The Evolution of Project Inter-Dependencies in a Software Ecosystem: the Case of Apache

Gabriele Bavota[1], Gerardo Canfora[1], Massimiliano Di Penta[1], Rocco Oliveto[2], Sebastiano Panichella[1]
[1]University of Sannio, Benevento, Italy
[2]University of Molise, Pesche (IS), Italy
{gbavota, canfora, dipenta}@unisannio.it, rocco.oliveto@unimol.it, spanichella@unisannio.it

***Abstract*—Software ecosystems consist of multiple software projects, often interrelated each other by means of dependency relations. When one project undergoes changes, other projects may decide to upgrade the dependency. For example, a project could use a new version of another project because the latter has been enhanced or subject to some bug-fixing activities. This paper reports an exploratory study aimed at observing the evolution of the Java subset of the Apache ecosystem, consisting of 147 projects, for a period of 14 years, and resulting in 1,964 releases. Specifically, we analyze (i) how dependencies change over time; (ii) whether a dependency upgrade is due to different kinds of factors, such as different kinds of API changes or licensing issues; and (iii) how an upgrade impacts on a related project. Results of this study help to comprehend the phenomenon of library/component upgrade, and provides the basis for a new family of recommenders aimed at supporting developers in the complex (and risky) activity of managing library/component upgrade within their software projects.**

I. Introduction

Software ecosystems [2], [16] are groups of software projects that are developed and co-evolve in the same environment. These projects share code, depend on one another, reuse the same code, and can be built on similar technologies. Examples of ecosystems—investigated in previous studies about software evolution—can be the plugins developed for a specific platform, such as the universe of Eclipse plug-ins [3], the programs developed with a specific programming language [25], or even using domain-specific language (see for instance the R ecosystem studied by German *et al.* [8]).

When one project undergoes changes and issues a new release, this may or may not lead other projects to upgrade their dependencies. On the one hand, using up-to-date releases of libraries/components may result useful, because these releases can contain new and useful features, and/or possibly some faults may have been fixed. On the other hand, the upgrade of a component may create a series of issues. For example, some APIs may have changed their interface, or might even be deprecated [25], which makes necessary the adaptation of its client. In addition, let us suppose a program uses multiple libraries, namely lib_1 and lib_2, and lib_1 depends on lib_2. It can happen that if one upgrades lib_2, then lib_1 no longer works because does not support the new release of lib_2. Last, but not least, a library/component might have changed its license making it legally incompatible with the program using it [6]. All these scenarios suggest that *managing the upgrades of libraries/components in large ecosystems is a complex and daunting task, which requires to ponder several factors*. In principle, the problem is dealt with update management tools available in many operating systems—e.g., Windows, Linux, MacOS—however such update tools either work with entire applications or with operating system related upgrades. Also, they are not able to decide when performing the upgrade and when it might be avoided or postoponed.

This paper presents the results of an exploratory study aiming at (i) investigating how dependencies between projects change among the Java subset of the Apache ecosystem; and (ii) exploring and understanding the likely reasons and consequences of such changes. Specifically, the paper investigates:

1) how the projects composing the ecosystem evolve and how the dependencies between them change. In the context of this study, we limit our attention to dependencies related to API usage and/or framework usage through extension;
2) to what extent are dependencies upgraded (i.e., to a new release of the target project), and what are the drivers of such upgrades;
3) how the upgrade of a dependency impacts on the source code of a project.

The investigated ecosystem contains software projects generally related to the domain of Web application (and not only) development, ranging from JSP/Servlet engines (e.g., Tomcat) to Web service containers (Axis), XML parsers (Xerces), and various kinds of support library (e.g., Apache commons or log4j). Overall, *we observed the evolution of 147 projects over a period of 14 years, resulting in a total number of 1,964 releases*.

Results indicate a tangible increase of the dependencies over time. When a new release of a project is issued, in 69% of the cases this does not trigger an upgrade. When, instead, this happens, the likely reasons have to be found in major changes (e.g., new features/services) as well as in large amount of bug fixes. Instead, developers are reluctant to perform an upgrade when some APIs are removed. The impact of upgrades is generally low, unless it is related to frameworks/libraries used in crosscutting concerns.

The paper is organized as follows. Section II describes the study definition and planning, while results are reported in Section III. Section IV discusses the threats that could affect the validity of the results achieved. Section V relates our study with existing literature about the evolution of software ecosystems and evolution/adaptation of APIs. Finally, Section VI concludes the paper and outlines directions for future work.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.39

II. Study Definition and Planning

The *goal* of our study is to analyze how project inter-dependencies evolve in a software ecosystem, with the *purpose* of understanding the likely reasons and consequences of such changes. The *quality focus* is software maintainability, which could be improved by understanding the phenomenon of library/component upgrade. The *perspective* is of researchers interested in understanding when and why developers upgrade dependencies in software ecosystems.

The *context* of our study consists of the entire history of the Java subset of the Apache ecosystem, that represent the vast majority of it (75% of the projects). To date, the entire Apache ecosystem is composed of 195 software projects spread in 23 different categories (e.g., big-data, FTP, mobile, library, testing, XML) and developed by using a total of 29 programming languages. We analyzed the change history of the 147 Java software systems, in the period of time going from June 1999 to April 2013 resulting in 1,964 releases. The size of the ecosystem in the analyzed period of time ranges from 32 up to 28,584 KLOCs, while the number of classes (methods) ranges from 113 to 114,000 (1,386 to 780,731). For sake of clarity, in the following we refer to the project having a dependency toward another project as the "client project".

A. Research Questions

The study aimed at providing answers for the following three research questions:

- **RQ**$_1$: *How does the Apache ecosystem evolve?* This research question is preliminary to the other two, and aims at providing a picture of the context of our study. Specifically, we analyzed how the number of projects, their size, the dependencies among them, and the declared software licenses changed in the Apache ecosystem during time. Such information represents the foundation for the other research questions.
- **RQ**$_2$: *What are the reasons driving a client project to upgrade a dependency toward a new available release of a project it depends on?* Our conjecture is that the client project does not always upgrade a project it depends on when a new release of the latter is available. In this research question we not only aimed at verifying our conjecture, but we also tried to understand what are the reasons driving a client project to upgrade (or not) toward a new available release of a project it depends on. We analyzed as possible factors (i) *structural* changes, captured by analyzing changes in source code of the used project (major/minor); (ii) *functional* changes, captured by analyzing release notes, and (ii) *legal* changes, i.e., those occurring in the declared licenses, that might result in legal incompatibilities between the client project and the project it uses.
- **RQ**$_3$: *What is the impact on the client project code of an upgrade of a dependency toward a new available release of a project it depends on?* This research question aims at quantitatively investigating the impact on the source code of the client project when it upgrades a project it uses toward a new available release.

B. Data Extraction Process

To answer our research questions we first *downloaded the source code* of the 1,964 software releases considered in our study. We used a crawler and a code analyzer developed in the context of the Markos European project[1]. The crawler was able to identify for a given project of interest the list of available releases with their release date as well as its svn address. This information was extracted by crawling DOAP (Description Of A Project) files[2] available on the Internet.

Using the information extracted by the crawler, the code analyzer checked-out the SVN repository and identified the folder containing each of the project releases identified by the crawler. This was done by exploiting the SVN tag mechanism. In other words, the versioning system of Apache projects has a separate directory for each release (where files belonging to such a release are stored), besides keeping the project history in the SVN main trunk. In case the code analyzer did not identify any folder containing a particular release, it reported the problem. During data extraction, such an issue happened for 278 releases that were manually downloaded from the Apache archives, available online for each project[3].

Once downloaded all the software releases, we *extracted dependencies* existing between them. Note that in this study we focus on dependencies existing between Java Apache projects, ignoring those toward projects external to the Apache ecosystem or not written in Java. Also in this case, the Markos code analyzer has been used. The identification of the inter-project dependencies was performed in different steps. Given a set of software releases, the code analyzer searched—in each folder release—for files that explicitly reported inter-project dependencies. These files in the Apache ecosystem are generally of three types: `libraries.properties`, `deps.properties`, or the Maven `pom.xml` file. Note that the dependency information reported in these files is generally detailed (i.e., both the name of the project as well as the used release are reported) and reliable.

When the code analyzer was not able to find none of these files, it searched for all jar files contained in the release folder and tried to match each of those files with one of the other software releases provided. This is done by computing the Levenshtein distance [17] between the name of the jar file and the name of each provided release. The output of the code analyzer is a list of candidate dependencies between the set of provided software releases.

In our study, we assumed the dependencies extracted by parsing the files `libraries.properties`, `deps.properties`, and `pom.xml` as correct. Instead, when the dependencies were extracted by analyzing jar files in the release folder, we manually validated all candidate dependencies classifying them as *true dependencies* or as *false positives*. This operation was done by two of the authors that analyzed a total of 3,742 dependencies, classifying as correct 832 of them. Overall, the final number of dependencies found in the analyzed 14 years and considered in our study is 3,514.

[1] http://markosproject.berlios.de

[2] http://projects.apache.org/doap.html

[3] An example of archive for the Ant project can be found here http://archive.apache.org/dist/ant/source/

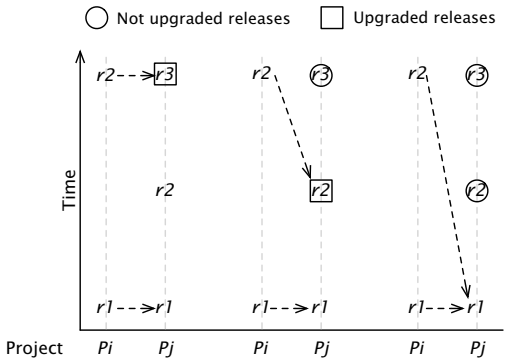


Fig. 1. Process used to divide *upgraded* and *not upgraded* releases.

All the extracted data is not enough to answer all research questions. Indeed, to answer $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ we also identified the software licenses declared in the downloaded software releases. To this aim we used Ninka[4] [10], a lightweight license identification tool for source code that consists on a sentence-based matching algorithm that automatically identifies license from *license statements*. We ran Ninka on each file contained in the 1,964 software releases considered in our study obtaining as output the license type and version declared in its licensing statement (if present).

As for the other factors considered in $\mathbf{RQ}_2$, we computed the bug-fixed in each software release and the changes performed between each pair of subsequent releases of the same project. As for the bug-fixing we mined bug-tracking systems of the various projects, extracting only the bugs fixed in each specific release, while the Markos code analyzer was used to extract changes performed among two subsequent releases of each project. We extracted the number of (i) added and deleted classes; (ii) added and deleted public methods; and (iii) changes in existing methods (by distinguishing between public and non public methods).

To classify releases from a functional point of view, we manually analyzed release notes of all the analyzed releases, and classified them using the following tags: *minor* (only improvement of existing features), *major* (new features added), and *bug fixing*. Clearly, the tag *minor* excludes the tag *major*, while the tag *bug fixing* is orthogonal to *minor/major*, and can be assigned to any release note talking about fixed bugs, despite it underwent minor or major changes. This classification has been performed by two of the authors who individually analyzed and tagged the release notes. Then, they performed an open discussion to resolve any conflicts and reach a consensus on the assigned tags.

To answer $\mathbf{RQ}_3$, we identified—using again the Markos code analyzer—the source code potentially impacted when an upgrade of a dependency is performed by a client project. The impacted source code is overestimated considering as candidate impact set all the classes of the client project importing at least one class of the upgraded project.

[4]http://ninka.turingmachine.org/

C. Analysis Method

In order to answer $\mathbf{RQ}_1$ we analyzed the history of the Apache ecosystem, considering snapshots captured every month. In particular, starting from June 1999, we compute, for each month: (i) the number of existing projects; (ii) the size of the ecosystem in terms of KLOCs; (iii) the dependencies existing between projects; (iv) the software licenses declared in source files.

Concerning the quantitative analysis performed to answer $\mathbf{RQ}_2$, and given the dependencies existing between the different releases during time, we verified if releases that are upgraded by client projects (hereby referred as *upgraded releases*) have more changes and/or bug-fixing than releases ignored by client projects (hereby referred as *not upgraded releases*).

To create the two sets of releases (i.e., *upgraded releases* and *not upgraded releases*) we adopted the process depicted in Fig. 1. For each pair of Apache projects, P_i and P_j, having at least one dependency between their releases, when P_i upgrades the dependency toward P_j, we determined whether P_i upgrades the dependency toward to the last existing release of P_j or to another release. In the former case, we put the upgraded P_j release in the set *upgraded releases*. Instead, when the upgrade was not toward the last available release we still put the upgraded P_j release in the set *upgraded releases*, however we also put the newer ignored releases of P_j in *not upgraded releases*.

To better understand how we computed such sets, Fig. 1 shows three different evolution scenarios of dependencies between two projects P_i and P_j. Let us assume that the release $r1$ of P_i depends on the release $r1$ of P_j. Then, a new version of project P_i is released ($r2$). In the first scenario, when $r2$ for P_i is released, its dependency is upgraded to $r3$ of P_j, the last available P_j release. In this case, $r3$ is included in the set *upgraded releases*, while no releases are added to the set *not upgraded releases*, since P_i correctly upgraded its dependency to the last available P_j release. In the second scenario (reported in the middle of Fig. 1), the release $r2$ of P_i upgrades its dependency to the release $r2$ of P_j, even if a newer release (i.e., $r3$) is available. In this case the release $r3$ of P_j has been "ignored" by P_i and thus, it is added to the set *not upgraded releases*, while release $r2$ of P_j is added to the set *upgraded releases*. In the third and last case, P_i does not upgrade at all the dependencies toward P_j, i.e., the new release of P_i continues to use the release $r1$ of P_j, despite the availability of more recent releases (i.e., $r2$ and $r3$). In this case, $r2$ and $r3$ are added to the set *not upgraded releases*, while no releases are added to the set *upgraded releases*.

Note that, if a release r_i of a project P_j belongs to the set of *upgraded releases* when analyzing dependencies between P_i and P_j, and the same release belongs the set of *not upgraded releases* when analyzing dependencies between a project P_s and P_j, the release r_i is removed from both sets, and not considered any longer in the comparison between *upgraded releases* and *not upgraded releases*. This is done (i) to avoid overlap between the two sets, does not allowing their fair comparison; and (ii) to strongly isolate only releases that are generally upgraded (and not) by client projects.

Besides comparing descriptive statistics, we also used the

Mann-Whitney test [4] to compare the distribution of changes and bug-fixing for the above described two sets of releases (information extracted through the process described in Section II-B). We assumed a significance level of 95%. We also estimated the magnitude of the difference between the number of changes for the two considered groups of releases (upgraded and not upgraded by clients) using the Cliff's Delta (or d), a non-parametric effect size measure [14] for ordinal data. We followed the guidelines in [14] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Finally, to answer $\mathbf{RQ}_3$ we report descriptive statistics of the impacted client code in terms of percentage of impacted classes, and percentage of impacted LOCs.

D. Replication package

The study described in this section can be replicated using the replication package available online[5]. Such a replication package includes information to download all analyzed projects, as well as working data sets used to answer the study research questions.

III. Analysis of the Results

This section discusses the results achieved aimed at answering the three research questions formulated in Section II-A.

A. RQ1: How does the Apache ecosystem evolve?

Fig. 2 reports the evolution over time of the Java Apache ecosystem, in terms of size measured in KLOCs (see Fig. 2(a)), number of projects (black line in Fig. 2(b)) and number of dependencies existing between them (gray line in Fig. 2(b)). As expected, during the analyzed 14 years, the size of the Apache ecosystem grows up exponentially (model fitting resulted in an adjusted $R^2 = 0.56$). From the single Java project existing in 1999 (i.e., Apache ECS[6]) the Apache ecosystem grows up to the 147 Java projects existing today (reflecting also the new developers that started to work in the Apache project teams). Such a growth is linear (adjusted $R^2 = 0.98$). With the increasing of the number of projects also the size—see Fig. 2 (b)—of the entire ecosystem grows, by reaching almost 30 Million LOCs in April 2013. A very strong peak in the size of the ecosystem can be observed between the end of 2006 and the begin of 2007, where the size to the Apache ecosystem redoubled. In this period, several new and big projects have been added to the ecosystem, e.g., Apache UIMA[7] with its 2 millions of LOCs.

Also, dependencies between projects increase continuously during evolution. Similarly to the size, but differently from the number of projects, dependencies follow an exponential trend (adjusted $R^2 = 0.56$). In fact, until 2003 (when about 25 projects were in the ecosystem) there were few dependencies between the projects. After 2003, dependencies sensibly grow in the following years. This is mainly due to the fact that several projects added after 2003 are projects implement reusable components—like those belonging to the Apache Commons[8]—that are used as libraries by several Apache projects. For example the number of client projects for Apache Commons Compress[9] grows up to 20 (April 2013).

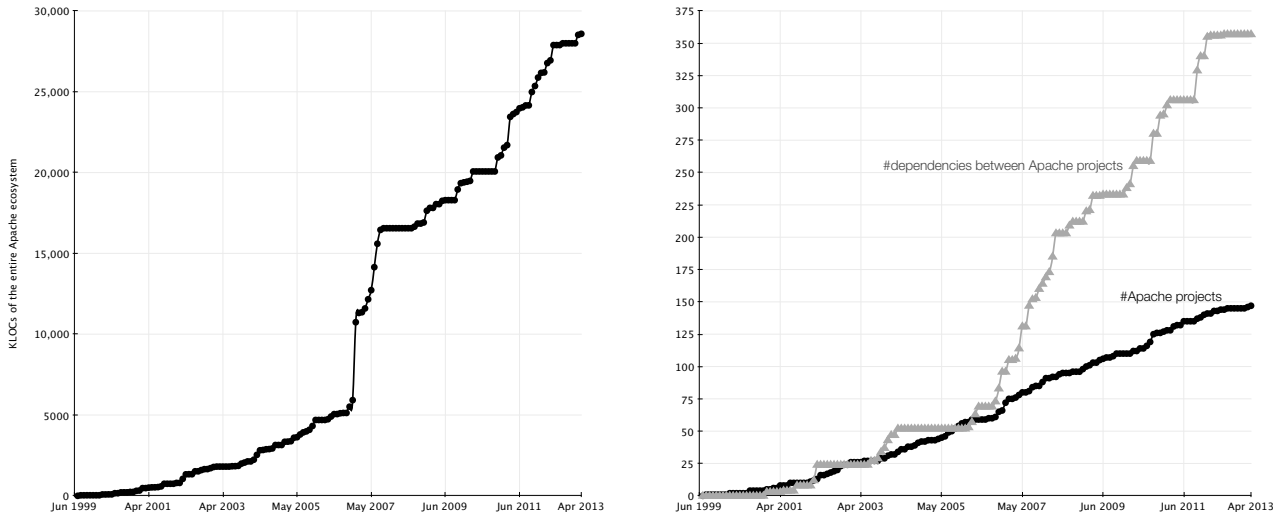
To get a better view on how the Apache software projects and the dependencies between them evolved during time, Fig. 3 shows snapshots of the Apache ecosystem from 2002 to 2013. We ignored the years before 2002 since, as reported in Fig. 2(b), the number of projects (and dependencies) is quite low. In the graphs of Fig. 3, each node represents a project, while an edge connecting two nodes represents a dependency between two projects. By looking at the figure it is clear as the net of dependencies in the ecosystem grows during evolution. Also, focusing on the 2013 snapshot, several *hub projects*, i.e., projects having a lot of client projects, can be noticed. Besides the previously cited Apache Commons project, other *hub projects* are for example Apache Log4j[10] (having 31 client projects), Apache Geronimo[11] (30), and Apache Ant[12] (29). It is worth noting that all these projects implement quite general and reusable features, useful for software projects having different purposes.

As explained in Section II-C, we also analyzed the evolution of the software licenses declared by the Apache projects during time. From 1999 until 2003 we found *Apache Software License (ASL) v1.1* as the only license present in all source code files of all the existing projects. Starting from 2004 all projects started to migrate toward ASL *v2.0* and, by the end of 2004, 86% of the source code files in the Apache ecosystem already completed such a migration, leaving the remaining 14% to *v1.1*. This migration was complete in 2008. In addition to these two licenses, we just found one Apache project (i.e., ApacheTapestry[13]) containing in the majority of its source code files a different license, namely *BSD 3*. However, this does not create any legal issues for potential client projects interested in using ApacheTapestry as library. In fact, the ASL is largely inspired to the *BSD* license and, contrarily to the *GPL* one, source code files having a *BSD* license can be used by source code files having an ASL. Given that the changes in terms of licenses observed during the Apache ecosystem history cannot generate legal issues, in our $\mathbf{RQ}_2$ we will not analyze licenses as a possible factor motivating the upgrade of a dependency for client projects.

B. RQ2: What are the reasons driving a client project to upgrade a dependency toward a new available release of a project it depends on?

Regarding ($\mathbf{RQ}_2$), we first verified if *not upgraded* releases exist in the Apache ecosystem history. Among the 1,964 releases considered in our study, 950 have been involved in at least one dependency (as client or as library) during their history. Of these 950, 140 releases belong to the 38 projects that have been used as "library project", i.e., have at least one client project using them. Thus, these are the 140 releases that we classified as *upgraded* or as *not upgraded* by client projects, following the process described in Section II-C. It

[5] http://distat.unimol.it/reports/icsm-apache/
[6] http://projects.apache.org/projects/ecs.html
[7] http://uima.apache.org/
[8] http://commons.apache.org/
[9] http://commons.apache.org/proper/commons-compress/
[10] http://logging.apache.org/log4j/
[11] http://geronimo.apache.org/
[12] http://ant.apache.org/
[13] http://tapestry.apache.org/



(a) (b)

Fig. 2. Evolution of the size (a) and of the projects and dependencies (b) in the Apache ecosystem.

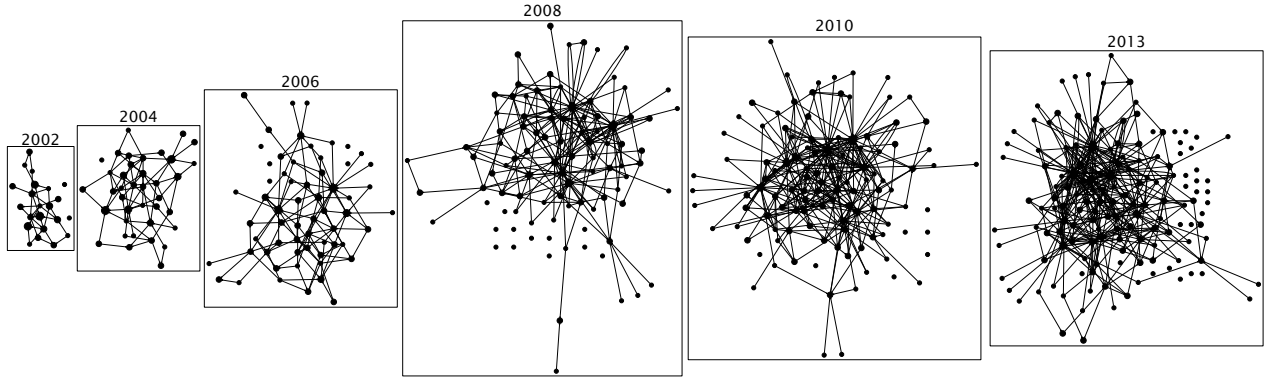


Fig. 3. Snapshots of projects and their dependencies in the Apache ecosystem history.

is worthwhile to note that 14 of these releases have not been assigned to one of the two sets, due to the fact that they are upgraded by some client projects and not upgraded by other clients. Af for the other releases, 87 belong to the *not upgraded* set, while 39 have been assigned to the *upgraded* set. This means that 69% of new releases of Apache software projects are "ignored" by client projects that depend on such projects.

Fig. 4 reports the boxplots for different type of changes for releases that are ignored by client projects (i.e., *not upgraded releases*), and releases used by client projects to upgrade their dependencies (i.e., *upgraded releases*). Moreover, Table I reports the results of the Mann-Whitney test (p-value) and the Cliff's d effect size when comparing the distributions for the different types of changes performed on *upgraded* and *not upgraded* releases. On average, in *upgraded releases* there are 25 times more added classes than in *not upgraded releases* (125 *vs* 5)—see Fig. 4(a). As shown in Table I, this difference is statistically significant (p-value $<$0.0001) with a large effect size (0.62). From Fig. 4 it is clear as, generally, there are no deleted classes (with respect to the previous release) in both kinds of releases—see Fig. 4(b)–and thus, no statistically significant difference.

As for the changes applied to existing methods (i.e., methods already present in a previous release of the project), we observed almost three times more changes for the *upgraded releases* when analyzing all methods in the system (705 *vs* 217)–see Fig. 4(c)—as well as when just focusing on public methods (that are those used by the client projects), 527 *vs* 160—see Fig. 4(d). For both types of changes, results in Table I highlight statistically significant differences between *upgraded* and *not upgraded* releases, with a large effect size (0.48) when considering all methods, and a medium effect size (0.46) when just focusing on public methods. Also the number of added public methods is higher in the *upgraded releases* (six times more) than in *not upgraded* releases—see Fig. 4(e)—with statistically significance and a large effect size (0.57).

All these results quantitatively highlight that *upgraded releases* contain changes affecting the interfaces and substantial changes, as compared to the *not upgraded releases*. This is particularly evident when focusing on added classes (29 times more) that are likely related to new features provided by the new project release, and on added public methods (six times more than for *not upgraded releases*), that represent new services available to the client projects. Also, the higher number of overall method changes (three times more than

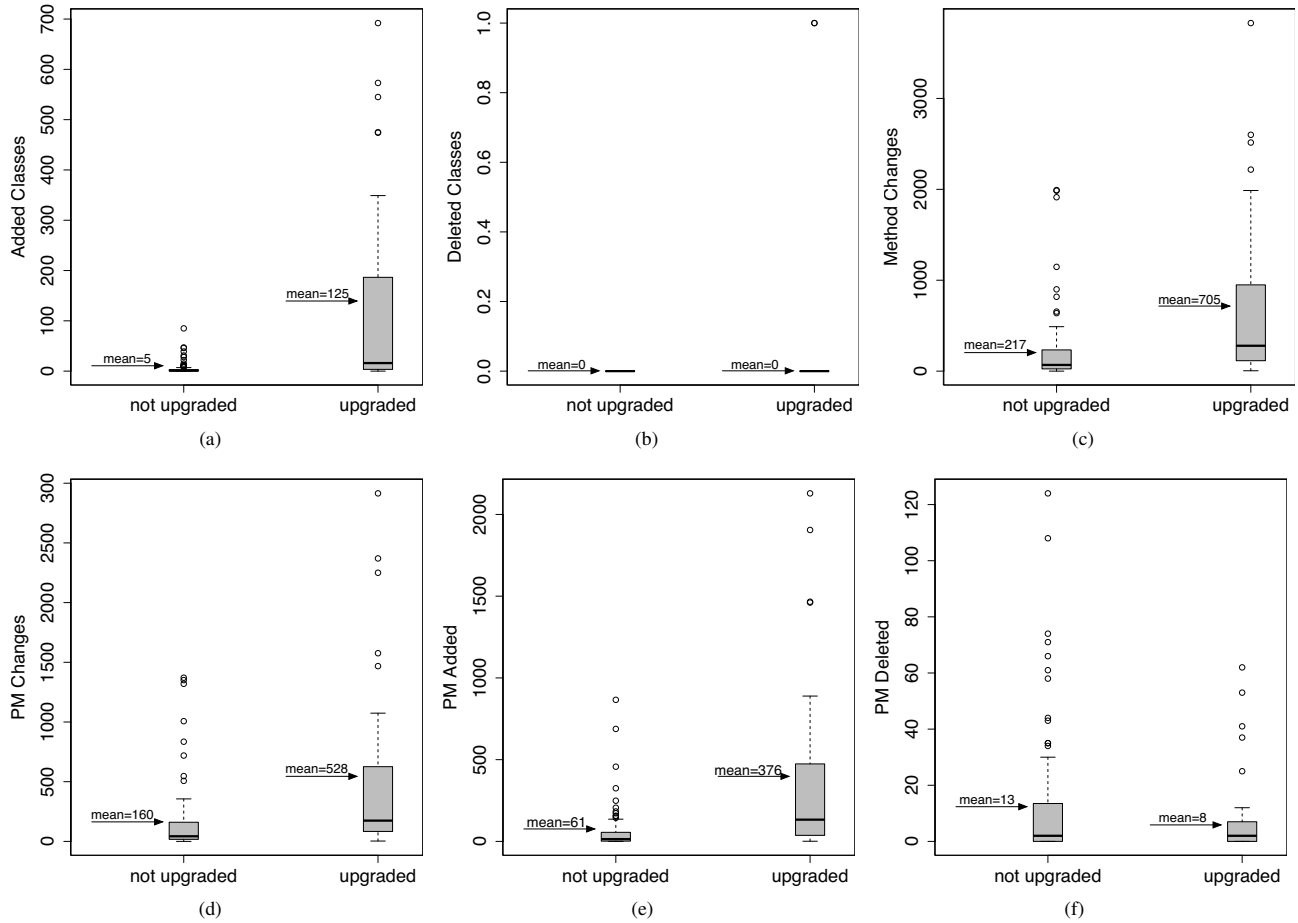


Fig. 4. Changes in upgraded and not upgraded releases.

TABLE I. CHANGES AND FIXED BUGS IN UPGRADED AND NOT UPGRADED RELEASES: MANN-WHITNEY TEST (ADJ. P-VALUE) AND CLIFF'S (d).

Tested	p-value	d
Added Classes	<0.0001	0.62 (Large)
Deleted Classes	0.51	0.05 (Small)
Method Changes	<0.0001	0.48 (Large)
PM Changes	<0.0001	0.46 (Medium)
PM Added	<0.0001	0.57 (Large)
PM Deleted	0.48	-0.01 (Small)
Fixed Bugs	<0.0001	-0.35 (Medium)

not upgraded releases) highlights substantial changes in the *upgraded releases* as compared to the *not upgraded releases*. The only change for which we did not observe a higher proportion in the *upgraded releases* are the deleted methods (-63%)–see Fig. 4(f). Note that deleted public methods mean removed services for the client projects. Thus, it is reasonable to think that client projects using the removed services tend to not upgrade the dependency towards the new release until they fix the client code in order to properly works with the new release. This could explain the lower number of deleted methods for *upgraded releases*, compared to *not upgraded releases*. However, this difference is not statistically significant (see Table I).

Concerning the number of bugs fixed in *upgraded* and *not upgraded* releases, Fig. 5 reports their distribution. On average,

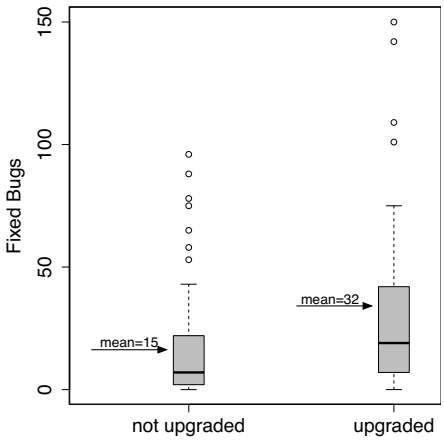


Fig. 5. Fixed bugs in upgraded and not upgraded releases.

the number of bugs fixed in the *upgraded releases* is more than two times greater than for *not upgraded releases* (32 *vs* 15). Also, this difference is statistically significant with a medium effect size (-0.35)—see Table I.

Summarizing, the results achieved highlighted that on average *upgraded releases* as compared to *not upgraded releases*:

TABLE II. ANALYSIS OF RELEASE NOTES FOR UPGRADED AND NOT UPGRADED SETS OF RELEASES.

Release type	Minor	Major	Bug fixing
not upgraded	79%	21%	82%
upgraded	59%	41%	92%

1) *include more new classes and public methods*, likely indicators of new features and services provided by the new release;
2) *underwent to more changes* performed on already existing methods, highlighting major changes in the new release;
3) *underwent to more bug fixing activities*, removing possible issues experienced by client projects when using the previous release;
4) *exhibit less deleted methods*, reducing compatibility problems for developers using them in the client code.

As explained in Section II-C, to provide further evidence to the results reported above, we inspected the release notes of both *upgraded releases* and *not upgraded releases* to understand what are the changes generally declared by developers when releasing both types of releases.

First, we found release notes of *upgraded releases* much longer than those of *not upgraded releases*. For instance, Apache log4j release notes for the six *not upgraded releases* considered in our study are composed, on average, of 676 words each, against the 2,339 words of the five *upgraded releases*. The same difference can be observed between the release notes of the four Apache Ant *not upgraded releases* having an average length of 1,417 words and those of the eight *upgraded releases* with an average of 10,476 words. This suggests that release notes for *upgraded releases* have a longer content, which often means (as confirmed by a manual analysis) describing much more novelties, improvements, and bug-fixes. For example, Apache log4j releases from 1.2.5 to 1.2.8 (4 releases), plus 1.2.11 and 1.2.12 belong to the *not upgraded releases* set. Their six release notes describe, in total, 29 bug fixes and one perfective maintenance activity, the latter being a different option to initialize the system. Instead, release notes for the five *upgraded releases* (i.e., 1.2.9, 1.2.13, 1.2.14, 1.2.16, and 1.2.17) include 123 fixed bugs, two perfective maintenance activities, and one new feature.

Among the six log4j *not upgraded* releases, five have been tagged as *minor* (83%) while one (i.e., 1.2.12) as *major* (17%). Also, all of them have been tagged as *bug fixing*. Concerning the five *upgraded releases*, two (i.e., 1.2.9 and 1.2.13) have been tagged as minor (40%), while three as *major*. Also in this case, all five releases have been also tagged as *bug fixing*.

The classification of the inspected release notes is reported in Table II. As we can see, 79% of *not upgraded releases* have been tagged as *minor*, against 59% of the *upgraded releases*, while 41% of *upgraded releases* have been tagged as *major*, against 21% of the *not upgraded releases*. Concerning the bug-fixing activities declared in release notes, overall 82% of the release notes for *not upgraded releases* have been tagged as *bug fixing*, against 92% of the *upgraded releases*.

Overall, the inspection of the release notes confirms that *client projects tend to upgrade their dependencies when substantial changes in the projects they depend on are released, including bug-fixing activities*.

TABLE III. IMPACTED SOURCE CODE COMPONENTS IN CLIENT PROJECTS.

	#Classes (%)	#KLOC (%)
Mean	58 (5%)	65 (6%)
Median	6 (1%)	12 (1%)
St. Dev.	122 (9%)	14 (12%)
Max	518 (41%)	77 (62%)
Min	1 (0%)	39 (0%)

C. RQ3: What is the impact on the client project code of an upgrade of a dependency toward a new available release of a project it depends on?

Table III reports descriptive statistics of the impacted source code of client projects upgrading one of their dependencies. The values are reported in terms of impacted number (percentage) of classes and number (percentage) of KLOCs of the client project.

On average, the impacted source code of the client project is quite limited, about 5% of the total number of classes and 6% of the KLOCs. This is quite expected, since most the dependencies a client project has are just due to few classes exploiting the services provided by this dependency. For instance, all the dependencies toward the Apache Commons projects are generally due to few methods in the client code exploiting the offered services, like the `compressors` and `archivers` services provided by the Apache Commons Compress project to manipulate archive files, or the collection of I/O utilities available in the Apache Commons IO project. Since these services support the implementation of specific tasks, it is expected that they just impact on classes having such tasks among their responsibilities.

Instead, there are some projects offering very wide services, representing crosscutting concerns exploited by a great part of the client project source code. This consideration is derived by the analysis of the row "Max" in Table III, reporting the maximum value of impacted client source code we measured in our study. This value is referred to a dependency that the project Apache Accumulo[14] (client project) has toward the project Apache Hadoop. Accumulo is a database system, while Hadoop is a framework supporting distributed processing of large data sets across clusters of computers using simple programming models. Accumulo exhibits dependencies toward Hadoop in 518 of its 1,263 classes (41%) for a total of 77 KLOCs impacted (62% of the total size). Other projects exhibiting an high impact on the client code are Apache Tomcat[15], impacting on average 23% of the client projects KLOCs, and Apache MINA[16] with an average of 10%. Again, both projects offer very generic services that could be reasonably exploited by several classes in the client projects. In fact, Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages (JSP) technologies, while Apache MINA is an application framework helping users in developing high performance and high scalability network applications.

On summary, results of $\mathbf{RQ}_3$ highlight that *the proportion of source code of client projects impacted by changes in the projects they depend on is quite limited, around 5%. However, there are specific dependencies, generally toward*

[14]http://accumulo.apache.org/
[15]http://tomcat.apache.org/
[16]http://mina.apache.org/

frameworks/libraries offering very wide, crosscutting services, that could strongly impact the client project source code when a dependency is upgraded.

IV. Threats to Validity

This section discusses the threats that can affect the validity of the results achieved. Threats to *construct validity* concern the relation between the theory and the observation. They can be mainly due to imprecisions in the measurements we performed. This is a summary of the main sources of imprecision:

- the mapping between dependencies declared within a project and other projects was performed using a set of heuristics, as explained in Section II-B. To cope with the imprecision of such heuristics, results were manually verified;
- the analysis of change impact done in $\mathbf{RQ}_3$ includes all client classes importing an API class that underwent a change. To determine whether a changed method was used or not, a fine-grained analysis would have been necessary. However, this was not our intent. Instead, we were interested to determine the potential set of clients for the changed API class, i.e., a set of classes that might need some verification/testing activities;
- analysis of licensing relies on the precision of Ninka, which is deemed to be higher than 90% [10].

Threats to *internal validity* concern factors internal to the study that could influence our results. Such kind of threats typically do not affect exploratory studies like the one in this paper. The only case worthwhile of being discussed is about $\mathbf{RQ}_2$ (reasons for upgrades) and to some extent $\mathbf{RQ}_3$ (why some changes in libraries have more impact than others). In the first case, although we have found some correlation between certain kinds of changes and upgrades decisions, we cannot claim there is a cause-effect relation. Nevertheless, we manually inspected release notes to support our findings. Similar considerations apply to $\mathbf{RQ}_3$, where the cases of large impact were fairly limited—i.e., to framework such as Accumulo and Hadoop—and it was possible to manually verify our findings.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. The analyses performed in this paper mainly have an observational nature, although we used, where appropriate ($\mathbf{RQ}_2$), statistical procedures and effect size measures to support our claims.

Threats to *external validity* concern the generalizability of our findings. Such a generalizability is clearly limited to the ecosystem being analyzed, i.e., Apache, and specifically Java projects of the Apache ecosystem. Also, in terms of assessing dependency upgrades, such assessment is confined to *within-ecosystem* dependencies, as we are not interested to analyze dependencies to projects that are not part of the ecosystem. Future studies need to be done to investigate upgrades with respect to external dependencies too, and to repeat the study on other ecosystems.

V. Related work

In recent and past years several papers have analyzed software ecosystems to investigate how and why a single project became an ecosystem of more than one software project. Authors of these works focused their attention on methods to analyze the evolution of software projects, as well as methods to extract dependencies between projects belonging to the ecosystem. In this section, we discuss studies aimed at analyzing software ecosystems. Also, we discuss studies that observed changes/deprecations of APIs and their impact on software evolution and stability.

A. Analysis of Software Ecosystems

During the software development/evolution of a software project, the complexity and dimension of the project increase in terms of (i) the number of components the software system is composed by; and (ii) number the developers teams. For example, the size of the Debian ecosystem doubles in size every 2 years [11]. In addition, large projects evolve rapidly through the evolution of a set of depending sub-projects [9], [11], [12]. This means that a software system, during its evolution, becomes part of a larger software ecosystem, developed in the context of an organization or an open-source community [18].

Software ecosystems have been studied in the last decade from several different perspectives. Lungu [18], [19] show how reverse engineering an ecosystem is a natural and complementary extension to the traditional system reverse engineering. In a previous work [20], Lungu *et al.* focused their effort on reverse engineering a software ecosystem by generating high-level views capturing various aspects of its structure and evolution.

Gonzalez-Barahona *et al.* [12] studied and analyzed the Debian Linux distribution founding that large source code data does not necessarily involve an ecosystem. However, in large software systems, knowing the dependencies between modules or components is critical to assess the impact of changes. In software ecosystems, which is composed by a collections of software projects, recover all the dependencies is not a simple problem. For this reasons several authors [21], [23] focused their effort on methods for the extraction of dependencies between projects in an ecosystem. In our study, we use some specific heuristics (see Section II-B) to identify the dependencies in the Apache ecosystem.

Grechanik *et al.* [13] have studied the structural characteristics of the source code of 2080 randomly chosen Java open source projects, by answering 32 research questions related to (i) classes and packages, (ii) constructors and methods, (iii) fields, (iv) statements, (v) exceptions, (vi) variables and basic types, and (vii) evolution/maintenance activities occurred on the projects.

The Eclipse ecosystem has been studied by several authors from different perspectives. Wermelinger *et al.* [26] identified a stable core of Eclipse plugins whose dependencies have remained stable over time. Other studies analyze the evolution of Eclipse of both core [22] and third-party Eclipse plugins [3]. In particular, Mens *et al.* [22] found that the Eclipse core plugins adhere to the laws of continuing change and growth, but not to the law of increasing complexity. Businge *et al.* [3] instead, analyzed the dependencies and the survival of 467 Eclipse third-party plugins, altogether having 1,447 versions.

They found how plugins depending on only stable and supported Eclipse APIs have a very high source compatibility success rate, compared to those that depend on at least one of the non-APIs that are those that depend on at least one of the potentially unstable, discouraged and unsupported Eclipse non-APIs. This means that third-party plugins that depend from the Eclipse ecosystems (stable and supported Eclipse APIs) have a higher source compatibility success rate than discouraged and unsupported Eclipse non-APIs. In addition, they found that the majority of plugins hosted on SourceForge do not evolve beyond the first year of release.

Recently, German *et al* [8] analyzed the evolution of the statistical computing project GNU R, with the aim of analyzing the differences between code characteristics of core and user-contributed packages. They found that the ecosystem of user-contributed packages has been growing steadily since the R conception at a significantly faster rate than core packages, yet each individual package remains stable in size. In our study, similar to the work by German *et al.* [8], we analyzed the evolution of the Apache ecosystem. Differently from R, in Apache the changes of a sub-project (package in R) happen very often during the year. This recall the need to study how and why a client project upgrades a dependency toward a new available release of a project it depends on.

Recently, Annosi *et al.* [1] proposed a framework to support developers in the upgrade of third-party components. The decision is driven by various factors, partially related to the kind of change occurred in the component (as mined from release notes or issue trackers), partially on expert judgements collected within the company. The work presented in this paper is complementary to the work of Annosi *et al.*, because it helps to identify what are the factors and events that trigger component upgrades in a large software ecosystem.

B. Analysis of API Changes

Theoretically, the API of a component should never change. In practice, when a new version of a software component is released, it is very likely that its interface changes. This requires projects that use the component to be changed before the new release of the component can be used. How and why API changes during software evolution has been studies by several authors. Dig *et al.* [7] studied the changes between two major releases of four frameworks (one proprietary and three open-source) and one library written in Java. They found that on average 90% of the API breaking changes[17] are represented by refactoring operations.

Hou *et al.* [15] analyzed the evolution of AWT/Swing at the package and class level. They found that—during 11 years of the JDK release history (i.e., since JDK 1.0 to Java SE 6)—the number of changed elements was relatively small as compared to the size of the whole API, and the majority of them happened in release 1.1. Thus, the main conclusion of their study is that the initial design of the APIs contributes to the smooth evolution of the AWT/Swing API. Raemaekers *et al.* [24] studied changes in APIs to measure the stability of the Apache Commons library. Their findings indicated that a relatively small number of new methods were added in each snapshot to the "Commons Logging" library, and there is more work going on in new methods of "Common Codec" than in old ones.

Recently, Robbes *et al.* [25] observed how much the API of a framework (or library) changes. They studied API deprecations that led to ripple effects across an entire ecosystem. The results showed that a number of API changes caused by deprecation can have a very large impact on the ecosystem and consequently on projects or developers that are impacted by the change, or the measure of the overall number of changes.

Changes in APIs and frameworks require the adaptation of clients, that can, sometimes, be automated. To this aim, Degenais and Robillard [5] proposed SemDiff, a tool to recommend client adaptation required when the used framework evolve. The authors evaluated SemDiff on the evolution of the Eclipse-JDT framework and three of its clients.

We share with the aforementioned papers the need for studying how the evolution of projects used as libraries in software ecosystems impacts on the evolution of client projects. However, instead of proposing how client projects should be adapted, we aimed at analyzing to what extent are dependencies upgraded—i.e., towards a new release of the target project—and what are the drivers of such upgrades. Our study provides some insights on the design of recommendation systems for supporting developers in the activity of library/component upgrade.

VI. Conclusion and Future Work

In this paper we analyzed the evolution of dependencies between projects belonging to the Java subset of the Apache ecosystem. Our study aims at providing some insights on (i) how the dependencies between projects composing the ecosystem change during software evolution; (ii) to what extent are dependencies upgraded, and what are the drivers of such upgrades; and (iii) how the upgrade of a dependency can impact on the source code of a project. In the context of our study, we observed the evolution of 147 projects over a period of 14 years, resulting in a total number of 1,964 releases.

Results of our study indicate that projects and their dependencies increase continuously during evolution. However, dependencies follow a different trend as compared to the number of projects of the ecosystem. Specifically, while the trend of number projects is linear, the number of dependencies between them grows exponentially. As for the upgrade of dependencies, we observed that, when a new release of a project is issued, in 69% of the cases this does not trigger an upgrade. Client projects tend to upgrade their dependencies when substantial changes in the projects they depend on are released, including major bug-fixing activities, change to API interfaces or addition of new API and features. Instead, developers are reluctant to perform an upgrade when some APIs are removed. The impact of upgrade is generally low, unless it is related to components providing features which is used in different points of a project.

On the one hand, the findings of this study allow to understand the phenomenon of software ecosystem evolution and, specifically, of library/component upgrade. Because of the addition of new features, the relationship between projects

[17] API breaking changes would cause an application built with an older version of the component to fail under a newer version.

in the ecosystem become more and more complex, and the upgrade management becomes cumbersome. On the other hand, achieving a deep understanding on when upgrades were performed and when not is useful to pose the basis for the development of smart upgrade management systems that, instead of just triggering updates whenever a new version of a component is available, are able to support the software engineer in the decision of whether to (i) perform the upgrade immediately (e.g., in case of a major release or important bug-fixing on the component), (ii) pospone it (minor fixes, that would just require to allocate effort for performing change impact analysis without any major advantage), or (iii) even avoid to perform it, e.g., when APIs are deprecated or when the upgrade can create licensing incompatibility issues.

In future work, we plan to continue studying how dependencies evolve in software ecosystems, considering other factors. For instance, the social/community aspects of ecosystems could play an important role in the evolution of dependencies between projects, i.e., the presence of joint development teams, and/or communication between teams could be important drivers for dependency upgrade as well as to promote code reuse between projects.

ACKNOWLEDGEMENTS

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, and Sebastiano Panichella are partially funded by the EU FP7-ICT-2011-8 project Markos, contract no. 317743. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] M. Annosi, M. Di Penta, and G. Tortora. Managing and assessing the risk of component upgrades. In *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pages 9–12, 2012.

[2] J. Bosh. From software product lines to software ecosystems. In *Proceedings of the 13th International Conference on Software Product Lines (SPLC)*, pages 111–119, 2009.

[3] J. Businge, A. Serebrenik, and M. van den Brand. Survival of eclipse third-party plug-ins. In *28th IEEE International Conference on Software Maintenance (ICSM 2012), Trento, Italy, Sep 23-28, 2012*, pages 368–377. IEEE Computer Society, 2012.

[4] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 481–490. ACM, 2008.

[6] M. Di Penta, D. M. Germán, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 145–154. ACM, 2010.

[7] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:83–107, 2006.

[8] D. German, B. Adams, and A. E. Hassan. Programming language ecosystems: the evolution of r. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 243–252, Genova, Italy, 2013.

[9] D. M. German, J. M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 140–149, Washington, DC, USA, 2007. IEEE Computer Society.

[10] D. M. German, Y. Manabe, and K. Inoue. A sentence-matching method for automatic license identification of source code files. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, New York, NY, USA, 2010. ACM.

[11] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 131–140, Washington, DC, USA, 2000. IEEE Computer Society.

[12] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Softw. Engg.*, 14(3):262–285, 2009.

[13] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16-17 September 2010, Bolzano/Bozen, Italy*. ACM, 2010.

[14] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.

[15] D. Hou and X. Yao. Exploring the intent behind api evolution: A case study. In *18th Working Conference on Reverse Engineering (WCRE'11), Limerick, Ireland, Oct 17-20, 2011*, pages 131–140, 2011.

[16] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. In *31st International Conference on Software Ecosystems, New and Emerging Research Track*, pages 187–190, 2005.

[17] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–716, 1966.

[18] M. Lungu. Towards reverse engineering software ecosystems. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 428–431. IEEE, 2008.

[19] M. Lungu. Reverse engineering software ecosystems. Technical report, University of Lugano, 2009.

[20] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *Sci. Comput. Program.*, 75(4):264–275, 2010.

[21] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *In Proceedings of ASE 2010*, pages 309–312. ACM Society Press, 2010.

[22] T. Mens, J. Fernández-Ramil, and S. Degrandsart. The evolution of eclipse. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 386–395. IEEE, 2008.

[23] J. Ossher, S. K. Bajracharya, and C. V. Lopes. Automated dependency resolution for open source software. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pages 130–140. IEEE, 2010.

[24] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference on Software Maintenance (ICSM'12), Trento, Italy, Sep 23-28, 2012*, pages 378–387, 2012.

[25] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 56:1–56:11, New York, NY, USA, 2012. ACM.

[26] M. Wermelinger and Y. Yu. Analyzing the evolution of eclipse plugins. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 133–136, New York, NY, USA, 2008. ACM.