Generating the Blueprints of the Java Ecosystem

Vassilios Karakoidas*, Dimitris Mitropoulos†, Panos Louridas*, Georgios Gousios‡ and Diomidis Spinellis*
*Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{bkarak,dds,louridas}@aueb.gr
†Computer Science Department
Columbia University
New York, United States
dimitro@cs.columbia.edu
‡Radboud University Nijmegen
Nijmegen, the Netherlands
g.gousios@cs.ru.nl

***Abstract*—Examining a large number of software artifacts can provide the research community with data regarding quality and design. We present a dataset obtained by statically analyzing 22730 JAR files taken from the Maven central archive, which is the de-facto application library repository for the Java ecosystem. For our analysis we used three popular static analysis tools that calculate metrics regarding object-oriented design, program size, and package design. The dataset contains the metrics results that every tool reports for every selected JAR of the ecosystem. Our dataset can be used to produce interesting research results, such as measure the domain-specific language usage.**

I. Introduction

We present a dataset that contains popular metrics, calculated from the analysis of a large collection of software artifacts written in Java. All artifacts were taken from the *Maven Central Repository* [1].

Maven is a build automation tool used primarily for Java projects, and it is maintained by the Apache Software Foundation [1]. To describe the software project being built, its dependencies, and the build order, Maven uses XML. The central repository contains more than 400,000 JARs, in a variety of programming languages such as Java, Clojure, Groovy, and Scala. All supported languages use the JVM platform as their runtime environment. To build a software component, it dynamically downloads Java libraries and other plug-ins from the Maven central repository, and stores them in a local cache. The repository can be updated with new projects and also with new versions of existing projects that can depend on other versions.

To analyze the projects coming from Maven repository, we used the following tools: a) CKJM [2], b) JDepend [3], and c) CLMT [4]. All tools focus on three main aspects of a software system, namely: object-oriented design, program size, and package design. Such aspects are considered very important because they provide quantifiable information about the quality and structure of a software component.

In this paper we present: a) the construction process to obtain the collection of the metrics results that the three aforementioned tools produced for 22,730 JARs, b) our dataset and c) how researchers and practitioners can use the dataset and produce meaningful results.

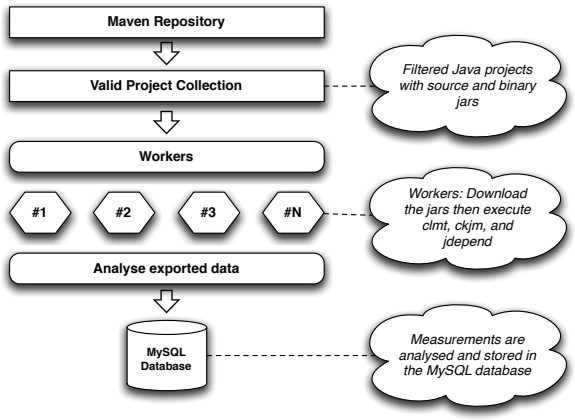

Fig. 1. The dataset construction process.

II. Dataset Construction Process

The dataset construction process is illustrated in Figure 1. In general, it follows the same methodology as Mitropoulos et al. [5].

Initially, a snapshot of the Maven repository was downloaded locally. The repository contains various projects that contain several versions. In our experiment, we only used Java projects. For every Java project all versions were filtered out and only the latest was kept. The process to identify them was the following; Every project consists of two JAR files, one that contains a compiled version of the project and one that contains the source code. The projects that did not have both binary and source JARs, were excluded from the experiment. When the source JAR file was downloaded, it was scanned for Java source files. If the source files were present in the archive, then the project was flagged as a valid Java project.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.76

TABLE I
THE SELECTED MAVEN PROJECTS' SIZE METRICS.

Metric	Value
Project Count	11,365
File Count	449,213
Package Count	59,436
Lines of Code	74,565,772
Source Lines of Code	40,921,287
Comment Lines of Code	25,268,959
Number of Classes	370,518
Number of Interfaces	66,352
Number of Enumerations	9,879
Measurement Count	32,844,836

When the selection process was finished, we created a series of processing tasks based on the selected JARs, and added them to a task queue mechanism. Then we executed a number of workers that checked out tasks from the queue, applied the static analysis tools on each JAR and stored the results to the data repository (a MySQL database system). The workers were written in Python. In addition, there were several shell scripts (bash) that performed several checks and sanitisation tasks (e.g. checking for malformed results). The tools that were applied on every JAR were the following:

- **CKJM-ext** *(ckjm)* [2]; which calculates many software metrics, including the *Chidamber and Kemmerer* set of object-oriented metrics [6]. The version of the tool that we used for this experiment can be found in GitHub [7].
- **JDepend** *(jdep)* [3]; a tool that analyses JAR files that contain compiled Java classes and calculates a series of design metrics.
- **CLMT** *(clmt)* [4]; which stands for *cross-language metric tool* and analyses the source code of several languages in order to calculate a series of size metrics.

Table I presents several size metrics for our dataset. It includes 11,365 projects, with more than 74 million lines of code and almost 33 million unique measurements. Table III, IV, V, and VI present the key metrics that are calculated and stored in the dataset. Each tool focuses on a different aspect of a software system; *ckjm* focuses on object-oriented design metrics [8], [6], *jdepend* on package design, while *clmt* focuses on program size metrics. Several metrics are calculated by more than one tool, like *cyclomatic complexity* [9], which is calculated by both *clmt* and *ckjm*. Both calculations are available in the database.

A series of newly introduced metrics were also stored in the database. Such metrics can count the usage of specific DSL application libraries in a software project and they are included in the Table VI. The methodology to identify and calculate the DSL metrics was the following: A set of standard DSL application libraries was identified, and the source was scanned for specific *import* statements (e.g. *java.util.regex*). These statements indicated that the standard package that implements regular expressions was used, thus regular expressions were used in the project. Build files or other resources that may contain DSLs were not included. One final assumption was also made; if XPath or XSLT were found in the source code, then the project would be marked as a project that utilizes XML. This is because both languages are used for query and transformations on XML DOM trees. Table II lists the selected DSLs application libraries. Note that we focused on libraries that were included as part of the Java SDK.

TABLE II
LIST OF SELECTED DSL APPLICATION LIBRARIES.

DSL	Java Package
Regular Expressions	`java.util.regex`
XML	`javax.xml`, `org.w3c` and `org.xml`
SQL	`java.sql` and `javax.sql`
XPath	`java.xml.xpath`
XSLT	`javax.xml.transform`
RTF	`javax.swing.text.rtf`
HTML	`javax.swing.text.html`

TABLE III
CLASS DESIGN METRICS.

Depth Of Inheritance Tree	*ckjm*
Coupling Between Objects	*ckjm*
Weighted Methods Per Class	*ckjm*
Response For Class	*ckjm*
Lack Of Cohesion In Methods	*ckjm*
Number Of Children	*ckjm*
Attribute Hiding Factor	*clmt*
Coupling Between Methods	*ckjm*
Average Method Complexity	*ckjm*
Cohesion Among Methods of Class	*ckjm*
Data Access Metric	*ckjm, clmt*
Inheritance Coupling	*ckjm*
Lack Of Cohesion In Methods3	*ckjm*
Measure Of Aggregation	*ckjm*
Measure Of Functional Abstraction	*ckjm*
Number of Attributes	*clmt*
Method Hiding Factor	*clmt*

TABLE IV
METHOD DESIGN METRICS.

Number of Method Parameters	*clmt*
Number of Methods	*clmt, ckjm*
McCabe Cyclomatic Complexity	*ckjm, clmt*

TABLE V
PACKAGE DESIGN METRICS.

Number of Concrete Classes	*jdep*
Afferent Couplings	*jdep, ckjm*
Efferent Couplings	*jdep, ckjm*
Instability	*jdep*
Abstractness	*jdep*
Distance Main Sequence	*jdep*

In Section V, we describe an experiment based on these metrics. The experiment measures the popularity of DSL usage for the dataset and therefore for the Java ecosystem in general.

III. DATABASE STRUCTURE

Figure 2 illustrates the database schema. Specifically, it consists of five tables; *measurement*, *category*, *identifiers*, *measurement_type*, and *project*. The central database table is *measurement*, which holds the measurement values. The other tables can be used for normalization.

TABLE VI
PROGRAM SIZE METRICS.

Number of Classes	*clmt, ckjm, jdep*
Number of Enumerations	*clmt, ckjm*
Number of Interfaces	*clmt, ckjm*
Module Count	*clmt, ckjm*
Comments Lines Of Code	*clmt*
Lines Of Code	*clmt, ckjm*
Source Lines Of Code	*clmt*
Function Oriented Code	*clmt*
DSL Usage Count	*clmt*
RTF Usage Count	*clmt*
Regex Usage Count	*clmt*
HTML Usage Count	*clmt*
XPath Usage Count	*clmt*
XSLT Usage Count	*clmt*
XML Usage Count	*clmt*
SQL Usage Count	*clmt*
File Count	*clmt*

Metrics are divided into six categories that define their scope; *module*, *class*, *method*, *code unit*, and *project-wide*. These values are stored in *category* database table.

The descriptive names for each metric are stored in the *measurement_type* database table. There are 65 metrics available in the dataset. The names are composed by the actual metric name and the tool that was used to calculate them. For instance, *McCabe_clmt* denotes that the metric name is the McCabe cyclomatic complexity generated by the *clmt* tool, while *McCabe_ckjm* means that it is the same metric calculated by the *ckjm* tool.

Filenames, methods, classes and package names along with other identifiers are stored in the *identifiers* table. Note that each measurement has a related filename and a related identifier that points to a software element. For example the *afferent coupling* metric for the module in the directory "com/scalagent/jmx" is related to the identifier *com.scalagent.jmx*. Finally, the database table *project* contains the related project identifiers, directly extracted from the maven repository.

IV. RESEARCH OPPORTUNITIES

The measurements provided by our dataset, can be used by researchers and practitioners alike. Researchers can use the data to experiment with new software quality models, or validate them with real projects. Practitioners who develop tools that calculate metrics can use the dataset to validate if their tools produce correct results.

The dataset also includes the DSL-related metrics, which quantifies the usage of basic DSL application libraries that are already shipped with the Java SDK. These kind of metrics, can be really useful for researchers that focus on DSL embedding, or study DSL usage patterns [10] as we illustrate in the following section. In addition, the dataset covers a significant portion of the Maven repository, it can be used for large scale empirical studies.

Finally, our dataset can be used side-by-side with the ones presented by Mitropoulos et al. [11] and Raemaekers et al. [12] in order to examine the datasets for metric correlations. This is because their datasets were also produced by analyzing projects coming from the Maven repository.

TABLE VII
TOP DSL USAGE COMBINATIONS.

DSLs	**Count**
XML	1,561
Regex	909
SQL	493
XML, XSLT	475
Regex, XML, XSLT	158
Regex, XML	303
SQL, XML	162
Regex, SQL	116
Regex, SQL, XML, XSLT	80
Regex, SQL, XML	71
SQL, XML, XSLT	54
XML, XPath	50

V. EXPERIMENTING WITH THE DATASET

The initial goal of this experiment was to provide quantifiable results that are indicative regarding the usage of DSLs in multiple Java projects. The related metrics can be found in the database table named *measuremnt_type*, and they are the following: *RTFUsage*, *RegexUsage*, *HTMLUsage*, *HTMLUsage*, *XPathUsage*, *XSLTUsage*, *XMLUsage*, *SQLUsage*, and *DSLCount*. A Python script that analysed the measurements stored in the database, we produced the results illustrated in Table VII. This table, lists popular DSL usage combinations found in our dataset. An interesting observation is that XML is the most widespread DSL with 1,561 occurrences in 11,365 projects (13%). Regular expressions are also popular with 909 occurrences (7%). Note that the numbers represents the projects where XML and regular expressions were used as the only DSL.

VI. LIMITATIONS

As we mentioned earlier, to produce our dataset we adopted a narrow selection process. In particular, we identified and analyzed only projects that were written in Java and had both the source code and the binary JAR available in the repository. The former reduced significantly the number of projects that were analyzed, because many of them provided only the binary JAR. Since only *clmt* analyses the source code, this resulted in the reduction of the number of measurements that exposed object-oriented and package design issues.

Finally, only one version was analyzed per project (the latest, unless it violated the aforementioned restriction). This decision renders the dataset unusable, for research focused on software evolution.

VII. RELATED WORK

The Maven ecosystem has been previously analyzed by Raemaekers et al. [12] to produce the *Maven dependency dataset*. Apart from basic information like individual methods, classes, packages and lines of code for every JAR, this dataset

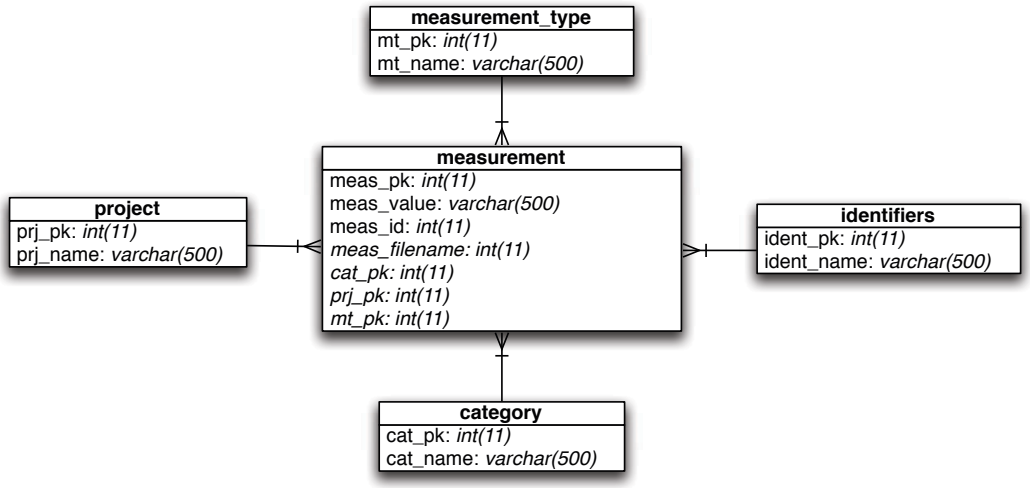

Fig. 2. The database schema.

also includes a database with all the connections between the aforementioned elements.

Also, Mitropoulos et al. performed a similar experiment [5]. In particular, they used the FindBugs [11] static analysis tool, to analyze a large part of the maven repository. Their dataset includes for each JAR, a corresponding bug collection produced by FindBugs. Our work differs from these two approaches since it presents the metrics calculated by the three different static analysis tools that calculate metrics regarding object-oriented design, program size, and package design.

VIII. Conclusions

We presented a dataset created by applying three different static analysis tools on Java projects coming from the Maven Central repository. The results involve a variety of software metrics from program size to object-oriented design. We have also shown how our data can be used to extract meaningful results concerning DSL usage in Java Projects.

As we discussed, our selection process was strict, filtering out projects that did not have the project's sources. However, we are planning to expand our project coverage in the future and include projects that cannot be analysed by all tools (due to source code unavailability). Finally, by including all versions for each project we can allow researchers to use our dataset in the evolution context.

IX. Availability

The dataset and the source code of this publication, along with some utility scripts are available at https://github.com/bkarak/data_msr2015. The SQL dump of the database is available at http://gaijin.dmst.aueb.gr/~bkarak/data_msr2015.bz2.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund esf) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (nsrf) - Research Funding Program: Thalis Athens University of Economics and Business - software engineering research platform.

References

[1] "Maven," February 2015, http://maven.apache.org/.

[2] D. Spinellis, "Tool writing: A forgotten art?" *IEEE Software*, vol. 22, no. 4, pp. 9–11, July/August 2005.

[3] "Jdepend," 2009, http://clarkware.com/software/JDepend.html.

[4] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos, "Evaluating the quality of open source software," in *SQM 2008: Second International Workshop on Software Quality and Maintainability—12th European Conference on Software Maintenance and Reengineering (CSMR 2008) satellite event*. The Reengineering Forum, Apr. 2008, pp. 5–28, electronic Notes in Theoretical Computer Science Volume 233 (March 2009).

[5] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The bug catalog of the maven ecosystem," in *MSR 2014, The 11th Working Conference on Mining Software Repositories*. ACM, May 2014.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[7] "Ckjm extended version," February 2015, https://github.com/bkarak/ckjm_ext.

[8] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[9] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[10] V. Karakoidas, "On domain-specific languages usage (why dsls really matter)," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 20, no. 3, pp. 16–17, March 2014.

[11] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[12] S. Raemaekers, A. v. Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, 2013, pp. 221–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487129