The Journal of Systems and Software 167 (2020) 110618

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

CODE reuse in practice: Benefiting or harming technical debt

Daniel Feitosa[a,*], Apostolos Ampatzoglou[b], Antonios Gkortzis[c], Stamatia Bibi[d], Alexander Chatzigeorgiou[b]

[a] Data Research Centre, University of Groningen, Groningen, the Netherlands
[b] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
[c] Department of Management Science and Technology, Athens University of Economics and Business, Greece
[d] Department of Informatics and Telecommunications, University of Western Macedonia, Kozani, Greece

ARTICLE INFO

Article history:
Received 1 July 2019
Revised 26 April 2020
Accepted 27 April 2020
Available online 23 May 2020

Keywords:
Technical debt
Reuse
Case study

ABSTRACT

During the last years the TD community is striving to offer methods and tools for reducing the amount of TD, but also understand the underlying concepts. One popular practice that still has not been investigated in the context of TD, is software reuse. The aim of this paper is to investigate the relation between white-box code reuse and TD principal and interest. In particular, we target at unveiling if the reuse of code can lead to software with better levels of TD. To achieve this goal, we performed a case study on approximately 400 OSS systems, comprised of 897 thousand classes, and compare the levels of TD for reused and natively-written classes. The results of the study suggest that reused code usually has less TD interest; however, the amount of principal in them is higher. A synthesized view of the aforementioned results suggest that software engineers shall opt to reuse code when necessary, since apart from the established reuse benefits (i.e., cost savings, increased productivity, etc.) are also getting benefits in terms of maintenance. Apart from understanding the phenomenon per se, the results of this study provide various implications to research and practice.

1. Introduction

Technical Debt (TD) is a software engineering metaphor that relates the construction of poor-quality software with incurring additional cost, and more specifically to going into debt (Kruchten et al., 2012). Based on the TD metaphor, software industries save an amount of money by not developing the system in optimal design-time quality levels—termed as *principal* (Ampatzoglou et al., 2015). However, later the maintenance costs increase—this amount is called *interest* (Ampatzoglou et al., 2015) due to lowered maintainability, whenever maintenance tasks occur (their frequency map to *interest probability* (Seaman and Guo, 2011)). By acknowledging the tremendous relevance of technical debt in software development industries, the TD community is striving to produce methods and tools for TD Management (TDM) that would reduce the amount of TD in the software, by either preventing the accumulation of additional TD, or by removing the existing one (Arvanitou et al., 2019). To this end, the roots of TD have been extensively studied (Kazman et al., 2015; Mo et al., 2015; Xiao et al., 2016) along with factors that encourage developers to manage it efficiently (Amanatidis et al., 2018; Ernst et al., 2014; Palomba et al., 2014; Potdar and Shihab, 2014).

Under the prism of understanding possible reasons that lead to TD accumulation, it becomes relevant to investigate existing software engineering practices, which might enforce TD accumulation. To this end, in this paper we focus on software reuse: through reuse, artifacts developed originally for one system (*source system*), are used again (either "*as are*" or after modification) in the construction of another *target system* (Krueger, 1992). The intensity of reuse as a phenomenon, becomes evident by considering that code reuse from 1.3 K popular Open Source Software (OSS) projects (e.g., log4j, jUnit, etc.) in other projects, represents approximately 316 K staff years and tens of billions of dollars in development costs (Ampatzoglou et al., 2013). Some of the main benefits that promoted reuse as a leading practice in software development is the increase of development productivity (Baldassare et al., 2005; Frakes and Kang, 2005), the improvement of several aspects of software quality (Ajila and Wu, 2007; Baldassare et al., 2005; Lim, 1994), and better software reliability in cases when the reused components are already tested when they are selected for reuse (Joos, 1994; Juristo and Moreno, 2001; Lim, 1994; Morisio et al., 2002; Poulin, 1999; Rine, 1997).

* Corresponding author.
E-mail addresses: d.feitosa@rug.nl (D. Feitosa), a.ampatzoglou@uom.edu.gr (A. Ampatzoglou), antoniosgkortzis@aueb.gr (A. Gkortzis), sbibi@uowm.gr (S. Bibi), achat@uom.edu.gr (A. Chatzigeorgiou).

https://doi.org/10.1016/j.jss.2020.110618
0164-1212/

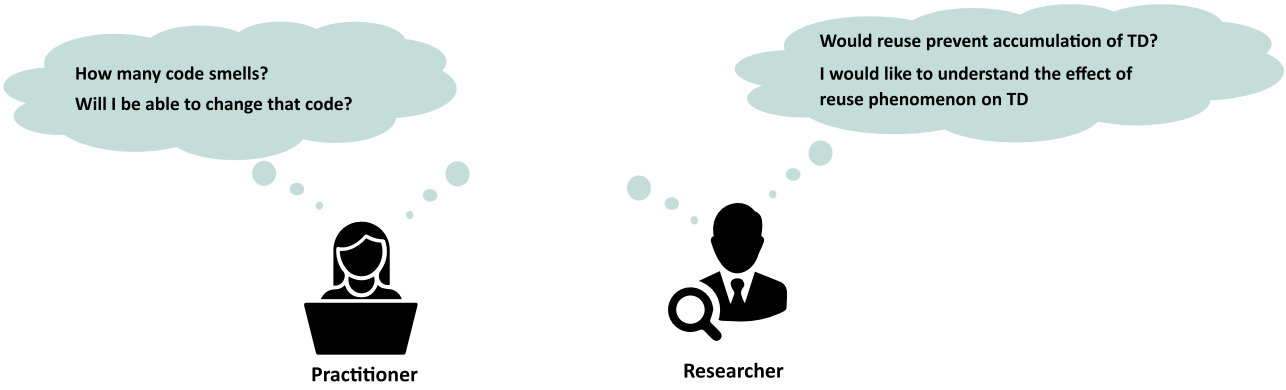

Fig. 1. Stakeholders' concerns—contributions of the study.

According to Barros-Justo et al. (2018), research efforts should focus on the use of quality models for testing the actual impact of reuse benefits, with maintainability appointed as the most important one, while linking them to specific practices. In this direction Mikkonen and Taivalsari (2019) stress that the revival of software reuse, due to the enormous amount of freely available source code on the web, poses new challenges to the software engineering community related to the systematic analysis of the compatibility and the properties of popular open source components. From the above, it becomes clear that although various aspects of business and product qualities have been studied, with respect to reuse, there is still the need to empirically explore the structural properties of the freely available reused components and their effect to the quality of the software in which they are integrated (see Section 2).

In this paper we target this specific knowledge gap, by investigating the relation of open-source code reuse to the structural quality of the target system. More specifically, we investigate if on average the structural quality of source code that is written from scratch (*native code*), is lower or higher compared to *reused code*. Additionally, by acknowledging the relevance of TD in modern software quality assurance processes, we focus our assessment of structural quality to technical debt measurements. Software reuse can be performed in two ways (Heinemann et al., 2011): (a) white-box, in which the reused code is inserted in the application as source code (i.e., directly editable); and (b) black-box, in which the reused code is inserted in the application in a binary form (i.e., it cannot be edited and maintained). Regarding black-box reuse, the notion of TD is not considered fitting, in the sense that artifacts reused in a black-box fashion, do not involve any maintenance. Therefore, for the purpose of our study we focus only on white box reuse. Finally, we note that TD is a far more multifaceted term, and that it is not restricted to code only. However, to keep the scoping of this study realistic, and by considering that reuse of small code-chunks (such as classes) are more likely to affect code TD rather than architecture, we focus this investigation on code TD only.

In particular, we scope our research so as to answer the following concerns of software practitioners and researchers, as illustrated in Fig. 1.

- **Practitioner**: "*Will the code that I want to reuse have a low number of code smells, so that I can easily bring it to the quality standards of the company?*"
- **Practitioner**: "*Will the code that I will reuse: follow object-oriented practices (*e.g.*, low coupling, high cohesion, etc.) that facilitate maintenance, or will it hinder fixing of defects and modification of functionality?*"
- **Researcher**: "*Is code reuse a practice that would be helpful in preventing the accumulation of code TD, or would writing native code yield better software quality?*"
- **Researcher**: "*Which particular aspects of the TD metaphor are hurt and which benefit from code reuse?*"

To answer the aforementioned concerns, we have performed a large-scale case study on approx. 50 Million (Mo) lines of code, from 400 different projects. The projects are first divided into its reused and native parts (i.e., classes), then reused classes are characterized as white-box or black-box, then we measure TD aspects for native and white-box reused classes, and perform statistical analysis, to draw meaningful conclusions. The main contribution of this study from a research point of view, is the exploration of the relation between white-box code reuse and code TD in a large-scale, which until now is rather unexplored. In terms of practical considerations, the results are expected to be useful for technical debt prevention, as explained in Fig. 1.

The rest of the paper is organized as follows: in Section 2, we present related work, i.e., studies that investigate the effect of reuse on software quality—since this is the first study on reuse and TD. In Section 3, we present background information, focusing on TD terminology and measurement/assessment strategies. In Section 4, we outline the case study design, whereas in Section 5 we present the obtained results. Next, in Section 6 we discuss them, by contrasting them to existing literature, providing tentative interpretations, and implications for researchers and practitioners. Finally, in Section 7 we discuss threats to validity, and in Section 8 we conclude the paper.

2. Related work

In this section we present related work to our study. Since to the best of our knowledge, this is the first study that investigates the effect of software code reuse (as discussed by de Almeida et al., 2005) on technical debt, in this section we broaden the scope of reporting to studies that explore the effect of reusing code to software quality. Special emphasis will be given to structural product quality, in the sense that it is closer to TD, compared to other quality views (Kitchenham, 1996). Nevertheless, the terms technical debt and software reuse (not restricted to code) have already been discussed in current literature.

First, Martinez-Fernandez et al. (2013) considered technical debt as a parameter for their economic model, while reusing at the software architecture level, by implementing reference architectures. Second, Yli-Huumo et al. (2013) investigates technical debt management techniques when using software product lines, i.e., one of the prominent ways of systematic reuse. To achieve this goal, they have conducted interviews with 12 practitioners; the

results suggest that: (a) TD is mostly formed as a result of intentional decisions made during the project to reach deadlines; and (b) customer satisfaction was identified as the main reason for taking TD in short-term but it turned to economic consequences and quality issues in the longer perspective. Also, the results suggested that product line managers did not had any specific plan for technical debt management. Both these studies are substantially different from this work, in the sense that they focus on architecture, rather than source code.

The positive effect of reuse on software quality has been verified by several studies (Lim, 1994; Frakes and Kang, 2005; Mahagheghi and Conradi, 2007, 2008). Lim (1994) analyzed metrics collected from two reuse programs completed by Hewlett-Packard and reported improved quality, in terms of defect density, increased productivity and reduced time to market. In this direction Frakes and Kang (2005) performed an exploratory study on the relationship between the amount of reuse and the quality of software modules developed in C/C++ within an industrial context. The authors analyzed four software projects and concluded that software reuse is positively correlated to software quality, as assessed by the developers, and negatively correlated to the error density. Another study that added evidence to the quality benefits acquired from reuse was performed by Mohagheghi and Conradi (2007), who examined the potentials of software reuse in a telecommunications project. The results of their case study revealed that software reuse contributed to lower fault-density and less modified code between the successive releases of the software product under study (Mohagheghi and Conradi, 2008). The quality benefits acquired from software reuse are also reported in the review performed by Mohagheghi and Conradi (2007) who assessed the effects of reuse in an industrial context. Concluding, by transferring the aforementioned results to the TDM context, one could argue that ***reuse leads reduced interest probability***, in the sense that the reused code has fewer defects; thus, it undergoes more rarely corrective maintenance, and therefore produces interest more sparsely.

In terms of structural quality, we have identified very few studies that investigate the effect of reuse on software product quality. Deniz and Bilgen (2014) performed a case study to test whether the quality of software code is improved as reuse rates of the products increase. The authors analyzed software modules developed by a defense industry (mainly developed in C++) in order to calculate complexity and class level metrics proposed by Chidamber and Kemerer (1994). Their findings show that some metrics (number of classes, lines of code, depth of inheritance tree) do not correlate with changing reuse rate. However, Coupling and Complexity metrics are significantly improved when the reuse rate increases, a fact that indicates the positive effect of reuse on structural quality of code. Constantinou et al. (2014) explored the effect of white-box reuse on software quality. In particular, they investigated more than 1 K Java projects and highlighted that on average reused classes where of higher complexity, less coherent, and more closed coupled to other classes, compared to system classes. Additionally, Zaimi et al. (2015) explored the effect reuse decisions on reusability, extendibility, flexibility, and effectiveness of the target software. To achieve this goal, the authors explored the reuse decision taken along the evolution of 5 well-known Java open-source projects. The results suggested that no statistically significant effect of reuse decisions to design-time quality attributes could be argued. Nevertheless, the update of a library version usually led to an (on average) improved quality. Finally, Nikolaidis et al. (2019) compared the levels of TD in source code reused from StackOverflow and suggested that reused code is in the majority of the cases of better quality, in terms of technical debt, compared to the code of the rest of the target system. This result was based on the analysis of approximately 50 reused code chunks of non-negligible size.

To summarize the aforementioned results, in Table 1 (for each identified study), we characterize it as directly or indirectly (e.g., through structural properties) associated to TD, we note the TD concept that is being analyzed, the used research method (qualitative, quantitative, descriptive), and the sign of the relation (positive or negative). Based on Table 1 (and the detailed descriptions of related works), we can conclude that: (a) there is limited evidence on the relation of TD Principal and Reuse; and (b) the results on the relation of TD Interest and Reuse are inconclusive, in the sense that some studies suggest positive correlations, other negative ones, and other no correlation all.

3. Technical debt terminology, measurement, and assessment

In this section we discuss all background information that is necessary for facilitating the understanding of this study. In particular, we present: (a) the TD metaphor; (b) an overview of TD concepts; and (c) the ways that they can be assessed, or measured. For the purpose of this study, we have decided to work at the source code level. To ease the understandability of this section, we present each concept along with each way of measure (or assess) and then we proceed to the next concept.

3.1. Introduction to technical debt

Maintenance is one of the most effort-intensive activities in the software lifecycle, since it stands for 50 - 75% of the total effort spent during the software lifecycle (van Vliet, 2008). Maintenance activities, such as requests for adding new functionality, or the correction of errors are hard to neglect and shall be performed between almost all pairs of successive software versions. On the contrary, changes that are not directly related to the external behavior of the system, but relate to design-time qualities, are often postponed or neglected, to shrink product time to market and reduce short-term costs. However, software systems are by definition highly evolving products, whose design-time quality will gradually decay (Parnas et al., 1994), and therefore deferring such maintenance activities (e.g., refactorings, resolution of bad smells, reverse engineering) might have a significant impact on several design-time qualities (e.g., maintainability, comprehensibility, reusability, etc.). This strategy leads to the creation of a financial overhead due to degraded quality, originally termed, by Cunningham (1992), as technical debt.

Technical debt (TD) is a metaphor that is used to draw an analogy between financial debt as defined in economics and the situation in which an organization decides to produce immature software artifacts (e.g., designs or source code), to deliver the product to market within a shorter time period (Cunningham, 1992). The most modern definition of technical debt is the 16,162 definition, that was one of the main conclusions of the TDM Dagstuhl Seminar in 2016, which is stated as follows: "*In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability*". In addition to trade-offs between design-time qualities and business goals (such as time-to-market, etc.), recent literature identifies trade-offs between run-time and design-time quality attributes (Feitosa et al., 2015), especially in software systems in which run-time properties cannot be compromised, such as embedded or real-time systems. These trade-offs, can also been considered as potential roots of neglecting design-time qualities, leading to the accumulation of TD ((Ampatzoglou et al., 2016) and (Martini et al., 2014)).

Table 1
Related work overview.

Study	Association to TD	TD Concept	Research Method	Outcome
Martinez-Fernandez et al., 2013	Direct	Principal	Qualitative	TD hinders reuse
Yli-Huumo et al., 2014	Direct	Principal	Qualitative	No strategy for TDM in SPLs
Lim, 1994	Indirect	Interest Probability	Quantitative	Reuse positively affects: (a) defect density; (b) productivity; and (c) time to market
Frakes and Kang, 2005	Indirect	Interest Probability	Quantitative	Reuse positively affects defect density
Mahagheghi and Conradi, 2007	Indirect	Interest Probability	Quantitative	Reuse positively affects defect density and change proneness
Mahagheghi and Conradi, 2008				
Deniz and Bilingen, 2014	Indirect	Interest	Quantitative	Reuse improves coupling and complexity. Not correlated to size and inheritance
Constantinou et al., 2015	Indirect	Interest	Quantitative	Reuse increases complexity and coupling. Lowers cohesion
Zaimi et al., 2015	Indirect	Interest	Quantitative	No relation found to reusability
Nikolaidis et al., 2019	Direct	Principal	Quantitative	Reuse decreases TD principal

TD is accumulated during all development phases, i.e. requirements analysis, architectural/detailed design, and implementation, and therefore should be monitored and handled during the complete software lifecycle (Kruchten et al., 2012). Nevertheless, code TD is reported as the most frequently studied type in research (Alves et al., 2016) and the most important one in the industry (Ampatzoglou et al., 2016). Although, TD is sometimes desirable (e.g., in cases when companies opt for investing on a different products, rather than improve the quality of an existing one) and its complete repayment is considered unrealistic (Eisenberg, 2013), its side-effects cannot be ignored, in the sense that TD severely hinders the maintainability of the software (Zazorwka, 2011). To this end, TD should be continuously monitored and managed. As a first step of any management process, it is important to identify the most crucial concepts that need to be monitored, and define a measurement plan for them—see Section 3.2.

3.2. Technical debt concepts and their measurement/assessment

The cornerstones of the TD metaphor are two concepts borrowed from economics: *principal* and *interest*. ***TD Principal*** is the effort required to eliminate inefficiencies in the current design or implementation of a software system (Ampatzoglou et al., 2015); typical examples of such inefficiencies are code and design smells. On the contrary, ***TD Interest*** is the additional development effort required to modify the software, due to the presence of such inefficiencies (Ampatzoglou et al., 2015): corresponding to the extra effort required to add new features or fix bugs because of the presence of TD (Buschman, 2011). The estimation of principal and interest depends on the type of TD (e.g., code, design, testing TD). In the next paragraphs we elaborate on estimating code TD principal and interest, which is the focus of this paper.

In Fig. 2, we visualize an overview of the two concepts, so as to allow the easy interpretation of TD terminology, based on the study of Chatzigeorgiou et al. (2015). In Fig. 2, we can observe the positioning of a random system in the y-axis ("*actual*"), which represents the level of design-time quality of the system. The actual quality is at some distance from the "*optimal*" quality: The effort required for the development team to close this quality gap, represents the *TD principal*. The negative consequence of principal, is *TD interest*, which represents the additional effort required to maintain the software in the *actual* state, compared to the effort that would be required if the system was of optimal quality.

According to two recent secondary studies on TD management by Ampatzoglou et al. (2015) and Li et al. (2015), SonarQube is the most frequently used tool for estimating ***TD principal***. SonarQube is representing TD principal through two different views: (a) the number of inefficiencies in the source code, and (b) the amount of

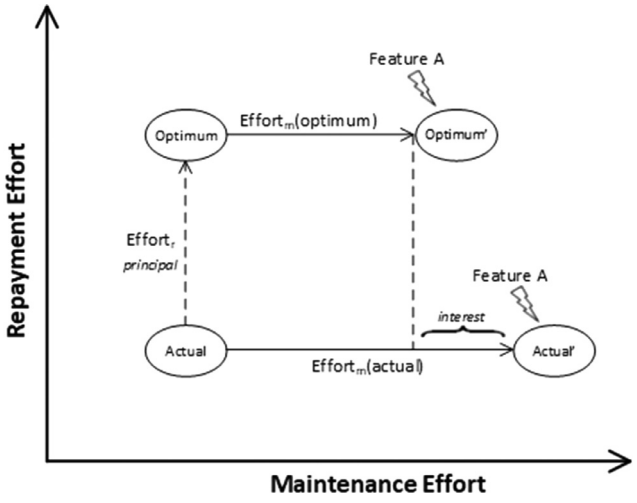

Fig. 2. TD Terminology visualization (Chatzigeorgiou et al., 2015).

time required to fix such inefficiencies. The platform algorithm was originally based upon an adopted version of the SQALE method proposed by Letouzey (2012), in which a remediation index is obtained for requirements of an applicable Quality Model. Since in this study we are adopting the Dagstuhl 16,162 definition of TD, we are not using the calculations of SonarQube, "*as-is*", but we consider only the effort to resolve *maintainability issues* (code smells, duplicated lines density, and coverage), since it is the only property discriminable at design-time. For *code smells* (by default) there are 334 rules—e.g., "*Method overrides should not change contracts*", "*Package declaration should match source file directory*", etc. SonarQube rules that are related to code smells are associated with code understandability, poorly written code, runtime security, and coding standard. Regarding *duplicated code*, SonarQube measures the portion of the code that contains duplicated logic—not necessarily only copy-pasted code, but also conceptual clones occurring at multiple places. Finally, SonarQube itself cannot assess which tests are actually executed and the code *coverage*; thus, it relies on third-party test *coverage* tools—e.g., JaCoCo for Java. All the aforementioned efforts are summed up as the *total TD principal*: calculated as the effort required to fix all the aforementioned maintainability issues. The measure is stored in minutes in the database. An 8-hour day is assumed, when values are shown in days. The value of the cost to develop a line of code is 0.06 days.

Software maintainability is inherently related to technical debt, and in particular to ***TD interest*** (Kruchten et al., 2012) (i.e., how easy it is for a software engineer to apply changes in a specific

Table 2
Maintainability properties and metrics.

Property	Metric	Description
Inheritance	DIT	*Depth of Inheritance Tree*: Inheritance level number, 0 for the root class.
	NOCC	*Number of Children Classes*: Number of direct sub-classes that the class has.
Coupling	MPC	*Message Passing Coupling*: Number of send statements defined in the class.
	RFC	*Response For a Class*: Number of local methods plus the number of methods called by class methods.
	DAC	*Data Abstraction Coupling*: Number of abstract types defined in the class.
Cohesion	LCOM	*Lack of Cohesion of Methods*: Number of disjoint sets of methods (a set of methods that do not interact with each other), in the class.
Complexity	CC	*Cyclomatic Complexity*: Average cyclomatic complexity of all methods in the class.
	WMPC / NOM	*Weighted Method per Class*: Weighted sum of methods. Each method of the class is assigned to a weight equal to 1.
Size	SIZE1	*Lines of Code*: Number of semicolons in the class.
	SIZE2	*Number of Properties*: Number of attributes and methods in the class

software system). Therefore, in this study we consider maintainability as a proxy for TD interest. The relation of interest and maintainability, as a consequence of the existence of TD principal, has been highlighted in the literature: "*the existence of compromises incur a "debt" in the software that should be repaid to restore the health of the system in the future and to avoid "interest" in the form of decreasing maintainability*" (Seaman and Guo, 2011). The set of metrics that we have selected to use in our study for quantifying maintainability (see Table 2) belong to well-known metric suites (Chidamber and Kemerer, 1994; Li and Henry, 1993).

The metrics selection was based on a secondary study by Riaz et al. (2009), which reported on a systematic literature review (SLR) aimed at summarizing software metrics that can be used as maintainability predictors. In particular, Riaz et al. (2009) have performed a quality assessment of maintainability models, through a quantitative checklist, in order to identify studies of high-quality score, i.e., studies that provide reliable evidence. More specifically, the checklist was comprised of 19 questions and each model was assessed for each criterion by a three-point scale: yes, no, or partially, with associated scores of 1, 0, and 0.5 respectively. The range of the total score of each study was between 0 and 19. All studies that have scored 7 or below were excluded from the list of selected studies, whereas among the studies with the highest scores were those of van Koten and Gray (2006), Zhou and Leung (2007) and Misra (2005). These studies have used the same definition of maintainability while the common metrics used in all three studies are the ones belonging to the metric suites proposed by Li and Henry (1993) and Chidamber et al. (1994), i.e., two well-known object-oriented set of metrics. The employed suites contain metrics that can be calculated at the source-code level, and can be used to assess well-known quality properties, such as inheritance, coupling, cohesion, complexity and size.

The employed suites contain metrics that can be calculated at the source-code level, and can be used to assess well-known quality properties, such as inheritance, coupling, cohesion, complexity and size.

- Regarding ***inheritance***, although we acknowledge its need as one of the main advantages of object-orientation, excessive levels of inheritance renders the design more complex, and therefore harder to maintain. More specifically, the DIT metric can be characterized as maintainability predictor, in the sense that a class placed very low in the inheritance tree has access to more properties or methods of super-classes and thus is hard to maintain. In such a case, it is more difficult to locate which class implements a method that needs to be changed or a property that need to be parsed. Similarly, for NOCC metric, the more direct sub-classes a class has, may affect its maintainability, in the sense that for understandability reasons it may be preferable to organize entities inside sub-hierarchies instead of giving excessive breadth to the design.
- Three ***coupling*** metrics are related to maintainability. In particular, RFC metric calculates the cardinality of the response of a class. Thus, a class that has many local methods and all these methods call others, RFC metric will score high, signifying a larger and more complex class in which it will be difficult to identify errors, due to excessive message delegation. Similarly, with RFC, the MPC metric depicts the dependence of a class to methods in other classes. Classes with high levels of MPC are more prone to ripple effects, i.e., changes propagated due to changes in other classes. Finally, a class that has multiple variables of abstract data types (DAC) is difficult to maintain, since method calls to abstract objects can potentially lead to concrete implementations located in sub-classes. Thus, identifying the proper implementation becomes more time consuming.
- Regarding ***cohesion***, LCOM characterizes the amount of responsibilities offered by a class. A class with many responsibilities is expected to change more frequently, and to include longer methods that are hard to maintain.
- For the ***complexity*** property we use two metrics: CC and WMPC. In particular, WMPC is the number of methods in a class. For a class that has a lot of methods, its' interface will be more frequently maintained. In addition, by focusing on the body of methods, CC measures the average cyclomatic complexity. A method with high CC, is harder to understand since it has more control flows (e.g. loops, if, etc.).
- Finally, the ***size*** of a class is very important, in the sense that a class that has a large number of lines of code, properties and methods will be more difficult to understand and maintain. For assessing this property, we use two metrics: SIZE1 and SIZE2.

4. Study design

The objective of this study is to investigate the relation between software reuse and technical debt. To achieve this goal, we compare the levels of the two pillars of the TD concept (i.e., principal, and interest) of reused and native classes, through a multi-case study. The study has been designed and reported according to the guidelines suggested by Runeson et al. (2012).

4.1. Objectives and research questions

The goal of the study is to "*compare white-box reused and native classes with respect to their TD principal and interest*". Based on this goal (and the two aspects of technical debt) we have derived two research questions that will guide the case study design and the reporting of the results:

***RQ*$_1$**: Is reused code having lower principal compared to native code?

This research question aims at investigating if the overall quality (as captured by TD) of the reused code is higher compared to the overall quality of the native code, in which the reused

code is to be introduced. This question is relevant for cases that development teams: (a) have to decide on whether to reuse code or develop it from scratch; and/or (b) want to refactor reused code so as to pass certain quality standards in the company. To answer this research question, we compare the average TD principal of native and reused code: TD principal sums-up the effort to refactor all code smells, as provided by SonarQube.

RQ$_2$: Is reused code having lower interest compared to native code?

This research question aims to investigate if the effort required to maintain reused code is higher or lower, compared to native code. The answer to this question is interesting to practitioners that aim at applying white-box reuse that will involve code maintenance in the target system. To answer this research question, we compare the average TD interest of native and reused code. TD interest is assessed through a set of proxies, i.e.*, well-known maintainability predictors: see Section 3 for more details.*

4.2. Case selection and units of analysis

According to Yin (2003), for every case study, researchers must determine the context, the cases, and the units of analysis. In this study, the context is open-source software and the cases / units of analysis are open source classes. We note that this case study is holistic: for each case one unit of analysis is extracted. To gather as many cases as possible, we queried the Reaper database[1] and selected the GitHub projects written in Java, using Apache Maven as an automation tool. We selected Java as a programming language so as to take advantage of the capabilities of exiting tools for quantifying the aspects of TD. We have selected Maven as a build tools (e.g., against Gradle), since it offers a large number of projects that could lead to a large-scale dataset, and since it is more generic-scoped compared to Gradle. In particular, most Gradle projects are Android applications; thus, they require manual customization and pre-build configurations. These tasks prevent the automated build and data-extraction of these projects for the needs of this large-scale analysis. Finally, to filter and select a subset of project in the Reaper database, we sorted them based on their popularity, i.e., their stars in GitHub API.

4.3. Data collection

The dataset that has been used in this study consists of 897,044 rows, one row for each class of the considered systems. For every class, we recorded 18 variables:

1 ***Software***: The name of the OSS project from which we extracted the data.
2 ***Class***: The name of the class under study.
3 ***Reuse***: Reused or Native
4 ***TD Principal***: The amount of TD principal in a specific class, based on SonarQube.
5 ***TD Interest***: The values of the 10 object-oriented metrics (V.5.1 – V.5.10) that can be used as proxies of TD interest, as calculated from the Percerons Client—see Table I.

For enabling the automated extraction of these variables, the following process has been used:

- ***Step 1: Download repositories***. After selecting the projects (see Section 4.2), using Git, we cloned locally the top 1000 ones. We selected this number of projects to improve the representativeness of the sample towards the population and strengthen the statistical analysis.
- ***Step 2: Build projects and retrieve dependencies.*** With the repositories at hand, we have then built each project. During the building process, the generated compiled package (i.e., a .jar or .war file) are placed in the local Maven repository (the .m2 directory by default). The dependencies (third party packages or libraries) of each project are also downloaded and placed in the local repository (in cases that the source code was not available as glass-box reuse, we downloaded it manually). From the total 1000, we discarded 598 projects that failed to build. For the remaining 402 successfully built projects, we stored their dependency tree, i.e., the paths to the packages of the project and its dependencies.
- ***Step 3: Collect project information.*** In this step, we analyzed each project's dependencies' tree and collected the first groups of variables (V1-V3). In particular, regarding V3, we used a two-step process. First, we marked as reused all systems classes that exist in the compiled packages that are downloaded from the Maven repository (black-box reuse – however black-box reused classes have not been studied in our analysis). Then, for each one of these classes from the Maven repository, we searched them in the source code of the 402 built projects, and when we identified them in a project (other than the source/original one), we marked them as reused (white-box reuse). The identification of the original project relied on the naming of the projects. Classes that are reused in more than one projects have been removed as duplicates (i.e. we retained only a single class). All other classes of the built projects (i.e. other than reused ones) are tagged as native, in the sense that we have no indication of reuse within our set of analyzed projects.
- ***Step 4: Measure TD Principal***. For quantifying TD principal (V4), we have used SonarQube (see Section 3). According to its documentation, SonarQube aims at the continuous evaluation of software quality. SonarQube can assess the quality of software on a multitude of programming languages, generating documentation on quality measures and issues, such as coding rule violations. The analysis has been performed according to the platform's default configuration. The TD Principal for each artifact corresponds to the total effort needed in order to resolve all existing maintainability issues in an artifact.
- ***Step 5: Measure TD Interest.*** For calculating the metrics of Table 1 that can be considered as interest proxies (see Section 3), we have used Percerons Client (Ampatzoglou et al., 2013). Percerons is a software engineering platform (Ampatzoglou et al., 2013) 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 in Java code through software metrics, and (c) design pattern instances. This step led to the recording of variables V.5.1 – V.5.10.

In the end of this process 897 thousand classes, retrieved from 402 projects, have been analyzed. The average size of the projects is approximately 2231 classes. The number of native classes in the dataset is 167 K (~19%) classes, whereas the rest are reused ones (~7% white-box reused and 74% black-box reused). Some additional demographics are presented in Fig. 3 and Table 3. From the figure we can observe that both the absolute, as well as, the normalized

Table 3
Reaper repo descriptives.

Variable	Min	Max	Mean	Std. Dev.
History	0	209	12,54	21,824
#Issues	0	67	2,38	6078
#Unit Tests	0	1	0,21	0,187
Stars	3	3440	176,91	325,200

[1] https://github.com/RepoReapers/reaper

Table 4
Hypothesis testing overview.

RQ	Dependent Variables	Grouping Variable	Null Hypothesis
RQ_1	[V4] Total TD Principal	[V3] Native or Reused	H_0: The population means for TD principal from the white-box and reused classes groups are equal
RQ_2	[V5.1] – [V5.10]		H_0: The population means for 10 proxy metrics for TD interest from the white-box and reused classes groups are equal

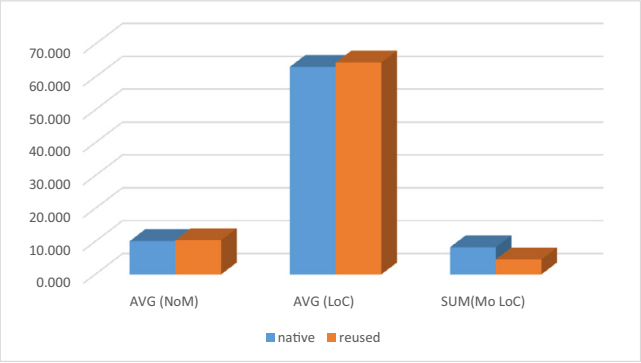

Fig. 3. Descriptives of the dataset.

values (divided by the number of classes) are quite close, constituting the two groups (the analysis is performed per class) comparable.

4.4. Data analysis

To answer the research questions set in Section 4.1, given the available dataset (see Section 4.3), the following data analysis process has been performed. Given the fact that all the analysis is built around subjects that can be split into two groups, we have selected tests and means of visualization for comparing the levels of a certain numerical variables between groups. To this end, for hypothesis testing, we have used the independent sample *t*-test. According to Field (2017) the proper execution of independent sample t-tests requires checking the following four assumptions:

- ***normal distribution***: We have checked that the differences between scores are normally distributed, using the Kolmogorov-Smirnov test (Field, 2017).
- ***data are measured at least at the interval level***: This assumption holds, since all the recorded variables are at a continuous scale.
- ***homogeneity of variance***: We have checked that the variances of the two groups are equal in the population, using the Levene's test (Field, 2017).
- ***independence of variables' scores***: This assumption holds, since all datapoints come from different classes.

Due to space limitations, here we report only the results on the TD Principal variable, but the same process has been performed for all ten variables that are proxies of TD Interest. In particular, in Fig. 4, we present the Q-Q plot, suggesting that the values of the variable are normally distributed for both groups. The Kolmogorov-Smirnov test for native classes is 0.087 (sig: 0.11), whereas for white-box reused classes is 0.072 (sig: 0.15). Additionally, the Levene's test of equality of variances suggested that the variances are equal (F: 0.266 and sig: 0.55).

The analysis on principal has been performed: (a) for the total TD principal; whereas (b) for interest, on all metrics that can be used as interest proxies—see Section 3. To ensure that the confounding factor of reused code size is factored out of the analysis, we performed hypothesis testing to compare the average size of reused and native classes, in terms of lines of code (LOC) and number of methods (NOM). The outcome of this comparison will be important during the interpretation of the results, since size is acknowledged as an important factor while performing quality comparisons. An overview of data analysis is presented in Table 4.

Table 5
Hypothesis testing for TD principal.

Code	Mean TD Principal (in minutes)	Std. Dev.	t-value	sig.
Native	0.472	40.79	-	<0.01
Reused	1.388	32.31	6.788	

5. Results

In this section we present the results of this study organized by research question. In Section 5.1, we answer RQ_1 (relation between reuse and TD principal), whereas, in Section 5.2 we answer RQ_2 (reuse and TD interest).

As a pre-processing step for our analysis, we explored the possible differences in the size of reused and not reused classes. The comparison has been made, by using two size metrics: (a) lines of code—LOC, and (b) number of methods—NOM. The results suggest that the two groups (native and white-box reused classes) have similar size in mean values (64.42 ± 191.04 vs. 65.22 ± 188.63 lines of code per class, and 10.38 ± 21.24 and 12.18 ± 22.48 methods per class respectively). However, the differences in their mean values are statistically significant (hypothesis testing with $p < 0.01$). Therefore, since any differences identified in the upcoming sections could be attributed to the different size of the reused vs. native code, mitigation actions shall be taken. To this end: to factor out this confounding factor all variables have been normalized against the lines of code of each class. Studying TD Principal Density instead of TD values per se has been adopted by other studies as well (e.g., by Digkas et al., 2018).

Reuse and TD Principal. In Table 5 we present the results that have been obtained by studying the TD principal accumulated in reused classes compared to native ones. Based on the results, we can conclude that TD Principal is higher in white-box reused code compared to native code. The difference apart from being statistically significant, is also important in an absolute value, in the sense that reused code has 290% more TD Principal Density, compared to native code. Despite the fact that standard deviation is quite high compared to the mean values, the standard deviation is comparable between the two groups (standard deviation ratio: 0.792).

Reuse and TD Interest. Following a similar analysis to RQ_1, in Table 6, we present the results of the independent sample t-tests for the variables that are proxies of TD interest. We note that from this analysis, we have omitted size metrics, since they have been factored out as explained in the beginning of Section 5. The results suggest that based on all metrics (except from Cyclomatic Com-

Table 6
Hypotehsis testing for TD interest.

TD Interest	Code	Mean	Std. Dev.	t-value	sig.
Depth of Inheritance Tree	Native	2.164	1.48	19.587	<0.01
	Reused	1.977	1.34		
Number of Children	Native	0.661	3.65	5.877	0.02
	Reused	0.602	4.29		
Cyclomatic Complexity	Native	1.623	2.45	-6.444	<0.01
	Reused	1.702	1.96		
Lack of Cohesion	Native	195.184	2481.79	0.889	0.78
	Reused	175.338	4217.11		
Response for a Class	Native	37.970	61.98	4.995	<0.01
	Reused	35.111	58.69		
Message Passing Coupling	Native	41.095	122.95	0.102	0.85
	Reused	38.698	111.78		
Data Abstraction Coupling	Native	0.335	1.21	11.172	<0.01
	Reused	0.295	1.73		

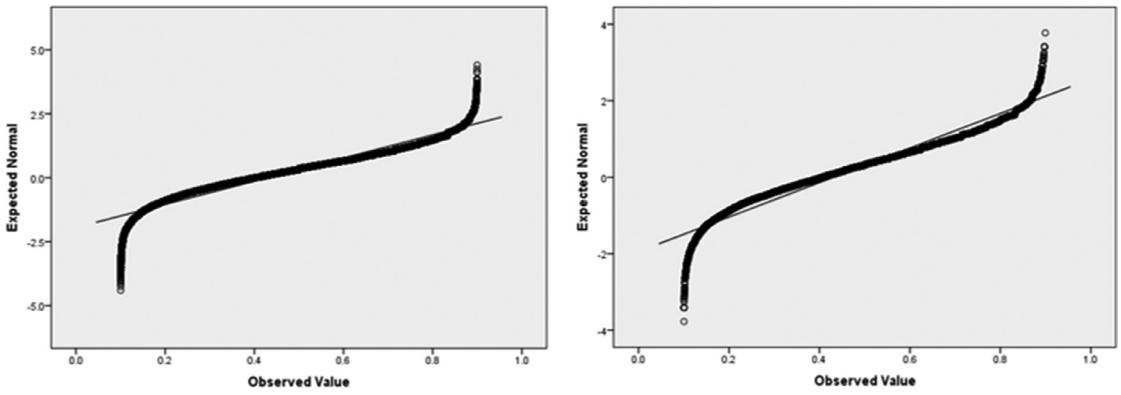

Fig. 4. Q-Q plots for checking normal distribution for TD principal.

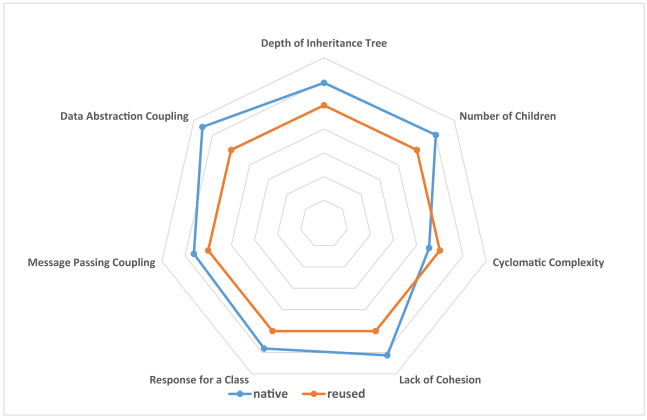

Fig. 5. Continuents of TD interest.

plexity) the reused code is more maintainable compared to the native one. Nevertheless, the differences are statistically significant only for the two inheritance metrics (DIT and NOCC), the complexity metric (CC), and two coupling metrics (RFC and DAC).

By focusing on the actual values of the metric scores (see Fig. 5), we can observe that the differences are rather small, ranging from 4.64% for CC to 15.22% for DAC, whereas for the majority of cases the difference is around 10%. This observation is in contrast to TD principal, in which: (a) the difference was more substantial in terms of absolute numbers, and (b) the native code excelled compared to the reused one.

The aforementioned findings are considered as expected in the sense that code that is organized into libraries is by definition paying special attention to modularity, so as to be reusable. Software modularity is composed by two structural properties: ***coupling*** and ***cohesion*** (van Vliet, 2008). Therefore, the fact that reused code excels in terms of coupling and cohesion can be consider expected. Additionally, reused code usually is a more conceptually difficult to implement code chunk, that offers advanced functionality, which inevitable contains necessary ***complexity***. Thus, the fact that native code is on average less complex can be attributed to the fact that it is a collection of trivial and advanced functionalities, in contrast to library code, which usually encapsulates more complex functionalities. Additionally, in terms of ***abstraction*** and ***inheritance***, the reused code is also expected to be superior, since it is meant to be reused and therefore offers extension points through well-known mechanisms such as patterns, open-close principle, etc., that rely on polymorphism.

6. Discussion

In this section we discuss the main findings of this paper, organized into two sub-sections. First, we present interpretations of the main findings of the case study, by providing comparisons to related work, when it is possible. Then, we provide implications to researchers and practitioners in the form of actionable outcomes and future work opportunities.

Interpretation of Results and Practical Considerations. This study compared the reused and native source code in terms of technical debt. The findings of the study are not uniform in the sense that the two aspects that have been investigated do not seem to be affected in the same way by software reuse as a phenomenon. On the one hand, the TD principal (i.e., the effort required to fix all source code inefficiencies) of reused code appears to be 3 times higher compared to native one. Interpreting this observation suggest that, supposing that software development industries want to

retain a certain standard of quality assurance, in terms of source code issues (i.e., code conventions, clumsy code, etc.), it is preferable to write their own code, in the sense that reused code is in more need of refactoring.

On the other hand, based on our findings the ***reused classes appear to be more maintainable*** than native classes (even marginally, ***less than 10%***)—i.e., ***having lower TD interest***. This observation has merit since it shows that in cases that the reused classes need to be maintained, their structure enables the easy extension of the code base. This finding is extremely interesting since it: (a) contradicts existing literature on the relation between TD principal and interest, which until now have been reported as positively correlated (e.g., (Kosti et al., 2017)); and (b) does not comply with the traditional relation between principal and interest in economics—a claim that it is also supported by others in the TD community (e.g., Schmid, 2013). This finding, suggests that reused code has some special characteristics that deserve further investigation. In particular, the findings of this study suggest that although the reused code is in-need of various refactorings (in terms of styling, coding conventions, etc.) the produced code obeys to good object-oriented practices; lowering complexity and coupling, and improving cohesion. Additionally, this finding suggests that although measuring TD principal (through SonarQube) and TD interest (through maintainability metrics) are having some overlap (e.g., SonarQube offers some rules, by setting thresholds on the value of Cyclomatic Complexity) the two amounts are not by-definition correlated, and therefore are valid and independent views of the two concepts.

Implications to Researchers and Practitioners. Based on the aforementioned observations various implications to researchers and practitioners can be highlighted. On the one hand, practitioners are encouraged to perform open-source code reuse, at least in terms of guaranteeing that technical debt can be sufficiently managed. Although the amount of TD principal that is brought to the system is higher compared to native code, reused code appears to be easier to maintain. In particular, the extra effort that shall be spent in refactoring existing inefficiencies is equalized at the first place by the effort saved during development, and in the long term by the interest savings along maintenance. However, each development team should monitor the TD principal and interest incurred by reuse and check whether it aligns with the team's overall quality assurance strategy. Additionally, in the special case of selecting between commercial components off-the-shelf (COTS) and OSS components, the results of the study can be used as part of the valuation of reuse alternatives, e.g., through real-option approaches (Mavridis, 2014). Such strategies consider the trade-offs between paying for get**ting access to propertiary components, against the need for paying for technology transfer.**

On the other hand, regarding TD ***research community***, we provide evidence that reuse is a promising technology for preventing the accumulation of TD, and for ensuring the future TD sustainability of the system. An interesting research implication that leads to a very interesting future work opportunity is studying why reuse does not have the same effect on TD principal and interest. This seems to be a special case for the TD literature in the sense that current empirical evidence suggest that TD principal and interest are correlated (Kosti, 2017) and since it contradicts the underlying financial concept that principal and interest are related through interest rate, as discussed by Schmid (2013). Deviating from these two observations constitute reuse at the class level as a candidate for more in-depth analysis, explanatory studies that goes beyond out exploratory ones. An interesting future work opportunity would be the replication of the study, by using additional building tools (e.g., Gradle), in order to investigate if the build tool related to the quality of the code that is brought inside the project.

7. Threats to validity

In this section, we present and discuss potential threats to the validity of our case study: construct validity, reliability, and external validity (Runeson et al., 2012).

7.1. Construct validity

Construct validity is related to the way in which the selected phenomena are observed and measured. In this study we quantified two TD concepts, namely TD principal and TD interest:

TD principal is quantified through SonarQube, which is the state-of-practice tool for measuring TD principal (Alves, 2016) in the sense that is the most widely used in research and practice. Although SonarQube is an established tool, it focuses on code TD, neglecting other types of TD, like architecture debt, requirements debt, etc. Despite the identified limitations, especially the lack of Architectural Technical Debt (ATD) identification and measurement, SonarQube is considered as extremely useful for code TD identification, monitoring, measurement and prioritization. According to Tsintzira et al. (2019) the TD principal as measured in this study is correlated at the level of 0.83 to the perception of practitioners in terms of the amount of effort required to refactor an existing industrial system.

In the literature there is no established way to measure ***TD interest***. This is due to the fact that an accurate measurement of interest would require the simultaneous maintenance of two software solutions: an optimal and an actual one and the anticipation of future maintenance activities. Besides the inability to fore-cast future changes, such an approach is unrealistic for two reasons: (a) there is no way to define a universally accepted optimal system, and (b) it is cost inefficient to maintain two real systems just aiming to accurately measure technical debt interest. Therefore, as the current state-of-the-art stands TD interest can only be assessed through proxies. In this study, as a proxy of interest we selected metrics that assess maintainability. Although in literature, maintainability has been linked to various metrics, in this study we selected ten object-oriented metrics (grouped in 5 categories/aspects of TD interest) measured at source code. Metrics' selection was based on empirical evidence in the literature suggesting that a combination of these metrics is the optimal maintainability predictors (Riaz et al., 2009). According to Tsintzira et al. (2019) the TD interest as measured in this study is correlated at the level of 0.73 to the perception of practitioners in terms of the amount of additional effort required to maintain an existing industrial system, due to the presence of inefficiencies.

Finally, a tentative threat to construct validity might arise by mixing up design and code TD, while calculating interest (we note that all calculations have been made at the source code level). Despite the fact that the interest proxy metrics are intended to be design ones, the majority of them cannot be calculated from design artifacts (e.g., a class diagram). For instance, LCOM requires for each calculation to be aware of the attributes that are being accessed in the body of a function. This information is only available at the implementation phase and from the source code artifact; despite the fact that the level of calculation is the class. The same holds for other metrics, e.g., the coupling ones, since the declaration of an extra variable in a method body would increase coupling, but it is highly unlikely that it would lead to the inclusion of an association in a class diagram. Therefore, the used metrics are in the border between code and design TD; and we consider their use as a proper decision.

Respect to reliability, we consider any possible researchers' bias, during the data collection and data analysis process. The design of the study, concerning data collection, does not contain threats, since all data are automatically extracted by tools, without any

subjective configuration. Moreover, with respect to the data analysis process, to mitigate any potential threats to reliability, three researchers were involved in the process, aiming at double checking the work performed and thus reducing the chances of reliability threats. Furthermore, the detailed case study protocol presented in Section 4 enables the repetition of the study, as well as the provision of a replication package.

7.2. Internal validity

Concerning internal validity, we note possible confounding factors that might have biased the results of this study. The main threat to internal validity is related to the characterization of classes with respect to reuse. First, regarding the characterization of a class as reused or native, we have used a systematic process for classifying classes. Through this process, we are certain that the classes that have been classified as reused ones are true-positive occurrences (high recall); however, we acknowledge that we might have characterized as native, some classes that have been reused in the white-box form (lowered precision—false positives). Due to the enormous size of the dataset, it was not realistic to perform a comprehensive check; however, to alleviate this problem, we have performed a manual check on a subset of our dataset (approx. 500 classes) and we have identified, no such cases. Second, regarding the characterization of classes as white-box, we note that we cannot differentiate between white- and glass-box reused classes: i.e., cases in which the reused code, is copied inside the code bases of the target application (as source), but it was never maintained. Getting definite results on this would require the analysis of the whole project evolution. We opted not to perform this task, since we believe that glass-box and white-box reuse do not differ substantially, and although some classes have not been maintained still, they contribute to the TD of the system, since they are candidates for accommodating future changes.

7.3. External validity

Concerning external validity, a potential threat to generalization is the possibility that performing the study on different projects of different languages might affect the obtained observations. However, we believe that the selected projects, given their size and complexity, represent a realistic real-world system. Additionally, the results of the study are not applicable to non-object-oriented systems, in the sense that TD interest in such systems could not be assessed through properties such as inheritance, coupling and cohesion, which are applicable only in OO software modules. Finally, the identified outliers (less than 1% of the sample) might influence the generalizability of results in the sense that in the population more extreme values might exist. However, we believe that this threat is substantially mitigated by the size of our sample and the small proportion of outliers.

8. Conclusions

Reuse is an established practice in software engineering that is yielding several benefits for the quality of the target system, and the development process, in terms of productivity. In this paper, we study the relation between software reuse at the class level and technical debt, which is a modern view of structural software quality, which valuates future maintenance actions. In particular, we have explored the reuse activities performed in ~400 projects (~890 K classes) and compared the TD principal and interest of reused and natively-developed classes. The results of the study suggested that reused classes tend to concentrate more principal, but are easier to maintain (lower interest). Unveiling the underlying relations between source-code reuse and technical debt, are useful to both practitioners and researchers, since they can get more informed decisions while reusing, and trigger some promising research opportunities.

Declaration of Competing Interest

The authors declare that they have no known competing financial interestsor personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Daniel Feitosa: Conceptualization, Methodology, Software, Formal analysis, Data curation, Writing - original draft, Writing - review & editing. **Apostolos Ampatzoglou:** Conceptualization, Methodology, Data curation, Writing - original draft, Writing - review & editing. **Antonios Gkortzis:** Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing. **Stamatia Bibi:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing. **Alexander Chatzigeorgiou:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing.

References

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

de Almeida, E.S., Alvaro, A., Lucredio, D., Garcia, V.C., de Lemos Meira, S.R., 2005. A survey on software reuse processes. In: 7th *International Conference on Information Reuse and Integration*, Las Vegas, USA. IEEE, pp. 66–71 15-17 August.

Alves, N.S.R., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Carolyn Seaman, 2016. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121 Elsevier.

Amanatidis, T., Mittas, N., Chatzigeorgiou, A., Ampatzoglou, A., Angelis, L., 2018. The Developer's Dilemma: Factors Affecting the Decision to Repay Code Debt. In: 1st International Conference on Technical Debt (TechDebt' 18), Gothenburg. IEEE/ACM, pp. 62–66 27-28 May.

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2015. The financial aspect of managing technical debt: A systematic literature review. Inf. Softw. Technol. 64, 52–73 ElsevierAugust.

Ampatzoglou, A., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., 2016. A Financial Approach for Managing Interest in Technical Debt. A Financial Approach for Managing Interest in Technical Debt. Springer.

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

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., Systa, K., 2016. The perception of technical debt in the embedded systems domain: an industrial case study. In: 8th International Workshop on Managing Technical Debt (MTD' 16), Raleigh, USA. IEEE, pp. 9–16 4 October.

Arvanitou, E.M., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Stamelos, I., 2019. Monitoring technical debt in an industrial setting. 23rd International Conference on the Evaluation and Assessment in Software Engineering (EASE' 19). ACM 14-17 April.

Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, G., 2005. An industrial case study on reuse oriented devel-opment. In: 21st International Conference on Software Maintenance (ICSM'05), Budapest, Hungary. IEEE, pp. 283–292 25-30 September.

Barros-Justo, J.L., Pinciroli, F., Matalong, S., Martínez-Araujo, N., 2018. What software reuse benefits have been transferred to the industry? A systematic mapping study. Inf. Softw. Technol. 103, 1–21 Elsevier.

Buschmann, F., 2011. To pay or not to pay technical debt. Software 28 (6), 29–31 IEEEJune.

Chatzigeorgiou, A., Ampatzoglou, A., Ampatzoglou, A., Amanatidis, T., 2015. Estimating the breaking point for technical debt. In: 7th International Workshop on Managing Technical Debt (MTD). IEEE, pp. 53–56 Bremen2 October.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. Trans. Softw. Eng. 20 (6), 476–493 IEEEJune.

Constantinou, E., Ampatzoglou, A., Stamelos, I., 2014. Quantifying reuse in OSS: A large-scale empirical study. Int. J. Open Source Softw. Process. 5 (3), 1–19 IGI–GlobalJuly.

Cunningham, W., 1992. The WyCash Portfolio Management System. In: 7th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '92), Vancouver, Canada, pp. 29–30 5-10 October.

Deniz, B., Bilgen, S., 2014. An Empirical Study of Software reuse and quality in an industrial setting. In: International Conference on Computer Science and its Applications, pp. 508–523 Springer30 June.

Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A., Ampatzoglou, A., 2018. How do developers fix issues and pay back technical debt in the apache ecosystem? In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER' 18). IEEE Computer Society, pp. 153–163 March.
Eisenberg, R., 2013. Management of technical debt: a lockheed martin experience report. 5th International Workshop on Managing Technical Debt (MTD' 13) 9 October.
Feitosa, D., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y., 2015. Investigating quality trade-offs in open source Critical Embedded Systems. In: 11th International Conference on Quality of Software Architectures (QoSA' 15), Montreal, Canada. ACM, pp. 113–122 4-7 May.
Field, A., 2017. Discovering Statistics Using IBM SPSS, fifth ed. SAGE Publications.
Frakes, W.B., Kang, K., 2005. Software Reuse Research: Status and Future. Trans. Softw. Eng. 31 (7), 529–536 IEEEJuly.
Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., Irlbeck, M., 2011. On the extent and nature of software reuse in open source java projects. Lect. Notes Comput. Sci. 207–222 Springer.
Joos, R., 1994. Software reuse at motorola. Software 42–47 September.
Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A., 2015. A Case study in locating the architectural roots of technical debt. In: 37th IEEE International Conference on Software Engineering (ICSE' 2015), IEEE/ACM, pp. 179–188 Florence, Italy, 16-24 May.
Kosti, M.V., Ampatzogirlou, A., Chatzigeorgiou, A., Pallas, G., Stamelos, I., Angelis, L., 2017. TD principal assessment through structural quality metrics. In: 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 17), Vienna, Austria. IEEE, pp. 329–333 30 August – 1 September.
Kruchten, P., Nord, R., Ozkaya, I., 2012. Technical debt: from metaphor to theory and practice. Software 29 (6), 18–21 IEEENovember.
Krueger, C.W., 1992. Software reuse. Computing Surveys 24 (2), 131–183 ACMJune.
Letouzey, J.L., 2012. The sqale method for evaluating technical debt. In: 3rd International Workshop on Managing Technical Debt (MTD '12), Zurich, Switze r land. IEEE, pp. 31–36 2–9 December.
Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. J. Syst. Softw. 23 (2), 111–122 ElsevierFebruary.
Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220 ElsevierMarch.
Lim, W.C., 1994. Effects of reuse on quality, productivity, and economics. Software 11 (5), 23–30 IEEEMay.
Martínez-Fernández, S., Ayala, C.P., Franch, X., Marques, H.M., 2013. REARM: a reuse-based economic model for software reference architectures. 13th International Conference on Software Reuse (ICSR' 13). Springer 18-21 June.
Martini, A., Bosch, J., Chaudron, M., 2014. Architecture Technical Debt: Understanding Causes and a Qualitative Model. In: 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA' 14), Verona, Italy. IEEE, pp. 85–92 27-29 August.
Mavridis, A., 2014. Valuation and Selection of OSS with Real Options. In: 26th International Conference on Advanced Information Systems Engineering (CAISE' 14). Springer, pp. 44–52 16-20 June.
Mikkonen, T., Taivalsaari, A., 2019. Software reuse in the era of opportunistic design. IEEE Softw. 36 (3), 105–111 May-June.
Misra, S.H., 2005. Modeling design/coding factors that drive maintainability of software systems. Softw. Qual. J. 13 (3), 297–320 Springer.
Mo, R., Cai, Y., Kazman, R., Xiao, L., 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: 12th Working IEEE/IFIP Conference on Software Architecture (WICSA '15), Ottawa, Ontario, Canada. IEEE, pp. 51–60 May.
Mohagheghi, P., Conradi, R., 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. Emp. Softw. Eng. 12 (5), 471–516 SpringerMay.
Mohagheghi, P., Conradi, R., 2008. An empirical investigation of software reuse benefits in a large telecom product. Trans. Softw. Eng. Methodol. 17 (3), 13 ACMpagesSeptember.
Morisio, M., Romano, D., Stamelos, I., 2002. Quality productivity and learning in framework-based development: an exploratory case study. Trans. Softw. Eng. 28 (9), 876–888 IEEESeptember.
Nikolaidis, N., Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., 2019. Reusing code from StackOverflow: the effect on technical debt. 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 19). IEEE 28-30 August.
Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D., 2014. Do they really smell bad? A study on developers' perception of bad code smells. In: 30th International Conference on Software Maintenance and Evolution (ISCME' 14), Victoria, Canada. IEEE, pp. 101–110 29 September – 3 October.
Parnas, D.L., 1994. Software Aging. In: 6th International Conference on Software Engineering (ICSE '94), Sorrento, Italy. IEEE Computer Society, pp. 279–287 16 -21 May.
Potdar, A., Shihab, E., 2014. An exploratory study on self-admitted technical debt. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 91–100.
Poulin, J.S., 1999. Reuse: been there done that. Communications 42 (5), 98–100 ACM May.
Riaz, M., Mendes, E., Tempero, E., 2009. A systematic review of software maintainability prediction and metrics. In: 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM' 09). IEEE, Florida, USA, pp. 367–377 15–16 October.
Rine, D.C., 1997. Success factors for software reuse that are applicable across domains and businesses. In: Symposium on Applied Computing (SAC' 97), ACM, San Jose, USA, pp. 182–186 28 February – 2 March.
Runeson, P., Höst, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. John Wiley and Sons.
Seaman, C., Guo, Y., 2011. Measuring and monitoring technical debt. Adv. Comput. 82, 25–46 Elsevier.
Schmid, K., 2013. On the limits of the technical debt metaphor some guidance on going beyond. In: 4th International Workshop on Managing Technical Debt (MTD '13), IEEE Computer Society, San Francisco, USA, pp. 63–66 18 - 26 May.
Tsintzira, A.A., Ampatzoglou, A., Matei, O., Ampatzoglou, A., Chatzigeorgiou, A., Heb, R., 2019. Technical Debt Quantification through Metrics: An Industrial Validation. *15th China-Europe International Symposium on Software Engineering Education (CEISEE' 19)*, IEEE 30-31 May.
van Koten, C., Gray, A., 2006. An application of Bayesian network for predicting object-oriented software maintaina-bility. Inf. Softw. Technol. 48 (1), 59–67 Elsevier.
van Vliet, H., 2008. Software Engineering: Principles and Practice. John Wiley & Sons.
Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q., 2016. Identifying and quantifying architectural debt. In: 38th International Conference on Software Engineering (ICSE), Austin, TX, USA. IEEE/ACM, pp. 488–498 May.
Yli-Huumo, J., Maglyas, A., Smolander, K., 2013. The sources and approaches to management of technical debt: a case study of two product lines in a middle-size finnish software company. *14th International Conference on Product-Focused Software Process Improvement (PROFES' 14)*, Springer 12-14 June.
Yin, R.K., 2003. Case Study Research: Design and Methods, third ed. Sage Publications.
Zaimi, A., Ampatzoglou, A., Triantafyllidou, N., Chatzigeorgiou, A., Mavridis, A., Chaikalis, T., Deligiannis, I., Sfetsos, P., Ioannis Stamelos, 2015. An empirical study on the reuse of third-party libraries in open-source software development. 7th Balkan Conference on Informatics Conference (BCI '15), ACM article 42-4 September.
Zazworka, N., Shaw, M., Shull, F., Seaman, C., 2011. Investigating the impact of design debt on software quality. In: 2nd Workshop on Managing Technical Debt (MTD '11), ACM, Hawaii, USA, pp. 17–23 21 -28 May.
Zhou, Y., Leung, H., 2007. Predicting object-oriented software maintainability using multivariate adaptive regres-sion splines. J. Syst. Softw. 80 (8), 1349–1361 Elsevier.

Dr. Daniel Feitosa is an Assistant Professor in the Faculty Campus Fryslân and the Chief Data Scientist at the Data Research centre of the University of Groningen. He is also an associated researcher in the group of Software Engineering and Architecture of the University of Groningen. He holds a BSc degree (2010) and MSc (2013) in Computer Science from the University of São Paulo, Brazil, and was awarded his PhD degree (2019) in Software Engineering by the University of Groningen. He currently has 20 publications among journal, conference papers and book chapters. His main research interests are in software architecture, software patterns and data analytics.

Dr. Apostolos Ampatzoglou is an Assistant Professor of Software Engineering, in the Department of Applied Informatics in University of Macedonia (Greece). Before joining University of Macedonia, he was an Assistant Professor in the University of Groningen (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). He has published more than 80 articles in international journals and conferences, and is/was involved in over 15 R&D ICT projects, with funding from national and international organizations. His current research interests are focused on technical debt, maintainability, reverse engineering, quality management, and design.

Antonis Gkortzis is a PhD Student at the Athens University of Economics and Business (Greece) in the Software Engineering and Security (SENSE) group. He holds an MSc degree in Software Engineering from University of Groningen (the Netherlands) and a BSc degree in Information Technology from the Technological Institute of Thessaloniki (Greece). His research interests include security, object-oriented design, maintainability, and software quality assessment.

Dr. Stamatia Bibi is an Assistant Professor 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 interests include process models, cost estimation, quality assessment, and cloud computing.

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, as a software designer. His research interests include object-oriented design, software maintenance and evolution. He has published more than 130 articles in international journals and conferences.