Large-scale, AST-based API-usage analysis of open-source Java projects

Ralf Lämmel[1,2] and Ekaterina Pek[2] and Jürgen Starek[1]

[1] Software Languages Team, Universität Koblenz-Landau, Germany

[2] ADAPT Lab, Universität Koblenz-Landau, Germany

Abstract

Research on API migration and language conversion can be informed by empirical data about API usage. For instance, such data may help with designing and defending mapping rules for API migration in terms of relevance and applicability. We describe an approach to large-scale API-usage analysis of open-source Java projects, which we also instantiate for the SourceForge open-source repository in a certain way. Our approach covers checkout, building, tagging with metadata, fact extraction, analysis, and synthesis with a large degree of automation. Fact extraction relies on resolved (type-checked) ASTs. We describe a few examples of API-usage analysis; they are motivated by API migration. These examples are concerned with analysing API footprint (such as the numbers of distinct APIs used in a project), API coverage (such as the percentage of methods of an API used in a corpus), and framework-like vs. class-library-like usage.

1. Introduction

The broader context of the reported research is *API*[1] *migration* [3, 24, 36, 2] (but also language conversion [17, 29, 22] to the extent that it involves API migration). Given a programming domain, and given a couple of different APIs for that domain, it can be challenging to devise transformations or wrappers for migration from one API to the other. The APIs may differ with regard to types, methods, contracts, and protocols so that actual API migration efforts must compromise with regard to automation and correctness [3, 2].

Several researchers, including ourselves, are working towards general techniques for reliable and scalable API migration. Because of the complexity of transformations and wrappers for migration as well as the difficulty of proving them correct, it is also advisable to leverage *diverse knowledge about actual API usage*.

[1]In this paper, we use the term API to refer both to a public programming interface and its actual implementation as a software library for reuse.

In the present paper, we describe an approach to large-scale *API-usage analysis* for the analysis of open-source Java projects. Our approach covers checkout, building, tagging with metadata, fact extraction, analysis, and synthesis with a large degree of automation. We describe a few examples of API-usage analysis; they are motivated by API migration. Overall, API-usage analysis helps with designing and defending mapping rules for API migration in terms of relevance and applicability.

While API migration remains the primary motivation for our efforts on API-usage analysis, we must say that the work reported in this paper has meanwhile grown into an effort of its own right: this kind of supported API-usage analysis also caters for *understanding simple structural properties of Java software* (such as the number of APIs used in open-source projects) and fundamental API characteristics (such as class library-like vs. framework-like API usage). In this sense, it supports potential empirical work on language usage in the line of Baxter et al.'s "*Understanding the shape of Java software*" [4].

Contributions of the paper

- We work out a few examples of API-usage analysis: (i) API-footprint analysis for projects; (ii) API-coverage analysis for the corpus; (iii) analysis of framework-like API usage. We discuss how such analyses inform API-migration efforts.
- We describe a process for obtaining a large corpus of built Java projects and involved Java APIs from an open-source repository in a scalable manner. Fact extraction uses precise, resolved (type-checked) ASTs.

Practical results and online access

In the *Software Languages Lab* at Koblenz, we are working on different aspects of API-usage analysis and several implementations. In this paper, we report on an infrastructure that we applied to SourceForge. All empirical data in this paper is based on this implementation. All figures and tables from the present paper as well as additional data material have been made available online.[2]

Road-map of the paper

§2 provides an overview of the approach to API-usage analysis. §3 instantiates the approach for the SourceForge open-source repository in a certain way. §4 describes some examples of API-usage analysis, and exercises them for the SourceForge-based corpus. §5 discusses major threats to validity. §6 discusses related work. §7 concludes the paper.

[2]Paper's web site incl. all support material: `http://softlang.uni-koblenz.de/sourceforge/`

2. Overview of the approach

API-usage analysis relies on methods and techniques that are commonly used in reverse engineering and program understanding. In particular, we need to set up a corpus of software projects to be used for data mining; we also need to provide a fact-extraction machinery to build a database of program facts. Additionally, we need to add metadata about APIs—as we operate in *the domain of the programming domains and their APIs*. Any specific form of API-usage analysis is then to be implemented through queries on the fact base (database) that is obtained in this manner.

2.1 Setting up the corpus

The objectives of a specific effort on API-usage analysis should obviously affect the provision of a corpus. In our work, so far, we have been mainly interested in extracting *evidence (facts) about API usage* from as many projects as possible. While corpora of dozens of well chosen projects (such as the one of [4]) are well suited for many data mining purposes (e.g., for the analysis of simple structural properties (metrics) of Java software), they are potentially limited with regard to API features that they exercise. For this reason, we are interested in large-scale efforts where API-usage data is systematically mined from open-source repositories. In principle, one could still include specific 'well-known' projects manually into the resulting corpus, if this is desired.

2.2 Provision of a fact extractor

We need to be able to reliably link facts of API usage to the actual APIs and their types, methods, etc. Hence, fact extraction must be syntax- and type-aware. (For instance, the receiver type in a method call must be known in order to associated calls with API methods.) We use fact extraction based on *resolved ASTs*. However, this choice basically implies that we only consider built ('buildable') projects, which may result in a bias. Therefore, we also incorporate an additional token-based (as opposed to AST-based) fact extractor into the architecture so that some more basic analyses are still feasible.

2.3 Addition of API metadata

Along with building many projects, one encounters many APIs. In the case of the Java platform, there are Core Java APIs and third-party APIs. Based on package and types names, one can identify these APIs, and assign names. For instance, certain types in the package `java.util` account for the 'Core Java API for collections'. One can also associate programming domains with APIs: GUI, XML, Testing, etc.

Third-party APIs reveal themselves in the process in two principle ways. First, projects may fail to build with 'class not found' errors, which are to be manually resolved by supplying (and tagging) the corresponding APIs through web search and download. Second, the corpus can also be analyzed for cross-project reuse. That is, one can automatically identify API candidates by querying for packages whose methods were called (and potentially declared and compiled) in at least two projects.

3. A study of SourceForge

We will now describe the instantiation of the above approach for the study of the present paper.

3.1 Project selection

Based on available metadata for all SourceForge projects, we determined the list of potential Java projects. In the present study, as a concession to scalability and simplicity of our implementation, we only downloaded projects with a SourceForge-hosted SVN source repository, and we only considered Java projects with Ant-based build management. (We discuss all threats to validity in §5.) We used a homegrown architecture for parallel checkout and building. The selected SourceForge projects were fetched in October 2008.

3.2 Resolution of missing API packages

Obviously, SourceForge projects may fail to build for diverse reasons: wrong platform, missing configuration, missing JARs, etc. We wanted to bring up the number of buildable projects with little effort. We addressed one particular reason: 'class not found' compilation errors. To this end, our build machinery compiles a summary of unresolved class names (package names) for a full build sweep over all projects. This summary *ranks* package names by frequency of causing 'class not found' errors so that the manual effort for supplying missing APIs can be prioritized accordingly. We searched the web for API JARs using unresolved package names as search strings. We downloaded these JARs, added them to the build path, and ran the automated build again. We repeated this step until the top of the list of missing packages would contain packages referenced only by 1-2 projects. This process provided us with 1,476 built projects, where approx. 15 % of these projects were made buildable by our resolution efforts. In the end, we were able to build 90.05 % of all downloaded SourceForge projects that satisfied our initial criteria (Java, SVN, ANT). The process resulted in an API pool of 69 non-Core Java APIs.

3.3 Fact extraction

We carried out resolved AST-based fact extraction by means of a compiler plug-in for `javac`, which is activated transparently as projects are built. In this study, we extracted facts about method declarations, method calls, and subtype relationships. **In this paper, we interpret the term *method* to include instance methods, static methods and constructors.** All facts were stored in a relational database using a trivial relational schema.

We only used AST-based facts from projects with successful builds. For all projects, we performed token-based fact extraction to count NCLOC (non-comment lines of code) for Java sources and to determine all package names from imports. The importing facts give an indication of, for example, the APIs that are used in projects that do not build.

3.4 Reference projects

We made an effort to identify a control group of (buildable) reference projects that could be said to represent well thought-out, actively developed and usable software. Such a control group allows us to check all trends of the analyses for the full corpus by comparison with the more trusted reference projects. Several charts in this paper show all projects vs. reference projects for comparison. We automatically identified the reference projects by means of SourceForge's metadata about maturity status of the project, number of commits, and dates of first and last commits. That is, we selected projects that rate themselves as 'mature' or 'stable', have a repository created more than two years ago, and have more than 100 commits to the repository. This selection resulted in 60 reference projects out of all the 1,476 built projects.

3.5 Size metrics for the corpus

Numbers of projects and their NCLOC sizes, and other metrics are summarized in Table 1 and Table 2. We use the metric *MC* for the number of method calls.

Metric	Value
Projects	6,286
Source files	2,121,688
LOC	377,640,164
NCLOC	264,536,500
Import statements	14,335,066

Table 1. Summary of token-based analysis (with all automatically identified Java/SVN projects on SourceForge).

Metric	Value
Projects with attempted builds	1,639
Built projects	1,476
Packages	46,145
Classes	198,948
Methods	1,397,099
Method calls	8,163,083

Table 2. Summary of AST-based analysis.

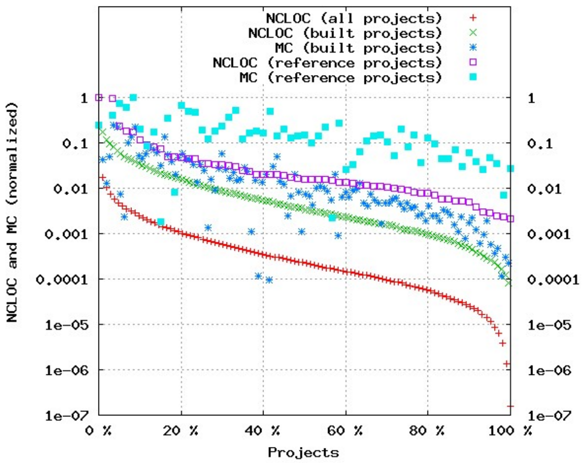


Figure 1. Size metrics (NCLOC, MC) for built and unbuilt projects (thinned out). The projects are ordered by the values for the NCLOC metric.

Figure 1 presents the distribution of size metrics (NCLOC, MC) for the corpus (y-axis is normalized w.r.t. the maximum of the metric in each group). As one can see, both metrics correlate reasonably. The maximum of NCLOC in the whole corpus (incl. unbuilt projects) is 25,515,312, the maximum of NCLOC among built projects is 1,985,977, which implies a factor 12.85 difference. Hence, we are missing the biggest projects currently. The maximum of MC among built projects is 228,242.

API	Domain	Core	# Projects	# Calls	# Distinct methods called
Java Collections	Collections	yes	1374	392639	406
AWT	GUI	yes	754	360903	1607
Swing	GUI	yes	716	581363	3369
Reflection	Other	yes	560	15611	154
Core XML	XML	yes	413	90415	537
DOM	XML	yes	324	52593	180
SAX	XML	no	310	13725	156
log4j	Logging	no	254	43533	187
JUnit	Testing	no	233	71481	1011
Comm.Logging	Logging	no	151	21996	88

Table 3. Top 10 of the known APIs (sorted by the number of projects using an API).

3.6 Provision of API metadata

Both Core Java APIs and manually downloaded JARs were processed by us to assign metadata: name of the API, the name of a programming domain, one or more package prefixes, and potentially a white-list of API types. Table 3 lists the top 10 of all the 77 manually tagged APIs together with some metadata and metrics. We used a special reflection-based fact extractor for the visible types and members of the API JARs. (Alternatively, one could also attempt to leverage API documentation, but such documentation may not be available for some of the APIs, and it would take extra effort to establish consistency between JARs and documentation.) These facts are also stored in the database, and they are leveraged by some forms of API-usage analysis.

4. Examples of API-usage analysis

We will introduce a few examples of API-usage analysis. In each case, we will provide a motivation related to API migration and language conversion—before we apply the analysis to our SourceForge corpus.

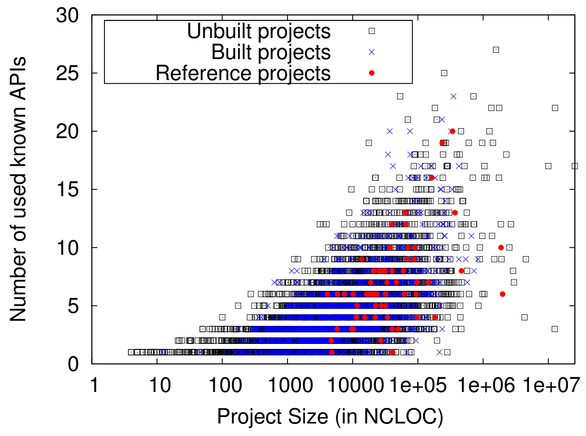
Admittedly, the statistical analysis of a corpus does not directly help with any specific migration project. However, the reported analyses as such are meaningful for single projects, too (perhaps subject to refinements). For instance, we will discuss API coverage below, and such information, when obtained for a specific project, directly helps prioritizing migration efforts. In this paper, which presents early research, we take a statistical view on the corpus to indicate the de-facto range for some of the measures of interest.

4.1 API footprint per project

We begin with a very simple, nevertheless informative API-usage analysis for the *footprint of API usage*. There are different dimensions of footprint. Below, we consider the numbers of used APIs and used (distinct) API methods. In the online appendix, we also consider the ratio of API calls to all calls. In extension of these numbers, we could also be interested in the 'reference projects × API pool' matrix (showing for each project the combination of APIs that it uses).

The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint-like data serves as a proxy for the API dependence or platform dependence of a project. In [16], we mention such API dependence as a form of *software asbestos*. In the following, we simply count the number of APIs used in a project as a proxy for the difficulty of API migration. Ultimately, a more refined analysis is needed such that *specific* (known to be difficult) API combinations are counted, and attention is payed to the status of whether these API combos are really exercised in one program scope or only separately.

In this context, we need to define what constitutes usage of an API. One option would be to count each method call with an API's type as static receiver type (in the case of an instance call), or as the hosting scope (in the case of a static call), or as the constructed type (in the case of a constructor call). Another option is to count any sort of reference to an API's types (including the aforementioned positions of API types in method calls, but counting additionally local variable declarations or argument positions of method declarations and method calls). Yet another option is to consider simply imports in a project. The latter option has the advantage that we can measure such imports very easily—even for unbuilt projects. Indeed, the following numbers were obtained by counting imports that were obtained with the token-based fact extractor.

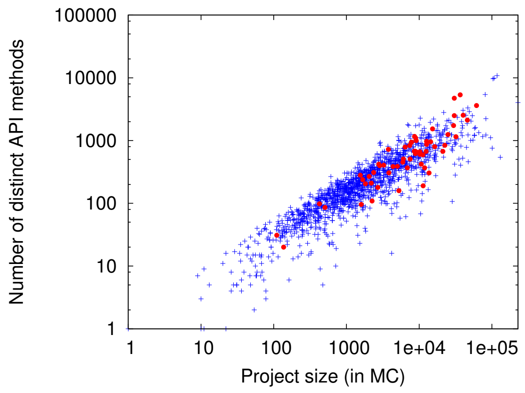


Projects	Min	1st Q	Median	Mean	3rd Q	Max
Unbuilt	1	2	4	4.409	6	27
Built	1	3	4	4.692	6	23
Reference	1	4	6	6.937	8	20

Figure 2. Numbers of known APIs used in the projects; reference projects are plotted on top of built projects which in turn are plotted on top of unbuilt projects.

Figure 2 shows the number of known APIs (y-axis) that are used in the projects ordered by NCLOC-based project size (x-axis). Unbuilt, built, and reference projects are distinguished. The listed maxima and quartiles give a sense of the API footprint in projects in the wild. The set of unbuilt projects exercises a higher maximum of used APIs than the set of built projects—potentially because of a correlation between the complexity of projects in terms of the number of involved APIs and the difficulty to build those projects.

We also need to clarify how to measure usage of API methods. That is, how to precisely distinguish distinct methods so that counting uses is well defined. Particularly, in the case of instance method calls, the situation is complicated due to inheritance, overriding, and polymorphism. As a starting point, we may distinguish methods by possible receiver type—no matter whether the method is overridden or inherited at a given subtype. Then, a method call is counted towards the *static receiver type* in a call. Additionally, we may also count the call towards subtypes (subject to a polymorphism-based argument: the runtime receiver type may be a subtype) and supertypes (subject to an inheritance-based argument: the inherited implementation may be used, if not inherited). Such inclusion could also be made more precise by a global program analysis.



Projects	Min	1st Q	Median	Mean	3rd Q	Max
All	1	94	199.5	370.7	423	10850
Reference	20	305.8	611	866.2	948.8	5351

Figure 3. Numbers of distinct API methods used in the projects (without distinguishing APIs).

Figure 3 shows the numbers of distinct API methods used in the built projects of the corpus; reference projects are highlighted. Methods on sub- and supertypes of static receiver types were not included. For simplification, we also considered overloaded methods as basically one method.

There is a trend of increasing API footprint with project size. Both axes are logarithmic, but project size grows more quickly than the count of distinct API methods. Most projects, even most of the largest ones, use less than 1,000 distinct API methods. As the table with maxima and quartiles shows, there are a few projects with exceptionally high counts. We have verified for these projects that they essentially implement or test large frameworks (such as `ofbiz.apache.org`). That is, these outliers embody large numbers of 'self-calls' for a large number of API methods.

4.2 API coverage by the corpus

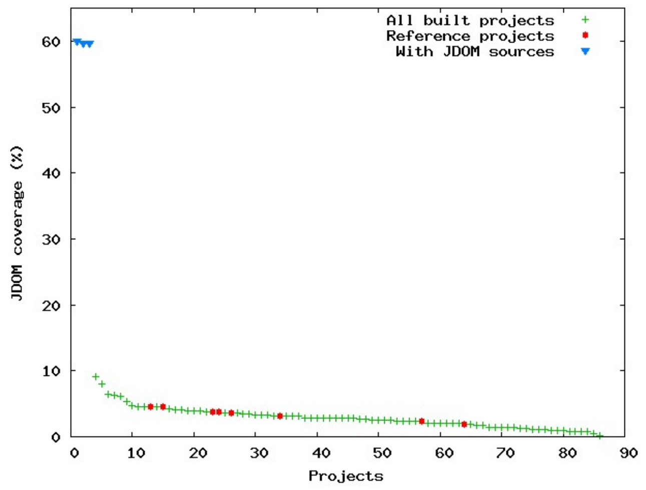
An important form of API-usage analysis concerns API coverage; see, for example, the discussion of coverage in the API migration project of [3]. That is, coverage information is helpful in API migration as means to prioritize efforts, and to leave out mapping rules for obscure parts of the API. Coverage information is also helpful in improving API usability [13, 12].

As it is the case with other forms of API-usage analysis, API coverage may be considered for either a specific project, or, cumulatively, for all projects in a corpus. For instance, for any given

API, we may be interested in the API types (classes and interfaces) that are exercised in projects by any means: extension, implementation, variable declaration, all kinds of method calls, and other, less obvious idioms (e.g., instance-of tests). At a more fine-grained level, we may be interested in the exercised members for each given API type. Hence, qualitative measurements focus on types and members that are exercised at all, while quantitative measurements rank usage by the number of occurrences of a type or a member or other weights.

Assuming a representative corpus, further assuming appropriate criteria for detecting coverage, we may start with the naive expectation that a good API should be covered more or less by the corpus. Otherwise, the API would contain de-facto unnecessary types and methods—which is clearly not in the interest of the API designer. However, it should not come as a surprise that, in practice, APIs are not covered very well—certainly not by single meaningful projects [3], but—as our results show—not even by a substantial corpus; see below.

We have actually tried to determine two simple coverage metrics for all 77 known APIs: i) a percentage-based metrics for the *types*; ii) another percentage-based metrics for all *methods*. However, we do not feel comfortable presenting a chart of those metrics for all known APIs here. Unless considerable effort is spent on each API, such a chart may be disputable. The challenge lies in the large number of projects and APIs, the different characteristics of the APIs (e.g., in terms of their use of subtyping), aspects of cloning, and yet other problems. Some of the issues will be demonstrated for selected APIs below.



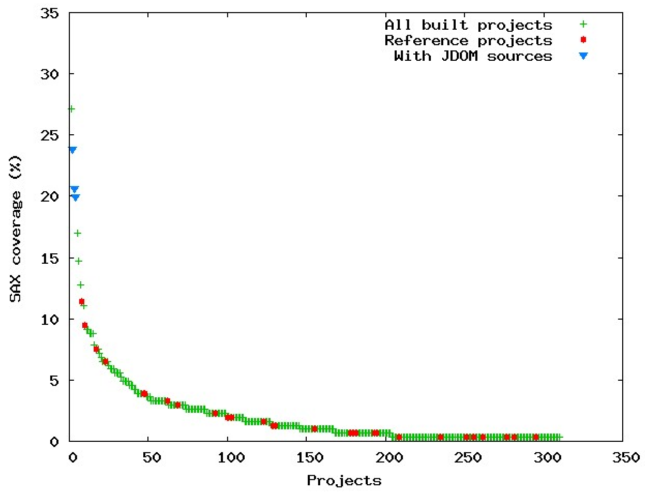
Projects	Min	1st Q	Median	Mean	3rd Q	Max
All built	0.1629	1.954	2.769	4.861	3.746	59.93
Reference	1.954	2.891	3.664	3.441	3.95	4.56

Figure 4. Usage of JDOM's distinct methods.

Let us investigate coverage for *specific* APIs. As our first target, we pick JDOM—a DOM-like (i.e., tree-based, in-memory) API for XML processing. We know that JDOM is a 'true library' as opposed to a framework. Regular client code should simply construct objects of the JDOM classes and invoke methods directly. We mention these characteristics because library-like APIs may be expected to show higher API coverage than framework-like APIs—if we measure coverage in terms of *called* methods, as we do here. In this paper, in the case of a call to an instance method, we only count the method on the immediate static receiver type as covered. We have checked that the inclusion of super- and subtypes, as discussed earlier, does not change much the charts described below.

Initially, we measured cumulative coverage for the methods of the JDOM API to be 68.89 %. We decided to study the contribution of the different projects. There are 86 projects with JDOM usage among the built projects of the corpus. Figure 4 shows the percentage-based coverage metrics for the methods of the JDOM API for those JDOM-using projects. The table with maxima and quartiles gives a good indication of the relatively low usage of the JDOM API.

Obviously, 3 projects stand out with their coverage. We found that these projects should not be counted towards cumulative coverage because these projects contain JDOM clones in source form. That is, it the API calls within the API's implementation imply artificial coverage for more than half of all JDOM methods. Without those outliers, the cumulative coverage is considerably lower, only 24.10 %.



Projects	Min	1st Q	Median	Mean	3rd Q	Max
All built	0.3268	0.3268	0.9804	2.22	2.614	27.12
Reference	0.3268	0.3268	1.144	2.369	3.023	11.44

Figure 5. Usage of SAX' distinct methods.

Let us consider another specific API. We pick SAX—a push-based XML parsing API. The push-based characteristics imply that client code typically extends 'handler' classes or implements handler interfaces with handler methods such as `startElement` and `endElement`—to which XML-parsing events are pushed. As a result, one should be prepared to find relatively low API coverage—*if* we measure coverage in terms of called methods.

We measured cumulative coverage for the methods of the SAX API to be 50.98 %. This relatively high coverage was suprising. There are 310 projects with SAX usage among the built projects of the corpus. Figure 5 shows the percentage-based coverage metrics for the methods of the SAX API for those SAX-using projects. We found that three of the projects with the highest coverage were in fact the previously discussed projects with JDOM clones in source

API	# Projects			# Methods		# Distinct methods		# Derived types		# API types	
	impl.	**ext.**	**any**	**impl.**	**over.**	**impl.**	**over.**	**interf.**	**classes**	**interf.**	**classes**
Swing	173	381	391	2512	11150	305	645	443	1859	39	92
AWT	194	75	225	4201	756	593	176	651	120	31	24
Java Collections	120	0	120	986	0	16	0	208	0	3	0
SAX	28	21	42	428	90	85	21	37	29	12	3
JUnit	3	38	40	4	344	4	19	3	46	2	2
Core XML	11	5	14	89	13	17	4	14	5	9	3
SWT	5	8	10	37	86	4	13	25	11	3	3
log4j	1	8	8	25	87	7	9	2	9	2	3
Reflection	7	0	7	10	0	1	0	7	0	1	0
JMF	4	2	6	8	6	6	3	4	3	3	3

Table 4. Top 10 of the APIs with framework-like usage (sorted by the sum of numbers of API-interface implementations and API-class extensions; see the first 3 columns).

form. Closer inspection revealed that the JDOM API implements, for example, a bridge from in-memory XML trees to SAX events, and hence, it pushes itself as opposed to regular SAX-based functionality that is pushed. This is an unexpected but correct use of the SAX API within the DOM API which brings up coverage of the SAX API. Even if we removed those 3 projects, the cumulative coverage only drops down a little to 49.34 %.

We also found other reasonable reasons for relatively high coverage. There are projects that use inheritance and composition to define new handlers (e.g., `http://corpusreader.sourceforge.net/`) so that API methods may get called through 'super' or delegation. As the quartiles show in the figure, most projects use a small percentage of the SAX API. Most of the relevant methods are concerned with processing the parameters of the handlers. Many of the SAX projects use (largely) predefined handlers, e.g., for validation—thereby implying a very low coverage.

4.3 Framework-like API usage

Finally, we introduce an analysis for framework-like API usage: What API types are typically implemented and extended, if any? Also, can we determine whether a given API is presumably more framework-like (less class library-like) than another API? What are the parts of an API that account for framework-like usage? In the context of API migration, proper framework-like usage is very challenging because it implies an 'inversion of control' in applications, which is very hard to come by with mapping rules [2].

More specifically, by *framework-like usage*, we mean any sort of idiomatic evidence for refining, configuring, and implementing API types within client code. In particular, we may measure i) extensions (i.e., client classes that extend API classes); ii) implementations (i.e., client classes that implement API interfaces); iii) overrides (i.e., client classes that subclass API classes and override inherited API methods). Obviously, there are facets that may be harder to identify generically. For instance, if a framework would involve plug-in or configuration support based on regular method calls, then such framework-like usage would be missed by i)—iii). There is again a way of defining framework-like usage in a cumulative way—very similar to coverage analysis. That is, for a given API, we may determine the set of API types that are ever involved in framework-like usage.

In reality, many APIs allow for both—class library-like and framework-like usage. For instance, the Core Java API DOM is essentially interface-based so that new providers can be implemented, but there are default implementations for the important use case of DOM as an in-memory XML API. In contrast, there are other APIs that are subject to framework-like usage more inevitably. For instance, the Core Java API Swing is often used in a way that the `JPanel` class is overridden.

In our corpus, 35 out of all 77 known APIs exercise a measurable facet of framework-like usage. Table 4 lists the top 10 of these APIs. In the table, we also show the numbers of API methods that are implemented or overridden throughout the corpus: we show both absolute numbers of implementations/overrides and the numbers of distinct methods. Further, we show the number of derived types in client code, and the number of API types ever exercised through framework-like usage.

Surprisingly, the table shows that there are only 7 APIs that are used in 10 or more projects in a framework-like usage manner. This clearly suggests that our corpus (of built projects) and our selection of APIs is trivial in terms of framework-like usage. Many APIs do not show up at all in the table—despite heavy usage in the corpus. For instance, DOM-like APIs like JDOM or XOM do not show up at all, which means that they are only used in a class library-like manner. The DOM API itself is subject to API-interface implementations in a number of projects. In the online appendix of the paper, we also break down the numbers of the table to show the types that are commonly involved in framework-like usage: just a hand full of GUI, XML and collection types account for almost all the framework-like usage in the corpus.

5. Threats to validity

5.1 Internal validity

The selected group may fail to be representative for the whole population. If we define the whole population to consist of all open-source Java projects, then we restricted ourselves in at least three ways, namely, the source of picking the projects (SourceForge only), the version control system (SVN only) and the build tool (Apache Ant only). These choices were concessions to the primary goal of the present research milestone: to prove the feasibility of large-scale, automated, resolved AST-based API-usage analysis.

If we were picking open-source projects manually and making sure these projects build, then we could be sure not to miss 'any projects of interest'. However, we are not exactly sure how to assemble a suitable hand-picked corpus; we are also not convinced that the internal validity issues of such a manual approach are less severe than ours (if we assume to further improve inclusion of projects in the future).

5.2 External validity

Interaction of setting and treatment: SourceForge is arguably not an appropriate source of representative contemporary software—open-source or not. (Hence, this threat is connected with the aforementioned threat.) A future, comparative study may determine how, for example, API coverage varies across different open-source repositories or 'well-known' hand-picked corpora.

5.3 Construct validity

Mono-method bias: We rely mostly on the results gained from the AST-based fact extractor gathered through a default build. This means, for example, that we miss sources that were not build in this manner. We also miss projects whose builds fail (and hence fact extraction is not completed). Further, there is the possibility of inconsistencies in fact extraction and performing the queries for API analysis. In fact, we fixed many such issues throughout the development.

6. Related work

We identify several categories of related work.

6.1 Analysis of open-source repositories

Our work relates to other work on analyzing open-source repositories. For instance, there has been work on the analysis of *download data*, *evolution data*, *metadata* (e.g., development status), or *user data* (e.g., the number of active developers), and simple *metrics* (e.g., LOC) for open source projects (e.g., on SourceForge) [11, 20, 10, 34]. In this context, we also refer to the project FLOSSmole `http://flossmole.org/` and the project FOSSology [7]. Usually, such large-scale efforts do not involve AST-based source-code analysis. For instance, in [9], the adoption of design patterns is studied in open-source software, but documentation (commit messages) as opposed to parse trees are analyzed. In [19], a very large-scale code clone analysis is performed.

In this context, our contribution is a scalable process for actually obtaining a large corpus of open-source projects that are directly amenable to AST-based analysis, which opens up new applications and improved precision of API-usage analysis, as we have shown in Sec. 4. There exist various code-search engines (see [27, 30, 31, 8] for discussions). Again, our approach distinctly leverages resolved ASTs.

6.2 Properties of Java software

There are several forms of related work that aim to analyse relatively simple properties of Java software (in the wild). In [4], key structural attributes of a hand-picked corpus of well-known programs are measured, and a careful analysis of the distribution for these properties is provided. We also refer to [35] for a related study on power-law distributions for class relationships. In [5], Java bytecode programs are studied empirically to determine simple counts (number of methods per class, number of bytecode instructions per method, etc.), structural metrics such as the complexity of control-flow and inheritance graphs. In [6], micro patterns (which are patterns at a lower level than design patterns) are analyzed. In [25], Java bytecode sequences are analyzed for most commonly used bytecode pairs.

These efforts are concerned with language aspects other than API usage. None of these efforts leverage large-scale corpora comparable to the one of the present paper. None of these efforts deal with the methodological challenges of obtaining such a large corpus that is amenable to the analysis of resolved ASTs.

6.3 API-usage analysis

There are other efforts that we would like to collectively label with *API-usage analysis*. The kinds of analysis in such related work are different from those considered in the present paper.

One important direction is concerned with the analysis of *API usage patterns*. In [23], data mining is used to determine API reuse patterns, and more specifically classes and methods that are typically used in *combination*. We also refer to [21] for a related approach. In [14], data mining is used, too, to determine frequently appearing ordered sets of function-call usages, taking into account their proximal control constructs (e.g., if-statements). In [37], data mining is used, too, to compute lists of frequent API usage patterns based on results returned from code-search engines; the underlying code does not need to be compilable. In [18], a machine learning-based approach is used to infer protocol specifications for API; see [33] for a related approach. In [1], inter-procedural, control-flow-sensitive static traces for C programs are used to mine API-usage patterns as partial orders from source code.

Perhaps the most closely related work is [31]: API hotspots and coldspots (say, frequently or rarely used classes and methods) are determined. This work is again based on the analysis of data obtained from a code-search engine. Such frequency analysis can be compared to our efforts on coverage analysis (c.f., Sect. 4.2)—except that we are using resolved ASTs. Also, we are specifically interested in the large-scale, cumulative coverage. Further, we are interested in interpreting analysis results in terms of API characteristics, and with a view on API migration.

In [32], library reuse is studied at a level of shared objects in the operating systems Sun OS and Mac OS X. One of the observations is that reuse seems to be low in the sense of Zipf's law. Thus the most frequent function will be referenced approximately twice as often as the second most frequent function, which occurs twice as often as the fourth most frequent function, etc. We have found a similar distribution for the frequency of method calls in our corpus; see the online appendix of this paper.

We have emphasized the utility of API-usage analysis for API migration, and we have described examples of such analysis that are tailored towards API migration. Let us mention a few more applications of API-usage analysis. In [28], a method for improving API documentation based on usage information is developed. In [13], the usability-improving use of wrapper libraries for APIs is discussed. In [26], APIs are analyzed to automatically extract ontologies about programming domains. In [15], source code is analyzed to find symptoms of API-method imitation, i.e., code that seems to reconstruct methods already offered by the API, thereby suggesting means of code improvement.

7. Conclusion

We have demonstrated a scalable approach to AST-based API-usage analysis for a large-scale corpus of open-source projects. Our implementation allows us to answer questions about usage of many well-known Java APIs. We have demonstrated this capability, for example, with specific aspects of XML programming APIs and generic aspects of framework-like API usage.

Our results should be advanced so that additional repositories, build systems, and 'well-known' projects are taken into account.

We are also working on advancing our method of fact extraction so that extra elements of the architecture provide better guarantees as to the completeness and correctness of fact extraction. For instance, we are working on a component for build forensics that tells us, subject to heuristics, whether an apparently successful project build is actually trustworthy.

Ultimately, we want to advance the suite of analyses so that this sort of empirical work becomes more directly useful for researchers and practitioners who are interested in API migration.

Acknowledgement The first author is grateful for the opportunity to present this research at an earlier stage in keynotes at the LaQuSo symposium on Verification and Validation of Software Systems, VVSS 2009, in Eindhoven, and at the Dutch PhD school IPA Herfstdagen 2008. The authors are grateful for Ruwen Hahn's major contributions to the technical realization of this research.

8. References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.

[2] T. T. Bartolomei, K. Czarnecki, and R. Lämmel. Swing to SWT and Back: Patterns for API Migration by Wrapping. In *Proceedings of 26th IEEE International Conference on Software Maintenance (ISCM 2010)*. IEEE, 2010. To appear. Paper available from the authors' web site.

[3] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API Migration for Two XML APIs. In *Software Language Engineering, 2nd International Conference, SLE 2009, Proceedings*, volume 5969 of *LNCS*, pages 42–61. Springer, 2010.

[4] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In *Proceedings of OOPSLA 2006*, pages 397–412. ACM, 2006.

[5] C. S. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software, Practice & Experience*, 37(6):581–641, 2007.

[6] J. Y. Gil and I. Maman. Micro patterns in Java code. In *Proceedings of OOPSLA 2005*, pages 97–116. ACM, 2005.

[7] R. Gobeille. The FOSSology project. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 47–50, New York, NY, USA, 2008. ACM.

[8] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 475–484, New York, NY, USA, 2010. ACM.

[9] M. Hahsler. A quantitative study of the adoption of design patterns by open source software developers. In *Free/Open Source Software Development, IGP*, pages 103–123, 2004.

[10] M. Hahsler and S. Koch. Discussion of a large-scale open source data collection methodology. *Hawaii International Conference on System Sciences*, 7:197b, 2005.

[11] F. Hunt and P. Johnson. On the Pareto distribution of sourceforge projects. In *Proceedings Open Source Software Development Workshop*, pages 122–129, 2002.

[12] J. M. D. III, U. Farooq, J. Stylos, and B. A. Myers. API usability: CHI'2009 special interest group meeting. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Extended Abstracts Volume*, pages 2771–2774. ACM, 2009.

[13] U. Jugel. Generating Smart Wrapper Libraries for Arbitrary APIs. In *Software Language Engineering, Second International Conference, SLE 2009, Revised Selected Papers*, volume 5969 of *LNCS*, pages 354–373. Springer, 2010.

[14] H. Kagdi, M. L. Collard, and J. I. Maletic. An approach to mining call-usage patterns with syntactic context. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 457–460. ACM, 2007.

[15] D. Kawrykow and M. P. Robillard. Improving api usage through automatic detection of redundant code. In *ASE'09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122, Washington, DC, USA, 2009. IEEE Computer Society.

[16] A. Klusener, R. Lämmel, and C. Verhoef. Architectural modifications to deployed software. *Science of Computer Programming*, 54(2-3):143–211, 2005.

[17] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code migration through transformations: an experience report. In *Proc. of the Conference of the Centre for Advanced Studies (CASCON)*, page 13. IBM, 1998.

[18] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM.

[19] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 106–115, 2007.

[20] L. Lopez-Fernandez, G. Robles, and J. M. Gonzalez-barahona. Applying social network analysis to the information in CVS repositories. In *Proceedings of the Mining Software Repositories Workshop*, 2004.

[21] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61. ACM, 2005.

[22] J. Martin and H. Muller. C to Java migration experiences. In *European Conference on Software Maintenance and Reengineering*, pages 143–153, 2002.

[23] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

[24] M. Nita and D. Notkin. Using Twinning to Adapt Programs to Alternative APIs. In *ICSE'10: Proc. of the 32nd Intern. Conf. on Software Engineering*, 2010.

[25] D. O'Donoghue, A. Leddy, J. Power, and J. Waldron. Bigram analysis of Java bytecode sequences. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 187–192, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland.

[26] D. Ratiu, M. Feilkas, F. Deissenboeck, J. Jürjens, and R. Marinescu. Towards a Repository of Common Programming Technologies Knowledge. In *Proceedings, International Workshop on Semantic Technologies in System Maintenance (STSM 2008)*, 2008.

[27] J. Stylos and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), Proceedings*, pages 195–202. IEEE Computer Society, 2006.

[28] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: improving API documentation using usage information. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009*, pages 4429–4434. ACM, 2009.

[29] A. A. Terekhov and C. Verhoef. The Realities of Language Conversions. *IEEE Software*, 17(6):111–124, 2000.

[30] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[31] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the Web. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, Washington, DC, USA, 2008. IEEE Computer Society.

[32] T. L. Veldhuizen. Software libraries and their reuse: Entropy, Kolmogorov complexity, and Zipf's law. *CoRR*, abs/cs/0508023, 2005.

[33] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *In TACAS*, pages 461–476, 2005.

[34] D. Weiss. Quantitative analysis of open source projects on SourceForge. In M. Scotto and G. Succi, editors, *Proceedings of the First International Conference on Open Source Systems, Genova*, pages 140–147, 2005.

[35] R. Wheeldon and S. Counsell. Power law distributions in class relationships. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:45, 2003.

[36] W. Wu, Y. G. Gueheneuc, G. Antoniol, and M. Kim. AURA: A Hybrid Approach to Identify Framework Evolution. In *ICSE'10: Proc. of the 32nd Intern. Conf. on Software Engineering*, 2010.

[37] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, New York, NY, USA, 2006. ACM.