Quantifying Reuse in OSS: A Large-Scale Empirical Study

Eleni Constantinou, *Aristotle University of Thessaloniki, Greece*
Apostolos Ampatzoglou, *University of Groningen, Netherlands*
Ioannis Stamelos, *Aristotle University of Thessaloniki, Greece*

ABSTRACT

Reuse is an established software development practice, whose benefits have attracted the attention of researchers and practitioners. In order for software reuse to advance from an opportunistic activity to a well-defined, systematic state of practice, the reuse phenomenon should be empirically studied in a real-world environment. To this end, OSS projects consist a fitting context for this purpose. In this paper, we aim at assessing the: (a) strategy and intensity of reuse activities in OSS development, (b)effect of reuse activities on design quality, (c) modification of reuse decisions from a chronological viewpoint and (d) effect of these modifications on software design quality. In order to achieve these goals, we performed a large-scale embedded multi-case study on 1,111 Java projects, extracted from Google Code repository. The results of the case study provide a valuable insight on reuse processes in OSS development, that can be exploited by both researchers and practitioners.

Keywords: Empirical Study, Open Source Software, Reuse Type, Reuse Decisions, Project Evolution

INTRODUCTION

Software reuse refers to the incorporation of existing software artifacts to systems, different than the ones that they have been originally developed for (Krueger, 1992). Nowadays, it is generally accepted that reuse practices are beneficial for systems' development since they increase productivity and potentially improve system product quality (Lim, 1994; Frakes & Kang, 2005; Ajila & Wu, 2007). In this context, the reuse opportunities that open source software repositories provide, through the enormous amount of available software projects, are considered an interesting research domain (Marri, Thummalapenta, & Xie, 2009). However, in order for the aforementioned OSS reuse benefits to be fully exploited, a systematic process is needed. As a first step towards such a systematic approach we consider the understanding of the underlying reuse mechanisms that exist in OSS projects.

Reuse is established as a common phenomenon during software projects' development (Heinemann, Deissenboeck, Gleirscher, Hummel, & Irlbeck, 2011). In order to understand the phenomenon of OSS reuse, the different types of reuse must be investigated. Software reuse is discriminated in three types: white-box, black-box and glass-box reuse. The first case, i.e. white-box reuse, refers to source code reuse, where the external source code is incorporated in the project files. In this case, the internal structure of the reused source code is exposed to the developers, and the source code can potentially be modified (Capiluppi, Stol, & Boldyreff, 2011). The second case, i.e. black-box reuse, refers to the reuse of external libraries in binary form, where the source code in not visible and therefore, not modifiable. The third case, i.e. glass-box reuse, refers to the reuse of the source code, but without changes. Since white-box reuse can entail additional effort for modification and maintenance (Sametinger, 1999), it is expected to occur less frequently compared to the other two types of reuse (Schwittek & Eicker, 2013). In cases of white-box reuse, maintenance effort is allocated both to project elements, i.e. elements internally developed by the project members, and reused elements. An indicator of the required maintenance effort is considered to be the design quality of the system (Bauer, Heinemann, & Deissenboeck, 2012). Although the design quality of OSS systems depends on additional factors, e.g. project maturity, we investigate the effect of reuse practices on systems' design quality. Thus, we measure projects' structural quality and investigate the effect of reused elements to the overall product quality.

A software reuse process includes the identification of candidate reuse artefacts and, whenever possible, their incorporation in systems under development. However, as projects evolve over time, their requirements and desired functionality change as well (Koch & Schneider, 2002). Therefore, engineers need to revisit their decisions on the employed reused elements, so as to examine if they need to be substituted, maintained, or removed (Bangerth & Heister, 2013). The modification of reused components can occur for several reasons: removal of reused functionality, upgrade of reused library or substitution of reused library (Bangerth & Heister, 2013). In this work, we investigate how the reuse decisions evolve over time, in order to provide insights on how real-world reuse occurs. More specifically, we argue that reuse modification decisions indicate how systematically reuse practices are applied throughout projects' lifecycle (Yu, 2006). Nonetheless, the reuse modification decisions entail additional effort during development. However, the aforementioned argument is not evaluated from previous studies, because of their research setup. Until now, studies that measure OSS reuse examine reuse practices in several projects, but only for isolated versions. In this study we measure the frequency and magnitude of such modifications. As a first approach to comprehend the rationale behind such modifications, we investigate if they are quality-driven, i.e. if the developers' focus on the design quality of the newly incorporated reused elements. The implications of modifying the reused elements during projects' evolution is studied by Dietrich, Jezek, & Brada, 2014 and Raemaekers, van Deursen, & Visser, 2012, although from a different perspective. More specifically, these studies focus on incompatibility issues that arise from library upgrades.

To sum up, this paper elaborates on the existing empirical knowledge on software reuse by providing the following contributions:

- **c1:** We extend empirical knowledge by quantifying software reuse in a large number of software projects. Therefore, we gain further insight on reuse practices (intensity, type of reuse, and effect on design quality) for projects of different scale, maturity and history.
- **c2:** We compare the design quality of white-box reused elements to the design quality of project elements. Subsequently we investigate how the incorporation of reused elements affects the overall quality of the system, i.e. the projects' design quality with respect to the magnitude of reuse.
- **c3:** We investigate the modification of reuse decisions over systems' successive versions for each reuse strategy. Thus, we show empirical evidence on developers' effort allocated on revisiting reuse decisions during projects' evolution.
- **c4:** We investigate the design quality of the modified sets of white-box reused elements across projects' versions. Consequently, we explore if the reuse decisions' modification positively contributes to the design quality of the existing software.

The rest of this paper is organized as follows. In the next section we describe related work on measuring reuse in OSS, and in the following section we describe the design of this case study. Next, we present the case study results and afterwards, we discuss our findings. This is followed by discussing the implications to researchers and practitioners. Next, we present the threats to validity of this work and finally, we discuss future research directions and conclude.

RELATED WORK

As related work for this study, we report research efforts that attempt to quantify reuse intensity or provide empirical evidence related to any of the aforementioned contributions. In order to increase the readability of this section we organize the presentation of related studies based on if they are focused on a specific type reuse or not.

Concerning studies that investigate only one reuse type, in (Mockus, 2007), the author investigates and quantifies source code reuse in OSS. He focuses on white-box reuse and more precisely, he identifies large sized components that are reused among several projects, since larger components are more likely to contain a more complete set of functionality. The algorithm used to quantify reuse, is based on the identification of directories in projects' source code tree that share several file names and checks if the fraction of shared file names exceeds a predefined threshold. The results were derived from 38.7 thousand projects, consisted of 5.3 million files, and showed that more than 50% of the files were used in more than one project. In (Schwittek & Eicker, 2013) the authors analyze 36 Java web applications to measure black-box reuse, since white-box reuse is rather limited. The results suggest that such applications reuse 70 external components on average, while 50% of the 40 most reused third party components are maintained by the Apache Foundation.

Research efforts that explore both black-box and white-box reuse are conducted by Haefliger, von Krogh, & Spaeth, 2008 and Heinemann, Deissenboeck, Gleirscher, Hummel, & Irlbeck, 2011. More specifically, in (Haefliger, von Krogh, & Spaeth, 2008) the authors present a multi-case study, in which they investigate code reuse by analyzing the source code artifacts and interviewing the developers of 6 projects. The results suggest that in all six projects, reuse activities are performed, while black-box reuse is the predominant type of reuse. Similarly, in (Heinemann, Deissenboeck, Gleirscher, Hummel, & Irlbeck, 2011), the authors conducted an empirical multi-case study in 20 popular OSS projects, in order to measure software reuse. They used static analysis and clone detection analysis to identify black-box and white-box reuse respectively. Their results suggested that reuse is common among OSS projects and that in most of the cases developers perform black-box reuse. However, they report that a threat to the validity of their results is that in order to quantify white-box reuse, they selected 22 commonly used libraries and focused only on them. Thus, the results might omit additional white-box reuse cases.

Table 1 compares existing empirical studies to our work, with respect to research design. The first two columns indicate which type of reuse is studied. The third and the fourth column specify if multiple versions are examined for each project and the number of distinct projects that are investigated, respectively. The fifth column describes how reuse is measured in each empirical study. By comparing the contributions of our study with related work, all studies provide evidence regarding contribution c1, whereas to the best of our knowledge this is the first study exploring contributions c2-c4. Finally, we note that in this study we randomly selected a large number of software projects to measure reuse, while related studies mainly focus on mature and popular OSS projects. Therefore, our results extend current knowledge on reuse activities, since we present findings on OSS activities regardless of projects' maturity or popularity.

Table 1. Empirical Studies on Software Reuse

	Research Design				
	Black-box Reuse	***White-box Reuse***	***Multiple Versions per Project***	***Projects***	***Reused Elements Identification***
(Mockus, 2007)		*Yes*	*Yes*	*38,700*	*Fraction of shared file names in directories above a predefined threshold.*
(Schwittek & Eicker, 2013)	*Yes*			*36*	*Automatic extraction of third party binary files from library subdirectory.*

(Haefliger, von Krogh, & Spaeth, 2008)	*Yes*	*Yes*		*6*	*Interviews with developers, inspection of source code and comments.*
(Heinemann, Deissenboeck, Gleirscher, Hummel, & Irlbeck, 2011)	*Yes*	*Yes*		*20*	*Static analysis and clone detection.*
Our approach	*Yes*	*Yes*	*Yes*	*1,111*	*Static analysis and manual inspection.*

CASE STUDY DESIGN

In order to provide evidence toward the aforementioned contributions, we performed an embedded multiple case study. The reason for selecting to perform a case study is that according to Runeson et al. (Runeson, Host, Rainer, & Regnell, 2012), case studies in software engineering are observational and used for monitoring projects in real-life context. The case study of this paper follows the case study design described by (Runeson et al., 2012)

Objectives and Research Questions

Following the Goal-Question-Metrics (GQM) approach (Basili, Caldiera, & Rombach, 1994), the goal of this study is *to analyze open source projects for the purpose of quantifying software reuse, monitoring its evolution, and evaluating the design quality of white-box reuse elements from the point of view of developers and maintainers in the context of software reuse in open source software.* According to the aforementioned goal, we investigate the following research questions that will guide the case study design and reporting of the results.

RQ_1: *How popular is reuse in open source software development* (contribution c1)*?*

$RQ_{1.1}$: *Which type of reuse (white-box or black-box) is more popular in open source software development* (contribution c1)*?*

$RQ_{1.2}$: *Is the frequency of invocation of the reused elements related to the type of the selected reuse strategy* (contribution c1)*?*

RQ_2: *What is the design quality of the classes that are being reused through white-box reuse* (contribution c2)*?*

RQ_3: *How frequently are the reused elements of software projects modified in later versions of the same system* (contribution c3)*?*

$RQ_{3.1}$: *Is the number of such modifications related to the type of reuse* (contribution c3)*?*

RQ_4: *Does the modified set of white-box reused elements provide improved scores of design quality metrics than the previous set* (contribution c4)*?*

Case Selection and Unit Analysis

Following the case study categorization of (Runeson et al., 2012), this paper presents an embedded multiple-case study, where the involved context is open source software projects, the cases are the reused packages and the units of analysis are the reused classes.

Data Collection

In order to support the dataset collection phase, the first author developed a tool in order to automatically identify, parse and download Java projects from Google Code repository. The initial dataset includes 1,459 projects from various domains and types of release. The dataset was filtered in order to exclude:

1. *Projects that do not provide executable files*. Several projects provide only their source code and therefore their compilation would be necessary in order to include them in this study, since the binary distributions of the projects are analyzed.
2. *Projects intended to serve as component repositories*. Although such projects could be analyzed, they are not included in this study since we aim beyond reuse measurement. Such projects require excessive effort for preprocessing in order to explore research question 4 and are therefore omitted. In order to analyze such projects, each of their components must be considered as a distinct project and this might impact the results since they could reuse other components of the same project.

After the exclusion phase, the filtered dataset consists of 1,111 projects. We note that the version identification for each project was performed by manually inspecting the release date of the corresponding files. For each project and its corresponding versions, static analysis was performed on the binary distribution in order to separate the classes into three categories: project classes, white-box reused classes and black-box reused classes. Project classes were distinguished from white-box reused classes by manually inspecting package names and by obtaining information from the projects' version control system. On the other hand, black-box reused classes were automatically identified by binary distribution analysis. The manual inspection process was repeated for all projects and their versions in order to conclude to the final dataset.

The dataset analysis was performed following a structured process for each project in order to provide valid and reproducible results[i]. Initially, for each project the files are sorted chronologically and assigned to a version. Additionally, the files that are uploaded on the same date are merged into the same version (e.g. binary, source code distribution). The software engineer inspects and if necessary, modifies the versioning configuration, prior to the finalization of the project's versions. Next, each version is analyzed as follows: the files that contain binary distributions are presented to the software engineer and in turn, he assigns each executable file either to the project-specific or to the external libraries category. The binary distributions of the external libraries category are the files that contain the project's black-box reused elements. Next, the project-specific binary distribution files are parsed and the project's packages are presented to the software engineer. In turn, he separates the project-specific packages from the white-box reused packages. Finally, the package-level and class-level measurements are stored in the database, where the entries are assigned to the project's version. The version analysis is performed for each project version. However, the tool implementation includes several heuristics in order to reduce the required effort to categorize the binary files: (1) the files whose parent folder is named "lib" are automatically included in the external libraries list, (2) the files whose name contains the project's name, are automatically included in the project-specific list. Finally, we use a heuristic to facilitate the packaging configuration of each version in our tool implementation. For a project that contains at least one analyzed version, the subsequent version analysis retrieves from the database the previous versions' project-specific and white-box reused packages. According to the project's previous configurations, the packages are automatically assigned to the package category. The automation of this step utilizes packages' names to identify common package structure at depth 3. For example, if in a previous version the org.apache.log4j package is in the white-box reused category, then if the package org.apache.log4j.helpers appears in a following version, it is automatically assigned to the white-box reuse category. The aforementioned process is repeated for each project by the first author and the results

are verified by the second author in order to eliminate possible errors in data collection and analysis. In the special case of an empty project, it is eliminated from our dataset.

Table 2. Collected Variables

Variable Name	***Variable***	***Description***
Project Name	*[A01]*	*The name of the project*
Version	*[A02]*	*Consecutive numbering according to the release date*
Reuse Type	*[A03]*	*The reuse strategy (white-box or black-box or both), if applied*
Design Metrics	*[A04-A09]*	*Chidamber and Kemerer design metrics (Chidamber & Kemerer, 1994)*

For each project and its corresponding versions, nine variables are documented as presented in Table 2. We note that in this study, glass-box reuse is considered as white-box reuse due to several limitations posed for the identification of this type of reuse. Firstly, identifying this type of reuse would require information about the exact version of the target system that is reused and additionally, textual analysis should be performed in order to verify if the source code has not been changed. The metrics used to measure the structural design quality originate from the Chidamber & Kemerer (CK) metrics suite (Chidamber & Kemerer, 1994), given that it is one of the best known metric suites for measuring object-oriented design quality. According a recent systematic literature review by Jabangwe, Börstler, Šmite, & Wohlin (2014), the CK metrics suite is the most frequently applied approach for quantifying quality attributes. The suite consists of six metrics:

[A04] WMC (Weighted Methods per Class): Measures the average Cyclomatic Complexity among methods of a class or if all methods are considered to be unity, then measures the number of methods (in this study the latter case is applied). High WMC values indicate greater potential impact on children, since they inherit all methods, and are harder to maintain and comprehend (Chidamber & Kemerer, 1994).

[A05] DIT (Depth of Inheritance Tree): Measures the depth of inheritance hierarchy. High values of DIT indicate classes that are difficult to understand and maintain (Chidamber & Kemerer, 1994).

[A06] NOC (Number of Children): Measures the number of immediate subclasses of the class. High NOC values imply improper abstraction of the class and therefore increased effort is required for their maintenance (Chidamber & Kemerer, 1994).

[A07] CBO (Coupling Between Objects): The number of other classes a class is connected to, through method calls, field accesses, inheritance, arguments, return types and exceptions. Excessive coupling between classes results to low maintainability and understandability (Chidamber & Kemerer, 1994).

[A08] RFC (Response For a Class): Calculates the number methods that can be executed in response to a message received by the class. High RFC values indicate classes that provide increased functionality to the system and therefore are potentially complex and require increased effort for testing and maintenance (Chidamber & Kemerer, 1994).

[A09] LCOM (Lack of Cohesion in Methods): Measures the dissimilarity of method pairs with respect to common attributes accesses. High values of LCOM indicate that classes

attempt to achieve many different objectives and therefore are error-prone and more difficult to test (Chidamber & Kemerer, 1994).

CK metric values were automatically calculated from the projects' binary distributions, using the CKJM tool (Spinellis, 2005).

Data Analysis

The data analysis step of this case study, includes descriptive statistics and independent samples t-test, in order to explore the research questions stated in the in the Objectives and Research Questions section. The analysis performed to investigate each research question is summarized in Table 3.

Table 3. Data Analysis and Presentation Overview

Research Question	***Variable***	***Analysis/Presentation***
RQ_1	*[A03]*	*Descriptive Statistics*
$RQ_{1.1}$	*[A03]*	*Descriptive Statistics*
$RQ_{1.2}$	*[A03]*	*Descriptive Statistics & Independent Sample t-test*
RQ_2	*[A04-A10]*	*Descriptive Statistics*
RQ_3	*[A03]*	*Descriptive Statistics*
$RQ_{3.1}$	*[A03]*	*Descriptive Statistics & Independent Sample t-test*
RQ_4	*[A04-A09]*	*Descriptive Statistics*

RESULTS

In this section we present the results of our case study, organized by research question. As a demographic view of our dataset, in Table 4 we present the most commonly reused libraries of our dataset. The results show that the majority of these libraries are of generic scope, e.g. XML processing, logging, unit testing, etc. As expected, none of the 15 most reused libraries provides project or domain specific functionalities.

Table 4. Reused libraries frequency

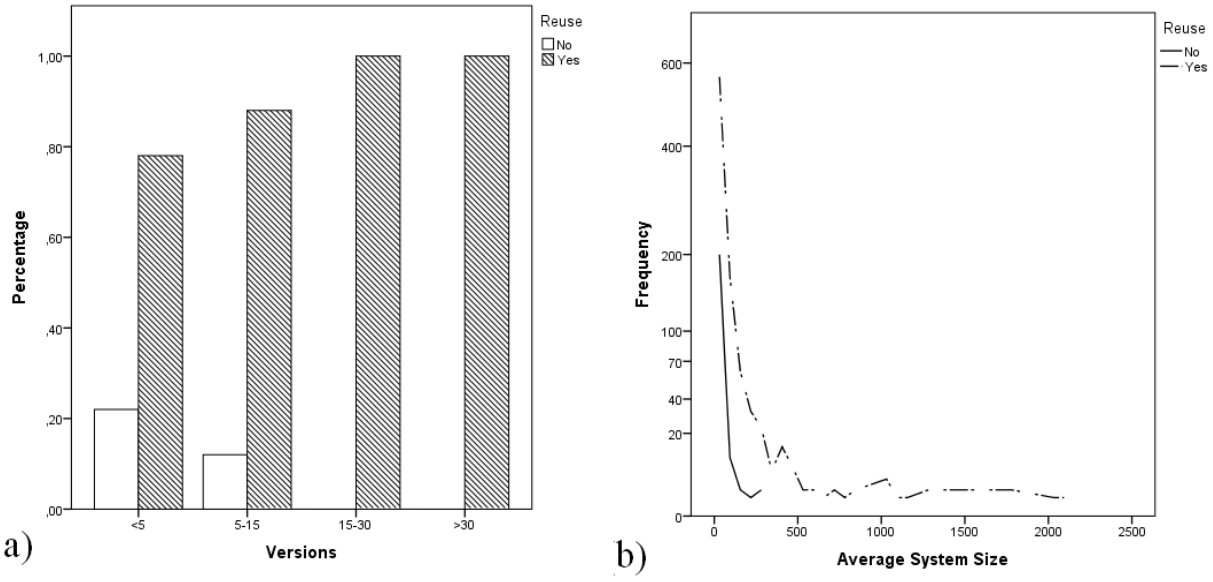
Reused Library	***Number of Projects***		
	Total	***Black-Box***	***White-Box***
Java API for XML Processing	*242*	*236*	*6*
Apache Logging Services	*165*	*150*	*15*
Apache Commons Logging	*113*	*106*	*7*
Simple Logging Facade for Java	*112*	*103*	*9*
Apache Commons Lang	*86*	*75*	*11*
Junit testing framework	*70*	*61*	*9*
Apache Http Components	*69*	*62*	*7*
Apache Ant	*52*	*52*	*0*
OSGI	*48*	*47*	*1*
Spring Framework	*46*	*42*	*4*

JDOM	*44*	*35*	*9*
Apache Commons IO	*42*	*34*	*8*
Dom4j	*34*	*34*	*0*
Apache Commons Codec	*37*	*33*	*4*
Eclipse	*33*	*33*	*0*

RQ1 : How popular is reuse in open source software development?

The analysis of the selected data indicates that the majority of OSS projects, reuse third party libraries and more precisely, reuse occurs in 897 of the 1,111 projects, i.e. in 80.74%. For the remaining 214 projects, i.e. 19.26%, no reused elements were identified. In order to further explore the differences between projects that perform reuse and others that do not, we investigate the levels of reuse from two different perspectives (see Figure 1): i.e. longevity and size in classes. Figure 1(a) presents the frequency of each type of reuse with regard to the number of different versions for each project. The results show that all projects with more than 15 versions reuse external libraries. The projects that reuse third party libraries present longevity, in terms of number of released versions (4.47±9.07) compared to projects that do not reuse (2.7±4.89). We note that the value in the parenthesis represents the mean value and standard deviation of the used metric. Figure 1(b) shows systems' average size with regard to their selection on reusing external libraries. According to Figure 1(b), projects that reuse appear to be larger, in terms of the average number of classes per version, (95.58±190.95) than projects that do not reuse external software (25.08±37.64). The results show that reusing external libraries, benefits projects' lifespan and size throughout their evolution.

Figure 1: Projects' Attributes



RQ1.1 Which type of reuse (white-box or black-box) is more popular in open source software development?

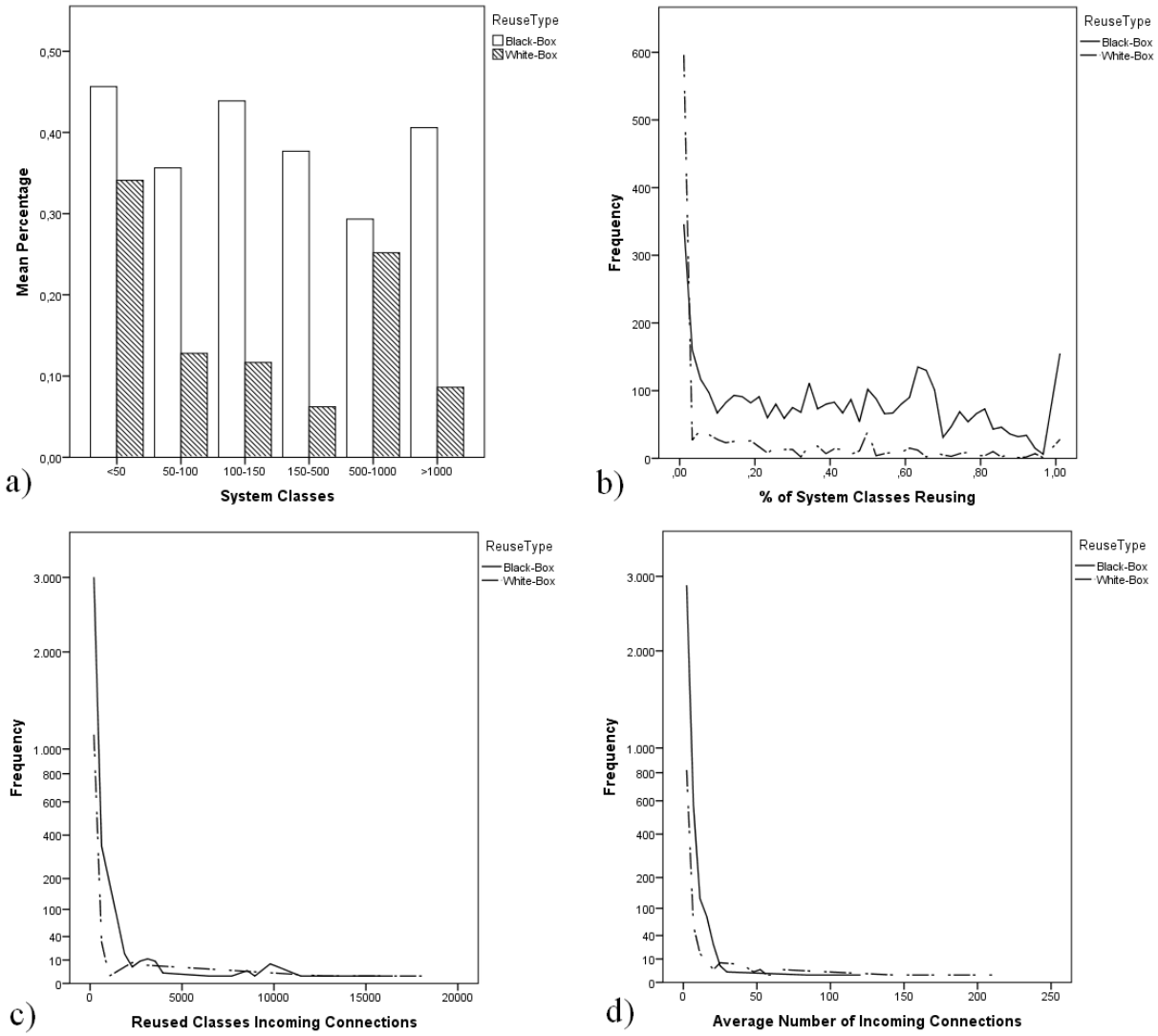
For the 897 projects that reuse third party components, we identified that in the majority of OSS projects black-box reuse strategy is employed. More precisely, 656 projects apply only the black-box reuse strategy, 46 projects apply only the white-box reuse strategy, while 195 projects use both strategies. In total, 851 projects employ a black-box reuse strategy and 241 a white-box reuse strategy. The results

coincide with other studies related to reuse quantification, which suggest that black-box is the most popular type of reuse.

RQ$_{1.2}$ ***Is the frequency of invocation of the reused elements related to the type of the selected reuse strategy*****?**

In Figure 2 we visualize how intensively reused elements are exploited across different reuse types. Figure 2(a) presents the proportion of classes in the systems under development that depend on reused elements according to each reuse strategy, whereas in Figure 2(b) the percentage of system classes that reuse third party elements is illustrated. Figure 2(c) shows the number of invocations from system classes per reuse strategy and finally in Figure 2(d), the average number of invocations per reused element is presented. These observations aim to provide insight to the intensity of reused elements exploitation for each type of reuse strategy.

Figure 2: Exploitation of Reused Elements



Concerning the number of system classes that reuse external elements, the results suggest that, on average, system classes depend on more black-box reused elements (46.53 calls) than white-box elements (26.21 calls), see Figure 2(a). In order to filter out the confounding factor of projects' size, we also present the percentage of systems' classes that reuse external elements. As depicted in Figure 2(b), on average system classes depend on more black-box reused elements compared to white-box reused elements. More specifically, on average 43.24% of system classes depend on black-box reused elements, while 28.4% depend on white-box reused elements. As observed in Figure 2(b), there are cases where all systems' classes depend on reused elements. In cases of white-box reuse, this occurs for 28 small systems that consist up to 22 classes. In the case of black-box reuse, this occurs for 129 small systems with less than 50 classes and for 26 larger systems that consist up to 135 classes. Figure 2(c) presents the use intensity of reused elements according to each reuse strategy. Black-box reused elements are on average used by more system classes (299.73±844.46), compared to white-box reused classes (117.9±743.5). Figure 2(d) presents the average number of incoming calls to reused elements for each type of reuse. On average each reused class is used by 3.6 system classes, regardless of the type of reuse.

In order to determine if there are significant differences in the number of system classes that exploit the reused elements and the number of links to each reused element, between black-box and white-box reuse cases, we performed independent samples t-test. The results show that systems' classes employ more black-box reused elements (44.7±95.54) compared to white-box reused elements (20.57±96.74). We note that the value in the parenthesis represents the mean value and standard deviation of the used metric. The results comply with the observations about the percentage of system classes that use reused elements, since on average larger percentage of system classes use black-box elements (0.41±0.29) compared to white-box elements (0.18±0.27). Both differences present a statistically significant difference at 0.01 level. Regarding the number of invocations from project classes to reused classes, a statistically significant difference between black-box reused elements (299.73±844.46) and white-box reused elements was found (117.9±743.50). However, the average number of incoming links per reused element did not present a statistically significant difference between the two types of reuse, i.e. 3.62±4.08 for black-box reuse and 3.64±10.99 for white-box reuse. Overall, the results indicate that projects that reuse external libraries employ more black-box reused elements compared to white-box reused elements. This observation is attributed to the required effort for white-box reuse adaptation.

RQ_2 What is the design quality of the classes that are being reused through white-box reuse?

The descriptive statistics of CK design quality metrics for system and white-box reused classes are presented in Table 5. The results show that on average, white-box reused elements present larger values compared to system classes, except for DIT metric. In order to investigate the effect of reusing external elements on the projects' design quality we measured for each system, the proportion of the system that originates from reuse and the average values of the design quality metrics. In Table 6, we present Spearman's correlation coefficient between the systems' reuse percentage and the design quality metrics. A significant positive correlation exists between the percentage of the system relies on reuse and the mean of the design metrics. This observation signifies that white-box reused elements always present larger values for the studied metrics; thus, they originate from projects with increased provided functionality, compared to the studied projects. However, as a side effect, some aspects of the design quality, such as coupling or cohesion, appear to be negatively affected.

Table 5. CK Metric Values

	System Classes				***White-Box Reused Classes***			
Metric	***Average***	***Median***	***Max***	***Sum***	***Average***	***Median***	***Max***	***Sum***
WMC	*8.88*	*5*	*1,086*	*771,839*	*11.22*	*6*	*548*	*174,5059*
DIT	*2.05*	*1*	*13*	*178,211*	*2.00*	*1*	*11*	*311,010*
NOC	*0.28*	*0*	*117*	*24,310*	*0.37*	*0*	*123*	*57,534*
CBO	*5.32*	*3*	*181*	*462,147*	*5.44*	*3*	*200*	*845,541*
RFC	*25.37*	*13*	*2,028*	*2,204,433*	*27.09*	*14*	*929*	*4,213,960*
LCOM	*91.58*	*3*	*67,896*	*7,957,641*	*194.28*	*4*	*10,2622*	*3,022,0781*

Table 6. Spearman's Correlation Coefficient

Metric	***Average***
WMC	*0.254***
DIT	*0.153***
NOC	*0.301***
CBO	*0.146***
RFC	*0.153***
LCOM	*0.340***

RQ$_3$ How frequently are the reused elements of software projects modified in later versions of the same system?

The results showed that black-box reused elements are involved in design changes more frequently than white-box reused elements, as presented in Table 7. Overall, our dataset consists of 3,477 version transitions, in which white-box and black-box reused elements are altered in 241 and 851 version transitions respectively. More precisely, for black-box reuse 3,515 packages were added, 1,841 were removed, while 47,607 packages were not altered during projects' evolution. For white-box reuse, 329 packages were added, 324 were removed and 2,954 packages were not altered. Additionally, Table 7 presents the average number of altered packages per version transition for each reuse type. When white-box reused packages are altered, on average 1.34 packages are added, 1.34 packages are removed and 12.25 packages are not altered. When black-box reused packages are altered, on average 4.13 packages are added, 2.16 packages are removed and 55.94 packages are not altered. Overall, once reused packages are imported to a project, they are rarely modified.

Table 7. Version transitions with modified reused packages

	White-box Reuse	***Black-box Reuse***
Version transitions	*241*	*851*
Added	*329 (1.34)*	*3,515 (4.13)*
Removed	*324 (1.34)*	*1,841 (2.16)*
Not Altered	*2,954 (12.25)*	*47,607 (55.94)*

RQ$_{3.1}$ Is the number of such modifications related to the type of reuse?

In order to determine if there are differences in the number of modified reused packages between the two reuse strategies, we run an independent-samples t-test. The results showed that there is a statistically significant difference at 0.01 level between the packages added and the packages that are not altered for the two types of reuse. More precisely, more black-box packages are added (5.36 ± 10.04) than white-box packages (3.43 ± 4.86) and more black-box packages are not altered (21.76 ± 29.98) than white-box packages (5.15 ± 9.33). However, concerning package removal no significant difference is

found for the two forms of reuse, i.e. (5.89 ± 21.45) for black-box packages and (5.87 ± 11.92) for white box packages. The increased rate of black-box reuse altered packages is attributed to the low effort that is required for their incorporation to a system, compared to white-box reuse.

RQ4 Does the modified set of white-box reused elements provide improved scores of design quality metrics than the previous set?

The observed differences of the average values of CK metrics for version transitions with modified white-box reused elements are depicted in Table 8. On average only DIT and CBO values present slightly enhanced values. In order to investigate whether or not the modified set of white-box reused elements better adhere to design quality metrics, we additionally measured the percentage of occurrences in which the difference of the average values of the metrics is increased and decreased. The results show that the percentage of cases where the metrics' values are increased is larger compared to the times that the values are decreased. Therefore, the structural design quality of the reused elements does not appear to be a criterion for white-box reuse elements' alteration. Thus, developers do not appear to focus on design quality attributes during the reusable elements' selection.

Table 8. Version transitions with modified reused packages

Average Metric Difference	***Average***	***Median***	***Min***	***Max***	***% of cases with D>0***	***% of cases with D<0***
DWMC	*1.04*	*0.001*	*-13.24*	*125.98*	*49.34*	*44.08*
DDIT	*-0.21*	*0.001*	*-2.5*	*3*	*43.42*	*31.58*
DNOC	*0.01*	*0.000*	*-0.78*	*0.78*	*31.58*	*31.58*
DCBO	*-0.24*	*0.005*	*-13.75*	*6.58*	*49.34*	*40.13*
DRFC	*1.35*	*0.079*	*-48.57*	*136.77*	*56.58*	*40.13*
DLCOM	*143.15*	*0.068*	*-3349.53*	*20459.14*	*50.66*	*43.42*

DISCUSSION

Interpretation of results

This study confirms existing literature (Mockus, 2007; Schwittek & Eicker, 2013; Haefliger, von Krogh, & Spaeth, 2008; Heinemann, Deissenboeck, Gleirscher, Hummel, & Irlbeck, 2011) which supports that the majority of the projects reuse third party libraries. Beyond related studies in OSS reuse, we investigate reuse practices with regard to projects' evolution and size. The results we obtained suggest that projects that reuse external libraries appear to have longer lifespan and size compared to projects that do not reuse, due to the additional functionality that can be incorporated with lower development effort (Capiluppi, Boldyreff, & Stol, 2011).

In this study, we extend empirical knowledge on reuse strategies since we provide empirical evidence about the exploitation intensity of reused elements. With respect to the reuse strategy, the results coincide with related studies since black-box reuse is the most frequently employed strategy. Concerning the reuse intensity, the results show that systems' classes reuse more black-box elements than white-box elements, an expected outcome due to the small effort required to reuse external libraries through black-box reuse. Although the black-box reused elements have more invocations than the white-box reused elements, on average each reused element is linked to 3.6 system classes. According to (Schwittek & Eicker, 2013), the black-box reused elements mainly contain libraries that enhance the basic functionality of Java platform (e.g. data structures). Our results are in agreement with (Schwittek & Eicker, 2013) in the sense that we confirm that systems excessively employ black-box reused elements. On the contrary, systems do not accordingly exploit white-box reused elements' functionality. A possible explanation for

this lies beneath the nature of the most frequently reused libraries with black box reuse. Specifically, since the most frequently black box reused libraries serve the systems for logging or testing purposes, they are more probable to be called more times inside the project scope. Considering the effort required for source code adaptation in white-box reuse strategy, the aforementioned usage exploitation implies that white-box reused elements functionality is closer to some aspects of the system under development. However, this assumption requires validation from the developers and in future work we plan to investigate it.

Concerning the structural design quality of white-box reused classes, the results showed that CK metrics' values of white-box reused classes present larger values compared to systems' classes except for DIT metric. Reused libraries mainly originate from mature and long-evolved projects and consequently, they present increased metrics' values due to the providing functionality (Sanjay, 2011). This observation, although intuitively explained, might be interpreted in two ways: (a) either the white-box reused code is hard to understand, extent and maintain, or (b) the source code that is copied and pasted into other OSS projects is not modified but reused as is (i.e. glass box reuse). Both the aforementioned claims need further investigation because in the case of (a) reuse elements selection strategies should be updated so as to take into account the components' design quality, whereas in case of (b) developers should be advocated to use the specified components through black-box reuse strategies.

In previous studies, the investigation of software reuse mainly focused on individual versions of the explored systems. On the contrary, in this study, we investigate reuse modification decisions. We measure the modification of reused packages for the systems in our sample according to each reuse strategy. For this analysis, package-level measurement is selected in order to provide valuable results regardless of the usage diversity of each reused library. For example, importing a logging library and a binary parser library into a project implies diverse usage of each library in the system. More specifically, in the former case the developed system is expected to exploit more reused classes compared to the latter case, where a java parser is usually used by an interface from a specific package. The obtained results show that reused elements are rarely modified in versions' transitions. Thus, once reused elements are imported in a system, they are scarcely modified. External packages are added more frequently during projects' evolution, while their removal appears less often for both reuse strategies. The package addition can be attributed to projects' need for additional functionality and therefore for importing new reused elements. The lower rate of package removal is expected since their removal indicates either their complete removal from the system or their substitution with new packages. However, this assumption is out of the scope of this study and additionally, the assumption of functionality substitution imposes great threat to possible justification of the developers' intention for packages' substitution. Nonetheless, the increased modification rate of black-box reused packages is attributed to the minimal effort required for incorporating them into a system under development. On the contrary, white-box reused packages are modified less frequently compared to black-box reused packages. This observation reveals that once the reused elements are included in the systems' source code they are more likely to remain during systems' evolution. Therefore, the asset selection for white-box reuse strategy should be based on quality criteria, among others, since during projects' evolution these assets require maintenance and testing effort.

White-box reuse strategy reveals the internals of the reused library since the source code is imported to the system under development. Therefore, in this study we examined developers' focus on structural design quality when their reuse decisions are modified during the projects' evolution by investigating if the modified set of white-box reused elements better adheres to structural design quality attributes. The results did not show significant differences between the modified sets of reused elements. Therefore, the reusable asset selection does not appear to be based on structural design quality attributes for a variety of reasons. Developers seem not to focus on design quality during their selection of reusable assets. This is attributed either to familiarity with alternative libraries or lack of interest on design quality. Additionally, this observation can be attributed to glass-box reuse where the internals of the reused

system are not necessarily investigated by the developers since they are used according to black-box reuse strategy.

Implications to researchers and practitioners

In this section we discuss the implications of our results to researchers and practitioners. Based on the expected contributions (stated in the Introduction), we believe that the reported results can prove beneficial to both researchers and practitioners. Firstly, we encourage *researchers* to:

- investigate if the extent of white-box reuse elements' exploitation is an indicator of the element's domain specificity. The results of this work reveal that elements reused according to white-box reuse strategy are moderately exploited, whereas black-box reused elements are more frequently invoked. The findings of this study support the aforementioned claim, by suggesting that black-box reused elements (such as logging mechanisms) are more generic.
- investigate the reuse decisions' rationale, by focusing on the modifications of reused elements across subsequent projects' versions. The results of this study suggest that white-box reused packages are rarely removed in projects' subsequent versions. However, lacking the requirements specification, the traceability of reuse decisions is not feasible.
- propose methods to facilitate the reuse element selection process, and specifically for the white-box reuse strategy, by taking into account the design quality attributes. The selection of reused assets needs to consider the understandability and adaptability of the candidate elements, in the sense that they will need to be incorporated in the target project's source code. The results of our study suggest that as the percentage of white-box reuse classes increases, the design quality is slightly diminished.
- investigate the complexity of reusable source code and its impact on reuse activities, especially for white-box reuse. Reusable source code tends to be more complex, since it more generic in the sense that it is intended to be used in several contexts. The complexity lies on the use of design patterns and interfaces with more functions that exploit the implemented functionality.

We encourage *practitioners* to:

- reuse external libraries in order to reduce the development effort and facilitate projects' evolution.
- consider the design quality attributes during the reusable asset selection since the effort required for their maintenance is increased during projects' evolution.
- base their reuse asset selection processes on quality criteria, since assets that have been imported according to white-box reuse are scarcely removed. Over-reusing external elements potentially leads to increased effort for maintenance and testing.
- prefer black-box instead of glass-box reuse strategy when the external reused elements are not prone to modifications.

THREATS TO VALIDITY

Internal validity deals with possible threats that can be caused by confounding factors, i.e. variables other than the independent variables that affect the dependent variable. Regarding internal validity in this study, a possible threat is the overestimation of white-box reuse due to glass-box reuse.

OSS developers often include the source files of external dependencies to the binary distributions. To mitigate this, during the analysis the classes found both in white-box reused elements and in external library jars of the projects, where only considered as black-box reused elements. Additionally, the analysis is based on the binary distributions of OSS and therefore, the white-box reused elements were manually inspected. To ensure the correctness of the project-specific and white-box reused elements, the first and second author examined the obtained dataset. We believe that the manual inspection limits the threat of biasing white-box reuse measurement.

Construct validity reflects to what extent the phenomenon under study really represents what is investigated according to the research questions. Concerning construct validity, we only measure reuse from OSS projects, either white-box or black-box and therefore, Java API calls are not considered during the reuse measurement process. Moreover, although reuse can involve different targets, i.e. design reuse, frameworks, etc., we only focus in reuse of OSS projects. An additional threat to the validity of our results is the inclusion of projects that are not actively developed for a long period, since such systems might not be representative samples of successful OSS projects. However, the primary goal of this study is to investigate the reuse intensity and strategies applied in OSS development. Thus, we do not consider the success or quality of the investigated projects. This threat to the validity of our results is mitigated by the fact that such systems follow similar reuse strategies as their peers during the period of time that they were actively developed. Finally, regarding the structural design quality measurement, we selected the Chidamber & Kemerer metrics suite and therefore, a possible threat to construct validity is that the results depend only on CK metrics. However, we acknowledge that design quality can be measured from other points of view, e.g. identification of design bad smells; therefore, our approach does not comprehensively assess design quality. Nonetheless, CK metrics suite is one of the most popular suites of design metrics with several variations that can be applied in order to confirm the results.

Case study reliability is related to whether the data are collected and the analysis is conducted in a way that it can be replicated by other researchers with the same results. In order to mitigate reliability we provide information about the open source projects investigated in this study[i]. Additionally, we explain in detail the data analysis process.

External validity deals with possible threats while generalizing the findings derived from sample to population. Concerning external validity, two factors can impose threats to the results of this study. The first factor is the selected projects are Java applications. The results cannot be generalized of the results for other programming languages, since the programming language has impact on reuse (Heinemann, Deissenboeck, Gleirscher, Hummel, & Irlbeck, 2011). The second factor is the generalization of the results for Java open source systems. To mitigate this factor, the implemented tool retrieves Java systems from Google Code repository without any selection criteria so as the dataset to consist of a representative sample of an OSS repository. Nonetheless, the selection of one repository, i.e. Google Code, poses a threat to the validity of the results, since it represents a small fraction of the developers that contribute to OSS.

CONCLUSIONS AND FUTURE WORK

Our study shows that third party reuse is a common practice among open source Java projects and provides insight about the intensity of incorporating reused elements. From the 1,111 studied projects, the majority reuses external libraries, mainly with the strategy of black-box reuse. Additionally, our study shows that although black-box reuse elements are used more often and their presence is more pervasive in software systems. Additionally, on average, the reused elements receive similar number of invocations regardless of the reuse strategy. Finally, the structural design quality of reused elements does not appear to be a factor for the selection of white-box reused assets. Following this study, we plan to extend it and focus on specific categories of software systems. In particular, we will study projects from other OSS repositories like Sourceforge. Additionally, we will study if reuse practices of different domains preserve

the results we obtained from this study. Finally, in our future work we will discriminate all three types of reuse by obtaining reused elements' information from repositories liken Maven, where more information that concerns reused elements is provided (e.g. versions of the reused libraries) (Raemaekers, van Deursen, & Vis, 2013).

ACKNOWLEDGMENT

This research work is co-founded by the European Social Fund and National Resources, ESPA 2007-2013, EDULLL, "Archimedes III" program. Also, the authors thank Prof. Lefteris Angelis for his valuable comments and help.

REFERENCES

Ajila, S. A., & Wu, D. (2007). Empirical study of the effects of open source adoption on software development economics. *Journal of Systems and Software*, *80*(9), 1517-1529.

Bangerth, W., & Heister, T. (2013). What makes computational open source software libraries successful? *Computational Science & Discovery*, *6*(1), 015010.

Basili, V. R., Caldiera, G., & Rombach, D. H. (1994). *Goal Question Metric Approach.* John Wiley & Sons.

Bauer, V., Heinemann, L., & Deissenboeck, F. (2012). A structured approach to assess third-party library usage. *International Conference on Software Maintenance* (pp. 483-492). Trento: IEEE.

Capiluppi, A., Boldyreff, C., & Stol, K.-J. (2011). Successful Reuse of Software Components: A Report from the Open Source Perspective. *International Conference on Open Source Systems* (pp. 159-176). Salvador: Springer Verlag.

Capiluppi, A., Stol, K.-J., & Boldyreff, C. (2011). Software Reuse in Open Source: A Case Study. *International Journal on Open Source Software and Processes*, *3*(3), 10-35.

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, *20*(6), 476-496.

Dietrich, J., Jezek, K., & Brada, P. (2014). Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. *Conference on Software Maintenance, Reengineering and Reverse Engineering* (pp. 64-73). IEEE.

Frakes, W. B., & Kang, K. (2005). Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, *31*(7), 529-536.

Haefliger, S., von Krogh, G., & Spaeth, S. (2008). Code Reuse in Open Source Software. *Management Science*, *54*(1), 180-193.

Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., & Irlbeck, M. (2011). On the Extent and Nature of Software Reuse in Open Source Java Projects. *International Conference on Top Productivity through Software Reuse* (pp. 207-222). Pohang: Springer-Verlag.

Jabangwe, R., Börstler, J., Šmite, D., & Wohlin, C. (2014). Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, 1-54.

Koch, S., & Schneider, G. (2002). Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal*, *12*(1), 27-42.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys*, *24*(2), 131-183.

Lim, W. C. (1994). Effects of reuse on quality, productivity, and economics. *IEEE Software, 11*(5), 23-30.

Marri, M. R., Thummalapenta, S., & Xie, T. (2009). Improving software quality via code searching and mining. *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation* (pp. 33-36). Vancouver: IEEE Computer Society.

Mockus, A. (2007). Large-Scale Code Reuse in Open Source Software. *First International Workshop on Emerging Trends in FLOSS Research and Development* (p. 7). Minneapolis: IEEE Computer Society.

Raemaekers, S., van Deursen, A., & Vis, J. (2013). The maven repository dataset of metrics, changes, and dependencies. *Working Conference on Mining Software Repositories* (pp. 221-224). Piscataway: IEEE.

Raemaekers, S., van Deursen, A., & Visser, J. (2012). Measuring software library stability through historical version analysis. *International Conference on Software Maintenance* (pp. 378-387). Trento: IEEE.

Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples.* John Wiley & Sons.

Sametinger, J. (1999). *Software Engineering with Reusable Components.* Springer-Verlag.

Sanjay, M. (2011). Evaluation Criteria for Object-oriented Metrics. *Acta Polytechnica Hungarica, 8*(5), 109-136.

Schwittek, W., & Eicker, S. (2013). A Study on Third Party Component Reuse in Java Enterprise Open Source Software. *International ACM Sigsoft Symposium on Component-based Software Engineering* (pp. 75-80). Vancouver: ACM.

Spinellis, D. (2005). Tool writing: a forgotten art? *IEEE Software, 22*(4), 9-11.

Yu, L. (2006). Indirectly predicting the maintenance effort of open-source software. *Journal of Software Maintenance and Evolution: Research and Practice, 18*(5), 311-332.

[i] We provide a list of the studied projects and a brief presentation of the projects' analysis process with our implemented tool at http://www.econst.eu/?p=72.