysis. To some extent, our work is analogous to [16,31]. Homan et al. studied the usage of the API entities from Java standard API over 39 projects [31]. Our study conducts a similar experiment on a larger corpus (containing over 5000 projects). Our results demonstrate that 15.3% of the classes and 41.2% of the method are not used at all, which is inconsistent with their results showing that 50% of the classes and 80% of the methods are never used. The coverage highly depends on the corpus scale. Both of the results are reasonable as the corpus sizes differ. Nevertheless, our results seem to be more convincing to readers because our corpus is much larger than theirs. Lämmel et al. also conducted an AST-based API-usage analysis on 1, 639 projects, including API footprint analysis, coverage analysis and framework-like API usage [16]. Our work involves similar analysis, but differs in several dimensions: (1) they manually downloaded the missing dependencies and used the shared libraries to make projects buildable. Significantly different from their approach, we automatically download the dependency libraries that each project requires, with the assistant of Maven. This approach greatly improves the precision of the API usage resolution and make our results more trustworthy; (2) we design more comprehensive and systematical metrics to measure the API usage; (3) we study both core API and third-party APIs exhaustively, including many extra interesting angles, *e.g.* the utilization of the compact profiles in the core API, the usage of the deprecated API entities and version issue of third-party APIs. In addition, some studies focused on the usage of Eclipse APIs [32,33]. We do not concentrate on one concrete API library and try to analyze almost all API libraries that are covered by our corpus. Thummalapenta and Xie proposed a code-search-engine-based approach

that detects API hotspots and coldspots [28] by mining code examples gathered from open-source repositories on the web. We also conduct this similar study through a completely different way, *i.e.* counting the API usage from the resolved ASTs.

API usage analysis has also inspired many applications, *e.g.* improve the usability of APIs [34,35], recommending the use of API entities [36], libraries [37] and versions [1], facilitating API migrations [6–11] and supporting software maintenance [12].

Studies of API properties. Much of the recent research has examined a multitude of API properties, *e.g.* stability [38,39], usage diversity [40] and usability [27,41,42]. McDonnell et al. conducted an in-depth case study on co-evolution behaviors of `android` API and dependent applications to understand the API stability [38]. They found 28% of API references in client applications were outdated with a median lagging time of 16 months. Our results also confirm that libraries with distinct versions are employed by projects simultaneously. The newest release is not always the preferred choice for programmers. Raemaekers et al. proposed four metrics (*i.e.* the weighted number of removed methods, the amount of changes in existing methods, the ratio of changes in new to old methods and the ratio of new methods) to calculate the stability of public interfaces and implementations of a library [39]. Mendez et al. studied API usage diversity (*i.e.* the different statically observable combinations of methods called on the same object), and found significant usage diversity for many classes [40]. Stylos and Myers found that method placement can have large usability impact in object-oriented APIs [41]. We believe our API analysis data can also assist in the studies of similar API properties.

Studies of software characteristics. As more open-source repositories (*e.g.* Github, Sourceforge and Bitbucket) have been publicly available, many researchers have started to understand software characteristics through empirical approaches. Dyer et al. conducted a large-scale study on Java features usage from the perspective of the language syntax [43]. Baxter et al. presented an in-depth study on the structures of Java programs by analyzing 56 projects [25]. They measured several key structural attributes to ascertain if they follow the power-laws. Likewise, Grechanik et al. mined structure usage in more than 2000 Java projects [44]. Gabel and Su studied *uniqueness* that software generally lacks uniqueness which most code snippets that developers write already exist [45]. Hindle et al. studied *naturalness* that the actual code is regular and predictable, like natural language utterances [46]. They followed the uniqueness study and confirmed the software "syntactic redundancy". Tu et al. further studied the *localness* that human-written programs were localized [47]. They introduced a cache language model that optimized the n-gram model by involving local regularities of the code to improve code suggestion accuracy. Other than the above studies that analyzed the language usage from the lexical or syntactic level, we study the language from another view, *i.e.* the API level.

9. Conclusion and future work

This paper has presented a large-scale study of how Java's APIs are used in practice by analyzing more than 5000 open-source Java projects. Our study has exposed interesting quantitative information to help understand how APIs from the core library and third-party libraries have been used. There are several interesting directions for future work. In detail, we plan to (1) conduct a more comprehensive study on a variety of other programming languages to increase the external validity of our findings; (2) analyze API usage under more scenarios: how are API entities co-used in practice (*e.g.* what API entities co-occur frequently); how are they used within certain syntactic structures (*e.g.* what API methods are employed frequently in nested loops); (3) apply our empirical data to optimize the design and construction of API libraries, especially to generate better compact profiles that minimize the resource consumption for devices; (4) investigate the possibility of adopting the actual API usage data to facilitate API-based code recommendation; (5) understand how the usage of API entities correlates with code quality (*e.g.* defect rate), and mine such correlations for API restriction and education.

Acknowledgments

The work is supported by the National Natural Science Foundation of China under grant no. 61572126, the Huawei Innovation Research Program (HIRP) under grant no. YB2013120195 and the Scientific Research Foundation of Graduation School of Southeast University grant no. YBJJ1313.

References

[1] Y.M. Mileva, V. Dallmeier, M. Burger, A. Zeller, Mining trends of library usage, in: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol), 2009, pp. 57–62.
[2] U. Sandberg, Tired of Date and Calendar?http://www.jayway.com/2006/09/16/tired-of-date-and-calendar/ (accessed 10.04.15).
[3] Your Language Sucks. https://wiki.theory.org/YourLanguageSucks (accessed 10.04.15).
[4] M. Robillard, What makes APIs hard to learn? Answers from developers, IEEE Softw. 26 (6) (2009) 27–34.
[5] L.A. Meyerovich, A.S. Rabkin, Empirical analysis of programming language adoption, in: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA), 2013, pp. 1–18.
[6] J.Y. Gil, I. Maman, Micro patterns in Java code, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005, pp. 97–116.
[7] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, MAPO: mining and recommending API usage patterns, in: Proceedings of the 23rd European Conference on Object-oriented Programming (ECOOP), 2009, pp. 318–343.
[8] G. Uddin, B. Dagenais, M.P. Robillard, Temporal analysis of API usage concepts, in: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 804–814.
[9] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, D. Zhang, Mining succinct and high-coverage API usage patterns from source code, in: Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013, pp. 319–328.
[10] H.A. Nguyen, T.T. Nguyen, G. Wilson Jr., A.T. Nguyen, M. Kim, T.N. Nguyen, A graph-based approach to API usage adaptation, in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2010, pp. 302–321.
[11] M. Nita, D. Notkin, Using twinning to adapt programs to alternative APIs, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE), 2010, pp. 205–214.
[12] V. Bauer, L. Heinemann, Understanding API usage to support informed decision making in software maintenance, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, pp. 435–440.
[13] Maven. http://maven.apache.org/ (accessed 10.04.15).
[14] J. Cocke, V. Markstein, The evolution of RISC technology at IBM, IBM J. Res. Dev. 34 (1) (1990) 4–11.
[15] Java SE. http://www.oracle.com/technetwork/java/javase/ (accessed 10.04.15).
[16] R. Lämmel, E. Pek, J. Starek, Large-scale, AST-based API-usage analysis of open-source Java projects, in: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), 2011, pp. 1317–1324.
[17] Eclipse EGit. http://www.eclipse.org/egit/ (accessed 10.04.15).
[18] Eclipse Aether. http://eclipse.org/aether/ (accessed 10.04.15).
[19] Maven API. http://maven.apache.org/ref/3.3.1/index.html (accessed 10.04.15).
[20] Eclipse JDT. http://www.eclipse.org/jdt/ (accessed 10.04.15).
[21] Apache Commons BCEL. https://commons.apache.org/proper/commons-bcel/ (accessed 10.04.15).
[22] W. Lim, Effects of reuse on quality, productivity, and economics, IEEE Softw. 11 (5) (1994) 23–30.
[23] L. Heinemann, Effective and Efficient Reuse with Software Libraries, Technische Universität München, 2012 Ph.D. thesis.
[24] R. Wheeldon, S. Counsell, Power law distributions in class relationships, in: Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003. Proceedings, 2003, pp. 45–54.
[25] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, E. Tempero, Understanding the shape of Java software, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2006, pp. 397–412.

[26] P. Louridas, D. Spinellis, V. Vlachos, Power laws in software, ACM Trans. Softw. Eng. Methodol. 18 (1) (2008) 2:1–2:26.
[27] J.M. Daughtry, U. Farooq, J. Stylos, B.A. Myers, API usability: CHI'2009 special interest group meeting, in: Proceedings of the 27th International Conference on Human Factors in Computing Systems (CHI), Extended Abstracts Volume, 2009, pp. 2771–2774.
[28] S. Thummalapenta, T. Xie, SpotWeb: detecting framework hotspots and coldspots via mining open source code on the web, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2008, pp. 327–336.
[29] Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/(accessed 10.04.15).
[30] F. Long, D. Mohindra, R. Seacord, D. Sutherland, D. Svoboda, The CERT Oracle Secure Coding Standard for Java, Addison-Wesley Professional, 2011.
[31] H. Ma, R. Amor, E. Tempero, Usage patterns of the Java standard API, in: Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC), 2006, pp. 342–352.
[32] J. Businge, A. Serebrenik, M.G.J.v.d. Brand, Eclipse API usage: the good and the bad, Softw. Qual. J. 23 (2013) 107–141.
[33] R. Holmes, R.J. Walker, Informing eclipse API production and consumption, in: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (ETX), 2007, pp. 70–74.
[34] J. Stylos, A. Faulring, Z. Yang, B. Myers, Improving API documentation using API usage information, in: IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC), 2009, pp. 119–126.
[35] C. De Roover, R. Lammel, E. Pek, Multi-dimensional exploration of API usage, in: Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC), 2013, pp. 152–161.
[36] Y.M. Mileva, V. Dallmeier, A. Zeller, Mining API popularity, in: Proceedings of the Fifth International Academic and Industrial Conference on Testing—Practice and Research Techniques (TAIC-PART), 2010, pp. 173–180.
[37] F. Thung, D. Lo, J. Lawall, Automated library recommendation, in: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 182–191.
[38] T. McDonnell, B. Ray, M. Kim, An empirical study of API stability and adoption in the android ecosystem, in: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 70–79.
[39] S. Raemaekers, A. van Deursen, J. Visser, Measuring software library stability through historical version analysis, in: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 378–387.
[40] D. Mendez, B. Baudry, M. Monperrus, Empirical evidence of large-scale diversity in API usage of object-oriented software, in: Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 43–52.
[41] J. Stylos, B.A. Myers, The implications of method placement on API learnability, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, pp. 105–112.
[42] J. Bloch, How to design a good API and why it matters, in: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, 2006, pp. 506–507.
[43] R. Dyer, H. Rajan, H.A. Nguyen, T.N. Nguyen, Mining billions of AST nodes to study actual and potential usage of Java language features, in: Proceedings of the 36th International Conference on Software Engineering (ICSE), 2014, pp. 779–790.
[44] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, C. Ghezzi, An empirical investigation into a large-scale Java open source code repository, in: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2010, pp. 11:1–11:10.
[45] M. Gabel, Z. Su, A study of the uniqueness of source code, in: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2010, pp. 147–156.
[46] A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 837–847.
[47] Z. Tu, Z. Su, P. Devanbu, On the localness of software, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 269–280.