An Empirical Study on the Reuse of Third-Party Libraries in Open-Source Software Development

Asimina Zaimi
Department of Information Technology Technological Education Institute, Thessaloniki, Greece

Apostolos Ampatzoglou
Department of Mathematics and Computer Science, University of Groningen, Groningen, The Netherlands
a.ampatzoglou@rug.nl

Noni Triantafyllidou
Department of Information Technology Technological Education Institute, Thessaloniki, Greece

Alexander Chatzigeorgiou
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
achat@uom.gr

Androklis Mavridis
Department of Informatics, Aristotle University, Thessaloniki, Greece

Theodore Chaikalis
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
chaikalis@uom.gr

Ignatios Deligiannis
Department of Information Technology Technological Education Institute, Thessaloniki, Greece
ignatios@it.teithe.gr

Panagiotis Sfetsos
Department of Information Technology Technological Education Institute, Thessaloniki, Greece

Ioannis Stamelos
Department of Informatics, Aristotle University, Thessaloniki, Greece
stamelos@csd.auth.gr

ABSTRACT

Software development based on third-party libraries is becoming increasingly popular in recent years. Nowadays, the plethora of open-source libraries that are freely available to developers, offer great reuse opportunities, with relatively low cost. However, the reuse process is in many cases rather ad-hoc. In this paper, we investigate reuse processes in five successful open-source projects, with respect to: (a) the extent to which software functionality is built from scratch or reused, (b) the frequency with which reuse decisions are modified, and (c) the effect of reuse on software product quality. The results of the study suggest that: (a) OSS projects heavily reuse third-party libraries, (b) reuse decisions are not frequently revisited, and (c) there is no clear evidence that reuse decisions are quality-driven.

Categories and Subject Descriptors

• **Software and its engineering ~ Software creation and management** • **Software and its engineering ~ Software evolution** • **Software and its engineering ~ Maintaining software** • *Software and its engineering ~ Object oriented development*

Keywords

Software libraries; open-source software; reuse; quality

1. INTRODUCTION

Software reuse, often defined as the use of existing engineering knowledge and artifacts to build new software systems [12], is a challenging and multifaceted topic, which attracted research interest since the late 1960s [19]. The reusable modules and classes reduce implementation time, increase the likelihood that prior testing and use has eliminated bugs and localizes code modifications when a change in implementation is required. Historically, software reuse is focused on reapplying code modules, data structures or entire applications in new software projects. Recently, however, it has been acknowledged as beneficial to redeploy software components across the entire development life cycle, starting with domain modeling and requirements specification, through software design, coding and testing, to maintenance and operation [1].

Hewlett-Packard has found that reuse can have a significant and largely positive effect on software development. Metrics drawn from two HP reuse programs document the improved quality, shortened time-to-market, and enhanced economics resulting from reuse. Because work products are used multiple times, the accumulated defect fixes result in a higher quality work product. Additionally, since reused work products have already been created, tested, and documented, productivity increases because adopters of reusable work products need to do less work [11]. In a different context, Sojer et al. [26] point out that code reuse does play a major role in OSS development; developers reported, on average, that 30 percent of the functionality they have implemented in their current main projects has been based on reused artifacts. Software reuse activities are categorized in two major types:

- **white-box reuse**, which refers to source code reuse, where the external source code is incorporated in the project files; and
- **black-box reuse**, which refers to the reuse of external libraries in binary form, where the source code is not visible and therefore, not modifiable.

According to Haefliger et al. [14], black-box reuse is the dominant type of reuse in software development. Additionally, in [4] the authors report that in 2007 over half of software developers used a

BCI'15, September 02-04, 2015, Craiova, Romania.

DOI: http://dx.doi.org/10.1145/2801081.2801087

part of open-source projects or OSS components off the self (COTS) in their most recent projects. To this end, black-box reuse of third-party libraries constitutes a field of great interest to both researchers and practitioners.

However, in order for software development companies to maximize the benefits from reuse, they should follow a specific reuse process and not perform it opportunistically [16]. For this purpose companies are expected to make reuse decisions based on a predetermined rationale, document them, update them if necessary and trace them along software evolution. Concerning black-box reuse, we catalogue three reuse decisions that software engineers could make:

- **add a third-party library** to the software system;
- **remove a third-party library** from the software system; or
- **update the version of a third-party library** of the software system.

In this paper, we first investigate the extent to which Open-Source Software (OSS) project reuse third-party libraries, second we investigate the frequency of each reuse decision, and finally we investigate possible relationships of these decisions with design-time quality attributes [10]. To achieve this goal, we perform an embedded multiple case study on five successful Java OSS projects. The rest of the paper is organized as follows: Section 2 presents related work, Section 3 discusses the case study design, Section 4 presents the results, whereas Section 5 discusses them. Finally, in Section 6 we describe the most important threats to the validity and in Section 7 we conclude the study.

2. RELATED WORK

Many earlier empirical studies have shown that systematic software reuse increases productivity [7, 18, 21] and software quality [9,12,14]. However, we will focus on those studies that quantify reuse intensity in OSS or provide empirical evidence related to our stated research questions.

Haefliger et al. [14], in a multi-case study, analyzed code reuse within six open-source projects by inspecting source code artifacts and interviewing the developers of the projects. Their study showed that all sample projects reuse software and the dominant form of reuse was black-box reuse. Similarly in another empirical multi-case study in 20 popular OSS Java projects [15], the authors investigated (1) whether open-source projects reuse third party code and (2) how much white-box and black-box reuse occurs. The results showed that reuse is common among OSS Java projects and that black-box reuse is the predominant form of reuse. Additionally, Raemaekers et al., examined a large dataset of available open source and proprietary software to identify the most frequently used third-party libraries [22]. The results suggest that logging frameworks (e.g., apache.*log4j* or apache.commons.*logging*) are the most frequently reused libraries. In a similar context, Schwittek and Eicker [24] examined the reuse intensity of third-party libraries in OSS web applications. The results suggested that web applications reuse on average 70 libraries, and that 50% of the most reused libraries come from Apache Foundation.

In [12] and [20] the authors focused only on white-box reuse, investigating and quantifying large-scale code reuse in open-source projects. They measured the overlap of filenames among OSS projects in their database of 38.7 thousand OSS projects and investigated what type of components are reused the most. The results for the studied projects showed that more than 50% of the components exist in more than one project. Moreover, data in [20] suggests that code reuse is more popular in OSS development than in the commercial closed source software. In a study on third party component reuse in Java enterprise OSS [24], the authors analyzed 36 Java web applications to measure only black-box reuse. The results showed that 70 third party components are being reused on average and 50% of the 40 most reused third party components are maintained by the Apache Foundation.

Sojer and Henkel [26] conducted a survey among 686 open-source developers to investigate the usage of existing open-source code for the development of new open-source software. More specifically they analyzed the degree of code reuse with respect to developer and project characteristics. Their results showed that an average of 30% of the implemented functionality in the projects of the survey participants is based on reused code. Another exploratory study that analyzes knowledge reuse in open-source software is reported by von Krogh et al. [17]. The authors surveyed the developers of 15 open-source projects to find out whether knowledge is reused among the projects and to identify different categories of reuse. Their study showed that all the considered projects do reuse software components.

3. CASE STUDY DESIGN

In order to explore the reuse of third-party libraries from OSS projects, we performed an embedded multiple case study on five well-known open-source software (OSS) projects provided by sourceforge[1]. The main benefits from conducting a case study is that the phenomenon under study is investigated in its real-life context, since large-scale reuse of third-party libraries cannot be easily monitored in a controlled environment. In this section we describe the case study, which was designed and reported according to the guidelines proposed by Runeson and Host [23].

3.1 Objective and Research Questions

The goal of this study, described using the Goal-Question-Metric (GQM) formulation [9], is: *"to analyze the reuse of third-party libraries from OSS projects for the purpose of evaluation with respect to:*

(a) the reuse intensity,

(b) the evolution of the reuse decisions, and

(c) the effect of the reuse on product quality,

from the point of view of software engineers in the context of OSS evolution".

Based on the abovementioned goal, we have extracted three research questions (RQs):

RQ$_1$: What extent of the system under study is based on reused third-party libraries and what extent is written from scratch?

RQ$_2$: What is the evolution of reuse decisions across time?

- **RQ$_{2.1}$**: In what percentage of the reused libraries the decision to reuse them is not revisited/unchanged during the lifetime of the software? (i.e. the library is not an updated version, not removed, not added compared to the library used in the previous version of the software),
- **RQ$_{2.2}$**: What percentage of the reused libraries are removed during the lifetime of the software?

[1] http://www.sourceforge.net/

RQ2.3: What percentage of the reused libraries are added during the lifetime of the software?

RQ2.4: In what percentage of the reused libraries is their version updated during the lifetime of the software?

RQ3: What is the effect of reuse decisions on product quality of the OSS projects?

3.2 Case and Unit Analysis

According to [23], case studies can be characterized either as holistic or embedded, based on the way they define their cases and units of analysis. This study is an embedded multiple case study, because we investigate multiple open-source projects, i.e., cases, and from each case we extract a multiple units of analysis, i.e., software versions.

3.3 Case Selection

In this study, we considered only Java projects, due to the tools used during data collection (see Section 3.4). The cases of our study have been selected so as to have more than 10 versions, and with variation in the third-party libraries that they reuse across their lifespan (i.e., versions). To this end, the following projects have been selected:

- **ArgoUML** is the leading open-source UML modeling tool and includes support for all standard UML 1.4 diagrams. In this study we explored versions 0.10 to 0.34, i.e., 19 versions.
- **dr Java** is a lightweight programming environment for Java designed to foster test-driven software development. It includes an intelligent program editor, an interactions pane for evaluating program text, a source level debugger, and a unit testing tool. In this study we examined 62 versions from 2002 until 2012.
- **Findbugs** is a static analysis tool to find bugs in Java programs. In this study we examined 10 versions of the project (from 1.2.1 to 2.0.2).
- **jFreeChart** is a free (LGPL) chart library for the Java(tm) platform. It supports bar charts, pie charts, line charts, time series charts, scatter plots, histograms, simple Gantt charts, Pareto charts, bubble plots, dials, thermometers and more. In this study we explored 52 versions, i.e., from version 0.5.6 until 1.0.14.
- **Mogwai** is a Java 2D & 3D tool for visualizing entity relationship design and modeling (ERD, SQL). We have examined 25 versions of the *ER_Designer* component, i.e., from 1.0 until 3.0.0.

3.4 Data Collection

For every unit of analysis various data points have been extracted, as shown below:

[V1] Number of reused third-party libraries;

[V2] Percentage of OSS functionality offered by reused third-party libraries (i.e., 100 * $DSC_{libraries}$ / DSC_{system})[2];

[V3] Reused third-party libraries that have remained unchanged (both retained in the project and with the same library version) compared to the previous version;

[V4] Reused third-party libraries that have been removed from previous version;

[V5] Reused third-party libraries that have been added from previous version;

[V6] Reused third-party libraries whose versions have been updated from previous version; and

[V7] Reused third-party libraries quality attribute (QA) metric scores (for QAs and metrics descriptions see below);

To quantify the design quality of classes, we used the Quality Model for Object-Oriented Design (QMOOD) [8]. QMOOD is a hierarchical quality model that assesses six high-level quality attributes (i.e., flexibility, effectiveness, extendibility, reusability, functionality, and understandability). To assess these attributes QMOOD provides a model based on several object-oriented (OO) properties (i.e., complexity, coupling, cohesion, design size, hierarchies, abstractions, messaging, encapsulation, composition, inheritance, and polymorphism). The definitions of the above-mentioned quality attributes and properties, and the equations used to calculate the score of each quality attribute, as defined by Bansiya and Davis, can be found in [8].

To automate the process of quality assessment (i.e., the calculation of metrics) for each project version we used Percerons Client[3]. Percerons is a software engineering platform [5], created by one of the authors, to facilitate empirical research in software engineering, by providing:

- identification of componentizable parts of source code [6],
- quality assessment [3], and
- design pattern instances [5].

The platform has been used for similar reasons in [2, 3, 13]. The extraction of variables [V1] to [V6] have been performed manually by the first author, and double-checked by the third. In particular, since the examined projects included the reused third-party libraries by placing them in a separate folder, it has been straightforward to extract the corresponding dependencies. The obtained data has been made accessible in the web[4].

3.5 Data Analysis

In order to explore the research questions set in section 3.1, we will perform descriptive statistical analysis and hypothesis testing. The analysis plan, per research question, is presented in Table 1.

Table 1. Data Analysis Plan

Research Question	Variables	Analysis
RQ_1	[V1] [V2]	Descriptive Statistics Line Chart
$RQ_{2.1}$	[V3]	Descriptive Statistics Line Chart
$RQ_{2.2}$	[V4]	Descriptive Statistics Line Chart
$RQ_{2.3}$	[V5]	Descriptive Statistics Line Chart
$RQ_{2.4}$	[V6]	Descriptive Statistics Line Chart
RQ_3	[V6] [V7]	Descriptive Statistics Paired-Sample t-test

[2] DSC: Design Size in Classes

[3] http://www.percerons.com

[4] http://se.uom.gr/portfolio/BCI_2015_third-party-libraries-oss

For answering RQ_1 and RQ_2 (and all of its sub-research questions), we followed a similar process:

- we present basic descriptive statistics (i.e., min, max, mean, and standard deviation) for the variable of interest, for each one of the cases separately;
- we visualize the evolution of the variable of interest, across all available project versions, for every case separately. We note that although a scatter plot might appear a more fitting representation for the time series of all research questions, we have preferred to perform visualization through line charts to improve the readability of the diagram.

For answering RQ_3, we first applied a filtering process (see below) and then applied hypothesis testing on the corresponding variables. The analysis strategy for answering RQ_3, is as follows:

- for all projects, we filtered pairs of successive versions, in which only one type of reuse decision was applied (i.e., only addition of libraries, only removal of libraries, only update of library version);
- for each type of reuse decision, we applied hypothesis testing (paired sample t-test) for every QA under study (i.e., flexibility, understandability, effectiveness, extendibility, reusability, and functionality). As pair we consider the value of the QA metric score, before and after the application of the reuse decision.

4. RESULTS

In this section we will present the results of our case study, organized by research question.

4.1 RQ_1: Library Reuse Intensity

Based on our data analysis planning, in order to answer RQ_1, we:

- quantify reuse in terms of the total number of third-party libraries that are reused in our five cases (i.e., OSS projects), and present descriptive statistics concerning all units of analysis (i.e., version) extracted for each case (see Table 2);
- quantify the percentage of the total number of classes reused from third-party libraries w.r.t. the total number of system classes (see Table 3); and
- graphically depict the evolution of the two aforementioned measures (see Figure 1 and Figure 2, respectively).

Table 2. Number of reused libraries

Project	Min	Max	Mean	Std. Dev.
dr Java	4	17	10.33	3.564
Findbugs	10	17	14.30	2.452
ArgoUML	6	37	19.42	10.297
jFreeChart	0	6	3.75	1.792
Mogwai	21	76	41.12	12.112

From the results of Table 2 and Figure 1, we can observe that the five OSS projects that we have studied are reusing third-party libraries with an increasing trend across time. In the final version, four projects reuse more than 15 libraries, whereas one project (i.e., *jFreeChart*) is reusing only six third-party libraries. A possible explanation for this is the fact that *jFreeChart* is itself a library that has to provide functionalities to other systems.

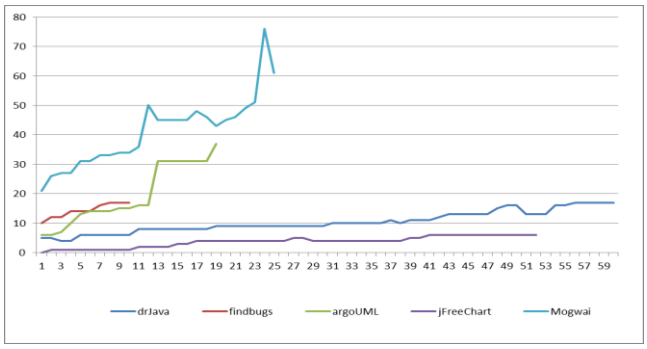

Figure 1. Evolution of Number of Reused Libraries

Additionally, from the results of Table 3 and Figure 2, we cannot observe a similar trend. Specifically, the relative size of libraries (compared to the total size of the system), in terms of classes, is not uniformly increasing or decreasing over time. In the peak of reuse intensity, most systems are basing 70% of their provided functionality on third-party libraries, whereas there is one project case (i.e., *Mogwai*), which reuses around 97% of its classes.

Table 3. Relative Reused Library Size

Project	Min	Max	Mean	Std. Dev.
dr Java	40.0%	72.9%	51.8%	9.14%
Findbugs	57.3%	64.6%	60.1%	2.38%
ArgoUML	35.6%	73.8%	54.2%	10.34%
jFreeChart	31.3%	66.0%	54.0%	7.18%
Mogwai	95.5%	99.5%	97.5%	2.00%

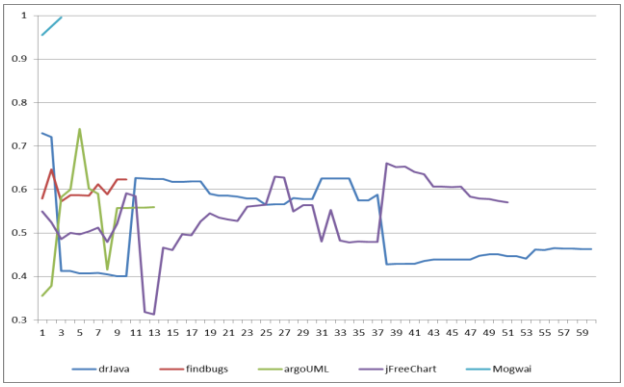

Figure 2. Evolution of the Relative Size of Reused Libraries

4.2 RQ_2: Reuse Decisions

Based on our data analysis planning, in order to answer RQ_2, we:

- quantify the percentage of reused libraries that remain unchanged across successive versions of an OSS project (see Table 4 for descriptive statistics, and Figure 3 for the evolution);
- quantify the percentage of reused libraries that have been removed between successive versions of an OSS project (see Table 5 for descriptive statistics, and Figure 4 for the evolution);
- quantify the percentage of reused libraries that have been added between successive versions of an OSS project (see Table 6 for descriptive statistics, and Figure 5 for the evolution); and

- quantify the percentage of reused libraries that have been updated between successive versions of an OSS project (see Table 7 for descriptive statistics, and Figure 6 for the evolution);

Table 4. Percentage of libraries remaining unchanged

Project	Min	Max	Mean	Std. Dev.
dr Java	50.0%	100%	93.0%	13.31%
Findbugs	25.0%	100%	82.8%	26.09%
ArgoUML	21.4%	100%	80.3%	24.15%
jFreeChart	0.0%	100%	57.0%	31.33%
Mogwai	58.3%	100%	93.1%	11.30%

By answering $RQ_{2.1}$, we observe that the majority (i.e., 80% - 93%) of the libraries are remaining unchanged between successive versions of the software, for four out of five cases (except *jFreeChart*). Therefore, when a library is imported in a system, it is rather unlikely to be removed, or updated to a more up-to-date version. On the other hand, concerning *jFreeChart*, we observe that in its early days developers experimented with the libraries that will be included (below 70% of unchanged libraries), while later they appear to finalize those that will be reused.

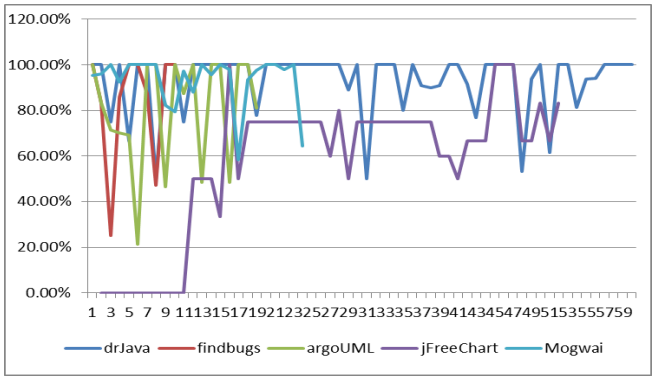

Figure 3. Evolution of the Number of Stable Libraries

Concerning the removal of libraries from one version of the system to the other, we observe that the number of removals is rather limited (1.5% - 2% for three projects). However, in the dataset, we can identify some extreme case, when more than 20% of the libraries from one version have been removed to the next one. Such extreme peaks in Figure 4, especially in cases when they are accompanied with similar peaks in the previous version in Figure 5, denote possibly unsuccessful mass reuse attempts that stayed only for one version in the project. On the other hand, as an extreme example from the opposite side, we observed that *jFreeChart* has removed no library for almost 50 versions.

Table 5. Percentage of removed libraries

Project	Min	Max	Mean	Std. Dev.
dr Java	0.0%	25.0%	1.9%	5.7%
Findbugs	0.0%	8.3%	1.4%	3.1%
ArgoUML	0.0%	35.7%	1.9%	8.2%
jFreeChart	0.0%	0.0%	0.0%	0.0%
Mogwai	0.0%	32.9%	3.7%	7.9%

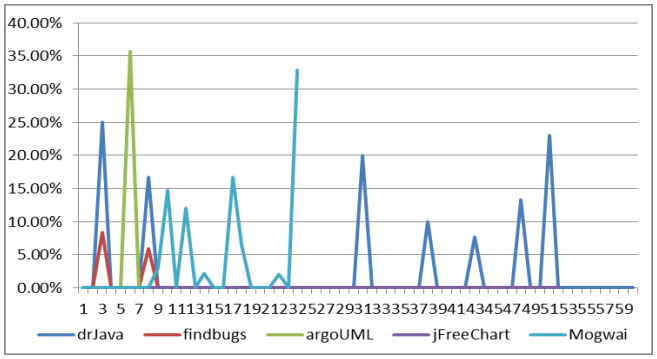

Figure 4. Evolution of the Number of Removed Libraries

Additionally, concerning the addition of third-party reused libraries along software evolution, one would expect that the addition of libraries would decrease over time, since the project matures. However, this is the case only for *jFreeChart*, whereas for the rest of the cases we observe peaks of similar size during the complete project evolution. The average addition of libraries for all cases varies from around 3% to 9% along their evolution.

Table 6. Percentage of added libraries

Project	Min	Max	Mean	Std. Dev.
dr Java	0.0%	33.3%	3.8%	8.4%
Findbugs	0.0%	16.7%	6.4%	7.0%
ArgoUML	0.0%	48.4%	9.2%	15.5%
jFreeChart	0.0%	50.0%	3.3%	9.9%
Mogwai	0.0%	49.0%	9.1%	12.7%

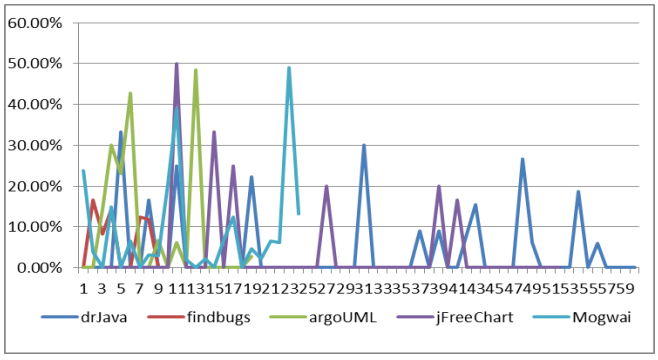

Figure 5. Evolution of the Number of Added Libraries

Finally, by answering RQ2.4, we suggest that developers only rarely update an existing library to a more up-to-date version. Similarly to other sub-questions regarding reuse decisions, *jFreeChart* is the only software, whose developers consistently update libraries (on average around 37%). On the other hand, the rest four systems update the versions of their libraries with a frequency between 1% and 9%. However, by taking into account the peaks demonstrated in Figure 6 (i.e., possible outliers), we can guess that the normal library update rate is even lower.

Table 7. Percentage of updated libraries

Project	Min	Max	Mean	Std. Dev.
dr Java	0.0%	20.0%	1.3%	4.2%
Findbugs	0.0%	58.3%	9.4%	20.5%
ArgoUML	0.0%	51.6%	8.6%	15.5%
jFreeChart	0.0%	100%	37.3%	29.7%
Mogwai	0.0%	25.0%	3.1%	5.8%

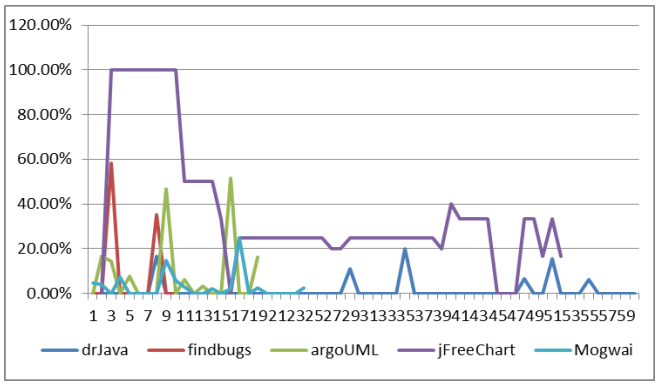

Figure 6. Evolution of the Number of Updated Libraries

4.3 RQ3: Reuse Decisions and Quality

Based on the aforementioned analysis strategy for answering RQ3, we have been able to isolate:

- 3 cases when only `remove library decisions` have been taken (see Table 8);
- 10 cases when only `add library decisions` have been taken (see Table 9); and
- 20 cases when only `update library version decisions` have been taken (see Table 10).

We note that the frequency of *update library version decisions* in this section is higher than the frequency of the other decisions, because we filtered version transitions, where only one type of decision was made. Therefore, since in many cases *remove library* and *add library* decisions were made in the same transition, such cases have been omitted, in the sense that the effect of the two decisions could not be separated. The results on the hypothesis testing concerning the aforementioned cases, as extracted by SPSS, are presented in Tables 8 - 10.

Table 8. Effect of *Remove Library Decisions* on Quality

Project	t-value	sig.	Mean before - after
Reusability	1.924	.194	2493.62 2109.92
Functionality	1.930	.193	1251.84 1054.69
Extendibility	-1.023	.414	-1.16 -1.02
Understandability	-1.180	.359	-1313.48 -1175.53
Effectiveness	-1.308	.321	0.08 0.09
Flexibility	-.902	.462	-1.55 -1.46

Table 9. Effect of *Add Library Decisions* on Quality

Project	t-value	sig.	Mean before - after
Reusability	-1.422	.189	858.14 1272.18
Functionality	.469	.650	855.99 634.64
Extendibility	-.826	.430	-0.68 -0.44
Understandability	1.417	.190	-376.57 -644.60
Effectiveness	.750	.472	0.18 0.17
Flexibility	-.479	.644	-1.24 -1.23

Table 10. Effect of *Update Library Version Decisions* on Quality

Project	t-value	sig.	Mean before - after
Reusability	-1.025	.318	306.62 314.06
Functionality	-.929	.365	153.83 156.98
Extendibility	-1.442	.166	0.03 0.07
Understandability	1.097	.286	-136.83 -134.74
Effectiveness	-1.525	.144	0.30 0.32
Flexibility	-.886	.387	-0.38 -0.36

The results of Tables 8 - 10, suggest that there is no statistically significant effect of reuse decisions to design-time quality attributes. The most important findings of RQ3, concern the update library version decisions, which suggest the new version of the library is on average of better quality than the previous one. However, none of these results are statistically significant, and therefore require further investigation.

5. DISCUSSION

In this section, we discuss the main finding of this study, from two perspectives: (a) their interpretations, and (b) the implications that they provide to both researchers and practitioners.

5.1 Interpretation of results

Most of the results of our study can be considered expected in the sense that they are either intuitive or in accordance to the existing literature. Specifically, the suggestion that:

- ***reuse intensity is increasing over time***, in terms of number of reused libraries, is intuitive, in the sense that developers, in order to implement new functionalities are in need of including more libraries in the systems
- ***the majority of reused decisions are not revisited after their establishment*** can be supported, by two possible facts: (a) the lack of a clear reuse process in many OSS projects – leading in many cases to opportunistic reuse, and (b) the fact that once a functionality is added to the system, it is highly unlikely to be removed.
- ***library removal is sparse*** can by supported by fact (b) of the previous bullet. In cases when ***massive library removals*** occur, the most possible reason is not the removal of a functionality, but a reconsideration of a reuse decision in the previous version, i.e., the addition of many libraries that did not fit well into the project. For example, at some point the *Mogwai* developers included the *jOGL* native libraries for linux, solaris and windows systems (although the functionality was already provided by *jogl-1.1.1*); and removed those libraries in exactly the next version of the system, probably due to revisiting the decision of working with native libraries. Library removal occurs in most of the cases simultaneously with library addition, implying a ***library substitution***.
- ***library versions update*** is also sparse, probably because of the opportunistic way that reuse is performed in OSS projects. In

other words, assuming that an employed library offers the required functionality that is being sought, the developers rarely consider the possibility of updating to a new, enhanced version.

- ***jFreeChart*** appears to be a project with a clear reuse strategy (i.e., regular update of libraries when newer versions arrive, experimentation with new libraries in the beginning of the project and gradual stabilization of reused functionalities), probably because *jFreeChart* is itself a framework.

5.2 Implications to researchers

The results of the study have pointed out several interesting future research opportunities and implications for researchers, as follows:

- The coarse-grain evaluation of reuse intensity in terms of ***library size in classes against system size in classes***, was not able to capture any trends. Therefore, it is suggested for researchers to investigate research intensity in terms of actual method calls, or actual number of reused classes.
- ***jFreeChart*** proved to be an OSS project that can be used as subject in future research efforts concerning reuse, in the sense that the results of our study imply that reuse is performed systematically by the developers of this project.
- The only ***reuse decision that seemed to be related to design-time quality attributes*** appears to be the update library version decisions. However, the results of this study were not statistically significant, possibly due to the small size of our sample. Therefore, researchers are encouraged to further investigate the subject. Specifically, design-time qualities like ***reusability*** and ***functionality*** are expected to be affected. On the contrary, since libraries are in most of the cases (at least in Java) reused through black-box approaches, ***extendibility, understandability, effectiveness*** and ***flexibility*** should not be considered a priority.

5.3 Implications to practitioners

Concerning practitioners, the results of the study have mainly pointed out implications related to reuse decisions and processes. Specifically, we encourage practitioners to:

- regularly ***revisit their decisions***. Specifically, they are advised to check for more up-to-date versions of the reused libraries since they are expected to be more thoroughly tested, provide more functionality, and may be developed with higher standard of quality. Also, they are encouraged to seek for opportunities for library substitution (i.e., replace one library with another), in the sense that the plethora of OSS third-party libraries provides excellent reuse opportunities.
- ***apply reuse more systematically***. Software engineers are encouraged to be cautious when importing a library in a project, in the sense that in our dataset, we have identified several cases when a large amount of libraries was reused in one version of the system and entirely removed in the immediately following one. This observation highlights some decisions that have not been properly weighted before their application.
- ***elaborate the reuse process***. Software reuse is a decision making process that would benefit from applying practices from other more mature domains. For example, decision documentation, traceability and sharing are actively discussed in the field of architecture and their benefits could be transferred to the reuse community.

6. THREATS TO VALIDITY

In this section we present and discuss threats to the construct validity, reliability, and external validity of this study. Internal validity is not applicable, as the study does not examine causal relationships. Construct validity reflects the mapping between the research questions and the measures that are used for answering them. Reliability concerns the case study design, and specifically if it is reported in a way facilitating its replication. Finally, external validity deals with possible threats when generalizing the findings derived from the examined sample to the entire population.

Concerning construct validity, we have identified two threats. First, in the second part of RQ_1, as a measure for reuse intensity, we use the ratio of the reused classes (library size) against the total system classes. This way of measurement is rather coarse-grained, in the sense that in many cases, only a small fraction of an imported library is actually reused. However, this strategy has not lead to any valuable conclusion and therefore the reported conclusions are not threatened. Second, the formulas, proposed by Bansiya and Davis [8], for assessing QAs, can pose an additional threat to construct validity. However, in the original introduction of the QMOOD model, the authors have validated it through an empirical study involving experienced practitioners.

In order to mitigate reliability, two different researchers were involved in the data collection phase, having all outputs double-checked. Also, the reporting of the case study protocol is presented in detail in this paper. These two mitigation actions make the case study results reproducible and the case study process replicable.

Additionally, concerning external validity, we have identified two possible threats to the validity of our results. First, all software systems that have been investigated are written in Java, thus, there is a possibility that results are different for other object-oriented languages, as well as for other paradigms. Second, since the number of cases in our study is rather limited, further validation is required to increase the confidence in the observed findings.

Finally, the fact that software quality has been assessed only through the perspective of design-time quality attributes (i.e., flexibility, effectiveness, extendibility, reusability, functionality, and understandability), excluding run-time qualities (e.g., correctness, performance, reliability, etc.) poses a limitation to the study. Therefore, replicating the study by taking into account different quality attributes, is deemed very valuable.

7. CONCLUSIONS

Nowadays, reuse is a standard procedure in modern software development. The most frequent method for reusing existing code is the incorporation, in systems under development, of third-party libraries, through black-box reuse. Although reuse constitutes a common activity in the software development lifecycle, its application process is far from being standardized.

In this paper, we investigate reuse processes, and more specifically reuse intensity and reuse decisions, as applied in the long-term development of five well-known OSS projects. The results of the study suggested that reusing third-party libraries is intensified along systems' evolution, but in a rather opportunistic way. Specifically, we have observed that:

- reuse decisions are not revisited along evolution,

- systems are not moving to more stable stages (in terms of the libraries they reuse) across time,
- cases when massive mishaps in reuse have been identified, i.e., large number of libraries are reused in one version of the system and all of them are removed in the next version of the system, and
- library substitution (i.e., replacing one library with another one) is not a common phenomenon.

The aforementioned results have been compiled to implications for researchers and practitioners, in terms of interesting future research directions and reuse process improvement suggestions.

ACKNOWLEDGMENTS

This research work is co-founded by the European Social Fund and National Resources, ESPA 2007-2013, EDULLL, "Archimedes III" program.

8. REFERENCES

[1] Aggarwal, D. and Naveeta, M. 2012. Software Reuse: A Compendium. *International Journal of Research in IT & Management*. 2, 2 (Feb. 2012), 93–100.

[2] Alhusain, S., Coupland, S., John, R. and Kavanagh, M. 2013. Towards machine learning based design pattern recognition. *2013 13th UK Workshop on Computational Intelligence (UKCI)* (Sep. 2013), 244–251.

[3] Ampatzoglou, A., Gkortzis, A., Charalampidou, S. and Avgeriou, P. 2013. An Embedded Multiple-Case Study on OSS Design Quality Assessment across Domains. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (Oct. 2013), 255–258.

[4] Ampatzoglou, A., Kritikos, A., Kakarontzas, G. and Stamelos, I. 2011. An empirical investigation on the reusability of design patterns and software packages. *Journal of Systems and Software*. 84, 12 (Dec. 2011), 2265–2283.

[5] Ampatzoglou, A., Michou, O. and Stamelos, I. 2013. Building and mining a repository of design pattern instances: Practical and research benefits. *Entertainment Computing*. 4, 2 (Apr. 2013), 131–142.

[6] Ampatzoglou, A., Stamelos, I., Gkortzis, A. and Deligiannis, I. 2012. A Methodology on Extracting Reusable Software Candidate Components from Open Source Games. *Proceeding of the 16th International Academic MindTrek Conference* (New York, NY, USA, 2012), 93–100.

[7] Baldassarre, M.T., Bianchi, A., Caivano, D. and Visaggio, G. 2005. An industrial case study on reuse oriented development. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05* (Sep. 2005), 283–292.

[8] Bansiya, J. and Davis, C.G. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*. 28, 1 (Jan. 2002), 4–17.

[9] Basili, V., Caldiera, G. and Rombach, H.D. 2002. Goal Question Metric (GQM) Approach. *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc.

[10] Bass, L., Nord, R., Wood, W., Zubrow, D. and Ozkaya, I. 2008. Analysis of architecture evaluation data. *Journal of Systems and Software*. 81, 9 (Sep. 2008), 1443–1455.

[11] Constantinou, E., Ampatzoglou, A. and Stamelos, I. 2015. Quantifying reuse in OSS: A large-scale empirical study. *International Journal of Open Source Software and Processes*. 5, 3 (2015).

[12] Frakes, W.B. and Fox, C.J. 1996. Quality Improvement Using A Software Reuse Failure Modes Model. *IEEE Trans. Softw. Eng.* 22, 4 (Apr. 1996), 274–279.

[13] Griffith, I. and Izurieta, C. 2014. Design Pattern Decay: The Case for Class Grime. *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2014), 39:1–39:4.

[14] Haefliger, S., von Krogh, G. and Spaeth, S. 2007. Code Reuse in Open Source Software. *Management Science*. 54, 1 (Nov. 2007), 180–193.

[15] Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B. and Irlbeck, M. 2011. On the Extent and Nature of Software Reuse in Open Source Java Projects. *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse* (Berlin, Heidelberg, 2011), 207–222.

[16] Jansen, S., Brinkkemper, S., Hunink, I. and Demir, C. 2008. Pragmatic and Opportunistic Reuse in Innovative Start-up Companies. *IEEE Software*. 25, 6 (Nov. 2008), 42–49.

[17] Von Krogh, G., Spaeth, S. and Haefliger, S. 2005. Knowledge Reuse in Open Source Software: An Exploratory Study of 15 Open Source Projects. *Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005. HICSS '05* (Jan. 2005), 198b–198b.

[18] Lim, W.C. 1994. Effects of reuse on quality, productivity, and economics. *IEEE Software*. 11, 5 (Sep. 1994), 23–30.

[19] McIlroy, D. 1968. Mass-Produced Software Components. *Proceedings of NATO Software Engineering Conference* (Garmisch, Germany, Oct. 1968), 138–155.

[20] Mockus, A. 2007. Large-Scale Code Reuse in Open Source Software. *Emerging Trends in FLOSS Research and Development, International Workshop on*. 0, (2007), 7.

[21] Morisio, M., Romano, D. and Stamelos, I. 2002. Quality, productivity, and learning in framework-based development: an exploratory case study. *IEEE Transactions on Software Engineering*. 28, 9 (Sep. 2002), 876–888.

[22] Raemaekers, S., van Deursen, A. and Visser, J. 2012. An Analysis of Dependence on Third-party Libraries in Open Source and Proprietary Systems. *Sixth International Workshop on Software Quality and Maintainability* (2012).

[23] Runeson, P., Host, M., Rainer, A. and Regnell, B. 2012. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley.

[24] Schwittek, W. and Eicker, S. 2013. A Study on Third Party Component Reuse in Java Enterprise Open Source Software. *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering* (New York, NY, USA, 2013), 75–80.

[25] Selby, R.W. 2005. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*. 31, 6 (Jun. 2005), 495–510.

[26] Sojer, M. and Henkel, J. 2010. Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. *Journal of the Association for Information Systems*. 11, 12 (2010), 868–901.