




What are the characteristics of popular APIs? A large-scale study on Java, Android, and 165 libraries

Caroline Lima¹ · Andre Hora² 

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Software systems are commonly implemented with the support of libraries, which provide features via APIs. Ideally, APIs should have some characteristics, for example, they should be well documented and stable so that client systems can confidently rely on them. However, not all APIs are equal in number of clients: while some APIs are very popular and used worldwide, other may face much lower usage rates. In this context, one question appears: *are there particular characteristics that differentiate popular APIs from ordinary APIs?* Answering this question can uncover how worldwide APIs are actually implemented and maintained, revealing practices to better support both research and development on APIs. In this paper, we assess the characteristics of popular APIs, including their size, legibility, documentation, stability, and client adoption. We analyze 1491 APIs provided by Java, Android, and 165 libraries. We detect that popular APIs are distinct from ordinary ones, for example, often, popular APIs are larger, have more comments, and are more unstable than ordinary APIs. Finally, we provide a set of lessons learned from the popular APIs on factors that developers can control, such as the exposure of public methods and the API stability.

Keywords Software library · Framework · API popularity · API usage · Software evolution · Software repository mining

1 Introduction

Software libraries¹ are commonly used to support development, providing source code reuse, improving productivity, and decreasing costs (Moser and Nierstrasz 1996; Konstantopoulos et al. 2009; Raemaekers et al. 2012). They cover distinct usage scenarios

¹We use the term *library* to designate both frameworks and libraries.

✉ Andre Hora
andrehora@dcc.ufmg.br

Caroline Lima
caroline.correa@aluno.ufms.br

¹ Faculty of Computing, UFMS, Belo Horizonte, Brazil

² Department of Computer Science, UFMG, Belo Horizonte, Brazil

(De Roover et al. 2013), such as mobile and web development, testing, and GUI creation. These features are provided to client systems via application programming interfaces (APIs), which are contracts that clients rely on to build their applications (Reddy 2011). Ideally, libraries should have some characteristics, for example, they should be well documented and stable so that clients can confidently rely on them. In this context, the literature shows the opposite; overall, libraries face lack of documentation or are unstable (e.g., Thummalapenta and Xie 2008; Eisenberg et al. 2010; Wu et al. 2010; Robbes et al. 2012; McDonnell et al. 2013; Bogart et al. 2016; Xavier et al. 2017; Hora et al. 2018b; Brito et al. 2018c; Kula et al. 2018b).

However, we notice that not all libraries (e.g., JUnit) nor their APIs (e.g., `org.junit.Test`) are equal regarding their number of client systems: while some of them are very popular and used worldwide, others may face much lower usage rates (Holmes and Walker 2007; Zerouali and Mens 2017). For example, by assessing 4532 software systems, it was detected that 97% relied on the JUnit testing framework while only 12% used its competitor TestNG (Zerouali and Mens 2017). At the API level, the differences are also large: extending this assessment to a much larger set with 260K systems (Dyer et al. 2013), we found that 20% adopted the JUnit API `org.junit.Test`, while only 1% used the TestNG API `org.testng.annotations.Test`. Similarly, the JDK utility API `java.util.List` is used by 50% of those 260K systems, while a closely related API provided by Google (`com.google.common.collect.Lists`) is much less popular, being adopted by only 2%.

Thus, we notice that some APIs clearly dominate the market due to their well-known credibility or usage convenience. For instance, JUnit APIs are the oldest and the de-facto standard for unit testing Java systems, while the usage of JDK standard APIs is very convenient for Java developers.

In this context, one important question appears: besides their credibility or usage convenience, *are there particular characteristics that differentiate popular APIs from ordinary APIs?* For example, are they distinct regarding documentation, stability, changeability, complexity, legibility, or other aspects? Answering this question can uncover how worldwide APIs are actually implemented and maintained, revealing practices to better support both research and development on APIs. In a related research line, the literature has been assessing the attributes of popular software artifacts to learn from them; for instance, previous work has studied the practices of popular Android mobile apps (Tian et al. 2014) and the attributes of popular GitHub projects (Borges et al. 2016).

In this paper, we assess the characteristics of popular APIs by analyzing both the provider and the client perspectives. Specifically, we analyze 1491 APIs provided Java standard APIs, Android framework, and 165 highly used libraries. We then categorize these APIs regarding their level of popularity in number of clients. We contrast these categories of APIs regarding a set of *source code* and *evolutionary* metrics. Finally, we assess the client perspective to better understand *when* these APIs are adopted and *how* they are used by clients. Particularly, we answer the following research questions:

- *RQ1 (API Code): What are the code characteristics of popular APIs?* We assess the source code of the studied APIs in terms of size, complexity, legibility, and documentation to better understand how worldwide APIs are actually implemented. We find that the majority of the popular APIs have more code comments than the ordinary ones; however, depending of the ecosystem, they may also be larger and more complex.

- *RQ2 (API Evolution): What are the evolutionary characteristics of popular APIs?* We analyze evolutionary aspects of the studied APIs by considering their changeability, contribution, and stability to assess how worldwide APIs are maintained. We find that the majority of the popular APIs have more changes and are more unstable than the ordinary ones; however, surprisingly, they do not have necessarily more contributors.
- *RQ3 (Client Adoption): When are popular APIs adopted by client systems?* We verify whether client systems establish dependencies to the popular APIs in initial or late stages of the development cycle. We detect that the popular APIs are often earlier adopted by the clients than the ordinary APIs, suggesting their importance since initial phases of development.
- *RQ4 (Client Usage): How are popular APIs used by client systems?* We assess whether the popular APIs tend to be used together with other APIs or alone. We find that the popular APIs are commonly used together with other APIs and packages; however, the difference is very small when compared with the ordinary APIs.

Overall, we find that (1) popular APIs are largely different from ordinary APIs; (2) the ecosystem plays an important role in the analysis of popular APIs; (3) popular APIs expose a large proportion of API elements; (4) popular APIs have more code comments; and (5) popular APIs are more likely to change over time. We then provide direct implications to researchers and practitioners. For example, when studying API characteristics, researchers should assess ecosystems separately, as they differ in the majority of the analyzed metrics. This paper has four contributions:

1. To our knowledge, we are the first to assess the characteristics of popular APIs provided by worldwide libraries in order to better understand their development and maintenance practices;
2. We contrast the characteristics of popular APIs against APIs in other level of popularity, particularly, ordinary and unpopular ones;
3. We compare the characteristics of popular APIs in three ecosystems: Java standard APIs, Android framework, and other 165 libraries;
4. We provide a set of lessons learned on factors that developers can control, such as the exposure of public methods, the ratio of commented code, the amount of contributors, and the API stability.

Structure of the paper: Section 2 motivates the study of popular APIs. Section 3 presents the study design, while Section 4 reports the results. Section 5 discusses our major findings and lessons learned. Section 6 presents the threats to validity. Finally, Section 7 discusses related work and Section 8 concludes.

2 API popularity

Libraries provide *interfaces* to software components created to be reused by client applications (Reddy 2011). We consider as an API any public interface that can be imported by a client system. In this study, we work at the API level, that is, we assess APIs rather than libraries. Examples of APIs commonly used by Java developers are `java.util.Date`, `org.junit.Assert`, and `org.slf4j.LoggerFactory`.

Figure 1 presents the top 500 most used APIs in Java, as extracted from a large dataset with 260K client systems (Dyer et al. 2013). We notice that a high usage rate is concentrated in few APIs and that this rate decreases very fast. For example, the most used API is `java.util.ArrayList`, which is imported by 143,454 (54%) out of those 260K clients. The API in position number 100 (`android.graphics.Color`) is used by 16,382 (6%) clients. The 300th most used API (`android.location.Location`) is adopted by 6786 (3%) clients while the API in position number 500 (`javax.swing.table.AbstractTableModel`) is used only by 3977 (~2%) clients.

Previous studies take into account the popularity of certain APIs to learn from and support software maintenance practices. For example, Mileva et al. (Mileva et al. 2009, 2010) mine usage trends of popular APIs to suggest whether an API should be adopted or avoided by clients. Hora and Valente (2015) assess the evolution of popular APIs to support library migration. Recently, Zerouali and Mens (2017) showed that some libraries are largely more popular than their competitors, and detected that sometimes they are used together.

To our knowledge, however, several aspects of the popular APIs have not yet been analyzed by the literature. It is not clear whether popular APIs are different from ordinary ones regarding internal and external characteristics. In this paper, therefore, we assess internal (i.e., code and evolution) and external characteristics (i.e., adoption and usage by clients) of popular APIs to better understand and to learn from their development and maintenance practices.

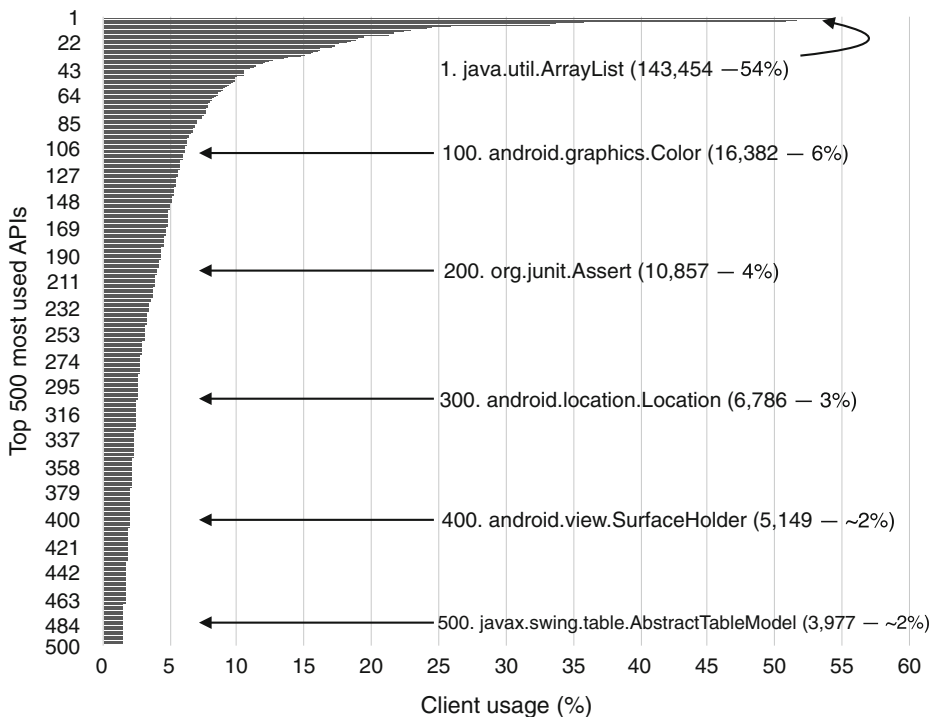


Fig. 1 Top 500 most used APIs in Java

3 Study design

To answer the proposed research questions, we need to detect and to assess popular APIs. Section 3.1 presents the process to detect popular APIs while Section 3.2 details the assessment of these APIs. Our dataset is publicly available.²

3.1 Detecting popular APIs

In this study, we measure API popularity in terms of the number of client systems. Specifically, we perform the detection of popular APIs in the following three steps. First, we collect a set of popular systems. Second, we detect the APIs used by these systems. Third, we categorize the detected APIs as popular, ordinary, and unpopular. These steps are summarized in Fig. 2 and described in the following subsections.

3.1.1 Step 1: Collecting client systems

1. GitHub projects. First, we detected the top 10,000 Java projects with more *stars* hosted on GitHub, the most popular social coding platform nowadays.

2. Projects with more than 1000 stars. To filter the most relevant projects, we selected the ones with more than 1000 stars, resulting in 1673.

3. Real systems. At this stage, it was also necessary to verify which of these projects were actually real systems. In this filtering, we manually inspected the first GitHub page (i.e., project description and readme file) and removed projects that were tutorials, examples, interviews, books, guides, among others non-systems, resulting in 1330 real systems. For example, *iluwatar/java-design-patterns*³ is the most popular Java project, but it is simply a collection of design pattern examples; therefore, it was excluded from our dataset. Some of the remained systems are created by organizations such as Google, Facebook, Twitter, Microsoft, and Apache. They also cover several domains, such as software tools (e.g., IntelliJ IDE and Eclipse Che), programming languages (e.g., Kotlin and Clojure), libraries (e.g., Facebook Fresco and Google Guava), and frameworks (e.g., Android and Spring).

To better characterize these 1330 systems, Fig. 3 presents their distribution regarding the number of stars, forks, and Java files. The median number of stars is 1812.5; the most starred systems are RxJava, Elasticsearch, and Retrofit. The median number of forks is 412, and the most forked ones are Spring Boot, Spring, and Dubbo. Finally, the median of the number of Java files is 56; the largest projects are IntelliJ Community, AWS SDK for Java, and Liferay.

3.1.2 Step 2: Detecting APIs used by client systems

4. Distinct APIs. The next step in our design was to detect the APIs used by the aforementioned 1330 systems. For this purpose, for each system, we extracted all the APIs imported by its Java files in the last release. Then, for each API, we computed the number of distinct

²<https://docs.google.com/spreadsheets/d/1VLR7nH2bMF9IjwwE7Hv4Ngkr9ANHm3sa4UPzAEtNyZY>

³<https://github.com/iluwatar/java-design-patterns>

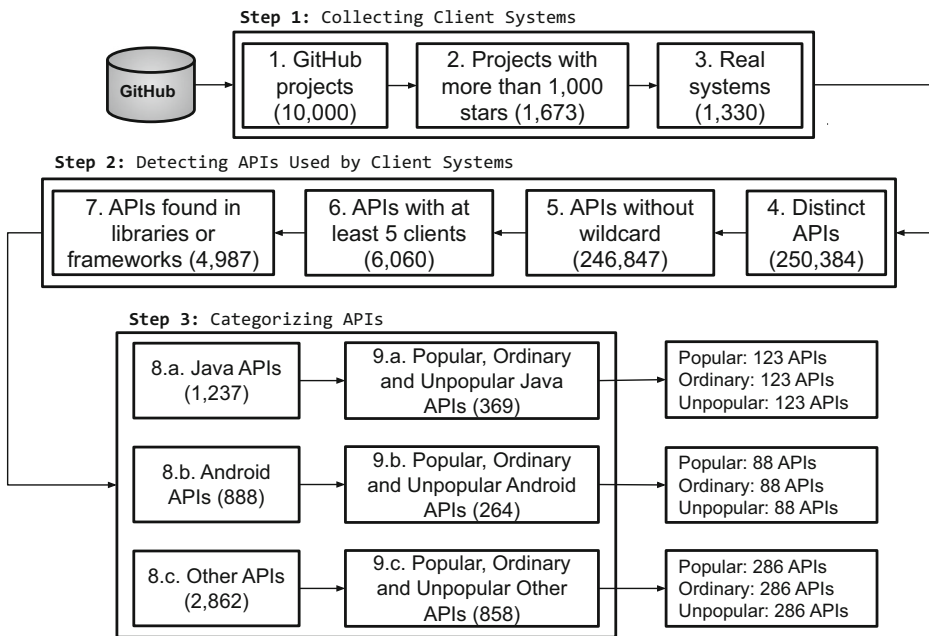


Fig. 2 Detecting popular, ordinary, and unpopular APIs

client systems using it. Thus, each detected API may have between 1 and 1330 clients. In total, we detected 250,384 distinct APIs imported by at least one client system.

5. APIs without wildcard. Next, we removed all APIs imported via wildcard (e.g., `java.util.*`) to keep just specific APIs. After this filtering, we are left with 246,847 APIs.

6. APIs with at least 5 clients. To filter out less popular and local APIs, we discarded the ones with less than 5 client systems. This resulted in 6060 APIs with at least 5 clients. The distribution of the number of clients for these APIs is presented in Fig. 4. The first quartile is 5 clients, the median is 9, and the third quartile is 19. We also

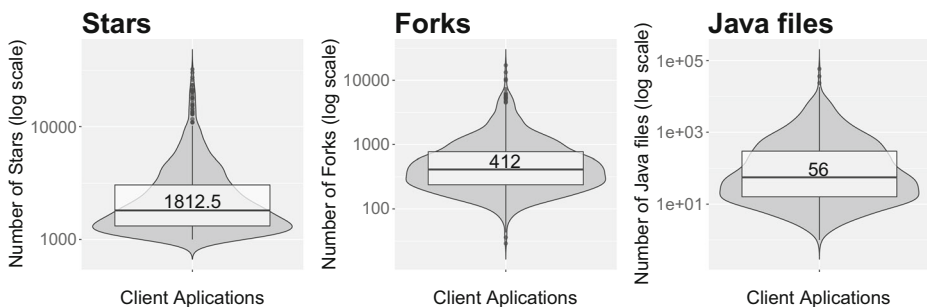


Fig. 3 Distribution of the real systems by stars, forks, and Java files

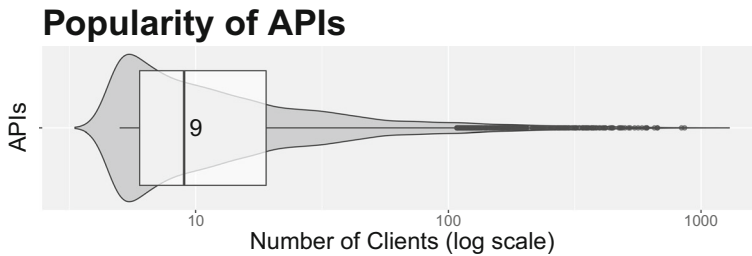


Fig. 4 API popularity in number of clients

notice a large amount of outliers (i.e., the dots in the boxplot), which represent highly popular APIs. For example, the top 3 most used APIs in our dataset are `java.util.ArrayList` (861 clients), `java.util.List` (837 clients), and `android.os.Bundle` (672 clients).

7. APIs found in libraries or frameworks. In this step, we manually detected the libraries or frameworks that the APIs belonged to. For example, the API `org.mockito.Mockito` belongs to the Mockito library. This stage was necessary to collect the source code of the studied APIs. We then downloaded the latest available release of these libraries and frameworks. For Java, we relied on their Mercurial repository⁴ and we analyzed the JDK 8. For Android, we analyzed two repositories available on GitHub repositories,⁵ which are mirrors of the official Google repository; in this case, we assessed the master branch, which represents the most recent Android version. For all the other systems, we analyzed the master branch of the most stable official repository on GitHub. For example, `junit-team/junit4` for JUnit, `mockito/mockito` for Mockito, `spring-projects/spring-framework` for Spring, etc. For Apache libraries, we relied on the `apache/*` repositories. Section 3.1.4 provides more information about the selected APIs. At the end, we were able to find the source code of 4987 APIs. Other APIs we did not find may have been removed or renamed in the last release or were outside of the master branch of the downloaded system.

3.1.3 Step 3: Categorizing APIs

8. Java, Android, and other APIs. We noticed that most of the selected APIs belonged Java and Android. Thus, to study them separately, we categorized the APIs in three ecosystems: Java, Android, and other. We considered as Java APIs those starting with `java`, `javax`, or `javafx`; we detected 1237 Java APIs. We considered as Android APIs those starting with `android`; we found 888 Android APIs. Finally, the other APIs that did not match Java nor Android were grouped together; it included 2862 APIs. In this step, we also discarded the APIs that were not public classes, such as Java interfaces, generics, enumerations, and annotations. This was performed due to two reasons: (i) to ensure the APIs have source code to be assessed (e.g., Java interfaces have only method signatures, so we could not perform

⁴<https://hg.openjdk.java.net>

⁵`aosp-mirror/platform_frameworks_base` and `aosp-mirror/platform_frameworks_support`

our code level metrics) and (ii) to ensure homogeneity on the analysis (e.g., it does not make sense to compare the complexity of a class with an enumeration).

9. Popular, ordinary, and unpopular APIs. The last step is to categorize the APIs as popular or unpopular. In this context, for each ecosystem (Java, Android, and other), we sorted their APIs in descending order based on the number of clients. Following the same experimental design to evaluate popular and unpopular software artifacts (Tian et al. 2014; Xavier et al. 2017; Brito et al. 2018c), we classified as popular the top 10%, i.e., APIs with the highest amount of client systems. We classified as unpopular the bottom 10%, i.e., APIs with the lowest amount of client systems. In addition, we also selected a set of 10% random APIs to contrast with both popular and unpopular ones; we name those as ordinary APIs. As a result, considering the three categories (popular, ordinary, and unpopular), we collected **369** Java APIs, **264** Android APIs, and **858** other APIs, totaling **1491** studied APIs.

3.1.4 About the selected APIs

The 369 Java APIs come from several packages, for example, `java.util`, `java.io`, `java.net`, `javax.swing`, and `javax.awt`; they help developers handling collections, files, network, ui, to name a few features. The 264 Android APIs also come from distinct packages, for example, `android.os`, `android.widget`, `android.util`, `android.content`, and `android.graphics`; they help developers dealing with Android features, such as permissions, picture/video manipulation, tasks, multiple touch events, and notifications. The 858 APIs in the category other belong to 165 libraries; they are mostly concentrated on Apache HttpComponents (33.7%), Spring Framework (5.6%), and Apache Hadoop (3.6%). In this case, the popular APIs come from 73 libraries; the top 3 are Apache HttpComponents (10.5%), JUnit (8.1%), and Apache Hadoop (5.24%). The ordinary APIs are provided by 112 libraries; the top 3 are Apache HttpComponents (6.7%), Apache Thrift (4.2%), and Apache Hadoop (3.9%). Finally, the unpopular APIs are provided 98 libraries; the top 3 are Spring Framework (9.1%), Elasticsearch (7%), and Apache Calcite (4.5%).

Table 1 presents an overview of 27 randomly selected Java, Android, and other APIs; for each ecosystem, we select nine APIs (three for each popularity level: popular, ordinary, and unpopular). We show two metrics of popularity; the first metric, number of clients (#Clients), is the one adopted in this study to compute API popularity. The second popularity metric, number of questions (#Questions), is computed with the support of Stack Overflow, which is the most import Question & Answers platform nowadays. This metric is extracted with the support of the Stack Exchange API⁶ and it reports the number of questions citing the API. Questions on Stack Overflow can be seen as a proxy of popularity for a given topic (Barua et al. 2014). As an example, consider the Java APIs `File`, `Duration`, and `FileDataSource`, which are classified as popular, ordinary, and unpopular, respectively. `File` has 500 clients and 29,668 questions on Stack Overflow. In contrast, `Duration` has 55 clients and 369 questions, while `FileDataSource` has only 7 clients and 171 questions. The same pattern happens in Android: the popular Android API `Bundle` has 63,070 questions, while the unpopular `Gallery` has only 224. Regarding the other APIs, the differences between popular and

⁶<https://data.stackexchange.com>

Table 1 Overview of 27 randomly selected Java, Android, and other APIs

Group	Name	Short description	#Clients	#Questions
Java	File	File abstraction	500	29,668
	URI	URI abstraction	239	2410
	Objects	Static utility methods	166	1012
	Duration	Time abstraction	55	369
	BoxLayout	Layout manager	13	882
	AWTException	AWT exception	9	517
	FileDataSource	Data typing services	7	171
	LinkRef	Reference abstraction	5	4
	PopupMenu	Popped up menu	5	65
Android	Bundle	Data passing	672	63,070
	Log	Log manipulation	473	24,607
	Toast	Little message view	360	21,368
	Html	Displayable HTML text	77	544
	ActivityInfo	Activity information	68	610
	MimeTypeMap	MIME-types mapping	38	66
	Gallery	Items listing	6	224
	GLU	Utilities	6	64
	ComposeShader	Composition of shaders	5	1
Other	Assert	Assertion test methods	232	2495
	Mockito	Mock test handling	119	819
	Guice	Injectors manipulation	34	101
	LocalFileSystem	File system representation	10	33
	LogicalFilter	Logical filter representation	6	0
	B64Code	Encoders and decoders	6	1
	Headers	Headers manipulation	5	6
	DataType	Data type static values	5	27
	StringContains	String test manipulation	5	0

unpopular APIs are also large. For example, the popular API `Assert` provided by JUnit has 2495 questions, while the unpopular API `Headers` provided by Undertow has only 6. Even though these 27 APIs represent only a fraction of all selected APIs, we notice the difference in popularity is quite significant regarding both clients and Stack Overflow questions.

3.1.5 Popularity of the selected APIs

Figure 5 explores the popularity difference of the selected APIs. It presents the distribution of the number of clients per API category for the three ecosystems. We clearly notice that the APIs classified as popular have more clients than the ordinary and the unpopular ones. Regarding the Java APIs, on the median, the popular have 139 clients. By contrast, the ordinary APIs have 17 clients while the unpopular have 6. The Android popular APIs have 151.5 clients, the ordinary 13, and the unpopular 5. As we can notice, the other APIs present the lowest variation in popularity: on the median, the popular APIs have 23 clients, the

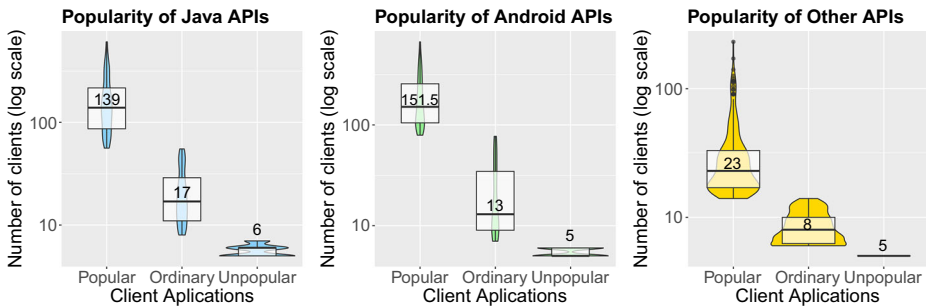


Fig. 5 API popularity: Java, Android, and other (165 libraries)

ordinary 8, and the unpopular 5. In all the cases, the categories are statistically significant different from each other with large difference (i.e., p value < 0.01 for Mann-Whitney test and $effect-size$ is 1 for Cliff's Delta).

3.1.6 Age of the selected APIs

To better understand how API popularity varies over time, we analyze their temporal aspects. Figure 6 presents the lifetime⁷ of the three API groups. Our first observation is that, independently of the level of popularity, all the analyzed APIs are relatively old. In Java, the popular, ordinary, and unpopular APIs have 3953 days, on the median. In Android, both the popular and ordinary have 3618 days, while the unpopular ones are a bit more recent: 3120.5 days. The other APIs have, respectively, 2460, 2421, and 2421 days. These numbers show that each ecosystem has APIs equivalent in terms of age; thus, the temporal aspects are overall similar. Indeed, in addition to Java and Android, the analyzed APIs belong to consolidated libraries, such as Elasticsearch, Hadoop, JUnit, HttpComponents, and Spring.

3.2 Assessing the APIs

After collecting the APIs, we compute several code, evolutionary, adoption, and usage metrics to answer the proposed research questions. These metrics are described in the following subsections and summarized in Table 2.

3.2.1 API code metrics (RQ1)

To answer the first question, we extract 10 source code metrics from the studied APIs, which cover aspects of maintainability. These metrics were computed with the support of the *Understand tool*⁸ and grouped in four categories:

Size: Includes metrics to assess the API size in terms of public methods and lines of code. These metrics are also computed relatively: the ratio between the number of public methods

⁷Lifetime is computed as the number of days between the first commit and the day of the analysis.

⁸<https://scitools.com>

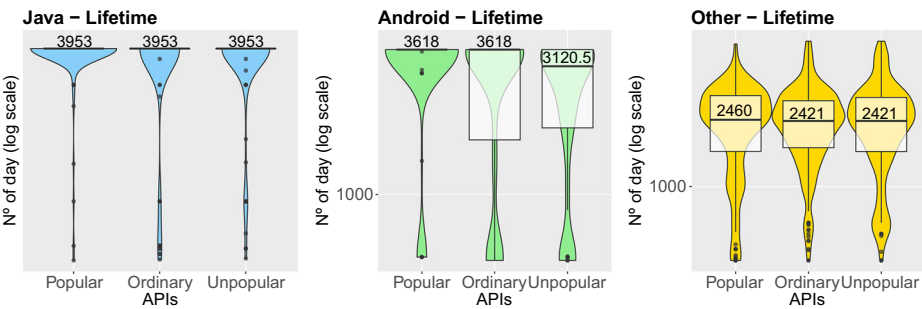


Fig. 6 API lifetime: Java, Android, and other (165 libraries)

and the total methods, as well as the ratio between the number of lines of code and the total lines.

Rationale: Larger APIs are harder to maintain (Henning 2007; Fowler and Beck 1999); however, they may provide more features to clients. By contrast, smaller APIs are recommended to increase their usability (Henning 2007). Clients using smaller APIs are also less dependent to external code (Myers and Stylos 2016) and, consequently, less likely to be affected by modifications on these elements. As a side effect, smaller APIs may provide less features, and thus be unattractive to clients. The balance between providing larger/smaller APIs is thus unclear. Moreover, APIs with a great amount of public methods may be an

Table 2 API code, evolutionary, adoption, and usage metrics

RQs	Categories	Metrics
RQ1	API size	○ Number of public methods
		○ Relative number of public methods
		○ Number of lines of code
		○ Relative number of lines of code
	API complexity	○ Cyclomatic complexity
RQ2	API legibility	○ Cyclomatic complexity per method
		○ Parameters per method
		○ Method name length
		○ Number of commented lines
	API documentation	○ Relative number of commented lines
RQ3	API changeability	○ Commits in API history
		○ Lifetime per commit
		○ Contributors in API history
		○ Lifetime per contributor
	API contribution	○ Number of API breaking changes
RQ4	API stability	○ Relative number of breaking changes
		○ Number of days since first API usage
		○ Relative first API usage
		○ APIs per commit
	Client adoption	○ Packages per commit

indicator of too many responsibilities or a bad design. In this case, APIs may improperly expose internal methods to clients. This is a bad practice because internal methods often change without any warranty, breaking the clients using them (Businge et al. 2012, 2013a, b, Hora et al., 2016).

Complexity: Assesses the cyclomatic complexity of the APIs. It computes the total cyclomatic complexity by summing the complexity for each method, i.e., the number of different paths through a source code (McCabe 1976). We also compute the cyclomatic complexity per method, i.e., the ratio between the total cyclomatic complexity and the number of methods.

Rationale: Complex APIs are difficult to learn, remember, and use correctly (Zibran 2008; Bloch 2006). APIs with complex code are also more likely to suffer from maintenance problems (Tufano et al. 2015; Henning 2007).

Striking the balance between code complexity, transparency, and flexibility is a hard challenge that API designers should handle, even with the help of guides, articles, and documentation (Zibran 2008).

Legibility: Analyzes the API legibility by computing the average method name length and average number of parameters.

Rationale: The name of classes, methods, and variables used in APIs may affect their usability (Zibran et al. 2011). While shorter names are cleaner and easier to remember, longer ones may be more meaningful. On the other hand, both extremes are problematic (Myers and Stylos 2016). Moreover, methods with a long list of parameters are harder to understand and use, decreasing the learning curve and productivity (Stylos et al. 2006; Stylos and Myers 2007; Fowler and Beck 1999).

Documentation: Includes metrics to assess API documentation, by computing the absolute and the relative number of lines with comments. Our analysis occurs at class level and considers the three Java comment statements: single line (`//`), multi line (`/* */`), and Javadoc comments (`/** */`).

Rationale: Documentation is important to any piece of code; however, it is fundamental to the success of an API. Documentation makes all the difference when developers start to learn, understand, use, or need to make a decision on which API will be adopted (Maalej and Robillard 2013; Robillard 2009). Moreover, code comments are often good to assist maintenance work (Lethbridge et al. 2003).

3.2.2 API evolutionary metrics (RQ2)

To answer the second research question, we assess the evolution of the studied APIs. That is, instead of analyzing a single file to compute the metrics as in the previous code analysis, we assess how this file evolved over time at commit level. By working at commit level, we are able to detect fine-grained evolutionary changes, for example, number of commits and number of contributors of the APIs; in contrast, at version level, it would not be possible to compute such kind of metrics to assess API evolution. Indeed, related previous studies also work at commit level to better capture API evolution (e.g., Xavier et al. 2017; Brito et al. 2018b). Particularly, for the Android and other APIs, which rely on the Git version

control system, we run the `git log`⁹ command on the file containing the API to extract information. For the Java APIs, which are maintained by Oracle and adopts the Mercurial version control system, we run an equivalent command: `hg log`.¹⁰ We also use a third-party library (Brito et al. 2018a) to compute the metrics about API stability. We measure six evolutionary metrics from the history of the APIs, which are grouped in three categories:

Changeability: Assesses terms: the number of commits and the ratio between lifetime and number of commits. Lifetime is computed as the number of days between the first commit and the day of the analysis.

Rationale: To cope with new features and bug-fixes, APIs should change over time. However, if the changes are quite frequent, the clients may be affected (Xavier et al. 2017) or may not keep up to date with new API versions (McDonnell et al. 2013). On the other hand, APIs with less changes may indicate more reliability to clients.

Contribution: Identifies the number of contributors of the APIs. It also computed relatively: the ratio between lifetime and the number of contributors.

Rationale: APIs with more contributors indicate collaborative work, which is fundamental in software development (Herbsleb 2007; Mistrík et al. 2010), making the knowledge widespread. In this case, several developers are able to add a feature or to fix a bug. APIs with fewer contributors show strong dependencies on specific developers; if this knowledgeable developer leaves the project, the system may be compromised (Avelino et al. 2016).

Stability: Computes the API stability by counting the number of API breaking changes over time, that is, modifications that may directly affect the client systems. Specifically, it measures the number public methods and attributes removed or modified in such a way that the clients are negatively impacted (e.g., loss of visibility, change in parameter list, change in return type, rename, move, among other Brito et al. 2018a). We also compute this metric relatively: the ratio between breaking changes and total API changes.¹¹

Rationale: Ideally, APIs should be stable so that clients are not impacted by API evolution (Xavier et al. 2017). For example, public methods and attributes should not be removed nor renamed from the interfaces unless they are deprecated (Brito et al. 2018c). However, recent research shows that API contracts may be broken due to many reasons, for instance, when developers want to implement new features or to simplify the interfaces (Brito et al. 2018b) or refactoring (Hora et al. 2018b).

3.2.3 Client adoption metrics (RQ3)

In the third research question, we focus on the client side. Specifically, we assess the version history of the client systems in order to investigate when the studied APIs are adopted. First, for each studied API, we detect when it was first used by its clients. With this

⁹Command: `git log --pretty=format:@"%H,%an,%ai" $API`

¹⁰Command: `hg log -f $API`

¹¹The *total API changes* include both the breaking changes (e.g., a removed public method) and the non-breaking changes (e.g., an added public method) (Brito et al. 2018a).

information, it is possible to verify at which point in time the clients have adopted a specific API. For example, the first time the API `java.util.ArrayList` was used by its client Retrofit was on Apr 26, 2011. Then, we extract, for each client, the date of the first and the last commit. Thus, we are able to compute two metrics: (i) first API usage in number of days and (ii) relative first API usage. For instance, the first commit of Retrofit happened on Oct 11, 2010 and the last commit on Aug 3, 2018 (so the system has 2853 days). Considering the API `java.util.ArrayList` and its client Retrofit, the *first API usage* happened 197 days after the first commit, and the *relative first API usage* is 1% (i.e., $197/2,853$).

Rationale: By assessing the client adoption, we aim to understand when APIs are imported by clients, in other words, to verify whether they are earlier or later adopted over time. APIs that are earlier adopted by clients may indicate their importance since the initial stages of the development cycle, i.e., APIs that are fundamental to kick start a project. By contrast, APIs that are later adopted may suggest they are relevant when the project is more mature, that is, a project can be started but not evolved without those APIs. The first case is more critical and brings another perspective to the studied APIs: in the case popular APIs are earlier adopted by clients, this shows they are key interfaces provided by their libraries.

3.2.4 Client usage metrics (RQ4)

In our last research question, we also assess the version history of the clients. In this case, for each studied API *a*, we detect all other APIs added over time in source code together with API *a*. For example, considering the API `java.util.List`, we see that it was added in source code together with API `java.util.ArrayList` in file `GLShader.java`.¹² Thus, for each studied API, we build a dataset including the list of other APIs used together. This way, we are able to compute two metrics: (i) number of added APIs and (ii) number of added packages. Considering the API `java.util.List` on the aforementioned example, the *number of added APIs* is 1 (i.e., `java.util.ArrayList`) and *number of added packages* is 1 (i.e., `java.util`).

Rationale: The client usage metrics aim to assess whether the studied APIs tend to be used with other APIs or be used alone. APIs that need to be imported together with other ones make the clients more coupled to a given library/framework. This may cause two problems on the clients. First, by depending on more APIs, the clients are more prone to suffer side API evolutionary effects (Xavier et al. 2017; Brito et al. 2018b). Second, the clients should know more APIs when using specific features, possibly making more difficult the API understandability. By contrast, APIs used alone or with few other APIs make the clients less coupled to the library/framework. This can also be an indication that the feature is simpler to reuse.

3.3 Presenting the results

To present the results and facilitate the comparison among the studied APIs, we use box plots and violin plots. The latter is a compact version of density plot, providing the full

¹²<https://goo.gl/FJKpXf>

distribution of the data (Hintze and Nelson 1998). To facilitate the visual distinction of the three studied ecosystems, we adopt the following colors on the plots: **blue** for Java; **green** for Android; and, **yellow** for other (i.e., the APIs coming from the 165 libraries).

We analyze the statistical significance of the difference between the categories by applying the Mann-Whitney test at p value = 0.05. To show the effect size of the difference between them, we compute Cliff's Delta (or d). Following the guidelines by Romano et al. (2006), we interpret the effect size values as negligible for $d < 0.147$, small for $d < 0.33$, medium for $d < 0.474$, and large otherwise. Our results are publicly available.¹³

4 Results

4.1 RQ1 (API Code): what are the code characteristics of popular APIs?

4.1.1 API size

Public methods. As presented in Fig. 7, the Java popular APIs provide more public methods (median 13) to clients than the ordinary (11) and the unpopular (8) ones. The Android APIs present the largest difference: the popular APIs have the double and the triple of public methods than the ordinary and the unpopular ones (medians 23, 12, and 7). The other APIs have the medians: 15, 11, and 11. Thus, particularly in Android, the popular APIs are larger, providing more public methods, and, possibly, more features to clients.

Relative public methods. In relative terms, the Java APIs have proportionally less public methods when compared with the other ecosystems (popular 30%, ordinary 25%, and unpopular 19.5%). That is to say, only 3 out of 10 methods in the Java popular APIs are public, while 7 out of 10 have other visibility modifiers, such as private, protected, and package. The Android popular APIs have relatively less public methods (55.8%) when compared with the ordinary (71.4%) and the unpopular (64.2%) ones. However, the other APIs have relatively more public methods as follows: 79.7%, 76.8%, and 66.7%. Thus, the Java developers are somehow very concerned with encapsulation, independently of the API popularity level. In Android, the popular APIs are more encapsulated than the ordinary ones. The other APIs present the highest rate of public methods, and their popular APIs expose proportionally more public methods to clients. This suggests that encapsulation, although very important to avoid the exposure of internal elements, varies among the studied APIs, being more adopted in Java and Android, but not when considering the 165 libraries (i.e., the other APIs).

Lines of code. Considering the lines of code (Fig. 8), we notice that the Java popular APIs are larger than the ordinary and the unpopular ones, with medians 150, 96, and 42. A great difference also happens in Android APIs, with medians 266, 118.5, and 55.5. The lowest difference occurs in the other APIs as follows: 150, 123, and 119 lines of code. Thus, the popular APIs are larger in terms lines of code, particularly in Java and Android.

¹³<https://goo.gl/tZernL>

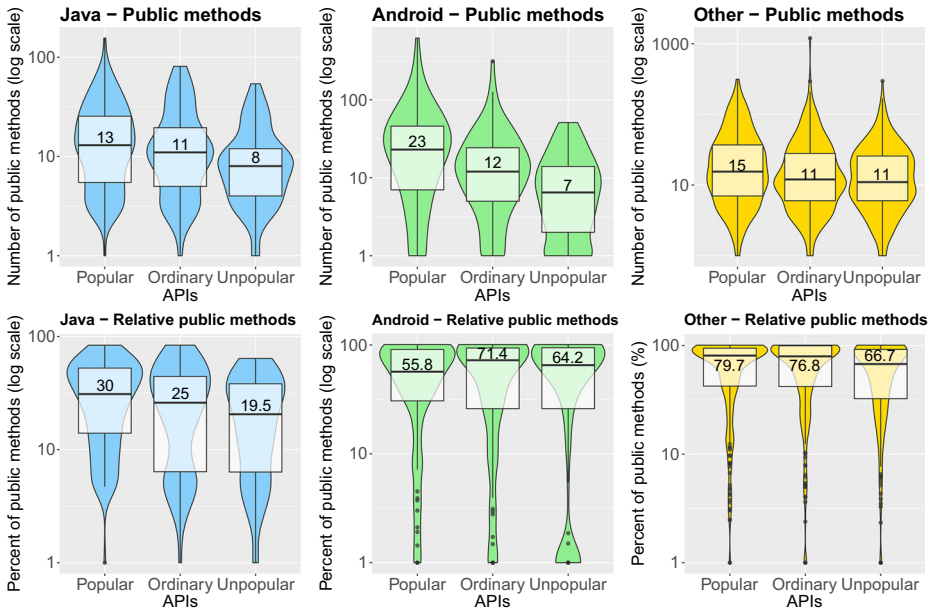


Fig. 7 API size—public methods

Relative lines of code. In relative terms (i.e., ratio between lines of code and total lines), the Java APIs are equally coded, having around one-third of lines of code, with medians 32.8%, 33.2%, and 31.6%. This means that around 70% of their code is composed by non-code

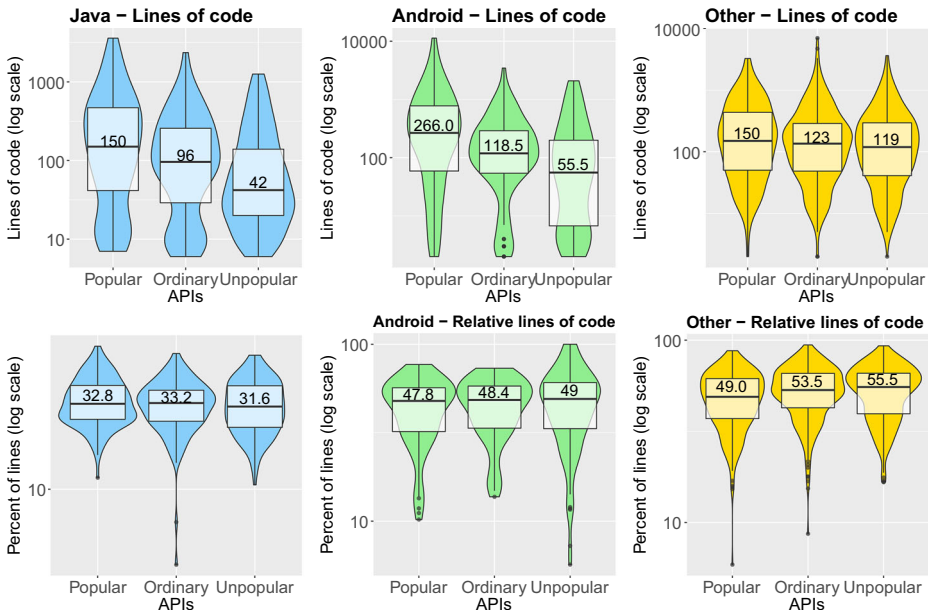


Fig. 8 API size—lines of code

lines, such as comments and blank lines. The same happens to the Android APIs; in this case, the APIs have around 48% of code lines (medians 47.8%, 48.4%, and 49%). The largest difference between the popular and the ordinary APIs occurs in the other APIs as follows: 49% against 53.5% and 55.5%. Thus, independently of the popularity and the ecosystem, we notice a great concern with writing non-code lines.

4.1.2 API complexity

Cyclomatic complexity. As presented in Fig. 9, the Java popular APIs are more complex (median 30 paths), than the ordinary (23 paths) and the unpopular (10 paths) ones. The same happens for the Android (45.5, 22.5, and 9.5) and the other (32.5, 22, and 23.5) APIs. Thus, in absolute numbers, the popular APIs are more complex than the ordinary ones.

Cyclomatic complexity per method. To ensure these results are not biased by the API size, we compute the relative complexity, i.e., the ratio between total cyclomatic complexity and the number of methods. In this case, the Java popular APIs remain more complex, with median 0.86 paths per method, than the ordinary (0.59) and the unpopular (0.27). However, the Android APIs are equally complex. The other popular APIs are only slightly more complex (1.29) than the ordinary (1.25) and the unpopular (1.13). Thus, only the Java popular APIs are relatively more complex than the ordinary ones.

4.1.3 API legibility

Method name length. As detailed in Fig. 10, we could not find any major difference regarding the method name length. The Java and the other API methods have around 12

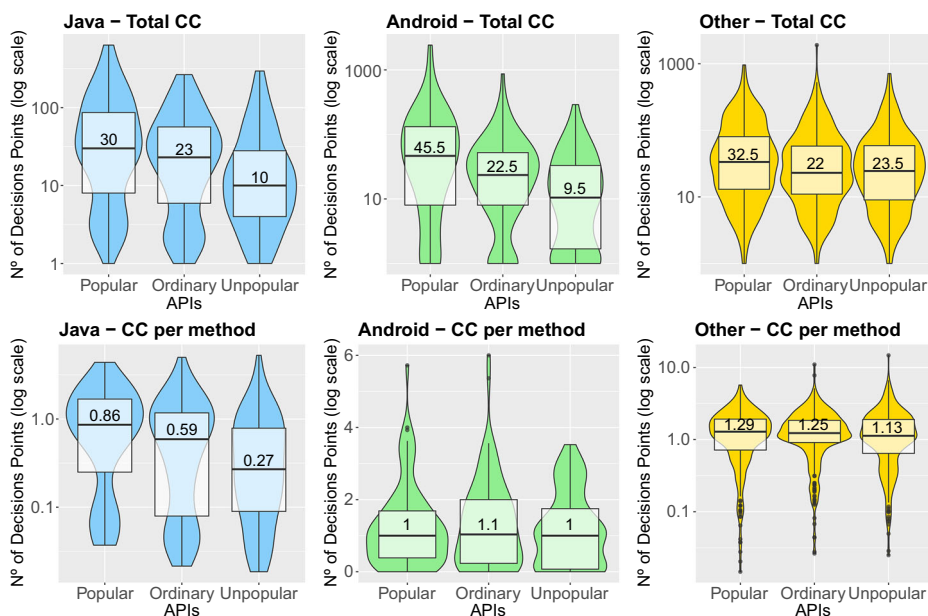


Fig. 9 API complexity—cyclomatic complexity

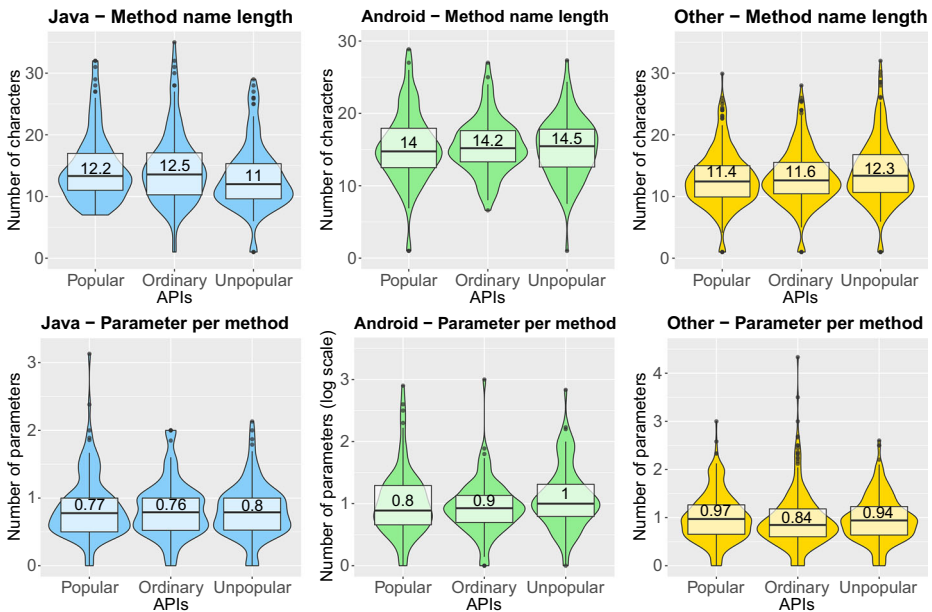


Fig. 10 API legibility

characters while the Android ones are slightly longer, with around 14 characters. Thus, independently of the API popularity and the ecosystem, on the median, method names vary between 12 and 14.

Parameters per method. In this case, again, the differences are not large. On the median, the Java popular APIs have 0.77 parameters per method, the Android ones have 0.8, and the other ones 0.97. Therefore, in most of the cases, the APIs have no more than one parameter per method, which is likely to facilitate their usage.

4.1.4 API documentation

Commented lines. As presented in Fig. 11, in all the cases, the popular APIs have more commented lines (median 258 in Java, 237 in Android, and 115 in other APIs) than the ordinary (178, 117, and 71) and the unpopular (84, 54, and 71) APIs. This may be explained by the fact that the popular APIs are often larger than the ordinary ones.

Relative commented lines. To avoid the possible bias caused by larger APIs, we measure the proportion of comments per API. Relatively, we do not see a large difference when comparing the APIs in Java (around 60% of code comments in all APIs) and Android (around 43% of code comments in all APIs). However, the other popular APIs have proportionally more comments (41.4%) than the ordinary ones (35.7% and 33.3%). Thus, independently of the popularity, Java and Android APIs are equally commented. By contrast, we notice a higher ratio of commented code in the popular APIs provided by the 165 libraries.

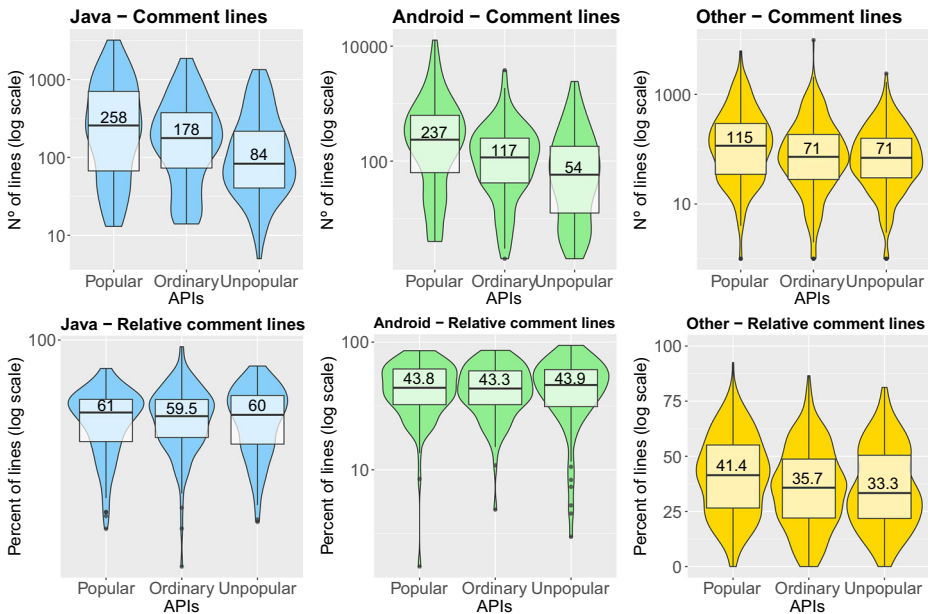


Fig. 11 API documentation

Summary of RQ1: Popular APIs are larger than ordinary ones, providing more public methods to clients or having more lines of code, independently of the ecosystem. The proportion of exposed public methods is much smaller in Java (30%) and Android (55.8%) than in the 165 libraries (79.7%). At class level, popular APIs are more complex than ordinary ones, providing more paths on their code. At method level, however, popular APIs are not necessarily more complex than the ordinary ones. Popular APIs are not different from ordinary ones regarding their method name length nor number of parameters per method. Popular APIs have more comments in code than ordinary ones. Particularly, in the case of the 165 libraries, popular APIs have 24% more comments than unpopular APIs (41.4% against 33.3%).

4.2 RQ2 (API evolution): what are the evolutionary characteristics of popular APIs?

4.2.1 API changeability

Number of commits. Figure 12 presents that the popular APIs have more changes than the ordinary ones. Specifically, the Java popular APIs have, on the median, 9 changes while the ordinary ones have 6 and 4. The Android popular APIs have 38.5 changes against 13 and 10.5, while the other popular APIs have 16 changes against 9.5 and 9. Thus, in absolute numbers, the popular APIs face more updates over time.

Lifetime per commit. To better understand the frequency of changes and avoid the possible bias caused by long life APIs, we use the metric ratio between lifetime and commits.

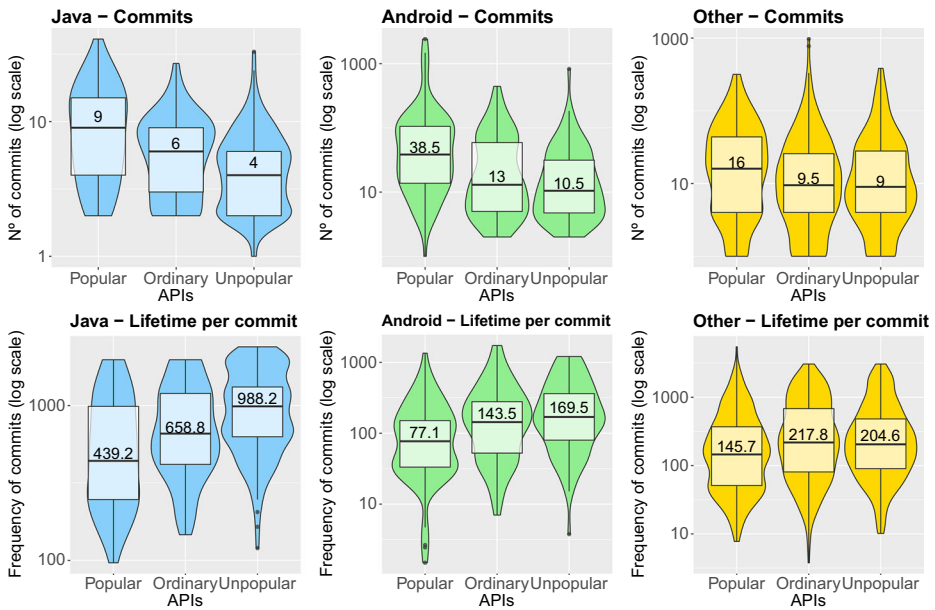


Fig. 12 API changeability

It shows that the Java popular APIs change more frequently (median 439.2 days per commit) than the ordinary ones (658.8 and 988.2). In Android, the changes in the popular APIs are even quicker (77.1 days per commit) than the ordinary APIs (143.5 and 169.5). The same happens to the other popular APIs: 145.7, 217.8, and 204.6. Notice the high difference among the ecosystems: while the Java popular APIs face one change on each 439.2 days, the Android popular ones have one change on each 77.1 days. The popular APIs provided by the 165 libraries are in the middle, with one change on each 145.7 days.

4.2.2 API contribution

Number of contributors. As shown in Fig. 13, the Java popular APIs have slightly more contributors (median 6) than the ordinary (5) and the unpopular (4) ones. In Android, the difference is higher: while the popular APIs have, on the median, 15 contributors, the ordinary have 7 and the unpopular 5. By contrast, we do not find any difference in number of contributors when comparing the other APIs.

Lifetime per contributor. This metric measures the concentration of contributors considering the API lifetime. The Java popular APIs have proportionally more contributors (median 658.8 days per contributor) than the ordinary and the unpopular ones (790.6 and 988.4). In Android, the popular APIs have median of 206.9 days per contributor, while the ordinary ones have 380.7 and 398.7. The same happens for the other popular APIs: they have relatively more contributors (median 494.8 days per contributor) than the ordinary ones (692.8 and 581); however, in this case, the overall difference is smaller when compared with Java and Android.

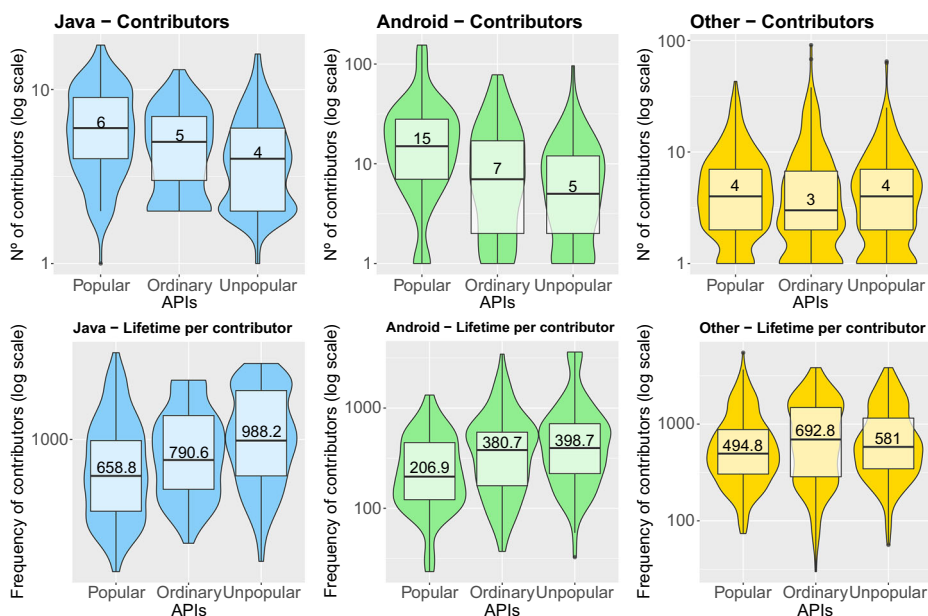


Fig. 13 API contribution

4.2.3 API stability

Breaking changes. Figure 14 shows that the Java APIs are very stable, with one breaking change per API, on the median. By contrast, the Android popular APIs have more breaking changes (median 6) than the ordinary (2) and the unpopular APIs (1). The same happens to the other APIs: the popular APIs have, on the median, 7 breaking changes, while the ordinary and the unpopular APIs have 2 and 3.

Relative breaking changes. In relative terms (i.e., ratio between API breaking changes and total API changes), the analysis reinforces the stability of the Java APIs. We notice, however, that the Android and the other popular APIs have proportionally more breaking changes (19.95% and 26%) than the ordinary (10.1% and 12.1%) and the unpopular ones (1% and 17.7%). This shows that popular APIs are more likely to break client contracts when evolving.

Summary of RQ2: Popular APIs have more changes than ordinary APIs over time. Popular APIs also change much more frequently, particularly in Android, with one change on each 77.1 days. Popular APIs do not necessarily have more contributors than the ordinary ones. However, popular APIs have a higher concentration of contributors over time, particularly in Java and Android. While Java APIs are very stable, popular APIs provided by Android have twice more breaking changes than ordinary APIs (19.95% against 10.1%). Popular APIs provided by the 165 libraries have more than double of breaking changes than ordinary ones (26% against 12.1%).

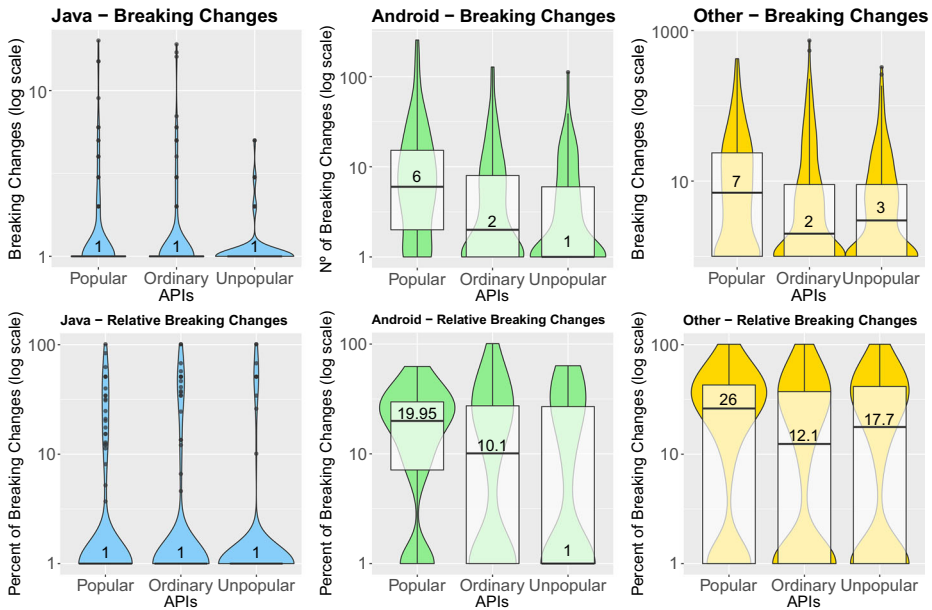


Fig. 14 API stability

4.3 RQ3 (client adoption): when are popular APIs adopted by clients?

First API usage. As shown in Fig. 15, the Java popular APIs are earlier adopted (median 114 days) than the ordinary and the unpopular ones (medians 420 and 770 days) by the client systems. The Android popular APIs are used as soon as possible by the clients (1 day), while ordinary ones are not used that fast (65 and 175.7 days). Similarly, the other popular APIs are earlier imported (429.7 days) than the ordinary ones (696.2 and 876.2 days).

Relative first API usage. To avoid bias caused by long lived client systems, we also assess the API first usage in relative terms, taking into account the age of the clients. In this case, we see the same tendency as in the absolute analysis: the Java, Android, and other popular APIs are earlier adopted (4%, 0%, and 22%) than the ordinary (16%, 5%, and 28%) and the unpopular ones (28%, 13%, and 35%). It is interesting to notice that, independently of the popularity, the Android APIs are the ones with the lowest frequency, indicating the importance of the Android framework APIs to kick start any mobile project. By contrast, the popular APIs coming from the 165 libraries are the latest adopted when compared with Java and Android.

Summary of RQ3: Popular APIs are earlier adopted than the ordinary APIs by the clients, suggesting their importance since initial stages of development. This may indicate that popular APIs are not simply interfaces with many clients, but they are APIs that make the foundation of their client systems.

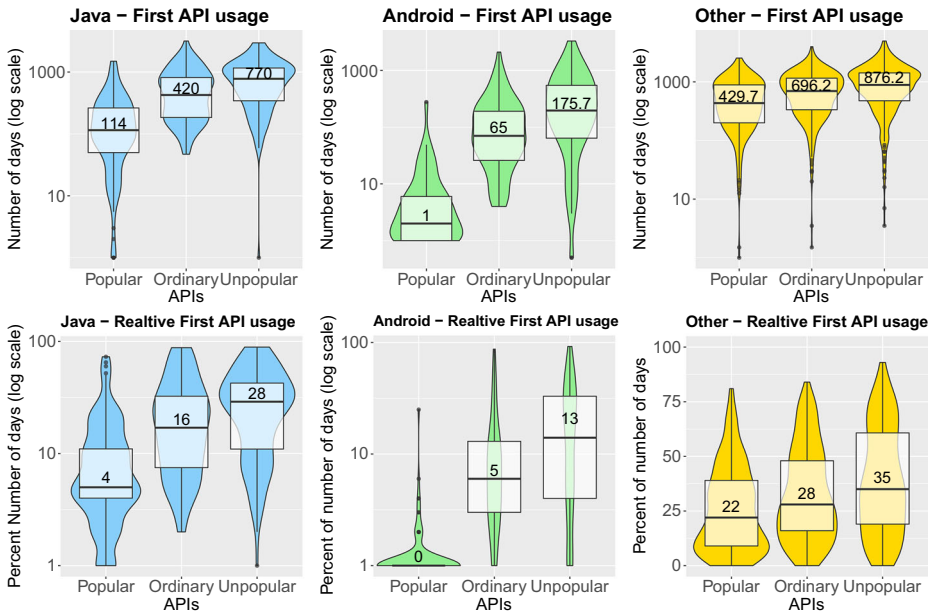


Fig. 15 Client adoption

4.4 RQ4 (client usage): how are popular APIs used by clients?

APIs per commit. Figure 16 presents that the Java popular APIs are imported, on the median, together with other 7.5 APIs; this number is not very different when compared with the ordinary and the unpopular APIs (6.9 and 7.2). In Android, the popular APIs are imported with other 9.4 APIs; their ordinary and unpopular APIs are imported with 11.9 and 14.8. The other APIs are imported together with 10.6, 11.9, and 12.7 APIs. Thus, independently of the popularity, we do not see a large difference in the number of imported APIs.

Packages per commit. At package level, we notice a similar tendency. The clients using Java popular APIs, on the median, import 4 distinct packages (ordinary 3.3, unpopular 3.6). In Android, the number of imported packages is a bit higher when compared with Java: 5.1, 5.3, and 7.3. Finally, the other APIs are imported together with 6, 6.1, and 6.8 distinct packages. Again, we do not notice any major difference regarding the number of imported packages.

Summary of RQ4: The number of APIs and packages imported together with the popular APIs is only slightly larger in Java and slightly smaller in the case of Android and the 165 libraries. Thus, we cannot confirm that the clients relying on the popular APIs become more or less coupled to libraries.

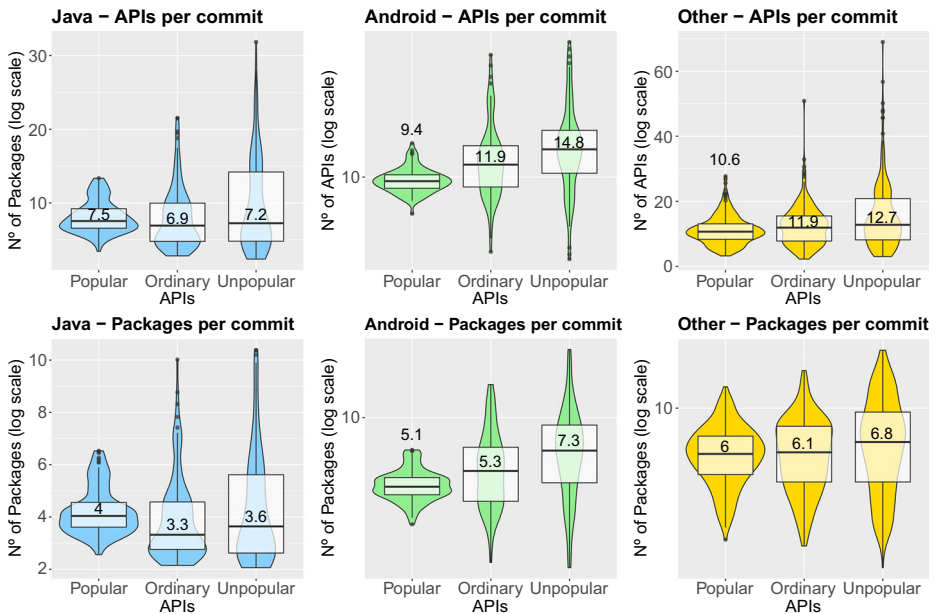


Fig. 16 Client usage

5 Findings and implications

5.1 Popular APIs are largely different from ordinary APIs

In this study, we detect that the popular APIs are different from the ordinary ones, both on code and evolutionary aspects, as well as on the client side. Table 3 summarizes the comparison between popular and ordinary APIs. Specifically, for each ecosystem and metric, it compares the popular against the ordinary APIs regarding their statistical significance difference (Mann-Whitney test) and effect size (Cliff's Delta). We notice that the Java and the Android ecosystems are the ones with more differences: in 13 and 14 (out of 20 metrics), respectively, the popular APIs are statistically significantly different from the ordinary ones. Both Java and Android agree in 12 metrics: 2 code level, all the 6 evolution level, and all 4 the client metrics. Thus, clearly, the popular APIs provided by Java and Android receive a different attention from both their developers and clients. By contrast, the other APIs have less differences: their popular APIs are statistically significantly different from ordinary ones in 9 out of 20 metrics: 3 code level, 4 evolution level, and 2 client metrics. Here, it is interesting to note that the two documentation metrics (i.e., commented lines and relative commented lines) are flagged as statistically significant different, suggesting that there is a greater effort to comment code on the popular APIs. Moreover, the two metrics related to contributors (i.e., contributors and lifetime per contributor) are not flagged as statistically significant different, indicating that the popular APIs do not have more/less collaborators than the ordinary APIs. Considering the three ecosystems (column “#All”), in 36 (60%) out of 60 comparisons, the popular APIs are statistically significant different with at least small effect from the ordinary

Table 3 Comparing the popular against the random APIs (Pop x Ran)

Metrics	Java Pop x Ran	Android Pop x Ran	Other Pop x Ran	#All
○ Public methods	—	0.27	—	1
○ Relative public methods	—	—	—	0
○ Lines of code	0.16	0.25	—	2
○ Relative lines of code	—	—	0.16	1
○ Cyclomatic complexity	0.15	0.23	—	2
○ CC per method	0.21	—	—	1
○ Parameters per method	—	—	—	0
○ Method name length	—	—	—	0
○ Commented lines	—	0.27	0.16	2
○ Relative commented lines	—	—	0.17	1
○ Commits	0.25	0.33	0.16	3
○ Lifetime per commit	0.21	0.27	0.18	3
○ Contributors	0.22	0.31	—	2
○ Lifetime per contributor	0.19	0.26	—	2
○ Breaking changes	0.24	0.32	0.21	3
○ Relative breaking changes	0.23	0.18	0.17	3
○ First API usage	0.59	0.85	0.24	3
○ Relative first API usage	0.49	0.84	0.19	3
○ APIs per commit	0.15	0.35	—	2
○ Packages per commit	0.28	0.24	—	2
Total	12	14	9	36

The numbers are the effect size for statistically significant different comparisons. Not statistically significant difference is represented by “—”

ones. Thus, independently of the ecosystem, the popular APIs have more changes, are more unstable, and are earlier adopted by client systems.

Implications: Researchers studying libraries should be aware that not all APIs are equals, and they vary in code, evolution, and client aspects, according to their popularity. In this context, we recommend the adoption of the API popularity metric to perform customized API analysis, avoiding handling them in an uniform way. For example, (i) recommendation tools can suggest to client developers which APIs they should firstly learn and use to kick start a project, based on API popularity metrics; (ii) library migration techniques (e.g., Wu et al. 2010; Hora et al. 2018a) can prioritize the most popular APIs in order to create API migration rules and to improve their accuracy; (iii) expert detection approaches (e.g., Fritz et al. 2010, 2014; Avelino et al. 2016) can focus on studying popular APIs separately, as they are the ones with major impact on the clients.

5.2 The ecosystem plays an important role in the analysis of popular APIs

Table 4 compares the *popular* APIs among the ecosystems. We see that in 17 (85%) out of 20 comparisons, the Java and the Android popular APIs are statistically significant

Table 4 Comparing the Java, Android, and other popular APIs

Metrics	Java x Android	Java x other	Android x other	#All
o Public methods	0.22	—	—	1
o Relative public methods	0.41	0.58	0.18	3
o Lines of code	0.15	—	0.16	2
o Relative lines of code	0.38	0.51	—	2
o Cyclomatic complexity	—	—	—	0
o CC per method	—	0.25	0.18	2
o Parameters per method	0.20	0.18	—	2
o Method name length	0.20	—	0.35	2
o Commented lines	—	0.25	0.21	2
o Relative commented lines	0.41	0.54	—	2
o Commits	0.66	0.27	0.36	3
o Lifetime per commit	0.80	0.56	0.28	3
o Contributors	0.57	0.24	0.62	3
o Lifetime per contributor	0.74	0.25	0.54	3
o Breaking changes	0.68	0.59	—	2
o Relative breaking changes	0.57	0.53	—	2
o First API usage	0.88	0.58	0.97	3
o Relative first API usage	0.85	0.58	0.97	3
o APIs per commit	0.48	0.46	0.22	3
o Packages per commit	0.62	0.61	0.29	3
Total	17	16	13	46

The numbers are the effect size for statistically significant different comparisons. Not statistically significant difference is represented by “—”

different. Comparing the Java and the other popular shows a difference in 16 cases (80%). The lowest amount occurs between Android and the other popular APIs: 13 differences (65%). Considering the three ecosystems (column “#All”), we notice that in 46 (77%) out of 60 comparisons, the popular APIs are statistically significant different with at least small effect. The three comparisons agree in 9 metrics: relative public methods, commits, lifetime per commit, contributors, lifetime per contributor, first API usage, relative first API usage, APIs per commit, and packages per commit. For example, the ratio of public methods in the Java popular APIs is different when compared with the Android and the other ratio.

Implications: When studying API characteristics, researchers should assess ecosystems separately, as they differ in the majority (77% in our case) of the analyzed metrics.

5.3 Popular APIs expose a large proportion of API elements

Based on the results provided by RQ1, we found that the Java and the Android popular APIs expose 30% and 55.8%, respectively, of their methods as public. By contrast, the popular APIs coming from the 165 libraries present a much higher ratio: they provide around

80% of public methods, even with the risks of exposing large APIs, which are harder to maintain (Henning 2007; Fowler and Beck 1999), and exposing internal methods, which are more likely to change without any warranty (Businge et al. 2012, Businge et al. 2013a, b, Mastrangelo et al. 2015, Hora et al. 2016).

Implications: API producers may expose more or less public methods on the popular APIs. Indeed, API producers may be conservative, by exposing few public elements (as in the Java ecosystem), or more liberal, exposing a high ratio of public elements (as in the 165 libraries). Overall, the majority of the API producers opt to provide a large proportion of elements on the popular APIs, possibly to accommodate more features.

5.4 Popular APIs have more code comments

Documentation is important to any software, and even more fundamental to libraries in order to support their clients. In this case, several types of documentation, such as manuals, web pages, wikis, and code comments, may help the clients. Our analysis is restricted to code comments, which are also good to help maintenance work (Lethbridge et al. 2003). In RQ1, we found that the Java and the Android APIs are equally commented; however, the popular APIs coming from the 165 libraries have up to 24% more comments.

Implications: API producers do treat the APIs differently: the majority of the popular APIs have proportionally more code comments than the ordinary ones, suggesting that the efforts to comment code are tailored to specific groups of APIs. Therefore, when and if necessary, API developers who are working in libraries may *prioritize* certain maintenance tasks according to the level of popularity faced by the API.

5.5 Popular APIs are more likely to change over time

Ideally, APIs should be stable when evolving so that client systems can rely on them (Brito et al. 2018b). Unfortunately, previous studies show the opposite: APIs provided to clients are commonly unstable (Wu et al. 2010; Robbes et al. 2012; McDonnell et al. 2013; Bogart et al. 2016; Xavier et al. 2017; Hora et al. 2018a). This occurs due to several reasons, such as new feature implementation, API simplification, and bug fixing (Brito et al. 2018b). For example, Xavier et al. (2017) mined 317 libraries and detected that, on the median, 14.78% of API breaking changes. Our results presented in RQ2 corroborate to the described scenario: the ordinary APIs in our largest dataset (i.e., the 165 libraries) have, on the median, 12.1% of breaking changes, which is very close to the ratio of 14.78% found by Xavier et al. (2017). Even more importantly and worrying, we detect that this issue is more critical on the popular APIs, which are much more unstable than the ordinary APIs. The Android popular APIs have 19.9% of breaking changes, while the popular APIs coming from the 165 libraries have 26% of breaking changes.

Implications: The developers of the majority of the popular APIs are more prone to evolve them, even with the cost of breaking contracts with clients. In this context, approaches to detect API evolution (e.g., Brito et al. 2018a, b; Xavier et al. 2017) can alert the developer who is introducing too many breaking changes on a popular API or warn the client that the API he is using everywhere is unstable (and, therefore, should be avoided).

6 Threats to validity

Association is not causation. We assessed whether there are metrics associated with popular, ordinary, and unpopular APIs. Notice, however, that association does not imply causation (Couto et al. 2014). In this case, more advanced statistical analysis, e.g., causal analysis (Retherford and Choe 2011), can be used to extend our study.

API popularity. In this research, we needed to measure popularity at file level (i.e., the file containing the API). Thus, we considered popularity in terms of number of client systems importing the given file. Measuring popularity in terms of client systems has been adopted by several previous studies, e.g., Mileva et al. 2010, 2011; Hora and Valente 2015; Zerouali and Mens 2017). At system level (i.e., not at file level like our study), other proxies of popularity can be used, such as number of stars, forks, contributors, and commits (Borges et al. 2016, 2018).

API categorization. We classified as popular APIs the top 10% with the highest amount of client systems. Similarly, we classified as unpopular APIs the bottom 10% with the lowest amount of client systems. Several previous studies have used a similar approach to differentiate popular and unpopular artifacts, for example, by assessing popular and unpopular mobile apps (Tian et al. 2014) and libraries (Xavier et al. 2017; Brito et al. 2018c).

Findings validation. We used statistical machinery (i.e., Mann-Whitney test and Cliff's Delta effect size) to reduce the possibility that our results are due to chance. Moreover, as stated in Section 3, our dataset is also publicly available for replication and further comparison.

Linking between APIs and libraries. The link between APIs with libraries described in Section 3.1 (Step 2.7) was performed manually. However, the chance of errors during this mapping is very reduced because the mapping was done with the support of documentation and code inspection to be sure about the libraries the APIs belonged to.

Commit vs. version level. In this study, we work at commit level when performing evolutionary analysis. Working at commit level allows us to perform more fine-grained assessment of the changes (Dagenais and Robillard 2008), particularly, when computing the API breaking changes. At version level (i.e., major releases), the assessment would be on coarse-grained changes (i.e., many commits may exist between two versions), which could possibly result in noisy data. We acknowledge, however, that by mining commits, we are subjected to assess changes that can be undone before arriving in production. Even though this scenario is feasible and may happen in practice, we believe this is rare and is not likely to affect our findings.

Generalization of the results. We focused on the analysis of 1491 APIs provided by Java, Android, and 165 credible libraries. These systems are open source and hosted on GitHub, the most popular software repository nowadays; thus, their code are easily accessible. Our findings, however, cannot be directly generalized to other libraries, particularly to the ones implemented in other programming languages and closed source.

7 Related work

7.1 Studies on API development guidelines

Many API-related studies are available to support developers creating better APIs. Robillard (2009) presented the aspects that make API learning difficult. By using questionnaires and interviews with Microsoft developers, some points were considered critical, for example, documentation with missing content and issues. In our analysis, we assessed code comment as a proxy of documentation, finding that the majority of the popular APIs have more comments. Some guidelines are associated to the design of good APIs (Bloch 2006; Henning 2007), for example, good documentation, relevant code examples, small amount of parameter, methods and classes with significant names, among other. Stylos and Myers (2007) showed that well-designed APIs are easier to use, hard to misuse, increase the productive of developers, prevent errors, and decrease chances of crashes. In our study, we analyze several aspects related to API quality as extracted from source code, such as size, complexity, and legibility.

7.2 Studies on API evolution

Other studies focus on assessing API evolution to measure their stability. Overall, the literature shows that APIs are often unstable and backward-incompatible (Wu et al. 2010; Robbes et al. 2012; McDonnell et al. 2013; Bogart et al. 2016; Hora et al. 2018a). Recently, Xavier et al. (2017) assessed 317 real-world libraries, finding that 28% out of 500K API changes break backward compatibility. The authors also found a rate of 14.78% of breaking changes at system level. In our study, we detected that the popular APIs have much more breaking changes than ordinary APIs, particularly in the case of Android and the 165 libraries. Our study, thus, complements the previous one (Xavier et al. 2017), showing that breaking changes do not happen equally among the APIs, but are associated with their popularity. Brito et al. (2018b) presented a list of reasons to break API contracts as reported by library designers. The authors found that developers often break APIs to implement new features, simplify APIs, improve maintainability, and fix bugs. Hora et al. (2018a) performed a large-scale case study in the Pharo ecosystem, by analyzing 118 API evolutionary changes. The authors found that the majority (61%) of the client systems are potentially affected by API changes; however, only a minority (5%) of them perform some reaction. In a related study, Robbes et al. (2012) verified the impact of deprecated APIs in the Smalltalk ecosystem. The authors extended their analysis to the Java ecosystem (Sawant et al. 2016, 2017b), finding that some deprecated APIs have large impact on the ecosystem under analysis and that the quality of deprecation messages should be improved. Recently, (Kula et al. 2018a) studied library dependency evolution and detected that most of the clients remain with outdated dependencies. Dig and Johnson (2005) studied API changes in five frameworks and libraries (Eclipse, Mortgage, Struts, Log4J, and JHotDraw). The authors reported that more than 80% of the breaking changes in these systems were caused by refactorings. McDonnell et al. (2013) investigated API stability in Android, finding that Android APIs evolve so fast that the clients are not able to catch up with the pace of API evolution. In our study, we also detect that the Android APIs are the ones with more updates, and that this issue is even more critical on their popular APIs.

7.3 Studies on API popularity

Some studies mine library popularity to learn and help developers in software maintenance. For example, Mileva et al. (2009, 2010) mined usage trends in popular libraries to suggest whether a certain library should be used or avoided by developers. The authors analyzed 250 Apache systems and 450 libraries, and identified which libraries are most used over the years. They also developed a tool (AKTARI) to check trends in library usage. Hora and Valente (2015) assessed API popularity to suggest interface migration through the analysis of 650 Java systems. Sawant and Bacchelli (2015) explored API popularity by assessing type-checked API method invocation information. Some studies point out that certain libraries and frameworks are more popular than others. Zerouali and Mens (2017) state that several test libraries are relatively more popular than their competitors and that sometimes they are used together. For example, the JUnit test library is much more adopted by client applications than the TestNG library (Zerouali and Mens 2017). In this case, the comparison is performed at library/framework level, i.e., not at the level of API as performed in our study. Thummalapenta and Xie (2008) provide a tool named SpotWeb to detect framework hotspot and coldspot. The hotspots can be compared with our popular APIs while the coldspot can be compared with our unpopular APIs. The authors detect such kind of spots to support framework usage practices. In our study, we focus on assessing the source code, evolution, usage, and adoption of APIs in three levels of popularity to uncover how world-wide APIs are actually implemented and maintained. Sawant and Bacchelli (2017a) propose fine-GRAPE, an approach that produces fine-grained API usage information. The study is performed in the context of five popular Java libraries and frameworks. Interestingly, the authors detect that a small number of API features are used by the clients and most of these features are introduced early in the APIs lifecycle. In our study, we complement this analysis by showing at large scale that the popular APIs are early adopted by clients in the case of Java, Android, and 165 libraries and frameworks.

7.4 Studies on popularity in other software artifacts

Popularity is a well-studied aspect not only in the context of software libraries, but in many other software artifacts. In the context of mobile apps, Tian et al. (2014) investigated 28 aspects to understand the differences between Android apps with high and low popularity. The authors looked at 1492 apps and focused on factors that developers can control, such as the quality of documentation in Google Play Market. As a result, out of the 28 investigated factors, 3 were more important to differentiate between popular and unpopular apps: installation size, number of images, and version of the SDK used. In our study, we also focus on aspects that the developer can control, related to source code (RQ1) and evolution (R2). Also in the Android ecosystem, Bavota et al. (2015) investigated whether popular apps are more or less likely to use APIs prone to fail. Specifically, they analyzed how 5848 apps from Google Play Market correlate with failures in imported APIs. The authors found that higher-rated apps (i.e., the most popular) use APIs that are less prone to crash than lower-ranked apps. In GitHub, Borges et al. (2016) investigated factors affecting the popularity of open-source projects; in this study, the metric star was used as a proxy for popularity. The authors found a strong correlation between stars and forks, and also that popularity may vary depending of the programming language, application domain, and repository owner.

In addition to Software Engineering, popularity is widely studied in other research areas. For example, studies analyzed the popularity of social media, such as YouTube

videos (Ahmed et al. 2013; Figueiredo et al. 2014; Chatzopoulou et al. 2010), Twitter hashtags (Lehmann et al. 2012; Ma et al. 2013), and likes and comments in Facebook (Swani et al. 2017). In this way, we can conclude that the study of aspects related to popularity is a topic of great concern and relevance in several research areas.

8 Conclusion

In this paper, we presented a large-scale study to better understand the characteristics of popular APIs. Particularly, we focused on code and evolutionary aspects (RQs 1 and 2), as well as on the client perspective (RQs 3 and 4). By assessing 1491 APIs provided Java standard APIs, Android framework, and 165 highly used libraries, we uncovered how worldwide APIs are actually implemented and maintained, and provided a set of lessons learned. We reiterate the most interesting findings from our experimental results:

- Popular APIs are largely different from ordinary APIs.
- The ecosystem plays an important role in the analysis of popular APIs.
- Popular APIs provide a large proportion of API elements.
- Popular APIs have more code comments.
- Popular APIs are more likely to change over time.

As future work, we plan to extend this study to analyze APIs provided by other programming languages and ecosystems, and assess other types of APIs than public classes (e.g., interfaces and generics). Moreover, we plan to enhance our set of metrics to cover other aspects of code quality and evolution (e.g., usability, design, and knowledge concentration), possibly with external artifacts, such as Stack Overflow questions and external documentation. We plan to introduce a qualitative analysis of our results, with the support of API developers and clients. Finally, we plan to study APIs in more fine-grained levels, such as methods, attributes, and annotations.

Funding information This research is financially supported by the CAPES and CNPq.

References

- Ahmed, M., Spagna, S., Huici, F., Niccolini, S. (2013). A peek into the future: predicting the evolution of popularity in user generated content. In *International Conference on Web Search and Data Mining*.
- Avelino, G., Passos, L., Hora, A., Valente, M.T. (2016). A novel approach for estimating truck factors. In *International Conference on Program Comprehension*.
- Barua, A., Thomas, S.W., Hassan, A.E. (2014). What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3), 619–654.
- Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C.E., Di Penta, M., Oliveto, R., Shihyanyk, D. (2015). The impact of API change-and fault-proneness on the user ratings of Android apps. *IEEE Transactions on Software Engineering*, 41(4).
- Bloch, J. (2006). How to design a good API and why it matters. In *SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*.
- Bogart, C., Kästner, C., Herbsleb, J., Thung, F. (2016). How to break an API: cost negotiation and community values in three software ecosystems. In *International Symposium on the Foundations of Software Engineering*.
- Borges, H., Hora, A., Valente, M.T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *International Conference on Software Maintenance and Evolution*.
- Borges, H., & Valente, M.T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*.

- Brito, A., Xavier, L., Hora, A., Valente, M.T. (2018a). APIDiff: Detecting API breaking changes. In *International Conference on Software Analysis, Evolution and Reengineering*.
- Brito, A., Xavier, L., Hora, A., Valente, M.T. (2018b). Why and how Java developers break APIs. In *International Conference on Software Analysis, Evolution and Reengineering*.
- Brito, G., Hora, A., Valente, M.T., Robbes, R. (2018c). On the use of replacement messages in API deprecation: An empirical study, (Vol. 137).
- Businge, J., Serebrenik, A., van den Brand, M. (2012). Survival of eclipse third-party plug-ins. In *International Conference on Software Maintenance*.
- Businge, J., Serebrenik, A., van den Brand, M. (2013a). Analyzing the Eclipse API usage: putting the developer in the loop. In *European Conference on Software Maintenance and Reengineering*.
- Businge, J., Serebrenik, A., van den Brand, M.G. (2013b). Eclipse API usage: the good and the bad. *Software Quality Journal*.
- Chatzopoulou, G., Sheng, C., Faloutsos, M. (2010). A first step towards understanding popularity in youtube. In *Conference on Computer Communications Workshops*.
- Couto, C., Pires, P., Valente, M.T., Bigonha, R., Anquetil, N. (2014). Predicting software defects with causality tests. *Journal of Systems and Software*, 93.
- Dagenais, B., & Robillard, M.P. (2008). Recommending adaptive changes for framework evolution. In *International Conference on Software engineering*.
- De Roover, C., Lämmel, R., Pek, E. (2013). Multi-dimensional exploration of api usage. In *International Conference on Program Comprehension*.
- Dig, D., & Johnson, R. (2005). The role of refactorings in API evolution. In *International Conference on Software Maintenance*.
- Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N. (2013). Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering*.
- Eisenberg, D.S., Stylos, J., Myers, B.A. (2010). Apatite: a new interface for exploring apis. In *Conference on Human Factors in Computing Systems*.
- Figueiredo, F., Almeida, J.M., Gonçalves, M.A., Benevenuto, F. (2014). On the dynamics of social media popularity: a youtube case study. *ACM Transactions on Internet Technology*, 14(4), 24.
- Fowler, M., & Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- Fritz, T., Ou, J., Murphy, G.C., Murphy-Hill, E. (2010). A degree-of-knowledge model to capture source code familiarity In *International Conference on Software Engineering*. ACM.
- Fritz, T., Murphy, G.C., Murphy-Hill, E., Ou, J., Hill, E. (2014). Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2), 14.
- Henning, M. (2007). API design matters. *Queue*, 5(4), 24–36.
- Herbsleb, J.D. (2007). Global software engineering: the future of socio-technical coordination. In *Future of Software Engineering*.
- Hintze, J.L., & Nelson, R.D. (1998). Violin plots: a box plot-density trace synergism. *The American Statistician*, 52(2), 181–184.
- Holmes, R., & Walker, R.J. (2007). Informing eclipse api production and consumption. In *OOPSLA Workshop on Eclipse Technology eXchange*.
- Hora, A., & Valente, M.T. (2015). apiwave: keeping track of API popularity and migration. In: International Conference on Software Maintenance and Evolution. <http://apiwave.com>.
- Hora, A., Valente, M.T., Robbes, R., Anquetil, n (2016). When should internal interfaces be promoted to public? In *International Symposium on Foundations of Software Engineering*.
- Hora, A., Robbes, R., Valente, M.T., Anquetil, N., Etien, A., Ducasse, S. (2018a). How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*, 26(1), 161–191.
- Hora, A., Silva, D., Robbes, R., Valente, M.T. (2018b). Assessing the threat of untracked changes in software evolution. In *International Conference on Software Engineering*.
- Konstantopoulos, D., Marien, J., Pinkerton, M., Braude, E. (2009). Best principles in the design of shared software. In *International Computer Software and Applications Conference*.
- Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K. (2018a). Do developers update their library dependencies?. *Empirical Software Engineering*, 23(1), 384–417.
- Kula, R.G., Ouni, A., German, D.M., Inoue, K. (2018b). An empirical study on the impact of refactoring activities on evolving client-used APIs. *Information and Software Technology*, 93, 186–199.
- Lehmann, J., Gonçalves, B., Ramasco, J.J., Cattuto, C. (2012). Dynamical classes of collective attention in twitter. In *International Conference on World Wide Web*.
- Lethbridge, T.C., Singer, J., Forward, A. (2003). How software engineers use documentation: the state of the practice. *IEEE Software*, 20(6), 35–39.

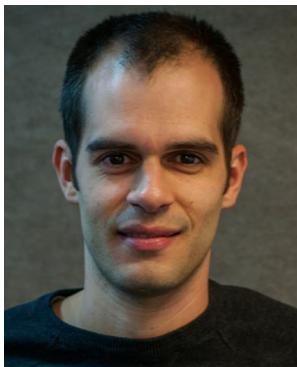
- Ma, Z., Sun, A., Cong, G. (2013). On predicting the popularity of newly emerging hashtags in twitter. *Journal of the Association for Information Science and Technology*, 64(7), 1399–1410.
- Maalej, W., & Robillard, M.P. (2013). Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 39(9), 1264–1282.
- Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., Nystrom, N. (2015). Use at your own risk: the java unsafe API in the wild. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
- McCabe, T.J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320.
- McDonnell, T., Ray, B., Kim, M. (2013). An empirical study of API stability and adoption in the Android ecosystem. In *International Conference on Software Maintenance*.
- Mileva, Y.M., Dallmeier, V., Burger, M., Zeller, A. (2009). Mining trends of library usage. In *Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops*.
- Mileva, Y.M., Dallmeier, V., Zeller, A. (2010). Mining API popularity. In *International Academic and Industrial Conference on Testing - Practice and Research Techniques*.
- Mileva, Y.M., Wasylkowski, A., Zeller, A. (2011). Mining evolution of object usage. In *European Conference on Object-Oriented Programming*.
- Mistrík, I., Grundy, J., Van der Hoek, A., Whitehead, J. (2010). Collaborative software engineering: challenges and prospects. In *Collaborative Software Engineering*.
- Moser, S., & Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9).
- Myers, B.A., & Stylos, J. (2016). Improving API usability. *Communications of the ACM*, 59(6), 62–69.
- Raemaekers, S., van Deursen, A., Visser, J. (2012). Measuring software library stability through historical version analysis. In *International Conference on Software Maintenance*.
- Reddy, M. (2011). *API Design for C++*. Morgan Kaufmann Publishers.
- Retherford, R.D., & Choe, M.K. (2011). *Statistical models for causal analysis*. Wiley.
- Robbes, R., Lungu, M., Röthlisberger, D. (2012). How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *International Symposium on the Foundations of Software Engineering*.
- Robillard, M.P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6), 27–34.
- Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J. (2006). Appropriate statistics for ordinal level data: should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *Annual Meeting of the Florida Association of Institutional Research*.
- Sawant, A.A., & Bacchelli, A. (2015). A dataset for API usage. In *Conference on Mining Software Repositories*.
- Sawant, A.A., Robbes, R., Bacchelli, A. (2016). On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In *32nd International Conference on Software Maintenance and Evolution*.
- Sawant, A.A., & Bacchelli, A. (2017a). Fine-grape: fine-grained api usage extractor—an approach and dataset to investigate api usage, (Vol. 22).
- Sawant, A.A., Robbes, R., Bacchelli, A. (2017b). On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK. *Empirical Software Engineering*.
- Stylos, J., Clarke, S., Myers, B. (2006). Comparing API design choices with usability studies: a case study and future directions. In *Workshop of the Psychology of Programming Interest Group*.
- Stylos, J., & Myers, B. (2007). Mapping the space of API design decisions. In *Symposium on Visual Languages and Human-Centric Computing*.
- Swani, K., Milne, G.R., Brown, B.P., Assaf, A.G., Donthu, N. (2017). What messages to post? evaluating the popularity of social media communications in business versus consumer markets. *Industrial Marketing Management*, 62, 77–87.
- Thummalapenta, S., & Xie, T. (2008). Spotweb: detecting framework hotspots via mining open source repositories on the web. In *International Working Conference on Mining Software Repositories*.
- Tian, Y., Nagappan, M., Lo, D., Hassan, A.E. (2014). What are the characteristics of high-rated apps? a case study on free Android applications. In *International Conference on Software Maintenance and Evolution*.
- Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyanyk, D. (2015). When and why your code starts to smell bad. In *International Conference on Software Engineering*.
- Wu, W., Gueheneuc, Y.G., Antoniol, G., Kim, M. (2010). AURA: a hybrid approach to identify framework evolution. In *International Conference on Software Engineering*.
- Xavier, L., Brito, A., Hora, A., Valente, M.T. (2017). Historical and impact analysis of API breaking changes: a large scale study. In *International Conference on Software Analysis, Evolution and Reengineering*.

- Zerouali, A., & Mens, T. (2017). Analyzing the evolution of testing library usage in open source java projects. In *International Conference on Software Analysis, Evolution and Reengineering*.
- Zibran, M.F., Eishita, F.Z., Roy, C.K. (2011). Useful, but usable? factors affecting the usability of APIs. In *Working Conference on Reverse Engineering*.
- Zibran, M. (2008). What makes APIs difficult to use. *International Journal of Computer Science and Network Security (IJCSNS)*, 8(4), 255–261.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Caroline Lima is a professor in the Faculty of Computing at the Federal University of Mato Grosso do Sul (UFMS). Lima received a MSc in Computer Science from the Federal University of Mato Grosso do Sul. Her research interests include software evolution and software repository mining.



Andre Hora is a professor in the Computer Science Department at the Federal University of Minas Gerais (UFMG). His research interests include software evolution, software repository mining, and empirical software engineering. Hora received a PhD in Computer Science from the University of Lille. He was a Postdoctoral researcher at the ASERG/UFMG group during two years and a software developer at Inria/Lille during one year. Contact: www.dcc.ufmg.br/~andrehora.