Noname manuscript No.
(will be inserted by the editor)

fine-GRAPE: fine-Grained APi usage Extractor – An Approach and Dataset to Investigate API Usage

Anand Ashok Sawant · Alberto Bacchelli

the date of receipt and acceptance should be inserted later

Abstract An Application Programming Interface (API) provides a set of functionalities to a developer with the aim of enabling reuse. APIs have been investigated from different angles such as popularity usage and evolution to get a better understanding of their various characteristics. For such studies, software repositories are mined for API usage examples. However, many of the mining algorithms used for such purposes do not take type information into account. Thus making the results unreliable. In this paper, we aim to rectify this by introducing fine-GRAPE, an approach that produces fine-grained API usage information by taking advantage of type information while mining API method invocations and annotation. By means of fine-GRAPE, we investigate API usages from Java projects hosted on GitHub. We select five of the most popular APIs across GitHub Java projects and collect historical API usage information by mining both the release history of these APIs and the code history of every project that uses them. We perform two case studies on the resulting dataset. The first measures the lag time of each client. The second investigates the percentage of used API features. In the first case we find that for APIs that release more frequently clients are far less likely to upgrade to a more recent version of the API as opposed to clients of APIs that release infrequently. The second case study shows us that for most APIs there is a small number of features that is actually used and most of these features relate to those that have been introduced early in the APIs lifecycle.

Keywords Application Programming Interface · API usage · API popularity · Dataset

A.A. Sawant
Mekelweg 4,
2628 CD, Delft
Tel.: +31 (0)15 27 89803
E-mail: A.A.Sawant@tudelft.nl

A. Bacchelli
E-mail: A.Bacchelli@tudelft.nl

1 Introduction

An Application Programming Interface (API) is a set of functionalities provided by a third-party component (*e.g.*, library and framework) that is made available to software developers. APIs are extremely popular as they promote reuse of existing software systems [1].

The research community has used API usage data for various purposes such as measuring of popularity trends [2], charting API evolution [3], and API usage recommendation systems [4].

For example, Xie *et al.* have developed a tool called MAPO wherein they have attempted to mine API usage for the purpose of providing developers API usage patterns [5,6]. Based on a developers' need MAPO recommends various code snippets mined from other open source projects. This is one of the first systems wherein API usage recommendation leveraged open source projects to provide code samples. Another example is the work by Lämmel *et al.* wherein they mined data from Sourceforge and performed an API usage analysis of Java clients. Based on the data that they collected they present statistics on the percentage of an API that is used by clients.

One of the major drawbacks of the current approaches that investigate APIs is that they heavily rely on API usage information (for example to derive popularity, evolution, and usage patterns) that is *approximate*. In fact, one of the modern techniques considers as "usage" information what can be gathered from file imports (*e.g.*, `import` in Java) and the occurrence of method names in files.

This data is an approximation as there is no type checking to verify that a method invocation truly does belong to the API in question and that the imported libraries are used. Furthermore, information related to the version of the API is not taken into account. Finally, previous work was based on small sample sizes in terms of number of projects analyzed. This could result in an inaccurate representation of the real world situation.

With the current work, we try to overcome the aforementioned issues by devising fine-GRAPE (fine-GRained APi usage Extractor), an approach to extract type-checked API method invocation information from Java programs and we use it to collect detailed historical information on five APIs and how their public methods are used over the course of their entire lifetime by 20,263 client projects.

In particular, we collect data from the open source software (OSS) repositories on GitHub. GitHub in recent years has become the most popular platform for OSS developers, as it offers distributed version control, a pull-based development model, and social features [7]. We consider Java projects hosted on GitHub that offer APIs and quantify their popularity among other projects hosted on the same platform. We select 5 representative projects (from now on, we call them only *API*s to avoid confusion with client projects) and analyze their entire history to collect information on their usage. We get fine-grained information about method calls using a custom type resolution that does not require to compile the projects.

The result is an extensive dataset for research on API usage. It is our hope that our data collection approach and dataset not only will trigger further research based on finer-grained and vast information, but also make it easier to replicate studies and share analyses.

For example, with our dataset the following two studies can be conducted:

First, the evolution of the features of the API can be studied. An analysis of the evolution can give an indication as to what has made the API popular. This can be used to design and carry out studies on understanding what precisely makes a certain API more popular than other APIs that offer a similar service. Moreover, API evolution information gives an indication as to exactly at what point of time the API became popular, thus it can be studied in coordination with other events occurring to the project.

Second, a large set of API usage examples is a solid base for recommendation systems. One of the most effective ways to learn about an API is by seeing samples [8] of the code in actual use. By having a set of accurate API usages at ones' disposal, this task can be simplified and useful recommendations can be made to the developer; similarly to what has been done, for example, with Stack Overflow posts [9].

In our previous work titled "A dataset for API Usage" [10], we presented our dataset along with a few details on the methodology used to mine the data. In this paper, we go into more detail into the methodology of our mining process and conduct two case studies on the collected data which make no use of additional information.

The first case is used to display the wide range of version information that we have at our disposal. This data is used to analyze the amount of time by which a client of an API lags behind the latest version of the API. Also, the version information is used to calculate as to what the most popular version of an API is. This study can help us gain insights into the API upgrading behavior of clients.

The second case showcases the type resolved method invocation data that is present in our database. We use this to measure the popularity of the various features provided by an API and based on this mark the parts of an API that are used and those that are not. With this information an API developer can see what parts of the API to focus on for maintenance and extension.

The first study provided initial evidence of a possible distinction between upgrade behavior of clients of APIs that release frequently compared to those that release infrequently. In the former case, we found that clients tend to hang back and not upgrade immediately; whereas, in the latter case, clients tend to upgrade to the latest version. The results of the second case study highlight that only a small part of an API is used by clients. This finding requires further investigation as there is a case to be made that many new features that are being added to an API are not really being adopted by the clients themselves.

This paper is organized as follows: Section 2 presents the approach that has been applied to mine this data. For the ease of future users of this dataset an overview of the dataset and some introductory statistics of it can be found in section 3. Section 4 presents the two case studies that we performed on this dataset. In section 5 we describe the limitations of our approach and the dataset itself. Section 6 concludes this article.

2 Approach

We present the 2-step approach that we use to collect fine-grained type-resolved API usage information. (1) We collect data on project level API usage from projects mining open source code hosting platforms (we target such platforms due to the large number of projects they hosted) and use it to rank APIs according to their popularity to select an interesting sample of APIs to form our dataset; (2) apply our technique, fine-GRAPE, to gather fine-grained type-based information on API usages and collect historical usage data by traversing the history of each file of each API client.

2.1 Mining of coarse grained usage

In the construction of this dataset, we limit ourselves to the Java programming language, one of the most popular programming languages currently in use [11]. This reduces the types of programs that we can analyze, but has a number of advantages: (1) Due to the popularity of Java there would be a large source of API client projects available for analysis; (2) Java is a statically typed language, thus making the collection of type-resolved API usages easier; (3) it allows us to have a more defined focus and more thoroughly test and refine fine-GRAPE Future work can be to extend it to other typed-languages, such as C#.

To ease the collection of data regarding project dependencies on APIs, we found it useful to focus on projects that use build automation tools. In particular, we collect data from projects using Maven, one of the most popular Java build tools [12]. Maven employs the use of a Project Object Model (POM) files to describe all the dependencies and targets of a certain project. POM files contain artifact ID and version of each project's dependency, thus allowing us to know exactly which APIs (and version) a project uses. The following is an example of a POM file entry:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
</dependency>
```

In the dependency tag from a sample POM file pictured above, we see that the JUnit dependency is being declared. We find the APIs name in the `artifactId` tag. The `groupId` tag generally contains the name of the organization that has released the API, in this case it matches the artifactId. However, there are other cases such as the JBoss-Logging API for which the `groupID` is `org.jboss.logging` and the `artifactId` is `jboss-logging`. The version of JUnit to be included as a dependency is specified in the version tag and in this case it is version `4.8.2`.

2.2 Fine-grained API usage

To ensure that precise API usage information is collected, one has to reliably link each method invocation or annotation usage to class in the API to which it belongs. This can be achieved in five ways:

Text matching: This is one of the most frequently used techniques to mine API usage. For example, it has been used in the investigation into API popularity performed by Mileva *et al.* [2]. The underlying idea is to match explicit imports and corresponding method invocations directly in the text of source code files.

Bytecode analysis: Each Java file produces one or more class files when compiled, which contain Java bytecode that is platform independent. Another technique to mine API usage is to parse byte code in these class files to find all method invocations and annotation usages along with the class to which they belong to. This approach guarantees accuracy as the class files contain all information related to Java program in the Java file in question.

Partial program analysis: Dagenais *et al.* have created an Eclipse plugin called Partial Program Analysis (PPA) [13]. This plugin parses incomplete files and recovers type bindings on method invocations and annotations, thus identifying the API class to which a certain API usage belongs.

Dynamic analysis: Dynamic analysis is a process by which the execution trace of a program is captured as it is being executed. This can be a reliable method of determining the invocation *sequence* in a program as it can even handle the case where type of an object is decided at runtime. Performing dynamic analysis has the potential of being highly accurate as the invocations in the trace are type-resolved being recovered from running of bytecode.

AST analysis: Syntactically correct Java files can be transformed into an Abstract Syntax Tree (AST). An AST is a tree based representation of code where each variable declaration, statement, or invocation forms a node of the tree. This AST can be parsed by using a standard Java AST parser. The Java AST parser can also recover type based information at each step, which aids in ensuring accuracy when it comes to making a connection between an API invocation and the class it belongs to.

All five of the aforementioned approaches can be applied for the purpose of collecting API usage data, but come with different benefits and drawbacks.

The text-matching-based approach proves especially problematic in the case of imported API classes that share method names, because method invocations may not be disambiguated without type-information. Although some analysis tools used in dynamic languages [14] handle these cases through the notion of 'candidate' classes, this approach is sub-optimal for typed languages where more precise information is available.

The bytecode analysis approach is more precise, as bytecode is guaranteed to have the most accurate information, but it has two different issues:

1. Processing class files requires these files to be available, which, in turn, requires being able to compile the Java sources and, typically, the whole project. Even though all the projects under consideration use Maven for the purpose of building, this does not guarantee that they can be built. If a project is not built, then the class files associated with this project cannot be analyzed, thus resulting in a dropped project.
2. To analyze the history of method invocations it is necessary to checkout each version of every file in a project and analyze it. However, checking out every version of a file and then building the project can be problematic as there would be an ultra-large number of project builds to be performed. In addition to the time costs, there would still be no warranty that data would not be lost due build failures.

The partial program analysis approach has been extensively tested by Dagenais *et al.* [13] to show that method invocations can be type resolved in incomplete Java files. This is a massive advantage as it implies that even without building each API client one can still conduct a thorough analysis of the usage of an API artifact. However, the implementation of this technique relies on Eclipse context, thus all parsing and type resolution of Java files can only be done in the context of an Eclipse plugin. This requires that each and every Java file is imported into an Eclipse workspace before it can be analyzed. This hinders the scalability of this approach to large number of projects.

Dynamic analysis techniques result in an accurate set of type resolved invocations. However, they require the execution of the code to acquire a trace. This is a limitation as not all client code might be runnable. An alternative would be to have a sufficient set of tests that would execute all parts of the program so that traces can be obtained. This too might be unfeasible as many projects may not have a proper test suite[15]. Finally, this technique would also suffer from the same limitations as the bytecode analysis technique; where analyzing every version of every file would require a large effort.

2.3 fine-GRAPE

Due to the various issues related to first four techniques, we deem the most suitable technique to be the AST based one. This technique utilizes the JDT Java AST Parser [16], *i.e.*, the parser used in the Eclipse IDE for continuous compilation in background. This parser handles partial compilation: When it receives in input a source code file and a Java Archive (JAR) file with possibly imported libraries, it is capable of resolving the type of methods invocation and annotations of everything defined in the code file or in the provided jar.

This will allow us to parse standalone files, and even incomplete files in a quick enough way such that we can collect data from a large number of files and their histories in a time effective manner.

We created fine-GRAPE that, using the aforementioned AST parsing technique, collects the entire history of usage of API artifacts over different versions. In practice, we downloaded all the JAR files corresponding to the releases of the API projects chosen. Although this has been done manually in the study presented here, this process of downloading the JAR files has been automated in the current version for the ease of the user. Then, fine-GRAPE uses Git to obtain the history of each file in the client projects and runs on each file retrieved from the repository and the JAR with the corresponding version of the API that the client project declares in Maven at the time of the commit of the file. The fine-GRAPE leverages the visitor pattern that is provided by the JDT Java AST parser to visit all nodes in the AST of a source code file of the type method invocation or annotation. These nodes are type resolved and are stored in a temporary data structure while we parse all files associated with one client project. This results in accurate type-resolved method invocation references for the considered client projects through their whole history. Once the parsing is done for all the files and their respective histories in the client, all the data that has been collected is transformed into a relational database model and is written to the database.

An API usage dataset can also contain the information on the method, annotations, and classes that are present in every version of every API for which usage data has been gathered such that any kind of complex analysis can be performed. In the previous steps we have already downloaded the API JAR files for each version of the API that is used by a client. These JAR files are made up of compiled class files, where each class file relates to one Java source code file. fine-GRAPE then analyzes these JAR files with the help of the bytecode analysis tool ASM [17], and for each file the method, class and annotation declarations are extracted.

2.4 Scalability of the approach

The approach that we have outlined runs on a large number of API client projects in a short amount of time. In its most recent state, all parts of the process are completely automated, thus needing a minimum of manual intervention. A user of the fine-GRAPE tool has to just specify the API which is to be mined, and this will result in a database that contains type-resolved invocations made to an API.

We benchmarked the amount of time it takes to process a single file. To run our benchmark, we used a server with two Intel Xeon E5-2643 V2 processors. Each processor consists of 6 cores and runs at a clock speed of 3.5 GHz. We ran our benchmark on 2,045 files from 20 client projects. To get an accurate picture, this benchmark was repeated 10 times. Based on this we found that the average amount of time spent on a single file was 165 milliseconds, the median was 31 milliseconds, the maximum was 1,815 ms for a large file.

2.5 Comparison to existing techniques

Previous work mined API usage examples, for example in the context of code completion, code recommendation, and bug finding. We see how the most representative of these mining approaches implemented in the past relate to the one we present here.

One of the more popular applications of API usage datasets is in the creation of code recommendation tools. In this field one of the more known tools is MAPO by Xie *et al.* [5,6]. The goal of MAPO is to recommend relevant code samples to developers. MAPO runs its analyzer on source code files from open source repositories. MAPO uses the JDT compiler to compile a file and recover type-resolved API usages. These fine-grained API usages are then clustered using the frequent itemset mining technique [18]. In more recent developments tools such as UP-Miner [19] have been developed to mine high coverage usage patterns from open source repositories by using multiple clustering steps. Differently from fine-GRAPE, none of the approaches used here take version of the various APIs used into account. Moreover, our approach as opposed to theirs does not require the building of the files and has no need for all dependencies to the resolved to run.

Mining of API usage patterns has also been done to detect bugs by finding erroneous usage patterns. To this end, researchers developed tools such as Dynamine [20], JADET [21], Alattin [22] and PR-Miner [23]. All these tools rely on the same mining technique *i.e.*, frequent itemset mining [18]. The idea behind this technique is that statements that occur frequently together can be considered to be a usage pattern. This technique can result in a high number of false positives, due to the lack of type information. fine-GRAPE tackles this problem by taking advantage of type information.

The earliest technique that was employed in mining API usage was used by the tool CodeWeb [24] that was developed by Amir Michail. More recently it has been employed in the tool Sourcerer [25] as well. This technique employs a data mining technique that is called *generalized association rule mining.* An association rule is of the form $(\bigwedge_{x \epsilon X} x) \Rightarrow (\bigwedge_{y \epsilon Y} y)$. This implies that for an event x that takes place, then an event y will also take place with a certain confidence interval. The generalized association rule takes not just this into account but also takes a node's descendants into account as well. These descendants represent specializations of that node. This allows this technique to take class hierarchies into account while mining reuse patterns. However, just like frequent itemset mining this can result in false positives due to the lack of type information.

Recently, Moreno *et al.* [26] presented a technique to mine API usages using type resolved ASTs. Differently from fine-GRAPE, the approach they propose builds the code of each client to retrieve type resolved ASTs. As previously mentioned in the context of bytecode analysis, this could result in the loss of data, as some client projects may not build, and low scalability.

3 A Dataset for API Usage

Using fine-GRAPE we build a large dataset of usage of popular APIs. Our dataset is constructed using data obtained from the open source code hosting platform GitHub. GitHub stores more than 10 million repositories [27] written in different languages and using a diverse set of build automation tools and library management systems.

3.1 Coarse-grained API usage: The most popular APIs

To determine the popularity of APIs on a coarse-grained level (*i.e.*, project level), we parse POM files for all GitHub based Java projects that use Maven (ca. 42,000). The POM files were found in the master branch of approximately 250,000 active Java projects that are hosted on GitHub.[1] Figure 1 shows a partial view of the results with the 20 most popular APIs in terms of how many GitHub projects depend on them.

This is in-line with a previous analysis of this type published by Primat as a blog post [28]. Interestingly, our results show that JUnit is by far the most popular, while Primat's results report that JUnit is just as popular as SLF4J. We speculate that this discrepancy can be caused by the different sampling approach (he sampled 10,000 projects on GitHub, while we sampled about 42,000 on GitHub), further research can be conducted to investigate this aspect more in detail.

3.2 Selected APIs

We used our coarse-grained analysis of popularity as a first step to select API projects to populate our database. To ensure that the selected API projects offer rich information on API usage and its evolution, rather than just sporadic use by a small number of projects, we consider projects with the following feature: (1) have a broad popularity for their public APIs (*i.e.*, they are in the top 1% of projects by the number of client projects), (2) have an established and reasonably large code base (*i.e.*, they have at least 150 classes in their history), (3) and are evolved and maintained (*i.e.*, they have at least 10 commits per week in their lifetime). Based on these characteristics, we eventually select the five APIs summarized in Table 1, namely Spring, Hibernate, Guava, and Guice and Easymock. We decide to remove JUnit, being an outlier in popularity and having a small code base that does not respect our requirements. We keep Easymock, despite its small number of classes and relatively low amount

[1] As marked by GHTorrent [27] in September 2014

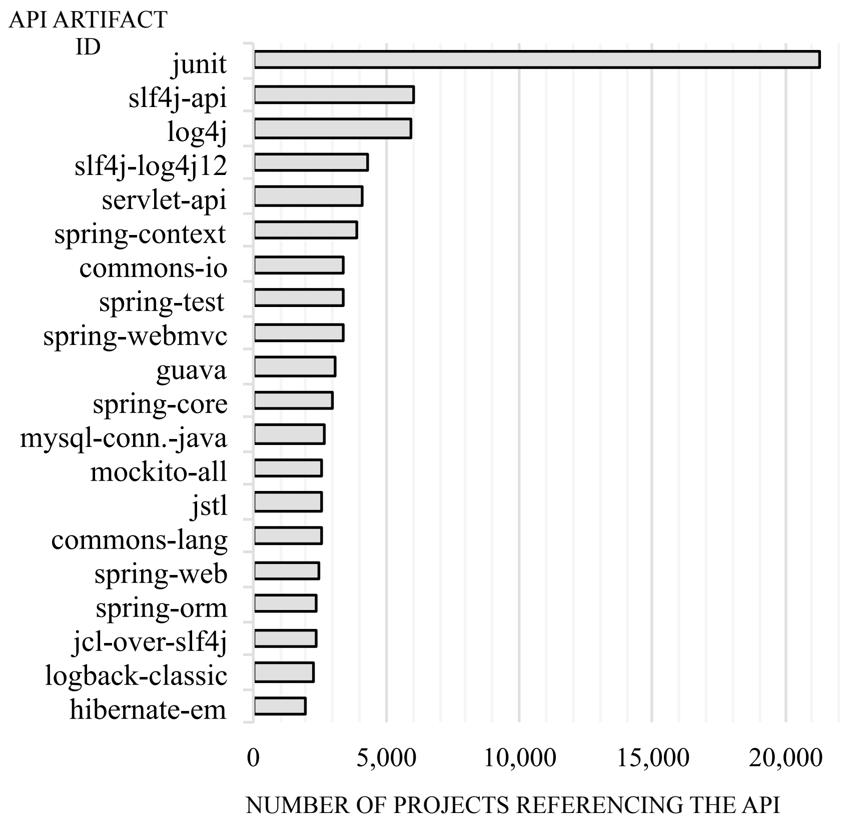

Fig. 1 Popularity of APIs referenced on Github

of activity in it's repository (ca. 4 commits per week) to add variety to our sample. The chosen APIs are used by clients in different ways: *e.g.*, Guice clients use it through annotations, while Guava clients instantiate an instance of a Guava class and then interact with it through method invocations.

In the following, a brief explanation of the domain of each API:

1. **Guava** is the new name of the original Google collections and Google commons APIs. It provides immutable collections, new collectsion such as multiset and multimaps and finally some new collection utilities that are not provided in the Java SDK. Guava's collections can be accessed by method invocations on instantiated instances of the classes built into Guava.
2. **Guice** is a dependency injection library provided by Google. Dependency injection is a design pattern that separates behavioral specification and dependency resolution. Guice allows developers to inject dependencies in their applications with the usage of annotations.

3. **Spring** is a framework that provides an Inversion of Control(IoC) container. This allows developers to access Java objects with the help of reflection. The Spring framework comprises of a lot of sub projects, however we choose to focus on just the spring-core, spring-context and spring-test modules due to their relatively high popularity. The features provided by Spring are accessed in a mixture of method invocations and annotations.
4. **Hibernate Object Relational Mapping (ORM)** provides a framework for mapping an object oriented domain to a relational database domain. It is made up of a number of components that can be used, however we focus on just two of the more popular one *i.e.*, hibernate-core and hibernate-entity manager. Hibernate exposes its APIs as a set of method invocations that can be made on the classes defined by Hibernate.
5. **Easymock** is a testing framework that allows for the mocking of Java objects during testing. Easymock exposes its API to developers by way of both annotations and method invocations.

Table 1 Subject APIs

API & GitHub repo	Inception	Releases	Unique Entities	
			Classes	Methods
Guava google/guava	Apr 2010	18	2,310	14,828
Guice google/guice	Jun 2007	8	319	1,999
Spring spring-framework	Feb 2007	40	5,376	41,948
Hibernate hibernate/hibernate-orm	Nov 2008	77	2,037	11,625
EasyMock easymock/easymock	Feb 2006	14	102	623

3.3 Data Organization

We apply the approach outlined in Section 2 and store all the data collected from all the client GitHub projects and API projects in a relational database, precisely PostgreSQL [29]. We have chosen a relational database because the usage information that we collect can be naturally expressed in forms of relations among the entities. Also we can leverage SQL functionalities to perform some initial analysis and data pruning.

Figure 2 shows the database schema for our dataset. On the one hand we have information for each client project: The PROJECTS table is the starting point and stores a project's name and its unique ID. Connected to this we have PROJECTDEPENDENCY table, which stores information collected from the Maven POM files about the project's dependencies. We use a DATE_COMMIT attribute to trace when a project starts including a certain dependency in its

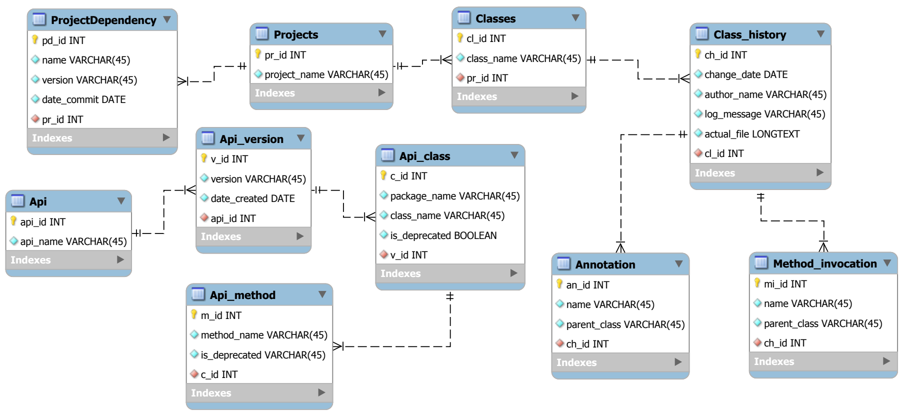

Fig. 2 Database Schema For The Fine-grained API Usage Dataset

history. The Classes table contains one row per each uniquely named class in the project; in the table Class_history we store the different versions of a class (including its textual content, actual_file) and connect it to the tables Method_invocation and Annotation where information about API usages are stored. On the other hand, the database stores information about API projects, in the tables prefixed with Api. The starting point is the table Api that stores the project name and it is connected to all its versions (table Api_version, which also stores the date of creation), which are in turn connected classes (Api_class) and their methods (Api_method) that also store information about deprecation. Note that in the case of annotations we do not really collect them in a separate table as annotations are defined as classes in Java.

A coarse-grained connection between a client and an API is done with a SQL query on the tables ProjectDependency, Api and Api_version. The finer-grained connection is obtained by also joining Method_invocation/Annotation and Api_class on parent class names & Method_invocation/Annotation and Api_method on method names.

The full dataset is available as a PostgreSQL data dump on FigShare [30], under the CC-BY license. For platform limitations on the file size the dump has been split in various tar.gz compressed files, for a total download size of 51.7 GB. The dataset uncompressed requires 62.3 GB of disk space.

3.4 Introductory Statistics

Table 2 shows an introductory view about the collected usage data. In the case of Guava for example, even though version 18 is the latest (see Table 1), version 14.0.1 is the most popular among clients. APIs such as Spring, Hibernate and Guice predominantly expose their APIs as annotations, however we see also a large use of the methods they expose. The earliest usages of Easymock and Guice are outliers as GitHub as a platform was launched in 2008, thus the repositories that refer to these APIs were moved to GitHub as existing projects.

Table 2 Introductory usage statistics

API	Most popular release	Usage across history	
		Invocations	Annotations
Guava	14.0.1	1,148,412	—
Guice	3.0	59,097	48,945
Spring	3.1.1	19,894	40,525
Hibernate	3.6	196,169	16,259
EasyMock	3.0	38,523	—

3.5 Comparison to existing datasets

The work of Lämmel *et al.* [31] is the closest to the dataset we created with fine-GRAPE. They target open source Java projects hosted on the Sourceforge platform and their API usage mining method relies on type resolved ASTs. To acquire these type resolved ASTs they build the APIs client projects and resolve all of its dependencies. This dataset contains a total of 6,286 client projects that have been analyzed and the invocations for 69 distinct APIs have been identified.

Our dataset as well as that of Lämmel *et al.* target Java based projects, though the clients that have been analyzed during the construction of our dataset were acquired from GitHub as opposed to Sourceforge. Our approach also relies on type resolved Java ASTs, but we do not build the client projects as fine-GRAPE is based on a technique able to resolve parsing of a standalone Java source file. In addition, the dataset by Lämmel *et al.* only analyzes the latest build. In terms of size this dataset is comprised of usage information gleaned from 20,263 projects as opposed to the 6,286 projects that make up the Lämmel *et al.* dataset. However, this dataset contains information on only 5 APIs whereas Lämmel *et al.* identified usages from 69 APIs.

4 Case studies

We present two case studies to showcase the value of our dataset and to provide examples for others to use it. We focus on case studies that require minimal processing of the data and are just on basic queries to our dataset.

4.1 **Case 1**: Do clients of APIs migrate to a new version of the API?

As with other software systems, APIs also evolve over time. A newer version may replace an old functionality with a new one, may introduce a completely

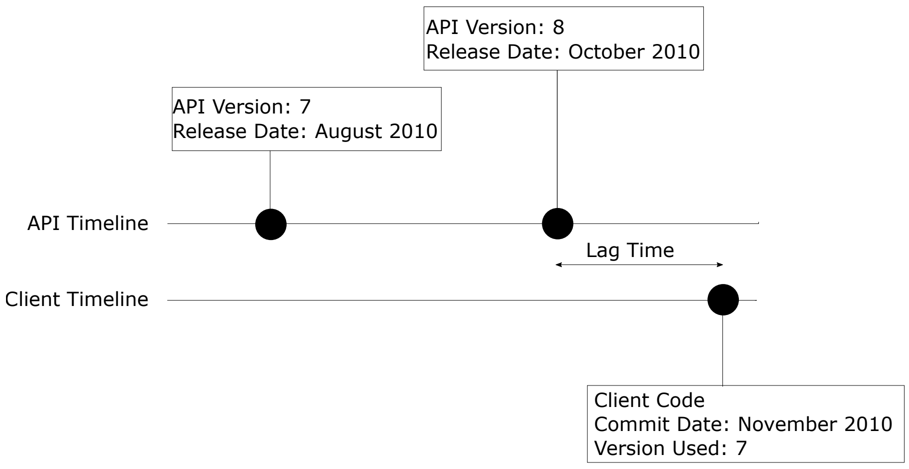

Fig. 3 An example of the lag time metric inspired by McDonnell *et al.* [32]

new functionality, or may fix a bug in an existing functionality. Some infamous APIs, such as Apache Commons-IO, are stagnating since long time without any major changes taking place, but to build our API dataset we took care of selecting APIs that are under active development, so that we could use it to analyze as to whether clients react to a newer version of an API.

4.1.1 Methodology

We use the *lag time* metric, as previously defined by McDonnell *et al.* [32], to determine the amount of time a client is behind the latest release of an API that it is using. Lag time is defined as the amount of time elapsed between a client's API reference and the release date of the latest version. A client lags if it uses an old version of an API when a newer version has already been released. For example, in Figure 3, client uses API version 7 despite version 8 being already released. The time difference between the client committing code using an older version and the release date of a newer version of the API is measured as the lag time.

In practice, we consider the commit date of each method invocation (this is done by performing a join on the METHOD_INVOCATION and CLASS_HISTORY tables), determine the version of the API that was being used by the client at the time of the commit (the PROJECT_DEPENDENCY table contains information on the versions of the API being used by the client and the date at which the usage of a certain version was employed), then consider the release date of the latest version of the API that existed at the time of the commit (this data can be obtained form the API_VERSION table in the database), and finally combine this information to calculate the lag time for each reference to the API and plot the probability density.

Lag time can indicate how far a project is at the time of usage of an API artifact, but it does not give a complete picture of the most recent state of all the clients using an API. To this end, we complement lag time analysis with the analysis of the most popular versions of each API, based on the latest snapshot of each client of the API (we achieve this by querying the PROJECT_DEPENDENCY table to get the latest information on clients).

4.1.2 Results

Results are summarized in four figures. Figure 4 shows the probability density of lag time in days, as measured from API clients, and Figure 5 shows the distribution of this lag time. Figure 5 shows frequency of adoption of specific releases: the three most popular ones, the latest release (available at the creation of this dataset), and all the other releases. Table 3 further specify the dates in which these releases were made public and provides absolute numbers. Finally, Figure 7 depicts the frequency and number of releases per API. The data we have ranges from 2004 to 2014, however for space reasons we only depict the range 2009 to 2014. Each year is divided into 3 slots of 4 month periods, and the number of releases in each of these periods is depicted by the size of the black circle.

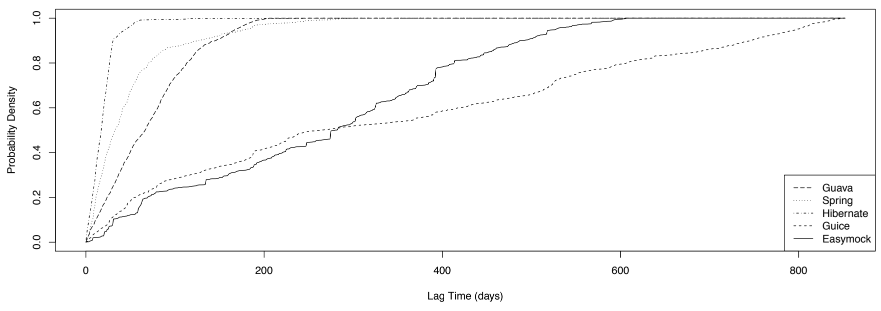

Fig. 4 Probability density of lag time in days, by API

Guava. In the case of the 3,013 Guava clients on GitHub the lag time varies between 1 day and 206 days. The median lag time for these projects is 67 days. The average amount of time a project lags behind the latest release is 72 days. Figure 4 shows the cumulative distribution of lag time across all clients. Since Guava generally releases 5 versions on average per year, it is not entirely implausible that some clients may be one or two versions behind at the time of usage of an API artifact.

Although the latest (as of September 2014) version of Guava is 18, the most popular one is 14 with almost one third of the clients using this version (as shown in Figure 5). Despite 4 versions being released after version 14 none of them figure in the top 5 of most popular versions. Version 18 has

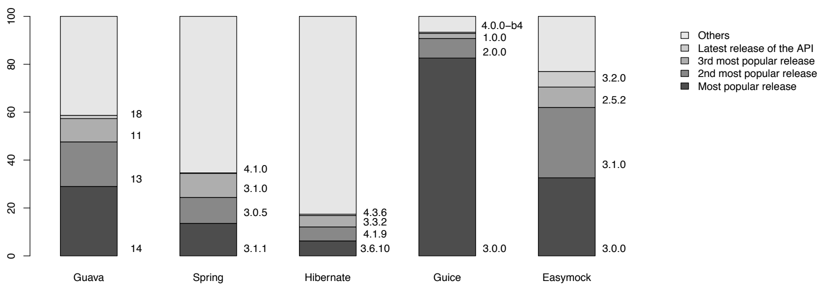

Fig. 5 Proportion of release adoption, split in the 3 most popular, the latest, and all the other releases, by API

Table 3 Publication date, by API, of the 3 most popular and latest releases, sorted by the number of their clients

API	Release	Release Date	Num of clients	(%)
Guava	14	03-2013	868	(29%)
	13	08-2012	557	(19%)
	11	02-2012	291	(10%)
	18	08-2014	41	(1%)
Spring	3.1.1	02-2012	2,013	(14%)
	3.0.5	10-2010	1,602	(11%)
	3.1.0	12-2011	1,489	(10%)
	4.1.0	10-2014	30	(0.2%)
Hibernate	3.6.10	02-2012	376	(6%)
	4.1.9	12-2012	352	(6%)
	3.3.2	06-2009	288	(5%)
	4.3.6	07-2014	32	(0.5%)
Guice	3.0.0	03-2011	536	(83%)
	2.0.0	07-2009	53	(8%)
	1.0.0	05-2009	14	(2%)
	4.0.0-b4	03-2014	3	(0.5%)
Easymock	3.0.0	05-2010	211	(33%)
	3.1.0	11-2011	190	(29%)
	2.5.2	09-2009	55	(9%)
	3.2.0	07-2013	42	(6%)

been adopted by very few clients (41 out of 3,013). None of the other newer versions (16 and 17) make it in the top 5 either.

Spring. Spring clients lag behind the latest release up to a maximum of 304 days. The median lag time is 33 days and the first quartile is 15 days. The third quartile of the distribution is 60 days. The average amount of lag time for the usages of various API artifacts is 50 days. Spring is a relatively active API and releases an average of 7 versions (including minor versions and revisions) per year (Figure 7).

At the time of collection of this data, the Spring API had just released version 4.1.0 and only a small portion (30) of projects have adopted it. The most popular version is 3.1.1 (2,013 projects) as is depicted in Figure 5.

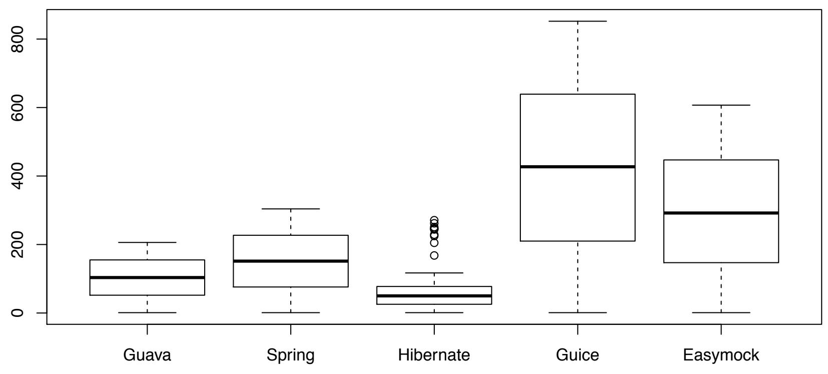

Fig. 6 Lag time distribution in days, by API

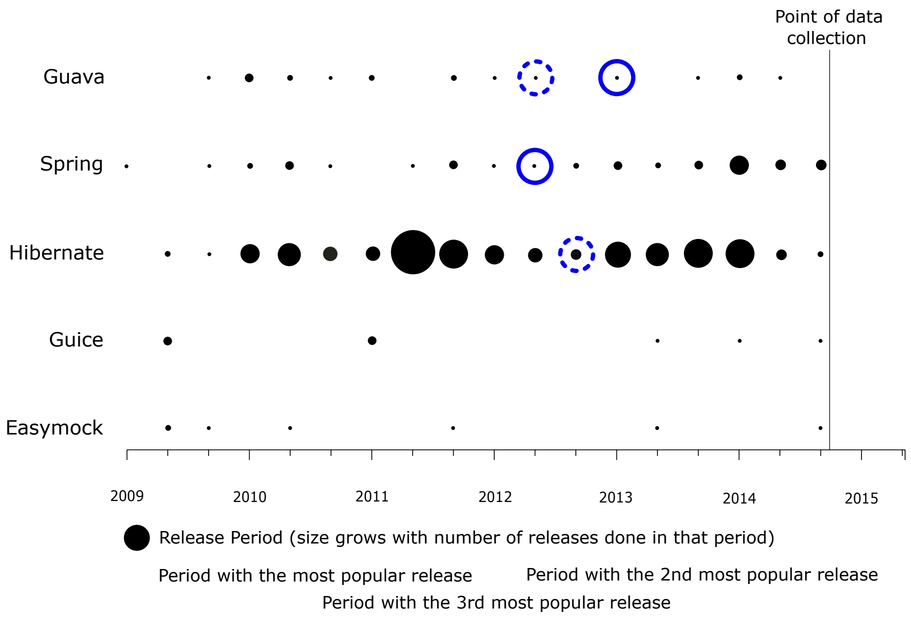

Fig. 7 Release frequency for each API from 2009 (the dataset covers from 2004)

We see that despite the major version 4 of the Spring API being released in December 2013, the most popular major version remains 3. In our dataset, 344 projects still use version 2 of the API and 12 use version 1.

Hibernate. The maximum lag time observed over all the usages of Hibernate artifacts is 271 days. The median lag time is 18 days, and the first quartile is just 10 days. The third quartile is also just 26 days. The average lag time over all the invocations is 19 days. We see in Figure 4 that most invocations to Hibernate API do not lag behind the latest release considerably, especially in relation to the other APIs, although a few outliers exist. Hibernate releases 17 versions (including minor versions and revisions) per year (Figure 7).

Version 4.3.6 of Hibernate is the latest release that available on Maven central at the dataset creation time. A very small portion of projects (32) use this version, and the most popular version is version 3.6.10, *i.e.*, the last release with major version 3. We see that a large number of clients have migrated to early versions of major version 4. For instance, version 4.1.9 is almost (352 projects versus 376 projects) as popular as version 3.6.10 (shown in Figure 5). Interestingly, in the case of Hibernate, from our data we see that there is not a clearly dominant version as all the other versions of Hibernate make up about three fourths of the current usage statistics.

Guice. Among all usages of the Guice API, the largest lag time is 852 days. The median lag time is 265 days and the first quartile of the distribution is 80 days. The average of all the lag times is 338 days. The third quartile is 551 days, showing that a lot of projects have a very high lag time. Figure 4 shows the cumulative distribution of lag times across all Guice clients. Guice is a young API and, relatively to the other APIs, releases are few and far between (10 releases over 6 years, with no releases on 2010 or 2012, Figure 7).

The latest version of Guice that has been released, before the construction of our dataset, is the fourth beta of version 4 (September 2014). Version 3 is unequivocally the most adopted version of Guice, as seen in Figure 5. This version was released in March of 2011 and since then there have been betas for version 4 released in 2013 and 2014. We speculate that this release policy may have lead to most of the clients sticking to an older version and preferring not to transition to a beta version.

Easymock. Clients of Easymock display a maximum, median, and average lag time of 607, 280, and 268 days, respectively. The first quartile and third quartile in the distribution are 120 and 393 days, respectively. Figure 4 shows the large number of projects that have a large amount of lag, relatively to the analyzed projects. Easymock is a small API, which had 12 releases, after the first, over 10 years (Figure 7).

The most recent version of Easymock is 3.3.1, released in January 2015. However, in our dataset we record use of neither that version nor the previous one (3.3.0). The latest used version is 3.2.0, released in July 2013, with 42 clients. Versions 3.0.0 and 3.1.0 are the most popular (211 and 190 clients) in our dataset, as seen in Figure 5. Version 2.5.2 and 2.4.0 also figure in the top three in terms of popularity, despite being released in 2009 and 2008.

4.1.3 Discussion

Our analysis lets emerge an interesting relation between the frequency of releases of an API and the behavior of its clients. By considering the data summarized in Figure 7, we can clearly distinguish two classes of APIs: 'frequent releaser' APIs (Guava, Hibernate and Spring) and 'non-frequent releaser' APIs (Guice and Easymock).

For all the APIs under consideration we see that there is a tendency for clients to hang back and to not upgrade to the most recent version. This is especially apparent in the case of the 'frequent releaser' APIs Guava and Spring: For these APIs, the older versions are far more popular and are still in use. In the case of Hibernate, we cannot get an accurate picture of the number of clients willing to transition because the version popularity statistics are quite fractured. This is a direct consequence of the large number of releases that take place every year.

For Guice and Easymock ('non-frequent releaser' APIs), we see that the latest version is not popular. However, for Guice the latest version is a beta and not an official release, thus we do not expect it to be high in popularity. In the case of Easymock, we see that the latest version (*i.e.*, 3.3.1) and the one preceding that (*i.e.*, 3.3.0) are not at all be used. In general, we do see that most clients of 'non-frequent releaser' APIs use a more recent version compared to clients of 'frequent releaser' APIs.

By looking at Figures 4 and 6, we also notice how the lag time of 'frequent releaser' APIs' clients is significantly lower than of 'non-frequent releaser' APIs' clients. This relation may have different causes: For example, 'non-frequent releaser' APIs' clients may be less used to update the libraries they use to more recent versions, they may also be less prone to change the parts of their code that call third-party libraries, or code that calls APIs that have non-frequent release policy may be more difficult to update. Testing these hypothesis goes beyond the scope of this paper, but with our dataset researchers can do so to a significant extent. Moreover, using fine-GRAPE, information about more APIs can be collected to verify whether the aforementioned relations hold with statistically significant samples.

4.2 **Case 2**: How much of an API is broadly used?

Many APIs are under constant development and maintenance. Some API producers do this to evolve features over time and improve the architecture of the API; others try to introduce new features that were previously not present. All in all, many changes take place in APIs over time [33]. Here we analyze which the features (methods and annotations) introduced by API developers are taken on board by the clients of these APIs.

This analysis is particularly important for developers or maintainers to know whether their efforts are useful and to decide to allocate more resources (e.g., testing, refactoring, performance improvement) in more used parts of their API, as resulting returns on investment may be greater. Moreover, API users may have more interest in reusing popular API features, as they are probably better tested through users [34].

4.2.1 Methodology

For each of the APIs, we have a list of features in the API_METHOD and API_CLASS tables [30]. We also have the usage data of all features per API that has been accumulated from the clients in the METHOD_INVOCATION and ANNOTATION tables. Based on this, we can mark features of the API have been used by clients. We can also count how many clients use a specific feature, thus classifying each feature as: (1) *hotspot*, in the top 15% of features in term of usage; (2) *neutral*, features that have been used once or more but not in the top 15% and (3) *coldspot*, if not used by any client. This is the same classification used by Thummalapenta and Xie [34] in a similar study (based on a different approach) on the usage of frameworks' features.

To see which used features were introduced early in an APIs lifetime, we can use the API_VERSION table to augment the date collected above with accurate version information per feature; then, for each of the used features, we see which version is the lowest wherein that feature has been introduced.

4.2.2 Results

The overall results for our analysis are summarized in Figures 8, 9, and 10. The first shows a percentage breakdown of usages of API features (left-hand side) and classes (right-hand side); the second and third report the probability distribution of the logarithm of the number of clients per API features, for 'non-frequent releaser' APIs and 'frequent releaser' APIs, respectively.

Generally, we see that the proportion of used features is never higher than 20% (Figure 8) and that the number of clients that use the features has a heavily right skewed distribution, which is slightly flattened by considering the logarithm (Figures 9 and 10). Moreover, we do not see a special behavior in this context of clients of 'non-frequent releaser' APIs vs. clients of 'frequent releaser' APIs.

In the following, we present the breakdown of the usage based on the definitions above.

Guava. Only 9.6% of the methods in Guava are ever used; in absolute numbers, out of 14,828 unique public methods over 18 Guava releases, only 1,425 methods are ever used. Looking at the used methods, we find that 214 methods can be classified as hotspots. The rest (1,211) are classified as neutral spots. The most popular method from the Guava API is `newArrayList` from the class `com.google.common.collect.Lists` class and it has 986 clients using it.
Guava provides 2,310 unique classes over 18 versions. We see that only 235 (10%) of these are ever used by at least client. Furthermore, only 35 of

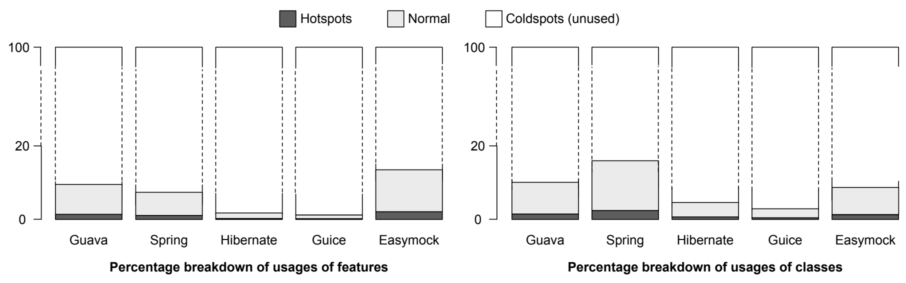

Fig. 8 Percentage breakdown of usage of features for each of the APIs

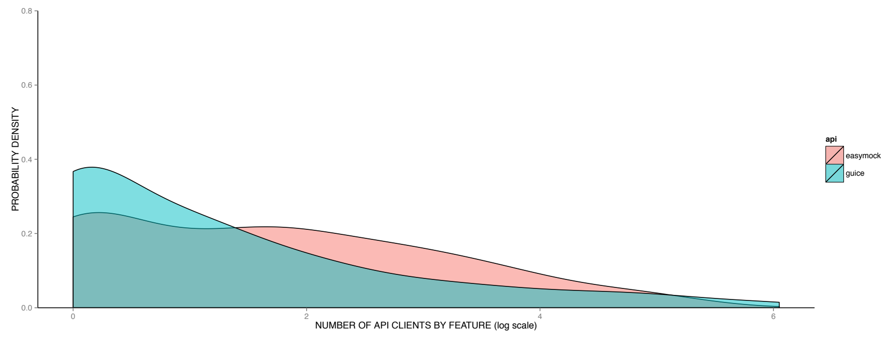

Fig. 9 Probability distribution of (log) number of clients per API features, by 'non-frequent releaser' APIs

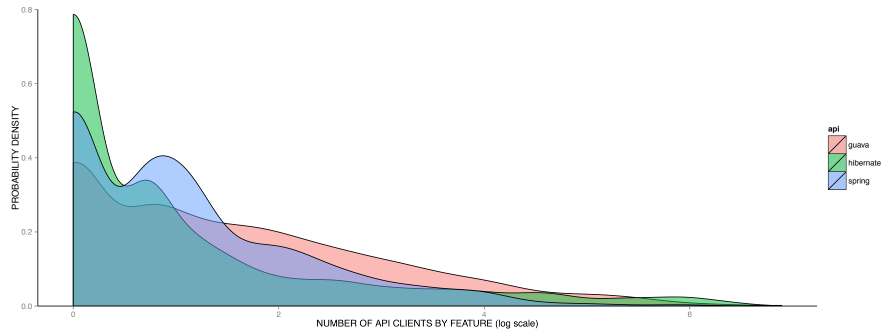

Fig. 10 Probability distribution of (log) number of clients per API features, by 'frequent releaser' APIs

these classes can be called hotspots in the API. A further 200 classes are classified as neutral. And we can classify a total of 2,075 classes as coldspots as they are never used. The most popular class is used 1,097 times and it is `com.google.common.collect.Lists`.

With Guava we see that 89.4% of the usages by clients of Guava relate to features that have been introduced in version 3 that was released in

April 2010. Following which 7% of the usages relate to features that were introduced in version 10 that was released in October 2011.

Spring. Out of the Spring core, context and test projects, we see that 7.4% of the features are used over the 40 releases of the API. A total of 840 features have been used out of the 11,315 features in the system. There are 126 features that can be classified as hotspots. Consequently, there are 714 features classified as neutral. The most popular feature is `addAttribute` from the class `org.springframework.ui.Model` and has been used 968 clients.

The Spring API provides a total of 1,999 unique classes. Out of these there are only 319 classes that are used by any of the clients of the Spring API. We can classify 48 of these classes as hotspot classes and the other 271 can be classified as neutral. We classify 1,680 classes as coldspots as they are never used. The most popular class has 2,417 clients and it is `org.springframework.stereotype.Controller`.

Looking deeper, we see that almost 96% of the features of Spring that are used by clients are those introduced in Spring version 3.0.0 that was released in December 2009.

Hibernate. From the Hibernate core and entitymanager projects we see that only 1.8% of the features are used. 756 out of the 41,948 unique public features provided over 77 versions of Hibernate have been used by clients in GitHub. Of these, 114 features that can be classified as hotspots and a further 642 features can be classified as neutral. The `getCurrentSession` method from the class `org.hibernate.SessionFactory` is the most popular feature, used by 618 clients.

Hibernate is made up of 5,376 unique classes. Out of these only 245 classes are used by clients. We can classify 37 of these classes as hotspots. The rest 208 classes are classified as neutral. We find that Hibernate has 5,131 coldspot classes. The most popular class is `org.hibernate.Session` with 917 clients using it.

In the case of Hibernate over 82% of the features that have been used were introduced in version 3.3.1 released in September 2008 and 17% of the features were introduced in 3.3.0.SP1 released in August 2008.

Guice. Out of the unique 11,625 features presented by Guice, we see that 1.2% (138) of the features are used by the clients of Guice. There are 21 features that are marked as being hotspots, 117 features marked as being neutral, and 11,487 classified as coldspots. The most popular provided by the Guice API is `createInjector` from class `com.google.inject.Guice` and is used by 424 clients.

The Guice API is made up of 2,037 unique classes that provide various features. Out of these only 61 classes are of any interest to clients of the API. We find that 9 of these classes can be classified as hotspots and the other 52 as neutral spots. This leaves a total of 1,976 classes as coldspots. The most popular class provided by Guice is `com.google.inject.Guice` and there are 424 clients that use it.

Close to 96% of the features of Guice that are popularly used by clients were introduced in its first iteration which was released on Maven central in May 2009.

Easymock. There are unique 623 features provided by Easymock, out of which 13.4% (84) are used by clients. This implies that 539 features provided by the API are never by used by any of the clients and are marked as coldspots. 13 features are marked as hotspots, while 71 features are marked as neutral. the The most popular feature is `getDeclaredMethod` from the class `org.easymock.internal.ReflectionUtils` and is used by 151 clients.
Easymock being a small API consists of only 102 unique classes. Out of these only 9 classes are used by clients. Only 1 can be classified as a hotspot class and the other 8 are classified as neutral spots. This leaves 93 classes as coldspots. The most popular class is `org.easymock.EasyMock` and is used by 205 clients.
We observe that 95% of the features that are used from the Easymock API were provided starting version 2.0 which was released in December 2005.

4.2.3 Discussion

We see that for Guava, Spring and Easymock, the percentage of usage of features hovers around the 10% mark. Easymock has the largest percentage of features that are used among the 5 APIs under consideration. This could be down to the fact that Easymock is also the smallest API among the 5. Previous studies such as that by Thummalapenta and Xie [34] have shown that over 15% of an API is used (hotspot) whereas the rest is not (coldspot). However, the APIs that they analyzed are very different to the ones that are here as they are all smaller APIs comparable to the size of Easymock, however none of them are of the size of the other APIs such as Guava and Spring. Also, their mining technique relied on code search engines and not on type resolved invocations.

In the case of Hibernate and Guice we see a much smaller percentage (1.8% and 1.2% respectively) of utilization of features. This is far lower than that of other APIs in this study. We speculate that due to the fact that the most popular features that are being used are also those that were introduced very early in the APIs life (version 3.3.1 in the case of Hibernate and version 1.0 in the case of Guice). These features could be classified as core features of the API. Despite API developers adding new features, there may be a tendency to not deviate from usage of these core features as these may have been the ones that made the API popular in the first place.

This analysis underlines a possibly unexpected low usage of API features in GitHub clients. Further studies, using our dataset, can be designed and carried out to determine which characteristics make certain feature more popular and guide developers to give the same characteristics to less popular features. Moreover, this popularity study can be used, for example, as a basis for developers to decide whether to separate more popular features of their APIs from the rest and provide them as a different, more supported package.

5 Limitations

Mining API usages on such a large scale and to this degree of accuracy is not a trivial task. We report consequent limitations to our dataset.

Master branch. To analyze as many projects as possible on GitHub, we needed to checkout the correct/latest version of the project on GitHub. GitHub uses Git as a versioning system which employs branches, thus making the task of automatically checking out the right version of the client challenging. We consider that the latest version of a given project would be labeled as the 'master' branch. Although this is a common convention [35], by restricting ourselves to only the master branch there is a non-negligible chance that some projects are dropped.

Inner and Internal classes. The method we use to collect all data about the features provided by the APIs, identifies all classes and methods in the API that are publicly accessible and can be used by a client of the API. These can include inner public classes and their respective methods. Or it can also consist of internal classes that are used by the features of the API itself but not meant for public consumption. The addition of these classes and methods to our dataset can inflate our count of classes and methods per API. If a more representative count is desired, it would be necessary to create a crawler for the API documentation of each API that is hosted online.

Maven (central) We target only projects based on a specific build automation tool on GitHub, *i.e.*, Maven. This results in data from just a subset of Java projects on GitHub and not all the projects. This may in particular affect the representativeness of the sample of projects. We try to mitigate this effect by considering one of the most popular building tools in Java: Maven. Moreover, the API release dates that we consider in our dataset correspond to the dates in which the API were published on Maven central, rather than the dates in which the API were official released on their websites. This could have an impact on the computed lag time.

GitHub. Even though GitHub is a very popular repository for open source software projects, this sole focus on GitHub leads to the oversight of projects that are on other open source platforms such as Sourceforge and Bitbucket. Moreover, no studies have yet ensured the representativeness of GitHub projects with respect to industrial ones; on the contrary, as also recently documented by Kalliamvakou *et al.* [36], projects on GitHub are all open source and many of the projects may be developed by hobbyists. This may result in developers not conforming to standard professional software maintenance practices and, in turn, to abnormal API update behavior.

Hibernate. In the case of Hibernate, we could not retrieve data for version 2 or 1. This is due to the fact that neither of these versions were ever released on the maven central repository. This may have an impact on both of the case studies as the usage results can get skewed towards version 3 of the API.

6 Conclusion

We have presented our approach to mine API usage from OSS platforms. Using fine-GRAPE we created a rich and detailed dataset that allows researchers and developers alike to get insights into trends related to APIs. A conscious attempt has been made to harvest all the API usage accurately. We mined A total of 20,263 projects and accumulated a grand total of 1,482,726 method invocations and 85,098 annotation usages related to 5 APIs.

We also presented two case studies that were performed on this dataset without using external scripts or data sources. The first case study analyzes how much clients migrate to new versions of APIs. Besides confirming that clients tend not to update their APIs, this study highlights an interesting distinction between clients of APIs that frequently release new version and those that do not. For the former, the lag time is significantly lower. Although our sample of APIs is not large enough to allow generalization, we deem this finding to deserve further research as it could potentially help API developers decide which release policy to adopt, depending on their objectives. In the second case study, we analyze which proportion of the features of the considered API is used by the clients. Results show that a considerably small portion of an API is actually used by clients in practice. We suspect that this may be a result of clients only using features that an API was originally known for as opposed to migrating to new features that have been provided by the API.

Overall, it is our hope that our large database of API method invocations and annotation usages will trigger even more precise and reproducible work in relation to software APIs.

7 Resources

The sample version of fine-GRAPE, titled fine-GRAPE-light can be found at: `https://github.com/theSorcerers/fine-GRAPE-light`. The dataset can be found on our Figshare repository at: `https://figshare.com/articles/API_Usage_Databases/1320591`.

References

1. R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.
2. Y. M. Mileva, V. Dallmeier, and A. Zeller, “Mining api popularity,” in *Testing–Practice and Research Techniques*, pp. 173–180, Springer, 2010.
3. D. Dig and R. Johnson, “How do apis evolve? a story of refactoring,” *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

4. D. Mandelin and D. Kimelman, "Jungloid mining : Helping to navigate the api jungle," *Synthesis*, pp. 48–61, 2006.
5. T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 3rd Working Conference on Mining Software Repositories*, MSR 2006, pp. 54–57, 2006.
6. H. Zhong, T. Xie, L. Zhang, J. Pei, and 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, Springer, 2009.
7. E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, "Cohesive and isolated development with branches," in *Fundamental Approaches to Software Engineering*, pp. 316–331, Springer, 2012.
8. M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.
9. L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the IDE," in *Proceedings of the 35th International Conference on Software Engineering, Tool Demo Track*, ICSE 2013, pp. 1295–1298, IEEE CS Press, 2013.
10. A. A. Sawant and A. Bacchelli, "A dataset for api usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR 2015, pp. 506–509, IEEE Press, 2015.
11. "Tiobe index." http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. accessed on 16 October 2015.
12. V. Saini, H. Sajnani, J. Ossher, and C. V. Lopes, "A dataset for maven artifacts and bug patterns found in them," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 416–419, ACM, 2014.
13. B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 313–328, 2008.
14. S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," in *Proceedings of the 2nd International Symposium Symposium on Constructing Software Engineering Tools*, CoSET 2000, 2000.
15. A. Bacchelli, P. Ciancarini, and D. Rossi, "On the effectiveness of manual and automatic unit test generation," in *The Third International Conference on Software Engineering Advances*, ICSEA 2008, pp. 252–257, IEEE, 2008.
16. L. Vogel, "Eclipse JDT-Abstract Syntax Tree (AST) and the java model-tutorial." http://www.vogella.com/tutorials/EclipseJDT/article.html.
17. E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: a code manipulation tool to implement adaptable systems," *Adaptable and extensible component systems*, vol. 30, 2002.
18. R. Agrawal, R. Srikant, *et al.*, "Fast algorithms for mining association rules," in *Proceedings of the 20th international conference on Very Large Data bases*, vol. 1215 of *VLDB 1994*, pp. 487–499, 1994.
19. J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR 2013, pp. 319–328, IEEE Press, 2013.
20. B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 296–305, 2005.
21. A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 07 of *ESEC/FSE 2007*, pp. 35–44, 2007.
22. S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE 2009, pp. 283–294, IEEE Computer Society, 2009.
23. Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

24. A. Michail, "Data mining library reuse patterns in user-selected applications," in *Proceedings of 14th IEEE International Conference on Automated Software Engineering, 1999.*, ASE 1999, pp. 24–33, IEEE, 1999.
25. S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 681–682, ACM, 2006.
26. L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?," in *Proceedings of the 37th International conference on Software engineering*, MSR 2015, p. in press, 2015.
27. G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 384–387, ACM.
28. O. Primat, "Github's 10,000 most popular java projects – here are the top libraries they use." http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/, nov 2013.
29. B. Momjian, *PostgreSQL: introduction and concepts*, vol. 192. Addison-Wesley New York, 2001.
30. A. Sawant and A. Bacchelli, "Api usage databases." `http://dx.doi.org/10.6084/m9.figshare.1320591`, 2015.
31. R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of the 26th ACM Symposium on Applied Computing*, SAC 2011, pp. 1317–1324, 2011.
32. T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proceedings of the 29th International Conference on Software Maintenance*, ICSM 2013, pp. 70–79, IEEE, 2013.
33. J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th International conference on Software engineering*, ICSE 2005, pp. 274–283, ACM, 2005.
34. S. Thummalapenta and 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, 2008.
35. S. Chacon, *Pro git.* Apress, 2009.
36. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 92–101, ACM, 2014.