The Journal of Systems and Software 134 (2017) 211–227

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Reusability of open source software across domains: A case study

Maria-Eleni Paschali[a], Apostolos Ampatzoglou[a,*], Stamatia Bibi[b], Alexander Chatzigeorgiou[c], Ioannis Stamelos[a]

[a] Department of Informatics, Aristotle University of Thessaloniki, Greece
[b] Department of Informatics & Telecommunications Engineering, University of Western Macedonia, Greece
[c] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 24 October 2016
Revised 3 August 2017
Accepted 8 September 2017
Available online 9 September 2017

Keywords:
Reusability
Open source
Application domains

ABSTRACT

Exploiting the enormous amount of open source software (OSS) as a vehicle for reuse is a promising opportunity for software engineers. However, this task is far from trivial, since such projects are sometimes not easy to understand and adapt to target systems, whereas at the same time the reusable assets are not obvious to identify. In this study, we assess open source software projects, with respect to their reusability, i.e., the easiness to adapt them in a new system. By taking into account that domain-specific reuse is more beneficial than domain-agnostic; we focus this study on identifying the application domains that contain the most reusable software projects. To achieve this goal, we compared the reusability of approximately 600 OSS projects from ten application domains through a case study. The results of the study suggested that in every aspect of reusability, there are different dominant application domains. However, *Science and Engineering Applications* and *Software Development Tools*, have proven to be the ones that are the most reuse-friendly. Based on this observation, we suggest software engineers, who are focusing on the specific application domains, to consider reusing assets from open source software projects.

1. Introduction

Open source software (OSS) projects can be utilized in many ways, one of the most important being the white-box reuse of their source code, which increases development productivity (Baldassarre et al., 2005), (Morisio et al., 2002) and product quality (Frakes and Fox., 1996), (Haefliger et al., 2007). Mockus suggests that 53% of OSS projects have performed reuse activities in 30% of their development process and that 49% of projects have reused more than 80% of their code. Additionally, it is suggested that most reused units have gone through major or minor modifications in order to be adapted in the target project (i.e., white-box reuse). Investing on OSS, as a "code reuser", involves two major tasks:

- ***Reusable asset identification***. In this step the reuser has to identify a piece of source code (e.g., method, class, set of classes, package, complete project, etc.) that implements the functionality that he/she wants to reuse. This task is a very difficult one, in the sense that: (a) the available amount of reusable assets in OSS is vast, and in some cases not well organized and documented, and (b) there is a lack of platforms that can act as search engines for providing access to OSS repositories.
- ***Reusable asset adaptation***. In this step (after the reusable asset has been identified), the reuser has to adapt the source code of the asset as extracted from the source system (in case of white-box reuse) to fit the architecture of the target system. Such an adaptation requires that the asset is well structured and maintainable. The assessment of maintainability is a non-trivial task in the sense that although there are plenty of maintainability indicators, a specific asset cannot be characterized as maintainable or not, in isolation, due to the lack of well-established thresholds values for the maintainability indices.

By taking into account that software reuse is more efficient when performed within the same application domain (Johnson et al., 2004), in this paper we focus on domain-specific reuse, i.e., reuse activities in which the target and the source systems belong to the same application domain (e.g., games, business applications, etc.). In the context of domain-specific reuse, it would be highly valuable for practitioners to know the extent to which a specific application domain can benefit from OSS reuse. In particular, it is expected that an application domain facilitates reuse if: (a) it offers a large variety of reusable assets (e.g., sets of classes

* Corresponding author.
E-mail addresses: mpaschali@csd.auth.gr (M.-E. Paschali), apamp@csd.auth.gr (A. Ampatzoglou), sbibi@uowm.gr (S. Bibi), achat@uom.gr (A. Chatzigeorgiou), stamelos@csd.auth.gr (I. Stamelos).

http://dx.doi.org/10.1016/j.jss.2017.09.009
0164-1212/

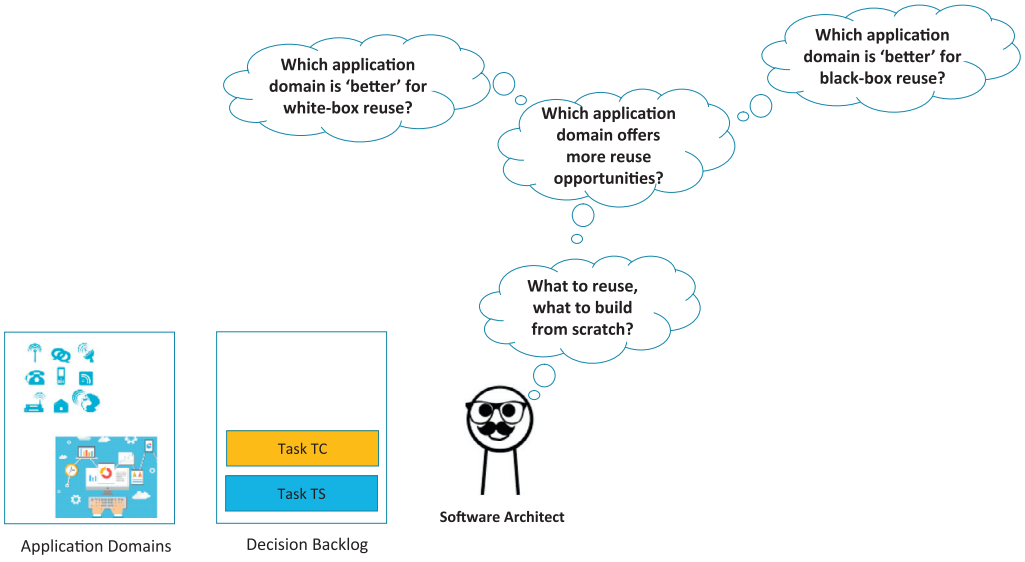


Fig. 1. Illustrative example of study motivation.

or projects), and (b) the offered assets can be easily adapted in target systems. Nevertheless, the research state-of-the-art lacks empirical evidence on which application domains offer the most reuse opportunities, in terms of number of offered reusable assets. Similarly, in the literature there is no evidence on the flexibility and adaptability of software systems across different application domains. Additionally, by considering that software reuse can be performed at different levels of granularity (Clements, 1995), (Lau and Wang, 2005), ranging from some lines of code to complete projects, packed as third-party libraries (e.g., as jar files) it is expected that various aspects of artifacts should be considered while assessing reusability. Based on the level of reuse granularity, there are different stakeholders that are interested in the reusability of projects: software architects that make high-level decisions on which third-party libraries to reuse are interested on reusability at the project level (black-box reuse), and software engineers who can select to reuse only parts of OSS projects are interested in the reusability at the set of classes level (white-box reuse). The motivation of this study is further illustrated in the example of Fig. 1.

In the illustration of Fig. 1, assume a company that focuses on the telecommunications domain. The company produces communications software (for sale) and some software development tools (for internal use in the company). The main architect of the company has specific tasks in his backlog for both projects (e.g., a task for the communications project (TC), and a task for the software development tool project (TS)). For both tasks the architect feels that there is a reuse potential from OSS. Other alternatives to reuse could be buying a component-off-the-self to provide the required functionality, or build it from scratch. Based on the manpower of the company, the decision is to reuse an OSS artifact for one of the two tasks and build the other from scratch. Therefore, one of the questions that the architect faces is: *For which one of the two projects should I reuse code and for which to build the code from scratch*? To answer this question many parameters need to be considered:

- Which application domain offers the most opportunities for reuse?
- Which factors are of interest in this decision (e.g., black-box vs. white-box reuse)?
- Which application domain excels in each one of these factors?

The example of Fig. 1 will be revisited in the discussion section (see Section 6.3) in which we will try to answer the aforementioned questions, based on the outcomes of our empirical study.

Although we can acknowledge the fact that the answer to the aforementioned question is influenced by many parameters, in this study we aim at exploring the extent to which specific application domains, offer more reuse opportunities. To achieve this goal, we investigate if there are statistically significant differences in the reusability of OSS projects from different application domains. To cover the concerns of both stakeholders we report results on the project level, synthesizing measures from both the class and the project level, based on the reusability model introduced by Hristov et al. (2012)—more information are available in Section 3. By taking into account the enormous amount of source code that is available in Open Source Software repositories (e.g., Sourceforge, GitHub, etc.), in this paper we perform an exploratory case study to investigate the opportunity to reuse OSS assets in software development. To achieve this goal, we exploit a large-scale meta-repository of OSS projects (namely Percerons[1]) that at this point offers quality assessments for approximately 600 open source software projects. The rest of the paper is organized as follows: In Section 2 we present an overview of related work and in Section 3 the employed reusability model. In Section 4 we present the study design in the form of a case study protocol. In Section 5 we provide the results, organized by research question, and discuss them in Section 6. In Section 7 we discuss the threats to validity of our study, and in Section 8, we conclude the paper.

2. Related work

2.1. Software reuse

Software reuse has been promoted as a key solution to face the "software crisis" since the early days of the software engineering discipline (Mili et al., 1991). The systematic use of existing software assets (referring to both artifacts and knowledge) to implement new software systems or update existing ones defines the process of software reuse (Jacobson et al., 1997). On the other hand, the degree to which a certain asset can be reused by other software systems determines the property of reusability.

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

Table 1
Software reusability models.

Quality properties	Metrics
Internal quality	Direct class coupling: Bansiya and Davies (2002)
	Coupling between objects: Kakarontzas et al. (2013)
	Lack of cohesion between methods: *Kakarontzas et al. (2013)*
	Cohesion among methods of class: *Bansiya and Davies (2002)*
	Class interface size: *Bansiya and Davies (2002)*
	Response for a class: *Kakarontzas et al. (2013)*; *Nair and Selvarani (2010)*
	Weighted methods for class: *Kakarontzas et al. (2013)*; *Nair and Selvarani (2010)*
	Design size in classes: *Bansiya and Davies (2002)*
	Number of classes: *Kakarontzas et al. (2013)*
	Depth of inheritance: *Kakarontzas et al. (2013)*; *Nair and Selvarani (2010)*
Customizability	Number of properties: *Sharma et al. (2009)*
	Setter methods: *Sharma et al. (2009)*
Interface Complexity	Overall complexity: *Sharma et al. (2009)*
Portability	Number of External dependencies: *Sharma et al. (2009)*; *Washizaki et al. (2003)*
	Self-completeness of component: *Washizaki et al. (2003)*
Understandability	Documentation quality: *Sharma et al. (2009)*; *Washizaki et al. (2003)*
	Existence of meta information: *Washizaki et al. (2003)*
	Observability: *Sharma et al. (2009)*
	Rate of component observability: *Washizaki et al. (2003)*
Adaptability	Rate of component customizability: *Washizaki et al. (2003)*

Early studies in the area examine mainly different types of reuse and problematic areas (Standish, 1984), (Krueger, 1992) as well as effective reuse process (Karlsson, 1995), (Crnkovic et al., 2002). A full reuse-oriented process (Karlsson, 1995) and maintenance model are expected to increase software development productivity (Boehm, 1999), (Mohagheghi and Conradi, 2007) and minimize quality degradation during maintenance operations of a software system (Baldassarre et al., 2005). Traditional and reuse-oriented software development are compared by Morisio et al. (2002) confirming the findings of Baldassarre et al. (2005), i.e., that software reuse has a positive effect on productivity and quality.

Several studies in literature can be found examining the possibility of successful software reuse considering one or more of the following aspects: (a) the application domain (Schwittek and Eicker, 2013), (Cho and Yang, 2008), (Folmer, 2007), (Lee et al., 2006) (b) the requirements specificity (Raemaekers et al., 2012), (c) and the type of reuse (Heinemann et al., 2011). The application domain reflects the environment in which a software system operates (enterprise software, operating systems, entertainment applications etc.). The requirements specificity affects software reuse success, as isolating the suitable features for potential reuse is of major importance. The features express the units of functionality of a software system that satisfy a requirement. The type of reuse is also of paramount importance distinguishing between white-box reuse and black-box reuse (Ruben, 1993). In white-box reuse (Frakes and Fox, 1996) there is knowledge regarding the internal concepts of the artifacts reused while in black-box reuse there is no such knowledge.

2.2. Software reusability metrics and models

In this section, we present the reusability models and the associated metrics and indices that have been identified in literature. The investigation of quality/ reusability models that incorporate metrics for addressing the potential reuse of a certain artifact is popular (Franch and Carvallo, 2003). Most of the reusability metrics and indices that are proposed as reusability indicators are source code or design metrics (see Table 1). Our target is to investigate all possible aspects, as recorded in literature that affect the reusability of a certain artifact along with the relevant metrics and indices for quantifying them in order to test the reusability of OSS artifacts across different domains.

A wide range of studies have been performed for assessing the reusability of a certain artifact based on structural properties (e.g., encapsulation, coupling and cohesion). Bansiya and Davies, 2002 proposed QMOOD, a hierarchical quality model, for assessing the quality of object-oriented artifacts that relates structural properties to high-level quality attributes (e.g., reusability, flexibility, etc.). QMOOD links reusability to Direct Class Coupling, Cohesion among methods of class, Class interface size and Design size in classes. An index for assessing the reuse potential of object-oriented software modules is proposed by Kakarontzas et al. (2013). In this study the C&K metrics suite (Chidamber et al., 1998) is explored for assessing the reusability of 29 OSS projects by applying logistic regression. The proposed model is based on the values of CBO, DIT, LOC, WMC, RFC, and LCOM that directly affect the value of FWBR index. Furthermore, Nair and Selvarani (2010) examine the reusability of a certain class based on the values of DIT, RFC and WMC, as defined in the Chidamber et al. (1998) suite. In total 688 classes were analyzed originating from two medium-sized java projects. The classes in the projects were grouped, based upon different metric values, according to thresholds found in literature. For each group of values a new index was calculated, termed percentage of influence in reusability. Multifunctional regression was then performed across metrics to define the index.

Many studies are also found in literature that consider apart from structural properties other high level properties that may affect reusability adopting the rationale that reusability can be performed at various levels of granularity where properties like customizability, understandability and adaptability are of crucial importance. Sharma et al. utilized Artificial Neural Networks (AAN) to estimate the reusability of software components (Sharma et al., 2009). They proposed four factors affecting component reusability, namely: (a) customizability, measured as the number of setter methods per total number of properties, (b) interface complexity measured in scale from low to high, (c) understandability, depending on the appropriateness of the documentation (demos, manuals, etc.), and (d) portability measured in scale from low to high. Washizaki et al. (2003) suggested a metric-suite capturing the reusability of components, based on data created outside the component environment. Four metrics were suggested for measuring understandability, adaptability, and portability namely existence of meta-information, rate of component observability, rate of component customizability, self-completeness of component and number of external dependencies. Another model found in literature considering a variety of high-level properties affecting reusability is Hristov's model (Hristov et al., 2012). This reusability model consists of eight main characteristics: reuse, adaptability, price, main-

tainability, quality, availability, documentation, and complexity. As this model consists of both high-level quality attributes and low level internal quality attributes we considered that it would be the most relevant and complete model to assess reusability of OSS projects. The reusability model along with indicative metrics is presented in Section 3.

2.3. Open source software reuse

The last years many research papers have explored the reuse potential of open source software artifacts, (Mockus, 2007, Heinemann et al., 2011, Raemaekers et al., 2012) . The plethora of freely available open-source libraries offers great reuse opportunities, with relatively low cost, (Ajila and Wu, 2007). However, no standard reuse process has been yet defined, as practitioners still adopt rather ad-hoc procedures for identifying the most suitable OSS artifacts for reuse.

Mockus (2007) and Heinemann et al. (2011) pointed out that source code reuse in OSS development is more intense compared to commercial/closed source software. Mockus (2007) explored white-box reuse of components originating from Linux and BSD operating systems, to other relatively large OSS projects concluding that the most widely reused components were relatively small in size while in some cases the authors observed a group of files reused without any change. Haefliger et al. (2007) concluded that black-box reuse is the dominant form of reuse by analyzing six open source projects and interviewing their developers. Both architectural and functional reuse was examined within the scope of the study. The first type of reuse required changes to the existing software architecture. The second type of reuse was based on previous architectural reuse. Through architectural reuse of a component, the developer makes functionality available to the program. Heinemann et al. (2011) investigated the occurrence of black-box and white-box reuse in 20 OSS Java projects applying static dependency analysis for quantifying black-box reuse and code clone detection for detecting white-box reuse. The results indicate that black-box software reuse is most common in OSS projects.

Schwittek and Eicker (2013) isolated 36 Java enterprise applications and explored the level of black box reuse in OSS applications. The findings point out that on average, seventy, third party components are being reused, with the majority of them being maintained by the Apache Foundation. Raemaekers et al. (2012) studied 284 Java systems and libraries representing a wide range of business domains and functions coming both from OSS projects and proprietary systems. The results showed that for logging frameworks (e.g., log4j), SDK and XML libraries are among the most frequently reused libraries. Sojer and Henkel (2010) conducted a survey among 686 Sourceforge developers regarding the usage of existing open-source code for the development of new open-source software. The developers' code reuse behavior point out that the more experienced developers with larger personal networks within the OSS community exploit reuse opportunities in more extent. Also the development paradigm that calls for releasing an initial functioning version of the software early leads to increased reuse.

This paper goes beyond current research by providing guidelines to select the relevant components for reuse according to the application domain of a project. In particular this paper:

- Explores the potential of open source software artifact reuse with respect to the application domain. In the study we consider 10 application domains covering a wide range of software categories.
- Adapts an analytical reusability model that consists of both high-level quality attributes and structural properties for assessing reuse potential of certain software artifacts.
- Examines a plethora of quality attributes affecting reusability, as recorded in literature, exploring the potential of both white-box reuse and black-box reuse.

3. Reusability model

To assess the reusability of OSS projects, we tailor the reusability model proposed by Hristov et al. (2012). According to Hristov et al. (2012), reusability can be assessed by quantifying eight main characteristics: *reuse, adaptability, price, maintainability, quality, availability, documentation*, and *complexity*. The quantification of the main characteristics is performed based on an "n-to-m" mapping to certain sub-characteristics, as presented in Fig. 2.

An analysis of the main characteristics of reusability is presented below:

- ***Reuse*** indicates the extent to which a software product is built upon reused components.
- ***Adaptability*** is reflecting the ease with which a component can be adapted when reused in a new system. Adaptability is affected by three aspects: the programming language, the structure of the reused component, and the existence of methods and interfaces for reusing the component.
- ***Price*** indicates how expensive or cheap a component is when reused.
- ***Maintainability*** represents the extent to which a component can be extended, after being added into the system (e.g., during a new version). Maintainability is reflected in the source code of the component.
- ***Quality*** describes the fulfillment of the component's requirements: (a) the errors and bugs of the component, (b) the inclusion of tests and test cases that verify it, and (c) the rating from its users.
- ***Availability*** describes how easy it is to find a component (e.g., instantly, after search, unavailable, etc.).
- ***Documentation*** reflects the provision of documents related to a component. The existence of such documents makes the component easier to understand and therefore reuse.
- ***Complexity*** reflects the internal structure of the component, and is depicted into many aspects of quality (e.g., the easiness to understand and adapt in a new context). Component/System complexity is measured through size, coupling, cohesion, and method complexity.

4. Case study design

In this section we present the design of the case study that we have performed for investigating the reusability of OSS projects across different application domains. In particular, we examined 596 open source software (OSS) projects that can be classified to ten application domains (namely: *Audio and Video Applications, Games, Science and Engineering Applications, Graphics Applications, Communications Applications, Business Applications, Security and Utilities, System Administration,* and *Software Development Tools*), so as to examine differences in their levels of reusability. The reusability model that we used for this assessment is a tailored version of the one proposed by Hristov et al. (2012) — see Section 3. The main reason that we performed a case study rather than another type of empirical evaluation—e.g., survey (Pfleeger and Kitchenham, 2001) or experiment (Wohlin et al., 2000) is that we wanted to investigate the reuse opportunities offered by real OSS projects, in the sense that OSS reuse constitutes a large portion of the complete population of software reuse. The following sub-sections are presenting the parts of the case study protocol, as described by Runeson et al. (2012).

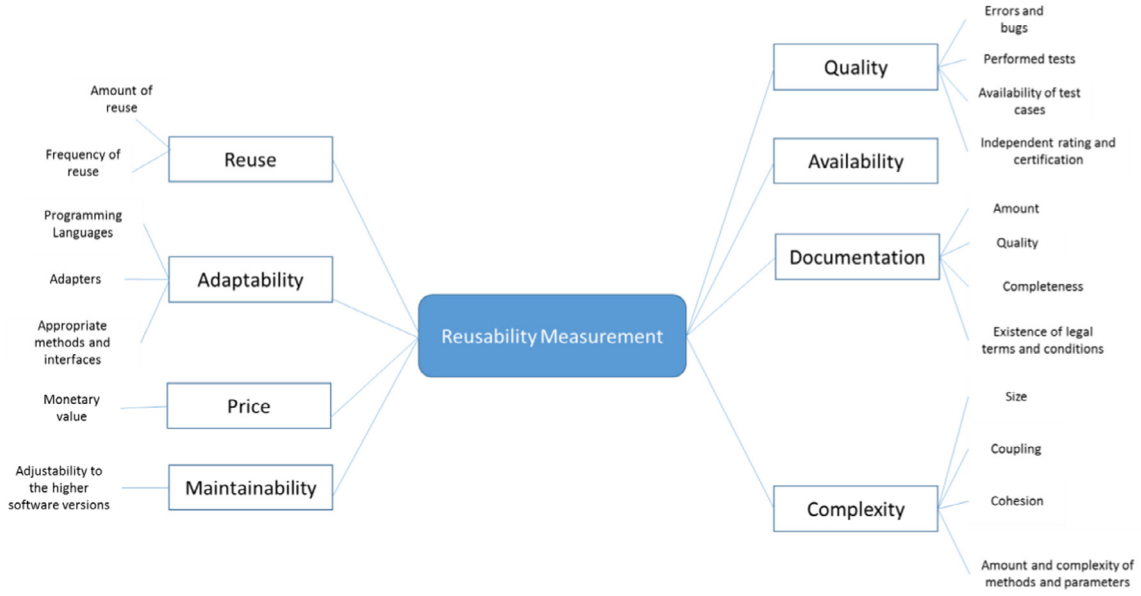


Fig. 2. Reusability measurement model.

4.1. Objectives and research questions

The goal of this case study is to compare the reusability of OSS projects in different domains. The evaluation will be made in terms of the seven characteristics, described in Section 3, i.e., all expect for price (which is not applicable for open source projects). In order to achieve this goal, we decompose the goal to seven research questions—one for each characteristic:

- [RQ_1] *Are there differences in the extent to which OSS projects of different domains are reused?*
- [RQ_2] *Are there differences in the adaptability of OSS projects of different domains?*
- [RQ_3] *Are there differences in the maintainability of OSS projects of different domains?*
- [RQ_4] *Are there differences in the external quality of OSS projects of different domains?*
- [RQ_5] *Are there differences in the availability of reusable sets of classes extracted from OSS projects of different domains?*
- [RQ_6] *Are there differences in the documentation of OSS projects of different domains?*
- [RQ_7] *Are there differences in the complexity of OSS projects of different domains?*

Answering the aforementioned seven research questions, will give us an overview on how the application domains compare in terms of each factor in isolation. Next, by synthesizing the information obtained by answering these questions, we will attempt to holistically evaluate the reusability of projects in different domains. Being able to rank the application domains in terms of reusability can provide useful information to both researchers and practitioners (see Section 6). For example, researchers can easily identify applications domains that are in need for applying methods and tools that improve reusability, whereas practitioners will be aware of application domains from which they can more easily reuse components.

4.2. Case selection and unit analysis

The case study of this paper is an embedded multiple-case study (Runeson et al., 2012), in which the context is OSS development, as cases we consider eight application domains, whereas as units of analysis the projects that belong to them. In order to select as many units of analysis as possible for our case study, we exploited a meta-repository[2] in which we have gathered data from various projects, hosted in various OSS repositories (e.g., Sourceforge, GitHub, etc.), namely Percerons[1]. Percerons is a software engineering platform (Ampatzoglou, 2013a) created by one of the authors with the aim of facilitating empirical research in software engineering, by providing: (a) indications of componentizable parts of source code, (b) quality assessment, and (c) design pattern instances. The platform is consistently used for empirical research in the last four empirical software engineering conferences—ESEM' 13 (Ampatzoglou, 2013b), ESEM'14 (Griffith and Izurieta, 2014), and ESEM' 15 (Arvanitou et al., 2015) and (Reimanis, 2015). The identification of units of analysis is performed automatically, by dumping the complete database of the repository. The collection of data for each unit of analysis is in some cases automatically performed by parsing the Percerons database, whereas in other cases manual parsing of projects webpages was necessary (see Section 4.3). To populate the Percerons database a number of projects have been selected for every application domain in *Sourceforge.net* (see Section 4.3 for more details on application domains). The criteria that we have used for selecting projects and populating the Percerons repository are discussed below:

- projects should be written in Java due to the limitations of the pattern detection, component identification, and metrics calculation tools that we have used for storing assessments in Percerons
- projects should provide a compiled version of the project since the tools use binary code as input
- projects should be highly ranked by popularity in Sourceforge. To sort projects we ranked them by popularity and then selected the top-100. From them we filtered those that meet the first two criteria.

The number of projects examined from each application domain varied from 25 to 135, as presented in Table 2.

[2] Examples of similar meta-repositories include FLOSSmole (http://flossmole.org/), which is a collaborative repository for OSS research data and analyses developed by Howison et al. (Howison et al., 2008).

Table 2
Study demographics (application domains).

Application domain	Number of projects
Video	50
Games	135
Business & enterprise	50
Home & education	32
Science & engineering	40
Communications	59
Development tools	125
Graphics	55
Security & utilities	25
System administration	25

Currently *Percerons* provides data on 596 projects that belong to ten application domains (consisting of ***more than 117,000 classes***—i.e., non-trivial systems with on average more than 200 classes each). Although this number is small compared to the population of available OSS projects we believe it is representative enough.

4.3. Data collection & analysis

To answer our research questions for every OSS project that we analyzed we recorded the following variables:

[*V*1] ***Software Name***: The name of the OSS project that we analyzed.

[*V*2] ***Application Domain***: The software domain names are derived from sourceforge.net, where the open source projects are being developed and maintained. Table 2 presents the 10 different application domains from which reusable projects are retrieved along with the distribution and frequencies of projects in each domain. We note that all application domain of *Sourceforge.net* have been considered in this study and therefore no selection bias has been introduced. Additionally, the mapping between projects and application domains has been recorded based on the *Sourceforge.net* entry in the sense that the developers of the software are expected to provide a valid classification for their software.

[*V*3] ***Reuse***: As a way to quantify the reuse aspect of the proposed model, we use three different metrics: (a) the number of reused external libraries in the project, and (b) the number of files reused from other projects as an indicator of *amount of reuse*, whereas (c) the amount of system files that have been reused in other projects of our dataset, as an indicator of *reuse frequency*.

[*V*4] ***Adaptability***: As a proxy of component adaptability, we use the reusability index defined by Bansiya and Davies (2002), namely AD_QMOOD. Although, based on the naming it seems that there is a quality mismatch in the selection of quality attributes, a closer examination of the provided definitions suggest that reusability as defined in QMOOD matches adaptability as defined by Hristov et al. (2012). In particular, they both define the ability of a component to be easily adapted from the source systems that it has been developed for, to the target system in which it will be reused (i.e., adaptation to the new context). According to Bansiya and Davies (2002) adaptability can be defined as a function of component coupling, cohesion, interface size, and component size in terms of classes.

[*V*5] ***Maintainability:*** As a way to quantify maintainability, we use the metric for extendibility, as defined by Bansiya and Davies (2002), namely MD_QMOOD. According to the model, extendibility can be calculated as a function of several object-oriented characteristics that either benefit or hinder the extension/easy maintenance of the system: abstraction, coupling, inheritance, and polymorphism.

[*V*6] ***External Quality:*** As a proxy of external quality, we use four metrics, based on the decomposition of external quality, as suggested by Hristov et al. (2012). More specifically, we use: (a) the number of opened and closed bugs as a measure of the *errors and bugs*, (b) the number of unit test files as a proxy for *performed tests* and *availability of test cases,* and (c) the average rating by the users of the software as a proxy for *independent rating and certification*.

[*V*7] ***Availability:*** The number of independent and compileable components that have been identified for the specific project in the *Percerons* repository are quantified into the variable NCOMP. The methodology that is used to identify components from open source projects and populate the repository has been proposed by Ampatzoglou et al. (2013b). The used algorithm is based on the identification of reusable sets of classes, by applying a path-based strong component algorithm (Gabow, 2000). In its current state *Percerons* provides 6.4 million candidate components that concern the 8 aforementioned application domains. However, we need to note that the majority of these components are not completely independent, since the algorithm (Ampatzoglou et al., 2012) stores components with efferent coupling (Martin, 2003) less than 10. As a measure of availability, we consider approximately 50,000 components that are completely independent and compileable (i.e., efferent coupling equals zero).

[*V*8] ***Documentation:*** Documentation is the only factor that has to some extent been subjectively evaluated. However, in order to: (a) assure the replicability of the process, and (b) guarantee the common understanding of our evaluation from the readers of the manuscript, we have defined a set of clearly defined, non-overlapping, and unambiguous criteria for OSS project assessment with respect to quantity and quality of documentation. To assess the *amount, completeness*, and *quality* of documentation, we performed manual inspection to the website of the project in *Sourceforge.net*. In particular, we counted the number of text files, multimedia files, or active forum topics. Based on this, we classified projects with 0–4 items as *very low*, projects with 5–10 items as *low*, 10–20 items as *moderate*, 20–50 as *high*, and projects with more than 50 items as *very high*. In case that no help was directly provided, but the community offered a number of emails as contacts, we assessed this project with *low*.

[*V*9] ***Complexity***: As proxies of complexity, we used four metrics that capture system size [***V9.1***], average class coupling [***V9.2***], average class cohesion [***V9.3***], and average method complexity [***V9.4***], as prescribed by the reusability model (see Section 3). We adopted the metrics on design quality from the suite of Chidamber et al. (1998). In particular, we used one coupling (CBO: Coupling between objects), one cohesion (LCOM: Lack of Cohesion of Methods), and one complexity metric (WMC: Weighted Method per Class).

All the metrics presented in Table 3 (except for documentation) are calculated initially at class level and then aggregated to project level. To aggregate measures from class to project level we selected the average (AVG) as an approach to assess the overall metric value for a certain project. Arvanitou et al. (2016) suggest that when measuring in-large (at project level) we need to employ the average function for producing stable versions of code-level metrics contrary to min, max and sum functions that are more relevant to class-level evaluations. Additionally, according to Fenton and Bieman (2014) stable metrics should be selected for large-scale evaluations that have the ability to derive the overall trend of the corresponding metric. Complementary, we selected to employ the relevant stable metrics (LCOM, CBO, RFC and NOC) compared to other

Table 3
Reuse factors and the associated metrics.

Reuse factor	Metric name	Values/ calculation method	Examined artifacts	Level of granularity
Reuse	Lib_Reuse	The number of reused external libraries in the project. Number of files in lib folder.	Binary Distribution	Project
	Reuse_Amount	The number of reused external libraries in the project that exist in our dataset (internal reuse).		
	Reuse_Frequency	The amount of files that have been reused in other projects of our dataset.	Source Code	
Adaptability:	AD_QMOOD	= **−0.25 DCC + 0.25 CAM + 0.5 CIS + 0.5 DSC** **DCC** is calculated as the number of different classes that a class is related to, based on attribute declarations. **CAM** is calculated using the summation of intersection of parameters of a method with the maximum independent set of all parameter types in the class. **CIS** is calculated as the number of public methods in a class **DSC** is calculated as the total number of classes	Source Code	Class
Maintainability	MD_QMOOD	= **0.25*ANA- 0.5*DCC+0.5*NOH+0.5*NOP** **NOH** is the number of class hierarchies in the design. **ANA** is calculated as the average number of classes from which a class inherits information. **NOP** is derived as a count of methods that can exhibit polymorphic behavior. Such methods in Java are marked as abstract. **DCC** is defined as in Adaptability factor	Source Code	Class
External Quality	BUGS	The number of opened bugs	Bug/Issue Tracker	Project
	C_BUGS	The number of closed bugs		
	N_TESTS	The number of unit test files	Source Code	
	RT	The average rating by the users of the specific software	OSS Repository	
	N_DOWN	The number of downloads of the specific project retrieved form sourceforge.org		
Availability	NCOMP	The number of independent and compileable components. (i.e., efferent coupling equals zero)	Source Code	Class
Documentation	DOC	The number of text files, multimedia files, or active forum topics. very low: projects with 0–4 items, low: projects with 5–10 items moderate: projects with 10–20 items high: projects with 20–50 items very high: projects with more than 50 items	Project webpage (official or repository)	Project
Complexity	CBO	**CBO** measures the number of classes that the class is connected to, in terms of method calls, field accesses, inheritance, arguments, return types and exceptions. High coupling is related to low maintainability and understandability. The range of values of the CBO metric is [0, NOC-1].	Source Code	Class
	LCOM	**LCOM** measures the dissimilarity of pairs of methods, in terms of the attributes being accessed. High Lack of Cohesion is an indicator of violating the single responsibility principle (Martin, 2003), which suggests that each class should provide the system with only one functionality.		
	WMC	**WMC** is calculated as the average Cyclomatic Complexity (CC) among methods of a class. High WMC results in difficulties in maintaining and understanding the system.		
	NOC	Number of Classes (**NOC**), in the sense that it provides an estimation of the amount of functionality offered by the system (Morisio et al., 2002). The size of the system needs to be taken into account, since smaller systems are expected to be less coupled, less complex, to have less classes as leafs in hierarchies and use less inheritance trees. Thus, assessing quality characteristics (apart from cohesion), without taking into account the size of the system would be unfair for application domains with larger projects.		

candidate ones (CIS, DAC, NOCC) for addressing the complexity of a certain project (Arvanitou et al., 2016).

4.4. Data analysis

The data analysis of this case study has been performed as a two-step process (see Table 4). In the first step we calculate descriptive statistics and perform Independent Sample Kruskal-Wallis test to check for difference across application domains. We note that we have selected to perform a non-parametric test for investigating the differences across domains, since the variables of interest are not normally distributed. Before the analysis, we "cleaned" the dataset from extreme values and outliers: (a) visually by using boxplots, and (b) statistically by checking iteration after iteration the cases with residuals with three or more standard deviations from the mean. Additionally, in order to reduce the effect size of metrics' range of values, we normalized all metric scores before the execution of the tests. The normalization has transformed all scores to fit the [0, 1] range, by retaining the proportions of scores for different projects. The formula used for normalization is:

$$metric_{normalized} = \frac{metric_{actual} - min}{(max - min)}$$

As a second step, we synthesize the indications provided by the different variables that are used as proxies for the same aspect of reusability. To be able to synthesize metrics in a safe way, without aggregating metrics with different range values, we transform the numerical indication to ranking. In particular, we characterize every software project with its rank compared to the rest of the population, as top-10%, top-20%, …, and top-100% included in the analysis. To achieve this, all of the software projects that partici-

Table 4
Data analysis and presentation overview.

RQ	Variable	Analysis
Reuse	**Testing Variables:**	Descriptive statistics
	• Lib_Reuse, Reuse_Amount, Reuse_Frequency	
	Grouping variable:	Independent Sample Kruskal-Wallis test
	• Application Domain	
Adaptability	**Testing variables:**	Descriptive statistics
	• AD_QMOOD	Independent Sample Kruskal-Wallis test
	Grouping variable:	
	• Application Domain	
Maintainability	**Testing variables:**	Descriptive statistics
	• MD_QMOOD	Independent Sample Kruskal-Wallis test
	Grouping variable:	
	• Application Domain	
External Quality	**Testing variables:**	Descriptive statistics
	• O_BUGS, C_BUGS, N_TESTS, RT, N_DOWN	Independent Sample Kruskal-Wallis test
	Grouping variable:	
	• Application Domain	
Availability	**Testing variables:**	Descriptive statistics
	• NCOMP	Independent Sample Kruskal-Wallis test
	Grouping variable:	
	• Application domain	
Documentation	**Testing variables:**	Frequencies (pie chart)
	• DOC	
	Grouping variable:	
	• Application domain	
Complexity	**Testing variables:**	Descriptive statistics
	• CBO, LCOM, WMC, CC, NOC	Independent Sample Kruskal-Wallis test
	Grouping variable:	
	• Application domain	

pate in the analysis are sorted based on the values of each metric and the percentages are then calculated with respect to the application domain of the software project. For each of these classes we perform frequency analysis, and visualize the outcome in radar charts, first grouped by aspect of reusability, and next by providing a synthesized view representing the complete reusability model.

5. Results

In this section we present the results of our case study, organized by research question. In each subsection, for each indicator of each reusability characteristic, we present: (a) the descriptive statistics for each application domain, and (b) the Kruskal-Wallis test across application domains. We note that in this section, no interpretation of results is performed, since it is preferred that all results are collectively discussed in Section 6.

5.1. Reuse

In this section, we are presenting results that concern the extent to which OSS projects are based on software reuse, and are reused in other systems. As explained in Section 4.3, reuse is quantified by three indicators: (a) the number of reused external libraries in the project (*lib-reuse*), (b) the number of files reused from other projects (*reuse-amount*), and (c) the amount of system files that have been reused in other projects of our dataset (*reuse frequency*). The descriptive statistics and the results of the independent sample Kruskall-Wallis test are presented in Table 5. Each row of the table corresponds to one application domain, whereas the columns represent the mean values and standard deviations of the aforementioned normalized indicators. The last row of the table presents the results of the Kruskall-Wallis test. We note that other descriptive statistics have been omitted due to the nature of the dataset. For example, MIN and MAX have not been presented since the variables are normalized ($\min = 0.0$ and $\max = 1.0$), whereas MODE values are not applicable for continuous variables. The domain with the higher mean value for each indicator is highlighted with bold font, whereas the indicators for which there are statistically significant differences across domains are highlighted with light grey cell shading. We note that with italics we highlight application domains that are close to the dominant one in terms of the mean value.

In addition to that, in Fig. 3, we present the percentage of application domains' projects that are ranked in the top-20% of the examined OSS population, in terms of each metric. We note that the higher this frequency the most reuse is exploited within the specific application domain. The scales in graph have been pruned to the maximum value of each category, since inter-charts comparisons are not applicable. Thus, we preferred to not have a uniform scale (i.e., 0% to 100%) for all of them to safeguard their readability. Based on the finding of both Table 5 and Fig. 3 we can observe that *System Administration* tools, *Software Development* tools and *Science and Engineering* applications are the domains in which most reuse activity takes place. On the other end, reuse seems to be a rather neglected software engineering technique, while developing OSS *Games*.

5.2. Adaptability

In this section, we are presenting results that concern the extent to which OSS projects are offering source code parts that are easily adaptable in the target system. As explained in Section 4.3, adaptability is quantified through the corresponding metric of the QMOOD suite. The descriptive statistics and the Kruskal-Wallis test are presented in Table 6. The presentation of data in the table is similar to the one in Section 5.1. Additionally, similarly to Section 5.1, the frequency of occurrence of OSS projects of each domain in the top-20% of the sample, is visualized in Fig. 4. The findings of the case study suggest that *Graphics, Business* and *Enterprise* Applications, as well as *Software Development* tools offer the most easily adaptable source code, compared to other application domains. On the other hand Security& Utilities and System Administration applications prove to be less adaptable compared to applications from other domains.

Table 5
Software reuse intensity.

Application Domain	Lib-Reuse		Reuse-Amount		Reuse-Frequency	
	Mean	*Std. Dev.*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*
Audio and Video	0.029	0.06	*0.054*	0.17	0.009	0.04
Business and Enterprise	*0.047*	0.16	0.027	0.05	0.001	0.01
Communications	0.029	0.06	*0.056*	0.09	*0.019*	0.08
Software Development	*0.050*	0.06	0.036	0.07	0.006	0.03
Games	0.011	0.03	0.032	0.09	0.016	0.07
Graphics	0.028	0.05	0.036	0.05	*0.022*	0.11
Home and Education	*0.048*	0.03	0.048	0.07	*0.020*	0.04
Science and Engineering	*0.050*	0.08	**0.063**	0.09	**0.029**	0.097
Security Utilities	0.035	0.14	0.032	0.12	**0.029**	0.14
System Administration	**0.055**	0.12	*0.060*	0.14	0.009	0.02
Kruskal-Wallis test (sig.)	0.000		0.950		0.468	

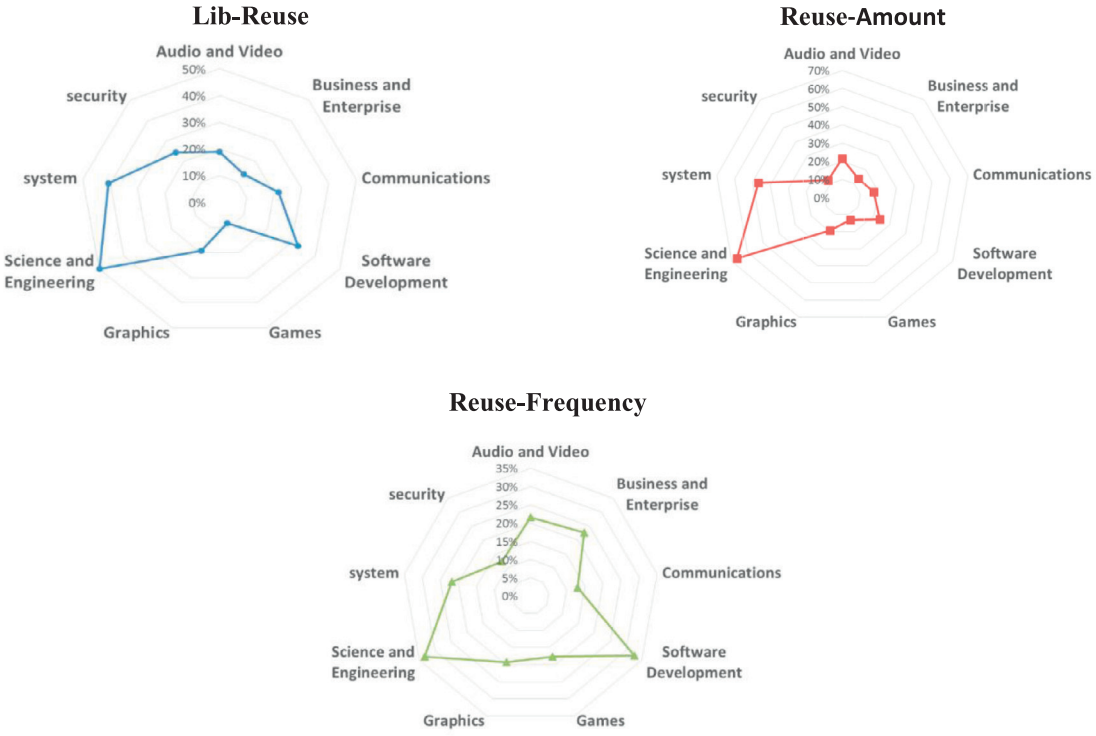


Fig. 3. Software reuse intensity.

Table 6
Software adaptability.

Application Domain	AD_QMOOD	
	Mean	*Std. Dev.*
Audio and Video	0.184	0.04
Business and Enterprise	*0.206*	0.15
Communications	0.177	0.05
Software Development	*0.197*	0.09
Games	0.187	0.06
Graphics	**0.217**	0.11
Home and Education	0.190	0.06
Science and Engineering	0.178	0.06
Security Utilities	0.093	0.01
System Administration	0.087	0.01
Kruskal-Wallis test (sig.)	0.000	

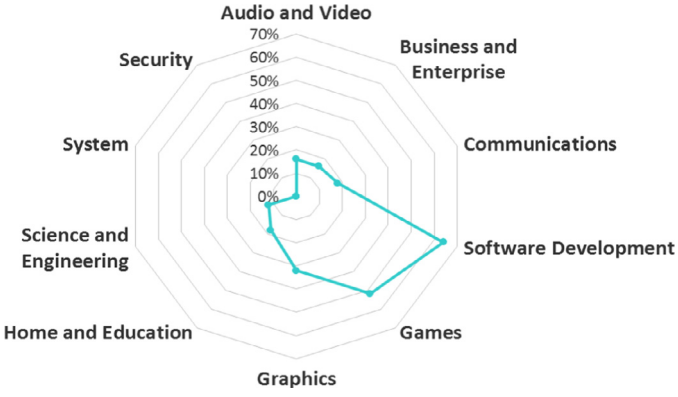


Fig. 4. Software adaptability.

5.3. Maintainability

In this section, we are presenting results that concern the extent to which OSS projects are offering source code parts that are easily maintainable. As explained in Section 4.3, reuse is quantified through the extendibility metric of the QMOOD suite. The descriptive statistics and the Kruskal-Wallis test (sig.) are presented

Table 7
Software maintainability.

Application Domain	MD_QMOOD	
	Mean	*Std. Dev.*
Audio and Video	0.376	0.05
Business and Enterprise	**0.406**	**0.08**
Communications	0.367	0.06
Software Development	0.372	0.07
Games	0.373	0.07
Graphics	0.371	0.04
Home and Education	0.365	0.06
Science and Engineering	*0.397*	0.06
Security Utilities	0.254	0.28
System Administration	*0.420*	0.32
Kruskal-Wallis test (sig.)	0.000	

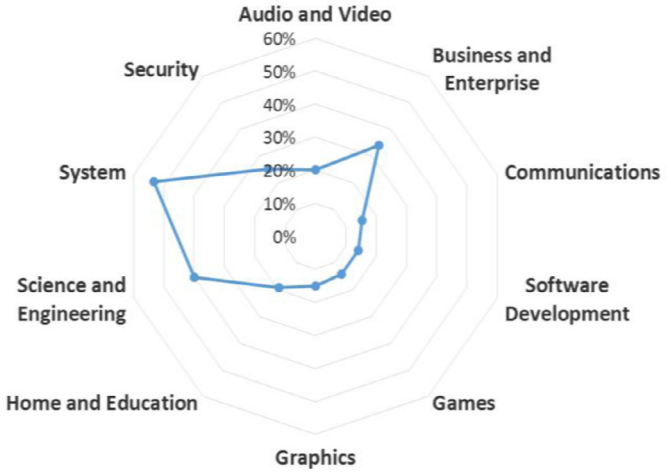


Fig. 5. Software maintainability.

in Table 7. The presentation of data in the table is similar to the one in Section 5.1. Based on the results presented in Table 7 and Fig. 5, we suggest that *Science, Engineering, Business* and *Enterprise* and *System Administration* applications offer the most maintainable source code implementations, compared to the rest application domains. Security and Utilities applications present lower maintainability than the rest applications.

5.4. External quality

In this section, we are presenting results that concern the external quality of OSS projects. As explained in Section 4.3, external quality is assessed by five indicators: (a) the number of test cases (*tests*), (b) the average rating by its users (*rating*), (c) the number of downloads during the last week (*downloads*), (d) the number of open bugs (*open bugs*), and (e) the number of closed bugs (*closed bugs*). The descriptive statistics and the Kruskal-Wallis test (sig.) are presented in Table 8. The presentation of data in the table is similar to the one in Section 5.1.

The results concerning external software quality are diversified across the different external quality metrics. Initially all application domains present very high values on *Rating*. This is explained by the fact that in the analysis we selected the most popular projects according to their rating on Sourceforge. In terms of *Number of Downloads* and *Closed Bugs* Software development applications appear to be more active. In terms of the *Tests* performed Science and Engineering applications present higher values. Game applications are generally ranked rather low. A possible explanation for this is that the rating of the OSS games, might mostly reflect users' enjoyment rather than the quality of the software per se. Finally, we need to note that, as expected the number of opened and closed bugs are naturally connected, since no bugs can be closed, unless they open in a previous version.

5.5. Availability

In this section, we are presenting results that concern the average number of independent components available from OSS of a specific application domain. As explained in Section 4.3, the components have been retrieved by an online component repository, named *Percerons*. The descriptive statistics and the Kruskal-Wallis test (sig.) are presented in Table 9. The presentation of data in the table and figure is similar to the one in Section 5.1. Based on the findings presented in this section, *Security & Utilities* and *System Administration* applications offer the most reusable assets in the OSS ecosystem.

5.6. Documentation

In this section, we are presenting results that concern the evaluation of the documentation in OSS projects of different application domains. As explained in Section 4.3, documentation is assessed by three indicators: (a) amount, (b) completeness, and (c) quality of the documentation that are all accumulated in a single scale variable, namely *Documentation*. To present the differences among application domains, we present bar charts, that represent the frequency percentages of different evaluations (varying from *very low* to *very high*), for different application domains. The results are presented in Fig. 8. We note that in order for an application domain to be properly documented, we expect to observe a right skewness on the histogram. Based on this clarification, *Science* and *Engineering* applications, *Business and Enterprise* applications, *System Administration* tools and *Software Development* tools, are the most documented ones in the Sourceforge repository.

5.7. Complexity

In this section, we are presenting results that concern the complexity of OSS projects. As explained in Section 4.3, complexity is quantified by four indicators: (a) the average cohesion classes (*cohesion*), (b) the average class coupling (*coupling*), (c) the average design size in classes (*size*), and (d) the average cyclomatic complexity of methods (*complexity*). The descriptive statistics and the Kruskal-Wallis test (sig.) are presented in Table 10. The presentation of data in the table and figure is similar to the one in Section 5.1. The results on complexity are diverse compared to the ones presented in the previous sections, in the sense that the most dominant application domains appear to lag in terms of structural complexity. In particular, *Communication* applications appear to be the ones with the least coupling between classes and the lowest average method complexity. However, a possible interpretation for this might be the relatively small size in classes of such applications. On the other hand, the most coherent projects can be found in the *Business* and *Enterprise* application domains, whereas the largest in terms of size in *Science* and *Engineering* applications.

6. Discussion

In this section we interpret the results obtained by our case study and provide some interesting implications for researchers and practitioners. Additionally, we illustrate a possible use case of exploiting our results in a reuse decision making process by elaborating on the example of Fig. 1 (see Introduction).

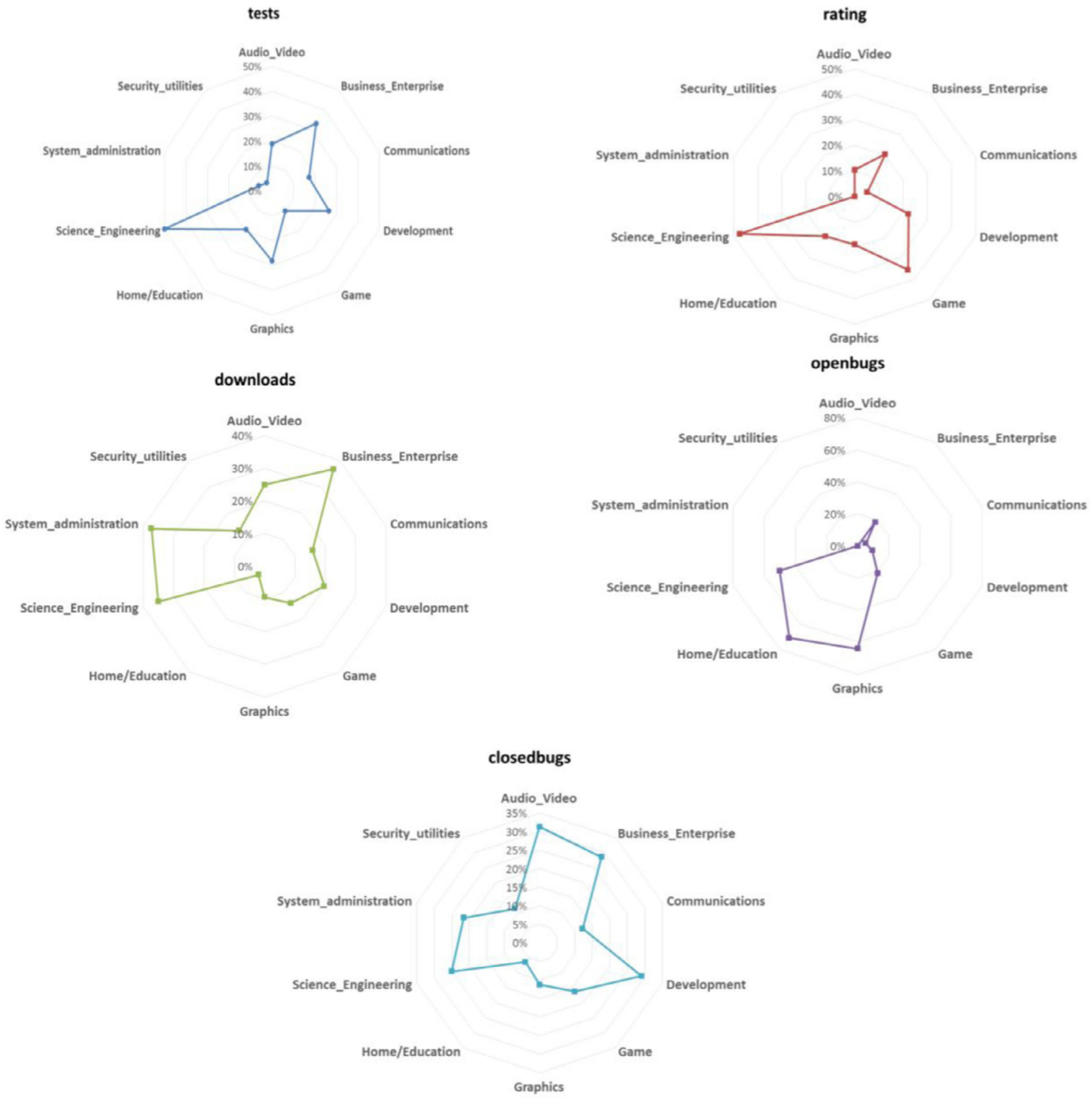


Fig. 6. Software external quality.

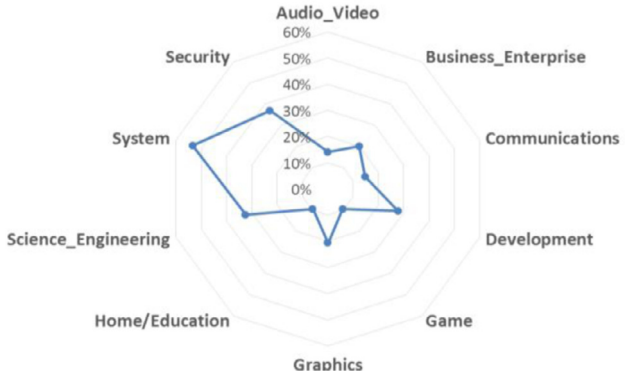


Fig. 7. Component availability.

6.1. Interpretation of results

First, we can observe that *System Administration, Business, Enterprise, Science*, and *Engineering* applications, and *Software Development* tools are the OSS projects with the higher levels of reusability. The rest application domains present good levels of quality in some sub-characteristics (e.g., Games for rating, Communication applications for class coupling and method complexity, etc.), but these exceptional cases are rather spread across the different reusability aspects. The top-3 optimal application domains per metric/reuse factor are summarized in Table 11. The application domains are ranked based on the radar charts presented for each metric in the corresponding results section (see Section 5.1–5.7)

The aforementioned results can be considered as intuitive in the sense that these application domains (i.e., *Business, System Administration, Enterprise, Science*, and *Engineering* applications, and *Software Development* tools) are the most "serious" ones, among those that are studied in this paper. Particularly, it is expected that developers of such applications are most aware of software engineering technologies, compared to those of other, most "soft-

Table 8
Software external quality across OSS application domains.

Application Domain	Tests		Rating		Downloads		Open Bugs		Closed Bugs	
	Mean	*Std. Dev.*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*
Audio and Video	0.008	0.03	0.956	0.08	**0.189**	0.45	0.451	0.15	**0.118**	0.21
Business and Enterprise	0.015	0.04	0.955	0.08	0.034	0.05	0.489	0.05	*0.111*	0.24
Communications	0.017	0.09	*0.973*	0.04	0.035	0.12	0.390	0.32	0.045	0.12
Software Development	*0.031*	0.11	0.968	0.05	*0.111*	0.30	0.507	0.05	*0.104*	0.20
Games	0.004	0.02	*0.977*	0.04	0.032	0.11	0.420	0.11	0.039	0.12
Graphics	0.007	0.02	*0.977*	0.03	0.014	0.05	*0.555*	0.01	0.038	0.11
Home and Education	0.010	0.03	*0.988*	0.03	0.010	0.05	*0.540*	0.01	0.034	0.08
Science and Engineering	**0.036**	0.12	*0.980*	0.03	0.035	0.06	0.463	0.16	*0.085*	0.17
Security Utilities	0.004	0.08	**0.991**	0.02	0.016	0.06	**0.560**	0.02	0.031	0.08
System Administration	0.007	0.04	0.926	0.13	0.091	0.21	0.435	0.06	0.028	0.06
Kruskal-Wallis test (sig.)	0.000		0.403		0.000		0.04		0.01	

Table 9
Component availability.

Application Domain	Availability	
	Mean	*Std. Dev.*
Audio and Video	0.013	0.02
Business and Enterprise	0.015	0.02
Communications	0.010	0.01
Software Development	0.018	0.02
Games	0.012	0.03
Graphics	0.016	0.02
Home and Education	0.015	0.02
Science and Engineering	*0.027*	0.02
Security Utilities	**0.072**	0.22
System Administration	*0.064*	0.07
Kruskal-Wallis test (sig.)	0.000	

skilled" or artistic application domains (e.g., *Graphics, Audio* and *Video* applications, or *Games*). Concerning the design quality of the applications with respect to their complexity we can distinguish applications belonging to the *Communications* domain. *Communication* projects present better average design metrics scores compared to other types of applications, being among the smallest in terms of size. *Home and Education* projects present the worse levels of design quality and complexity as well. Projects belonging to this application domain appear to be the least coherent and among the most coupled systems. The average size though, of *Home* and *Education* applications is not among the largest ones to explain the high complexity values. Nevertheless, it consistently had the largest variance in metric values, which is an indicator that this category is too broad and should be decomposed for further analysis. In fact, in Sourceforge the category consists of four very diverse sub-categories. The largest systems are *Science* and *Engineering, Development* Tools and *Business* and *Enterprise* projects. Among them, the best levels of design quality are provided by *Development Tools*, which can be due to the fact that they are created by developers with strong software engineering background. Furthermore, in the entertainment field we can find *Games, Graphics, Audio* and *Video* applications that share some common modules, presenting similar metrics (size, complexity and inheritance).

6.2. Implications to researchers and practitioners

The results of this study provide useful information both to researchers and practitioners. First, they can provide guidance on the existence of reuse opportunities for practitioners. Based on the results of this study, software developers can have indications on the feasibility of reuse in different application domains. Science and Engineering application developers can exploit the great reuse opportunities offered by OSS components in these domains (highest availability). These application domains offer the most components per project and are among the optimal among all characteristics of reusability. On the other hand, the results of this study can provide guidance on case selection for researchers. Nowadays, more and more researchers perform empirical studies on OSS projects. The results of the study can guide researchers in selecting appropriate application domains to identify as many units of analysis as possible. Finally, as future work we suggest the investigation of: (a) the actual reuse rates of OSS components in other applications, (b) the reusability of these components can be tested by software engineers through experiments, and (c) a process for systematically reusing these components can be introduced.

6.3. Applicability of empirical findings

By further focusing on the example of Section 1, we are now ready to present data that would be able to guide the software architect of Fig. 1 in his decision making approach. Assuming that

Table 10
Software complexity.

Application Domain	LCOM		CBO		NOC		WMC	
	Mean	*Std. Dev.*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*
Audio and Video	0.766	0.16	0.675	0.19	0.130	0.11	0.911	0.05
Business and Enterprise	**0.788**	0.15	0.654	0.14	0.145	0.15	0.862	0.18
Communications	0.736	0.26	**0.708**	0.17	0.261	0.15	**0.922**	0.05
Software Development	0.783	0.17	0.651	0.16	0.060	0.07	0.903	0.10
Games	0.780	0.18	0.662	0.15	0.221	0.21	0.911	0.05
Graphics	0.778	0.17	0.677	0.14	0.187	0.28	0.894	0.07
Home and Education	0.703	0.21	0.698	0.13	0.220	0.07	0.906	0.05
Science and Engineering	0.785	0.19	0.587	0.19	**0.345**	0.33	0.912	0.05
Security Utilities	0.725	0.18	0.687	0.12	0.210	0.23	0.911	0.04
System Administration	0.760	0.04	0.662	0.08	0.045	0.02	0.900	0.02
Kruskal-Wallis test (sig.)	0.39		0.00		0.04		0.02	

Table 11
Panorama of the results.

Reuse Factor	Metric Name	top-1	top-2	top-3	Optimal w.r.t. Factor
Reuse	Lib_Reuse	Science and Engineering	System Administration	Software Development	Science and Engineering
	Reuse_Amount	Science and Engineering	System Administration	Software Development	
	Reuse_Frequency	Science and Engineering	Software Development	Business and Enterprise	
Adaptability	AD_QMOOD	Software Development	Games	Graphics	Software Development
Maintainability	MD_QMOOD	System Administration	Science and Engineering	Business and Enterprise	System Administration
External Quality	O_BUGS	Home and Education	Graphics	Science and Engineering	Science and Engineering
	C_BUGS	Audio and Video	Software Development	Business and Enterprise	
	N_TESTS	Science and Engineering	Software Development	Business and Enterprise	
	RT	Science and Engineering	Games	Software Development	
	N_DOWN	Business and Enterprise	System Administration	Science and Engineering	
Availability	NCOMP	System Administration	Security Utilities	Software Development	System Administration
Documentation	DOC	Science and Engineering	Business and Enterprise	System Administration	Science and Engineering
Complexity	CBO	Science and Engineering	Business and Enterprise	Home and Education	Communications / Science and Engineering
	LCOM	Communications	Games	System Administration	
	WMC	Communications	Home and Education	Audio and Video	
	NOC	Science and Engineering	Software Development	Business and Enterprise	

Table 12
Exploration of white-box alternatives.

Reuse Factor	*Metric Name*	*Candidate Application Domains*	
		Software Development	*Communications*
Complexity	CBO	0	0
	LCOM	0	5
	WMC	0	5
	NOC	3	0
Complexity Score (35%)		0.75 (0.26)	2.5 (0.87)
External Quality	O_BUGS	0	0
	C_BUGS	3	0
	N_TESTS	3	0
	RT	1	0
	N_DOWN	0	0
Complexity Score (20%)		1.4 (0.28)	0 (0)
Maintainability	MD_QMOOD	0	0
Maintainability Score (20%)		(0)	(0)
Availability	NCOMP	1	0
Availability Score (25%)		1 (0.25)	0 (0)
Total Evaluation of Alternative		**0.79**	**0.87**

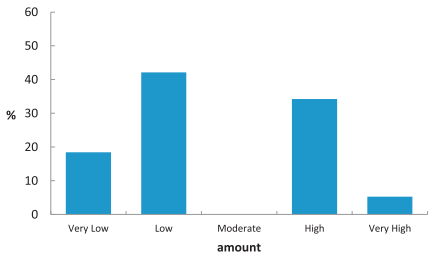
the company intends to perform black-box reuse, the reuse factor of interest do not include any internal characteristic. Therefore, the architect should focus on: ***reuse***, ***external quality***, and ***documentation***. Based on the findings presented in Table 11, the decision seems trivial, since the *Software Development Tools* application domain out performs the *Communications* domain for all metrics in these reuse factors. On the other hand, assuming that the company is interested in white-box reuse, the decision making becomes more complex. Assuming that the architect selects a weighted method with three factors: ***complexity (35%), external quality (20%), maintainability (20%)***, and ***availability (25%)*** a more elaborate selection process is required. For instance let us consider that the following point system is used: 5-points for the top-1 domain, 3-points for the top-2, and 1-point for the top-3. The results of the scoring process is presented in Table 12, based on which we can observe that for white-box reuse purposes (given the selected weights and criteria), the Communications application domain offers more reuse opportunities.

7. Threats to validity

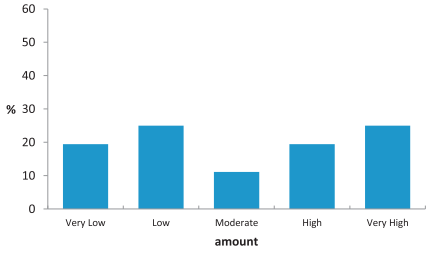
In this section we discuss the threats to validity that we have identified for this study. The section is organized based on the four main types of threats to validity for quantitative research within software engineering as appointed by Runeson et al. (2012): *construct, internal, external* and *reliability* validity. Construct validity defines how effectively a test or experiment measures up to its claims. Internal validity is related to the examination of causal relations examining whether an experimental environment is adequate to support the claim. External validity examines whether the results of a study can be generalized to other cases. Reliability is associated to the reproducibility of the study, i.e. the ability of other researchers to repeat the same process, collect data and reach the same results.

Construct Validity: The set of metrics selected to measure the adequacy of the extracted components could pose a possible threat to construct validity. For this reason we selected Hristov's framework (2012) for evaluating the adequacy of reusable components. Components evaluation was performed by quantifying a plethora of main characteristics: reusability, adaptability, maintainability, quality, availability, documentation, and complexity each of which synthesized by the values of several metrics as depicted in Fig. 1. Thus in total 18 metrics were addressed for the scope of this study in an attempt to avoid the case in which any important attribute is missing from the analysis. We note that the metrics that we selected for the quantification of the reuse factors are only few of the potential candidates. Therefore, we acknowledge the existence of alternative metrics, and we do not claim that the set we selected consists of optimal reusability predictors. Still the validity of the selected set and the level to which they affect reusability across OSS domains were addressed in Section 7. Additionally we attempted to normalize data by using the ranking of possible metrics instead of the actual metrics (see Section 4). In particular we should also mention specifically for the metric referring to *documentation* that we acknowledge that the documentation quantity (number of documents, webpages, etc.) is not a sole indicator of the documentation quality and we plan in the future to assign the value of this metric based on questionnaires provided to OSS developers. In this context another possible threat to construct validity is whether the identified candidate components can be stand-alone reusable artifacts that can be migrated to settings beyond their own derived application. Since in the literature there is no known algorithm that captures accurately 100% of all intended components our study was based on an exhaustive search process that provided adequate recall rates.

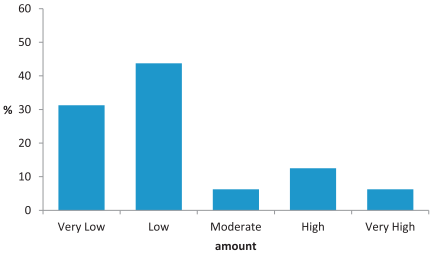
Internal validity: The proposed study attempted to form an association between quality characteristics (such as maintainability, availability and interactivity) and the application domain of the components extracted. We cannot claim that the quality attributes under study form a causal relationship with the application domain of the components but only that the results indicate trends and common practice when it comes to selecting components. Some of these trends can be verified intuitively while others may be surprising.



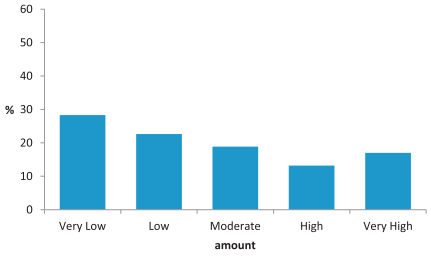
(a) Audio and Video



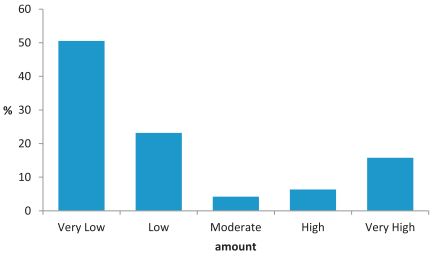
(b) Business and Enterprise



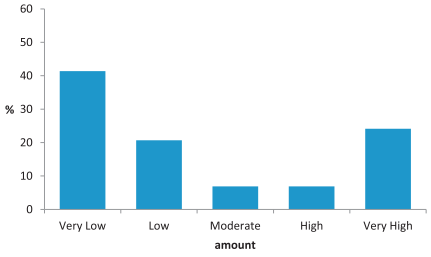
(c) Communication



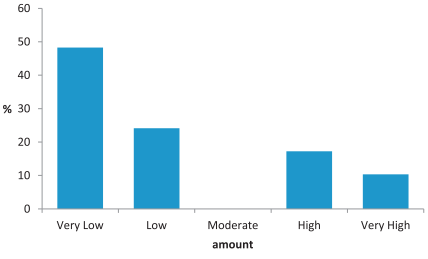
(d) Development Tools



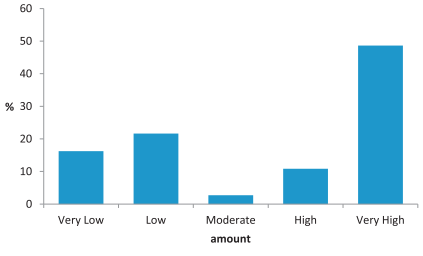
(e) Games



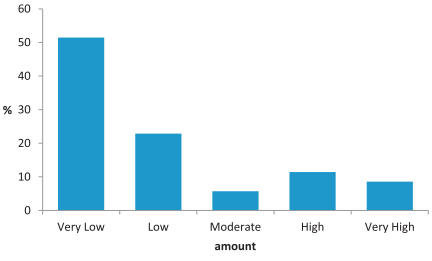
(f) Graphics



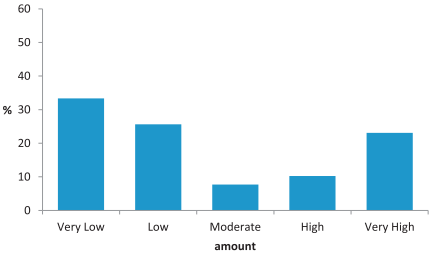
(g) Home and Education



(h) Science and Engineering



(i) Security & Utilities



(j) System Administration

Fig. 8. Software documentation.

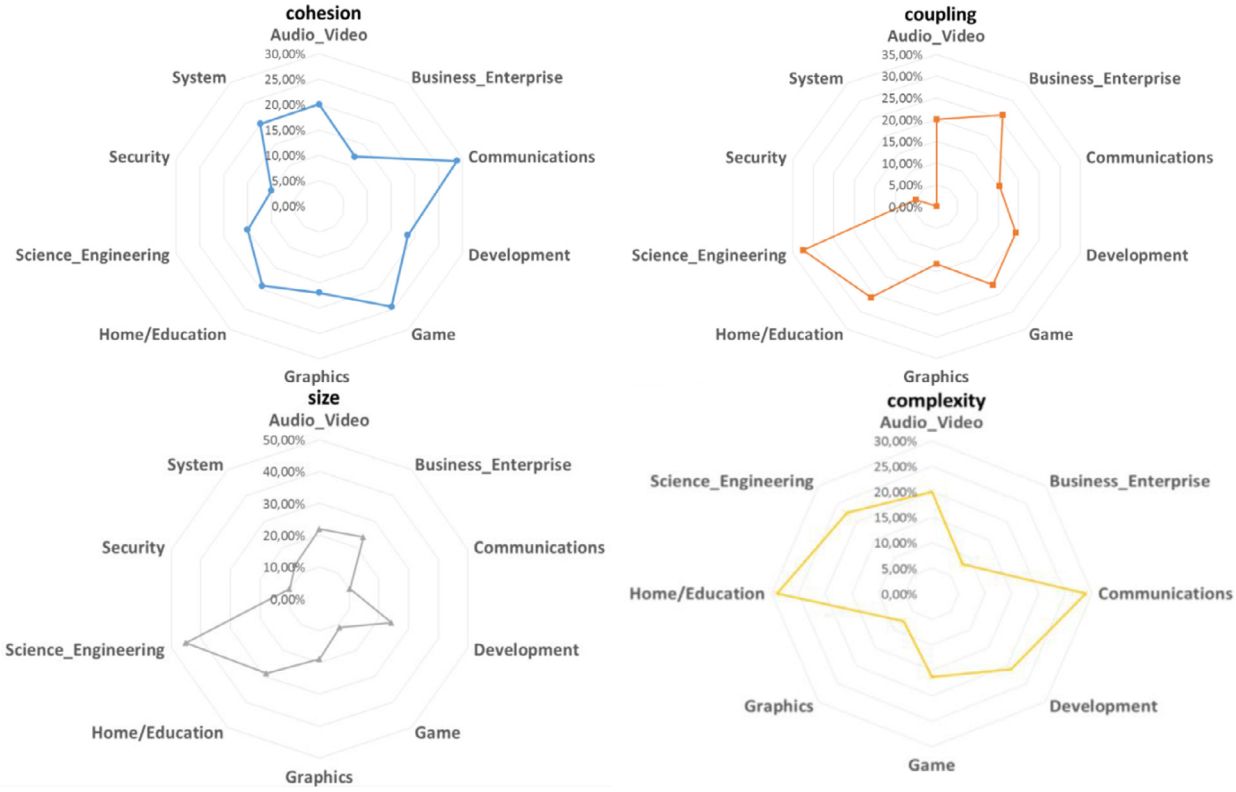


Fig. 9. Software complexity.

Reliability: With respect to reliability we believe that the followed research process ensures the reliability and safe replication of our study. The process that has been followed in this study has been thoroughly documented in the case study protocol, provided in Section 4. Therefore, the re-production of the case study can be easily performed by any interested researcher. Additionally the extraction of the dataset and the associated structural metrics was performed with the help of the publicly available meta- repository namely, Percerons and therefore any interested researcher can repeat the analysis and derive the same results. However concerning researcher bias, we should state that it was introduced during the data collection and data analysis process in one case. In particular in the data collection phase, the only possible bias can be identified while quantifying the metric value of the level of documentation provided for each project the component was derived from. To gather data on the documentation variable, a manual recording process was employed and performed by the first author. In order to increase the reliability of this process the second and the third author validated the results.

External validity: Concerning generalizability, we can say that, as in all such studies, in the case the sample of projects was altered and the component extraction was performed on additional or even different applications, then the results might be different as well. To ensure the generalizability of our results we have measured a wide range of applications providing a respectful number of reusable components, from a variety of application domains. Therefore we believe that the selected cases ($\sim$ 600 open source applications), offer a large and representative sample of the population. However, as the scope of the study are open source projects developed in Java, a possible bias regarding the programming language may affect our results limiting their transferability to other programming languages. Another possible threat comes from the choice of Percerons as a component identification repository, pooling reusable assets from Sourceforge, as source for the study objects could bias our selection, as a certain kind of open source developers could prefer other project repositories (such as maven, GitHub or Google Code). Additionally we should mention that a possible bias is inserted when selecting the most popular projects from each application domain (based on star rating provided by Sourceforge). This choice ensures that the relevant candidate projects for examining reusability aspects are included in the analysis but introduces bias in the values of the metric RATING that assess external quality, as the majority of projects present four-star rating and up. Though, a replication of this study in an even larger component set would be valuable in verifying current findings.

8. Conclusions

Software reuse is often viewed as a promising approach for reducing costs, shortening time-to-market and improving quality in software development. Especially in the vast world of open source projects it is expected that almost every piece of required functionality is somewhere to be found. However, reuse is far from trivial due to a number of reasons pertaining to the availability, adaptability, maintainability, documentation and complexity of existing components. In this study, we have performed an embedded multiple case study on 596 Java OSS projects to assess their reusability with respect to the application domain that they belong to. The findings of the study suggest that application domains differ when analyzed from the perspective of seven distinct aspects of reusability. The application domains *Science and Engineering Applications* and *Software Development Tools*, have proven to offer the most reuse-friendly components, while on the other hand, *Game* developers seem to pay limited attention to reuse. Such evidence can be valuable to both researchers and practitioners as a guide for selecting appropriate projects for studying or exploiting reuse opportunities.

Acknowledgement

This work was financially supported by the action "Strengthening Human Resources Research Potential via Doctorate Research" of the Operational Program "Human Resources Development Program, Education and Lifelong Learning, 2014-2020″, implemented from State Scholarship Foundation (IKY) and co-financed by the European Social Fund and the Greek public (National Strategic Reference Framework (NSRF) 2014–2020).

References

Ajila, S.A., Wu, D., 2007. Empirical study of the effects of open source adoption on software development economics. J. Syst. Softw 80 (September(9)), 1517–1529 Elsevier.

Ampatzoglou, A., Stamelos, I., Gkortzis, A., Deligiannis, I., 2012. Methodology on extracting reusable software candidate components from open source games. In: Proceeding of the 16th International Academic MindTrek Conference. Finland. ACM, pp. 93–100.

Ampatzoglou, A., Michou, O., Stamelos, I., 2013a. Building and mining a repository of design pattern instances: Practical and research benefits. In: Entertainment Computing, 4. Elsevier, pp. 131–142.

Ampatzoglou, A., Gkortzis, A., Charalampidou, S., Avgeriou, P., 2013b. An embedded multiple-case study on OSS design quality assessment across domains. In: 7th International Symposium on Empirical Software Engineering and Measurement (ESEM' 13) 10–11 October. ACM/IEEE Computer Society, Baltimore, USA, pp. 255–258.

Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2015. Introducing a ripple effect measure: a theoretical and empirical validation. 9th International Symposium on Empirical Software Engineering and Measurement (ESEM' 15). ACM/IEEE Computer Society, Beijing, China.

Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2016. Software metrics fluctuation: a property for assisting the metric selection process. Inf. Softw. Technol. 72 (April), 110–124.

Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, G., 2005. An industrial case study on reuse oriented development. 21st International Conference on Software Maintenance (ICSM'05). IEEE Computer Societ.

Bansiya, J., Davies, C.G, 2002. A hierarchical model for object-oriented design quality assessment. *Trans. Softw. Eng.* IEEE Comput. Soc. 28 (January(1)), 4–17 y, 283–292, September 2005.

Boehm, B., 1999. Managing software productivity and reuse. Computer 32 (9), 111–113.

Chidamber, S.R., Darcy, D.P., Kemerer, C.F., 1998. Managerial use of metrics for object oriented software: an exploratory analysis. Trans. Softw. Eng. IEEE Comput. Soc. 24 (August(8)), 629–639.

Cho, H., Yang, J.S., 2008. Architecture patterns for mobile games product lines. In: Proceedings of the 2008 International Conference on Advanced Communication Technology (ICACT'08). Korea. IEEE Computer Society, pp. 118–122. 17–20 February.

Clements, P.C, 1995. From subroutines to subsystems: component-based software development. Am. *Program.* 8 31–31.

Crnkovic, I., Hnich, B., Johnson, T., Kiziltan, Z., 2002. Specification, implementation, and deployment of components. Commun. Assoc. Comput. Mach. 45 (October(10)), 35–40.

Fenton, N., Bieman, J., 2014. Software Metrics: A Rigorous and Practical Approach, third ed. CRC Press, Inc., Boca Raton, FL, USA.

Folmer, E., 2007. Component based game development – a solution to escalating costs and expanding deadlines. In: 10th International Symposium on Component Based Software Engineering (CBSE' 07) 9–11 July. Springer-Verlag, Medford, MA, USA, pp. 66–73.

Frakes, W.B., Fox, C.J., 1996. Quality improvement using a software reuse failure modes model. *Trans. Softw. Eng.* IEEE Comput. Soc. 22 (April(4)), 274–279.

Franch, X., Carvallo, J.P., 2003. Using quality models in software package selection. Softw. IEEE Comput. Soc. 20 (January/February(1)), 34–41.

Gabow, H.N, 2000. Path-based depth-first search for strong and bi-connected components. Inf. Process. Lett. 74 (May(3–4)), 107–114 Elsevier.

Griffith, I., Izurieta, C., 2014. Design pattern decay: the case for class grime. 8th International Symposium on Empirical Software Engineering and Measurement (ESEM '14) 18–19 September. ACM/IEEE Computer Society, Torino, Italy.

Haefliger, S., Krogh, G.V., Spaeth, S., 2007. Code reuse in open source software. Manage. Sci. 54 (November(1)), 180–193 PubsOnline.

Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., MIrlbeck, M., 2011. On the extent and nature of software reuse in open source java projects. 12th International Conference on Top Productivity through Software Reuse (ICSR' 11). Springer.

Hristov, D., Hummel, O., Huq, M., Janjic, W., 2012. Structuring software reusability metrics for component-based software development. 7th International Conference on Software Engineering Advances (ICSEA).

Howison, J., Conklin, M., Crowston, K., 2008. FLOSSmole: a collaborative repository for FLOSS research data and analyses. In: Integrated Approaches in Information Technology and Web Engineering: Advancing Organizational Knowledge Sharing. IGI Global, pp. 18–27.

Jacobson, I., Griss, M., Jonsson, P., 1997. Software reuse: architecture. Process and Organization for Business Success. ACM Press/Addison-Wesley Publ. Co., New York, NY, USA.

Johnson, I., Snook, C., Edmunds, A., Butler, M., 2004. Rigorous development of reusable, domain-specific components, for complex applications. 3rd International Workshop on Critical Systems Development with UML (CSDUML'04). Springer.

Kakarontzas, G., Constantinou, E., Ampatzoglou, A., Stamelos, I., 2013. Layer assessment of object-oriented software: a metric facilitating white-box reuse. J. Syst. Softw. 86 (2), 349–366.

Krueger, C.W., 1992. Software reuse. Comput. Surv. 24 (2), 131–184 ACM.

Lau, K.K., Wang, Z., 2005. A taxonomy of software component models. In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA). IEEE, pp. 88–95.

Lee, W.P., Liu, L.J., Chiou, J.A., 2006. A component-based framework to rapidly prototype online chess games for home entertainment. In: Proceedings of the International Conference on Systems, Man and Cybermetrics (SMC'06) 8–11 October. IEEE Computer Society, Taipei, Taiwan, pp. 4011–4016.

Karlsson, .A., 1995. Software reuse: a Holistic Approach. John Wiley & Sons, Inc.

Martin, R.C., 2003. Agile Software development: Principles, Patterns and Practices. Prentice Hall, New Jersey.

Mili, H., Mili, F., Mili, A., 1991. Reusing software: issues and research directions. IEEE Trans. Softw. Eng. 21 (6), 528–562.

Mockus, A., 2007. Large-scale code reuse in open source software. 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS' 07). IEEE Computer Society.

Mohagheghi, P., Conradi, R., 2007. Quality, productivity reuse: a review of industrial and economic benefits of software studies. Empir. Softw. Eng. 12 (5), 471–516.

Morisio, M., Romano, D., Stamelos, I., 2002. Quality, productivity, and learning in framework-based development: an exploratory case study. *Trans. Softw. Eng.* IEEE Comput. Soc. 28 (September(9)), 876–888.

Nair, T.R.G., Selvarani, R., 2010. Estimation of software reusability: an engineering approach. SIGSOFT Softw. Eng. Notes 35 (January(1)), 1–6.

Pfleeger, S.L., Kitchenham, B., 2001. Principles of survey research part 1: turning lemons into lemonade. Special Interest Group Softw. 26 (November(6)), 16–18 ACM.

Raemaekers, S., Deursen, A.V., Visser, J., 2012. An analysis of dependence on third–party libraries in open source and proprietary systems. 6t*h International Workshop on Software Quality and Maintainability (SQM' 12)* March.

Reimanis, D., 2015. A research plan to characterize, evaluate, and predict the impacts of behavioral decay in design patterns. 13th International Doctoral Symposium on Empirical Software Engineering (IDOSE' 15).

Rubén, P.D, 1993. Status report: software reusability. IEEE Softw. 10.3, 61–66.

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

Schwittek, W., Eicker, S., 2013. A study on third party component reuse in java enterprise open source software. In: 16th International Symposium on Component-based Software Engineering (CBSE' 13). ACM, pp. 75–80.

Sharma, A., Grover, P.S., Kumar, R., 2009. Reusability assessment for software components. SIGSOFT Softw. Eng. Notes 34 (February(2)), 1–6.

Sojer, M., Henkel, J., 2010. Code reuse in open source software development: quantitative evidence, drivers, and impediments. J. Assoc. Inf. Syst. 11 (December(12)), 868–901.

Standish, T.A., 1984. An essay on software reuse. IEEE Trans. Softw. Eng. 5, 494–497.

Washizaki, H., Yamamoto, H., Fukazawa, Y., 2003. A metrics suite for measuring reusability of software components. In: Software Metrics Symposium, 2003. Proceedings. Ninth International. IEEE.

Wohlin, C., Host, M., Runeson, P., Ohlsson, M, Regnell, B., Wesslen, A., 2000. Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers.

Maria Eleni Paschali is a PhD Student at the Department of Informatics of the Aristotle University of Thessaloniki (Greece), in the group of Software Engineering. She holds an MSc degree in Applied Informatics from University of Macedonia, Greece (2011), and a BSc degree in Informatics and Communications from the Technological Education Institute of Serres, Greece (2008). Her research interests include software reuse, open source software development, and computer games.

Dr. Apostolos Ampatzoglou is a Guest Researcher at the Department of Informatics of the Aristotle University of Thessaloniki, where he carries out research in the area of software engineering. In the period from 2013 to 2016, he was an Assistant Professor at the University of Groningen (the Netherlands). He holds a BSc on Information Systems (2003), an MSc on Computer Systems (2005) and a PhD in Software Engineering by the Aristotle University of Thessaloniki (2012). His current research interests include technical debt, source code analysis, software maintainability, software quality management, open source software engineering and software design. He has published more than 55 articles in international journals and conferences. He is/was involved in over 10 research and development projects in Information & Communication Technologies with funding from national and international organizations. Finally, he serves as a reviewer in numerous leading journals of the software engineering domain, and as a member of various international conference program committees.

Dr. Stamatia Bibi is a Lecturer of software engineering in the Department of Informatics and Telecommunications at the University of Western Macedonia, Kozani, Greece. She holds a BSc in Informatics (2002) and a PhD (2008) in software engineering from the Aristotle University of Thessaloniki, Greece. Her research interests include software process models, estimation of software development cost and quality, cloud computing, and open source software.

Dr. Alexander Chatzigeorgiou is a Professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in Electrical Engineering and the PhD degree in Computer Science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999 he was with Intracom S.A., Greece, as a telecommunications software designer. Since 2007, he is also a member of the teaching staff at the Hellenic Open University. His research interests include object-oriented design, software maintenance, and software evolution analysis. He has published more than 150 peer-reviewed articles in international journals, conference proceedings and books. He is a member of the Technical Chamber of Greece.

Dr. Ioannis Stamelos is a Professor at the Department of Informatics of the Aristotle University of Thessaloniki, where he carries out research and teaching in the area of software engineering. He holds a diploma of Electrical Engineering (1983) and a PhD in Computer Science by the Aristotle University of Thessaloniki (1988). His current research interests are focused on open source software engineering, software project management and software education. He has published more than 200 articles in international journals and conferences. He is/was the scientific coordinator or principal investigator for his University in 30 research and development projects in Information & Communication Technologies with funding from national and international organizations.