Noname manuscript No.
(will be inserted by the editor)

On the Adoption and Effects of Source Code Reuse on Defect Proneness and Maintenance Effort

Giammaria Giordano · **Gerardo Festa** · **Gemma Catolino** · **Fabio Palomba** · **Filomena Ferrucci** · **Carmine Gravino**

Received: date / Accepted: date

Abstract Software reusability mechanisms, like inheritance and delegation in Object-Oriented programming, are widely recognized as key instruments of software design that reduce the risks of source code being affected by defects, other than to reduce the effort required to maintain and evolve source code. Previous work has traditionally employed source code reuse metrics for prediction purposes, e.g., in the context of defect prediction. However, our research identifies two noticeable limitations of the current literature. First, still little is known about the extent to which developers actually employ code reuse mechanisms over time. Second, it is still unclear how these mechanisms may contribute to explaining defect-proneness and maintenance effort during software evolution. We aim at bridging this gap of knowledge, as an improved understanding of these aspects might provide insights into the actual support provided by these mechanisms, e.g., by suggesting whether and how to use them for prediction purposes. We propose an exploratory study, conducted on 12 Java projects—over 44,900 commits—of the Defects4J dataset, aiming at (1) assessing how developers use inheritance and delegation during software evolution; and (2) statistically analyzing the impact of inheritance and delegation on fault proneness and maintenance effort. Our results let emerge various usage patterns that describe the way inheritance and delegation vary over time. In addition, we find out that inheritance and delegation are statistically significant factors that influence both source code defect-proneness and maintenance effort.

Keywords Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.

Giammaria Giordano, Gerardo Festa, Fabio Palomba, Filomena Ferrucci, Carmine Gravino
Software Engineering (SeSa) Lab - University of Salerno (Italy) — E-mail: giagiordano@unisa.it, g.festa22@studenti.unisa.it, fpalomba@unisa.it, fferucci@unisa.it

Gemma Catolino
Jheronimus Academy of Data Science & Tilburg University, The Netherlands — E-mail: g.catolino@tilburguniversity.edu

1 Introduction

Software reusability is the design principle that allows developers to reuse part of the existing software to implement new features [10,96]. This practice is widely recognized as one of the key assets of software development, as developers may have multiple benefits, such as the reduction of evolution time, effort, and cost, other than the reduction of risks of source code being affected by defects [90,58,86].

When it turns to Object-Oriented programming languages, many software reuse mechanisms have been provided over time. Design patterns [27,36], third-party libraries [113,85], and programming abstractions [95] are examples of these mechanisms. Focusing on Java, two very well-known types of programming abstractions are provided to developers: *inheritance* and *delegation* [8]. The former allows a class to take the properties and attributes of another class, establishing a hierarchical relation between them. The latter refers to when a class invokes an instance of another class to carry out operations without performing any other type of action.

The importance of these mechanisms has been remarked several times by researchers. In the early 90s, Chidamber and Kemerer [22] included the Depth of Inheritance Tree (DIT), i.e., a metric that measures the number of classes that inherit from another class, in their Object-Oriented metrics suite. Later on, researchers suggested more ways to measure different aspects of inheritance [13,65,84] and delegation [20,70,106], along with best and bad practices on how to use reusability mechanisms [43,51,66,75]. From the empirical standpoint, a noticeable amount of investigations targeted the role of inheritance and delegation in keeping source code quality under control. For instance, researchers have been studying the relationship between these mechanisms and Object-Oriented metrics [21,19,1], design patterns [4,50], code complexity [2], and source code maintainability [26,38,80]. Perhaps more interestingly, inheritance and delegation metrics have often been employed for building software maintenance predictive models. The key example is defect prediction [44,49], where researchers assessed how reusability mechanisms might contribute to the prediction of future source code defects [9,91,112,28,76]. Similarly, the contribution of inheritance and delegation has been experimented with for predicting maintenance effort change [17,72], code smells [7,29], software vulnerabilities [88], and infrastructure-as-code quality [24].

Despite the availability of a large body of knowledge on how inheritance and delegation mechanisms contribute to the prediction of source code attributes, most of the prediction models defined so far made a strong assumption: *developers make use of reusability principles while evolving source code.*

First, the extent to which these mechanisms are used in practice might notably impact their contribution to prediction models. Second, it is unclear how the relationship between reusability and source code attributes varies over time and, therefore, whether inheritance and delegation mechanisms should still be considered for prediction purposes as the system evolves.

In this paper, we propose an empirical investigation to fill the limitations of current research concerning the adoption of reusability practices and their evolutionary effects on two specific source code attributes such as *defect proneness* and *maintenance effort*. We select these attributes as they represent two interesting use cases to assess reusability mechanisms. On the one hand, these mechanisms are indeed supposed to reduce fault proneness and maintenance effort [90,58,86]. On the other hand, several prediction models targeted the early location of defects and estimation of the effort required to perform evolutionary tasks [17,77,72].

Our study focuses on Java projects, as Java (1) offers mechanisms that encourage the use of inheritance and delegation [23,102] and (2) is still among the most popular programming languages used in industry.[1] To conduct our experiment, we first mine the Defects4J dataset to extract commit-level information on the reusability mechanisms adoption. Then, we developed statistical models to assess the contribution of reusability mechanisms on defect proneness—as indicated by the number of defects over time—and maintenance effort—as indicated by the code churn of commits. The main results report on the inheritance and delegation usage patterns of the 12 projects considered, highlighting that (1) developers tend to frequently use these mechanisms and (2) their adoption varies over time in a significant manner. Furthermore, we identify a statistical relation, corroborated by a fine-grained qualitative investigation, between the adoption of inheritance and delegation and both defect-proneness and maintenance effort, hence concluding that software reuse is a relevant component that affects the way source code quality evolves.

This paper extends our registered report accepted at the *38th IEEE International Conference on Software Maintenance and Evolution* [40]. While in our previous work, we defined the research goals of the study and the envisioned data collected analysis methods, this submission analyzes the study's results achieved and discusses the implications, lessons learned, and actionable items that our work has for researchers and practitioners.

Structure of the paper. Section 2 overviews the research literature connected to our work, pointing out the main differences that let our investigation advance the state of the art. Section 3 defines the study's research questions, other than the research method applied to address them. In Section 4, we discuss the study's results, while in Section 5, we report on the implications that our findings have for researchers and practitioners. The main limitations of the study and the way we mitigated them are discussed in Section 6. Finally, Section 7 provides some final remarks.

2 Background and Related work

In this section, we first provide background information on the most widely used mechanisms in the Object-Oriented programming languages for reusing

[1] Programming language ranking - Year 2021: `https://www.tiobe.com/tiobe-index/`

code: *inheritance* and *delegation*. Then, we survey the related literature targeting code reusability and its impact on source code.

2.1 Background: Inheritance and Delegation Mechanisms in Java

Our study focuses on Java and, for this reason, we describe the way inheritance and delegation mechanisms can be employed in this programming language. In particular, in Java there are two forms through which it is possible to define a hierarchical dependency between two classes:

`'extends'`. Given two classes `A` and `B`, `A` is defined as super-class of `B` if `B` inherits variables or methods by `A`. In Java to establish this super-class – sub-class relation the sub-class must indicate it through the keyword "extends".

`'implements'`. Given a class `B`, and an interface `A`, we will claim that `B` inherits from `A` if `B` implements the interface `A`. In Java this mechanism is provided using the keyword "implements". In particular, when a class `A` inherits using an interface, it must provide a concrete implementation of methods defined as a blueprint on interface.

These definitions recall the concept of *reusability* in terms of specification inheritance, implementation inheritance, and delegation [15]. From a practical point of view, the first one refers to the possibility of replacing an object `A` with an object `B` using a combination of two principles:

- **Strict Inheritance.** When a sub-class `B` exposes behavior and properties of super-class `A` without making any changes [15].
- **The Liskov Substitution Principle.** According to Liskov and Wing [63], given two classes `A` and `B`, `B` is a sub-class of `A` if is possible to substitute the object `A` with the object `B` every time that the object `A` was expected.

The implementation inheritance occurs when a class indirectly reuses a super-class source code. The sub-class can wholly or partially override methods and/or properties and replace the super-class's original behavior with its own. However, the implementation inheritance violates, by definition, the encapsulation principle because a sub-class could accidentally invoke methods or use some proprieties of the super-class in a wrong manner [15]. To avoid this, it is possible to replace the implementation inheritance with the delegation in some cases. With this mechanism, a class `B` does not inherit anything from another class `A`, but `B` invokes methods of `A` directly by declaring itself a variable of type `A`.

2.2 Related Work: The Impact of Inheritance and Delegation Mechanisms on Source Code Quality

Source code reusability has been the subject of several research in the last decades. These touched various angles of the problem, by introducing novel

metrics to capture inheritance relations [22,13,65,84] and delegation [20,70, 106], defining best design practices to exploit the benefits of reusability [43,51], or identifying a number of source code quality issues that reusability can cause, e.g., code smells [66,75,35]. While the scope of our work targets inheritance and delegation mechanisms, it is worth mentioning the existence of close research areas such as the analysis of design patterns [34,115] and third-party libraries [114]. These are additional perspectives that we plan to investigate as part of our future research agenda, but that we leave out of the scope of this paper.

Reusability and code quality. As for the themes of our study, Albalooshi and Mahmood [2] conducted an empirical analysis on the implementation inheritance by considering three programming languages like C++, PYTHON, and JAVA. As a result, the authors found that the mechanisms of JAVA to define inheritance tend to degrade source code quality. Goel and Bathia [41] obtained similar results by analyzing the impact of multilevel inheritance on reusability considering three C++ projects. They found a negative correlation between the use of inheritance and the quality of source code in terms of maintainability. Other research efforts targeted the effect of inheritance and delegation on various aspects of source code quality. Chhikara et al. [21] conducted a case study on one small-scale software project, reporting on the correlation between inheritance metrics and other metrics belonging to the Chidamber and Kemerer suite. Chawla and Nath [19] took a closer look at how inheritance and delegation metrics may impact software coupling, concluding that these metrics can be useful to assess code quality. Similar findings were reported by Abreu et al. [1]. Additional experiments were conducted to assess the relation between reusability and design patterns [4,50] and code complexity [2]: all these studies converged toward the relevance of inheritance and delegation. More recently, we carried out a study to investigate the evolution of inheritance and delegation and their impact on the severity of code smells [38]. The results revealed that inheritance and delegation tend to increase over time, but not in a statistically significant manner. However, increasing the adoption of these mechanisms tends to decrease code smells' severity.

The potential benefits of reusability have led researchers to use inheritance and delegation metrics within prediction models. In this respect, most of the defect prediction models include reusability as a feature [44]. Perhaps more importantly, these metrics have been sometimes shown to significantly contribute to the predictions of those models: for instance, Jureczko and Madeyski [54] showed that the Depth of Inheritance Tree metric is among the best predictors of source code defectiveness. These results were later confirmed by other software maintenance and evolution researches [89,55].

Reusability and maintenance effort. From an empirical side, Prechelt et al. [80] carried out two experiments to investigate the relation between inheritance metrics and maintenance effort estimation. Their results revealed that maintaining a low level of inheritance depth positively impacts the (decrease of) developer's effort to maintain source code. Similarly, Daly et al.

[26] showed that as the inheritance depth level increases, so does the effort of developers to maintain code.

In terms of maintenance effort estimation, researchers have been mainly looking at process-level information (e.g., team data and measurements of the development activities), attempting to provide indications in terms of direct and indirect estimations of entire projects under maintenance [111]. Besides that, researchers have been also working on effort prediction of maintenance activities, which revolves around the prediction of the effort spent in performing specific activities such as code review [69] and bug fixing time [6,12]. The contribution provided by reusability metrics to those models are, however, unclear. Recently, Nagappan et al. [72] and Liu et al. [64] proposed the use of code churn, i.e., the amount of lines of code modified within commits, as an alternative metric of maintenance effort which better aligns with the actual effort spent by developers while performing evolutionary tasks.

Our work. With respect to the papers discussed above, ours has multiple differences. First, most of the previous work analyzed reusability by relying on the computation of metrics, e.g., Depth of Inheritance Tree (DIT); as further elaborated in Section 3, we operationalize reusability by means of specification inheritance, implementation inheritance, and delegation, being able to better map the adoption of reuse mechanisms over time. Second, we conduct a fine-grained analysis where the evolution and impact of reusability are investigated at commit-level. Furthermore, we address a key limitation of most previous works proposing prediction models: the contribution of code reuse to their capabilities indeed assumes that developers make use of reusability mechanisms. As such, our study provides more detailed insights into the potential benefits brought by inheritance and delegation to state-of-the-art prediction models.

3 Research Questions and Methods

The *goal* of the study was to (1) investigate the adoption of reusability mechanisms over time and (2) assess their impact on defect-proneness and maintenance effort. The *purpose* was to understand whether those mechanisms can provide developers with an indication of source code quality variation—considering the defect-proneness and effort to fix faults of a project. The *quality focus* was on the reusability in terms of implementation inheritance, specification inheritance, and delegation and their evolution within software projects. The *perspective* was that of practitioners and researchers: the former are interested in understanding whether the reusability mechanisms can be suitable for monitoring the quality of a system, while the latter are interested in improving their knowledge on how inheritance and delegation mechanisms can vary over time and impact source code quality. The *context* of our investigation was composed of publicly available Java projects, as detailed in Section 3.1.

Based on the goal of our study, we formulated three main research questions. The first aimed at understanding the use of source code reusability mechanisms by developers during software evolution. Specifically, we asked:

RQ$_1$. *How does the use of source code reusability mechanisms vary during software evolution?*

The goal of **RQ$_1$** was that of providing insights on the evolution of reuse mechanisms that might later be exploited to better interpret the findings of **RQ$_2$** and **RQ$_3$**. In other terms, the patterns observed in the context of this research question will also be useful to understand the effects of inheritance and delegation on defect-proneness and maintenance effort, e.g., should we identify an exponential growth in the adoption of delegation, this would potentially make this mechanism more relevant for software evolution, hence influencing more the amount of effort required to apply modifications.

Since we analyze three mechanisms for reusability, i.e., specification inheritance, implementation inheritance, and delegation [15], that can impact differently on software evolution, we considered three sub-research questions:

RQ$_{1.1}$. *How does the use of implementation inheritance vary during software evolution?*

RQ$_{1.2}$. *How does the use of the specification inheritance vary during software evolution?*

RQ$_{1.3}$. *How does the use of delegation vary during software evolution?*

Once the evolution of reusability mechanisms was analyzed, we investigated how the evolution might affect code quality, initially measuring it in terms of fault-proneness. Hence, we asked our second research question:

RQ$_2$. *How do source code reusability mechanisms impact fault-proneness over time?*

Finally, we assessed the impact of reusability mechanisms on the maintenance effort required to fix faults. Among the various *direct* and *indirect* metrics available in literature [111], we operationalize maintenance effort through *code churn*, that is, the amount of lines of code modified within a commit. This is an indirect metric that can proxy the actual effort spent by developers when maintaining source code [68, 71, 111]. In particular, we asked:

RQ$_3$. *How do source code reusability mechanisms impact code churn?*

Figure 1 overviews the research process applied to address our research questions. After a first phase of data extraction, where we collected data about inheritance, delegation, and other code quality indicators, we integrated the various pieces of information for further analysis. In this way, the research questions were addressed by employing statistical tests and models (see details in Section 3.3). To design and report the empirical study, we followed the

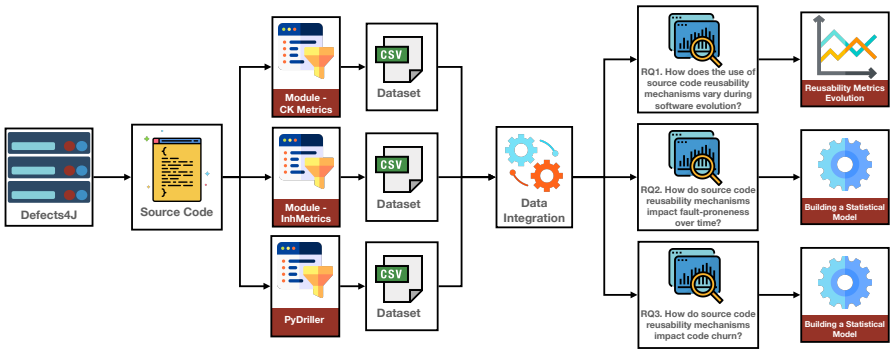

Fig. 1: Overview of the research process applied in the study.

guidelines proposed by Wohlin et al. [110] and the *ACM/SIGSOFT Empirical Standards*[2]. We made all the experimental materials (e.g., datasets, scripts) publicly available in an online appendix [39].

3.1 Context of the Study

The context of the study was composed of JAVA projects available within the DEFECTS4J dataset, which collects information on over 800 real bugs of open-source systems. According to the official documentation[3] each bug collected into the dataset is characterized by the following properties:

1. It is reported in the issue tracker of the project, has an associated commit message for resolution, and it is fixed in a single commit, i.e., the defect resolution never refers to more than one commit;
2. It is associated to a triggering test case that allows its reproduction;
3. It is minimized, meaning that the DEFECTS4J maintainers manually removed commits that would have induced noise, namely commits that did not actually provide information about the introduction of defects or fixing activity (e.g., commits where refactoring activities were done);
4. The fixing activities modified the source code. This means that the defect introduction can be caused by several factors, e.g., wrong parameters in configuration files and problems in the production class. However, the corresponding fixing only concerns changes within the source code.

By design, the dataset *does not* include all the defects reported in the issue trackers of the considered projects, but only those matching the inclusion criteria reported above. In this respect, there are some considerations to make. First, these criteria led to the definition of a set of defects having two

[2]Available at: `https://github.com/acmsigsoft/EmpiricalStandards`
[3]https://github.com/rjust/defects4j

key properties: (1) All the defects were *true positives*, *verifiable*, and *traceable*, meaning that there exists at least one test case letting the defective behavior of the code emerge, other than precise indications on the inducing-fix commit pairs reported by the developers, which were instrumental for our analysis, as further discussed in the following sections; (2) The dataset avoided, by design, possible bias due to the presence of *uncontrolled conditions*, e.g., tangled changes [48], that might have notably affected the validity of the conclusions reported by our study, e.g., refactoring actions targeting inheritance and delegation which were not related to defect fixing operations.

As a consequence of these two properties, the choice of Defects4J enabled the investigation of the impact of reuse mechanisms in a *noise-free* environment in which we could have provided more precise insights into the actual role played by inheritance and delegation. In any case, we are aware that the dataset contains a subset of the defects included in the issue trackers of the considered projects and that the missing analysis of some defects might potentially bias our conclusions. In response to this potential threat to validity, we (i) analyzed further the anatomy of the dataset to better characterize our sample - this is discussed in the remainder of this section; and (ii) conducted additional analyses aiming at assessing the types of defects that were not included in our analysis - these are part of Section 6.

In addition to the discussion on the use of Defects4J, it is worth remarking that, despite the defects being carefully selected, those defects are of different types and natures, hence representing various defects affecting real-world software systems [93]. Last but not least, Defects4J has been widely used in literature (e.g., [67,31]), hence representing a valuable asset that enables us to build additional knowledge on a state-of-the-art dataset - this would also be useful for other researchers interested in building on top of our work.

Project Name	# Bugs	Pull Request	Contributors	Stars	Forks	Commits	Branches	LOC	Analyzed
Commons-Codec	18	9	40	364	207	2,244	7	48k - 34k	●
Commons-Cli	39	8	42	255	154	1,169	4	5k - 16k	●
Commons-Collections	4	37	62	551	389	3,729	8	49k - 60k	●
Commons-CSV	16	8	37	281	220	1,796	4	166k - 166k	●
Commons-Compress	47	9	67	231	210	3,602	9	129k - 91k	●
Gson	18	151	125	21,2k	4,1k	1,668	14	68k - 70k	●
Jackson-Core	26	2	63	2.1k	690	2,124	21	33k - 66k	●
Jackson-Databind	112	19	198	3,1k	1,2k	6,578	22	98k - 235k	●
Jackson-Dataformat-XML	6	3	26	497	189	1,318	19	59k - 117k	●
Commons-JXPath	22	8	17	18	40	601	4	46k - 26k	●
Joda-Time	26	2	77	4,8k	922	2,196	6	103k - 164k	●
Closure-Compiler	174	6	472	6,5k	1,1k	17,962	76	60k - 60k	●
JSoup	93	43	99	9,6k	2k	1,693	3	39k - 34k	●
Commons-Lang	64	92	174	2,3k	176	6,859	8	160k - 190	●
Commons-Math	106	68	48	451	71	7,004	17	58k - 63k	●
Mockito	38	7	246	13,1k	2,3k	5,787	16	73k - 94k	●
JFreeChart	26	22	24	866	355	4218	3	250k - 290k	●

Table 1: Characteristics of the projects considered in the study. The column *'LOC'* provides a range reporting the minimum and maximum values observed over the history of the projects.

As mentioned in Section 2, little has been done to analyze code reuse mechanisms over time and how those may contribute to explaining fault-proneness and maintenance efforts during software evolution. For this reason, our anal-

ysis focused on the analysis of code reuse mechanisms from a low granularity perspective, i.e., commits. We analyzed over 44,900 commits. With respect to our initial plan [40], we had to discard five projects from the total amount of systems available in the dataset. This was mainly due to repository inconsistencies caused by developers' removal of defective commits. Table 1 reports statistics of the projects included in the Defects4J dataset. For each project, the table provides (i) the number of defects, (ii) process metrics such as number of commits, number of pull requests, and number of contributors; (iii) its minimum and maximum LOC; and (iv) if the project could have been analyzed. More particularly, we exploited the latest version of Defects4J (v2.0.0). The defects contained in this version were identified by the original authors using Java 1.8, which is the Java version used by all the projects considered in the study. The reliance on Java 1.8 had some implications on the number of defects reported in the dataset. More particularly, some behavioral changes introduced under Java 8 did not allow to verify anymore 29 of the defects reported in previous versions of Defects4J. As such, these 29 defects were considered *deprecated* and *no longer relevant* in Defects4J 2.0.0. In the light of this consideration, we excluded them from our study. These defects indeed violated the first property mentioned above: on the one hand, they were not verifiable; on the other hand, they were not necessarily true positives, as they were re-labeled by the original authors as non-defective when verifying them through the most appropriate Java version, namely the one employed within the corresponding systems.

3.2 Data Extraction Procedure

To answer our research questions, we quantified the reusability mechanisms employed within the considered software projects. To this aim, we operationalized three metrics capturing reusability mechanisms such as implementation inheritance, specification inheritance, and delegation. We did not rely on existing metrics, like the Depth of Inheritance Tree (DIT) or the Number of Children (NoC) [22], since we aimed at computing metrics that could have *directly* expressed the adoption of reusability mechanisms. Indeed, our metrics have a finer granularity and can indicate the exact constructs added by developers during a change/commit, e.g., the inclusion of a new method that delegates its operations or a change in the inheritance structure—this would not be possible using existing metrics, as they just provide the result of the actions done by developers, e.g., the increase of the depth of inheritance tree, without indications of how that was obtained. To compute the implementation inheritance, specification inheritance, and delegation metrics, we used a tool already validated in our previous work [38]. It was originally developed by the first author of this paper and compute the metrics following these patterns:

Specification Inheritance. Given a class B, the tool considers the specification inheritance as the arithmetical sum of each interface used by B. For

instance, suppose that B inherits methods from two interfaces A and C, and C in turn inherits methods from another interface D. In this case, the specification inheritance for B is 3.

Implementation Inheritance. Suppose that B is a sub-class of A, the tool considers the implementation inheritance as the arithmetical sum of each method in A called by some method in B. For example, suppose that B is a class with N methods, and A a class with just one method call `bar()`. To increase the number of implementation inheritance by one, one of the methods in B must invoke `bar()`.

Delegation. Given a class A, the tool considers the delegation metric as the arithmetical sum of each non-primitive variable (i.e., variables different from `int`, `double`, `String`, and so on) or variables that do not have a binding type provided by external libraries (e.g., `Checkbox` offered by `javax.swing` framework). For each variable, the tool verifies if it is only used to invoke external objects.

The metrics were computed over all the commits of the considered systems and were used to address $\mathbf{RQ}_1$. Specifically, for each commit we computed the sum of (i) specification and implementation inheritance uses and (ii) delegation uses by statically analyzing the files involved in the commit. As for $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we collected information on defects and code churn. To this aim, we mainly relied on the information made available by the Defects4J dataset. In particular, for each project of the dataset, Defects4J assigns to each defect a unique ID and stores an inducing-fixing commit pair, i.e., a pair of commits reporting when the defect was introduced and fixed, respectively, over the history of the project. Starting from these inducing-fixing commit pairs, we could reconstruct the defect history of each project by overlaying them on the full set of commits of the project and considering as defective all the commits between the inducing-fixing commit pairs. As for the code churn, these were collected by exploiting PyDriller, an automatic static analysis tool that can analyze Git repositories to extract information about commits, developers, modifications, diffs, and source code.[4] In our case, we run PyDriller over the commits of the considered systems and extracted the number of modifications performed by developers, i.e., the code churn.

The data extraction process described above was curated by the first two authors of the paper. More specifically, the first author was involved in the mining of the change history of the projects, while the second author had the responsibility to write the scripts for mining Defects4J.

[4] https://pydriller.readthedocs.io/en/latest/intro.html

3.3 Data Analysis Procedure

The collected data were further analyzed as follows:

1. **RQ**$_1$ - *Analysis of the evolution of reusability mechanisms over time.* To address this research question we analyzed how reusability metrics (implementation inheritance, specification inheritance, and delegation) vary over the evolution of the software systems considered. In particular, we employed basic statistical analysis and visualized results using plots.
2. **RQ**$_2$ - *Analysis of the impact on defect-proneness of reusability mechanisms over time.* In this respect, we built a statistical model to verify how reusability metrics impact the variability of defects in the source code.
3. **RQ**$_3$ - *Analysis of the impact on maintenance effort of reusability mechanisms over time.* Similarly to **RQ**$_2$, we built a statistical model to verify how reusability metrics impact the maintenance effort to fix a bug.

Specifically, the statistical models were devised as reported in the following.

Independent Variables. According to our previous considerations, we used the reusability metrics, i.e., implementation inheritance, specification inheritance, and delegation, as independent variables.

Response Variable. In the context of **RQ**$_2$ we were interested in understanding how the reusability metrics impact the defect-proneness of software systems over time. Starting from the defect history built by exploiting Defect4J, we modeled our response variable as follows. Let C_i be a generic commit of the change history of the project P. The number of defects affecting P at the time of C_i was computed through the $\#defects(C_i)$ function, which relies on the following system of equations:

$$\begin{cases} \#defects(C_i) = \#defects(D4J_{C_i}) - \#fixedDefects(D4J_{C_i}), \text{ if } i = 1; \\ \#defects(C_i) = \#defects(C_{i-1}) + (\#defects(D4J_{C_i}) - \#fixedDefects(D4J_{C_i})), \text{ if } i > 1; \end{cases} \quad (1)$$

where $\#defects(D4J_{C_i})$ indicates the number of defects in Defects4J having as inducing commit C_i, $\#fixedDefects(D4J_{C_i})$ indicates the number of defects fixed in the commit C_i, computed as the amount of defects fixed according to Defects4J in C_i, and $\#defects(C_{i-1})$ indicates the number of defects affecting P at commit C_{i-1}. As shown, we had to distinguish the case of the first commit (i=1) from the rest (i>1). When considering the first commit, there cannot indeed be previous fixing operations that influenced the number of defects and, as such, the number of defects at the first commit is only due to the difference between the number of defects pointed out by Defects4J and the number of defects fixed in the same commit. When considering the other commits, instead, the number of defects at the time of the generic commit C_i is given by the total number of defects at time C_{i-1} plus the operations performed within C_i, both in terms of defects introduced

and fixed. After computing the number of defects affecting the considered systems at each commit, we analyzed how this number varied over time.
Let C_i and C_{i+1} be two subsequent commits of the change history of the project P; we labeled the commit pair (C_i, C_{i+1}) as *stable*, *increased*, or *decreased* using the $label(C_i, C_{i+1})$ function described in the following:

$$label(C_i, C_{i+1}) = \begin{cases} `Stable' & \text{if } \#defects(C_i) = \#defects(C_{i+1}); \\ `Increased' & \text{if } \#defects(C_i) < \#defects(C_{i+1}); \\ `Decreased' & \text{if } \#defects(C_i) > \#defects(C_{i+1}). \end{cases} \quad (2)$$

In other terms, we exploited the information previously collected on the number of defects at each commit of the change history of the project P to describe how the amount of defects varied over time.
In $\mathbf{RQ}_3$, instead, we were interested in assessing the effect of reusability metrics on the effort required to fix defects, as measured by code churn. Starting from the defect history of each project, we considered, as relevant for the research question, the commits marked as fixing commits. Afterwards, we computed our response variable as the sum of the code churn of the files involved in those commits.

Control Variables. We computed a number of control variables. This step was required because the impact on the response variables of the statistical models might be due to various additional factors other than the independent variables. As such, we first computed the Chidamber and Kemerer (CK) metrics [22], namely *DIT* (Depth of Inheritance Tree), *NOC* (Number Of Children), *LOC* (Lines of Code), *LCOM* (Lack of Cohesion of Methods), *WMC* (Weighted Methods per Class), *RFC* (Response for a Class), and *CBO* (Coupling Between Objects).
In $\mathbf{RQ}_2$, we also considered the code churn as control variable as suggested by previous findings in the literature [72], i.e., we verified whether the variation of the number of defects was due to the amount of changes performed by developers within commits. This metric was not considered in $\mathbf{RQ}_3$, as it was directly connected to the response variable and could, therefore, bias the conclusions.
With respect to the control variables considered in the study, it is important to discuss the role of *NOC* and *DIT*. These two metrics are by definition connected to code reusability and measure indeed two aspects related to how developers reuse existing source code through inheritance. We included them with the intent of comparing their statistical power to the reusability metrics considered as independent variables. In other terms, the inclusion of *NOC* and *DIT* allowed us to assess the extent to which the reusability metrics we computed represent relevant factors for the response variables when compared to state-of-the-art metrics.
Before building the statistical models, we assessed the presence of possible multi-collinearity concerns. These arise when two or more variables are excessively correlated, possibly biasing the statistical model and the subsequent interpretation of the results [73]. In this respect, we followed well-established guidelines [3, 62]. For each pair of variables, we computed the Spearman's

correlation coefficient [101]. If this scored higher than 0.7, then we removed the variable having the most complex definition to favor explainability - for instance, we preferred keeping the *LOC* metric rather than *WMC* to make the interpretation of the results easier.
The scripts used to compute the dependent and control variables were developed by the second author of the paper, while the independent variables were computed through the tool originally developed by the first author.

Choosing the Statistical Model. To address $\mathbf{RQ}_2$ we built a *Multinomial Log-Linear Model* [103]. This model generalizes logistic regression to multi-class problems, matching our need to have a model able to handle our response variable composed of three values (*"stable"*, *"increased"*, *"decreased"*). As done in our previous work [38], we used R for running the analysis using the function MULTINOM available in the package NNET.[5]
In $\mathbf{RQ}_3$ we had to build a different model because of the nature of the response variable, i.e., code churn. In particular, we built a *Generalized Linear Model* [33] using the GLM function available in R.
The first two authors of the paper were involved in the development of the statistical models. In addition, the interpretation of the results involved all the authors of the paper: these were involved through open discussions and regular meetings with the first two authors.

3.4 Public Availability of Data

To guarantee the replicability of our work and enable other researchers to build on top of our analyses, we made all data and scripts publicly available in our online appendix [39].

4 Analysis of the Results

In the following sections, we report and discuss the results addressing the research questions of the empirical study. For the sake of comprehensibility, we split the discussion by **RQ**.

4.1 $\mathbf{RQ}_1$ - On the Variation of Reusability Mechanisms in Source Code

Figure 2 shows how the three reusability mechanisms considered in our study, i.e., implementation inheritance, specification inheritance, and delegation, evolve over time in the considered software projects. Each row of the figure reports the evolution of the metrics for two projects separately. To facilitate the interpretation of the results and enable a more seamless comparison of evolutionary trends across diverse projects, we normalized the reusability metrics

[5] https://cran.r-project.org/web/packages/nnet/nnet.pdf

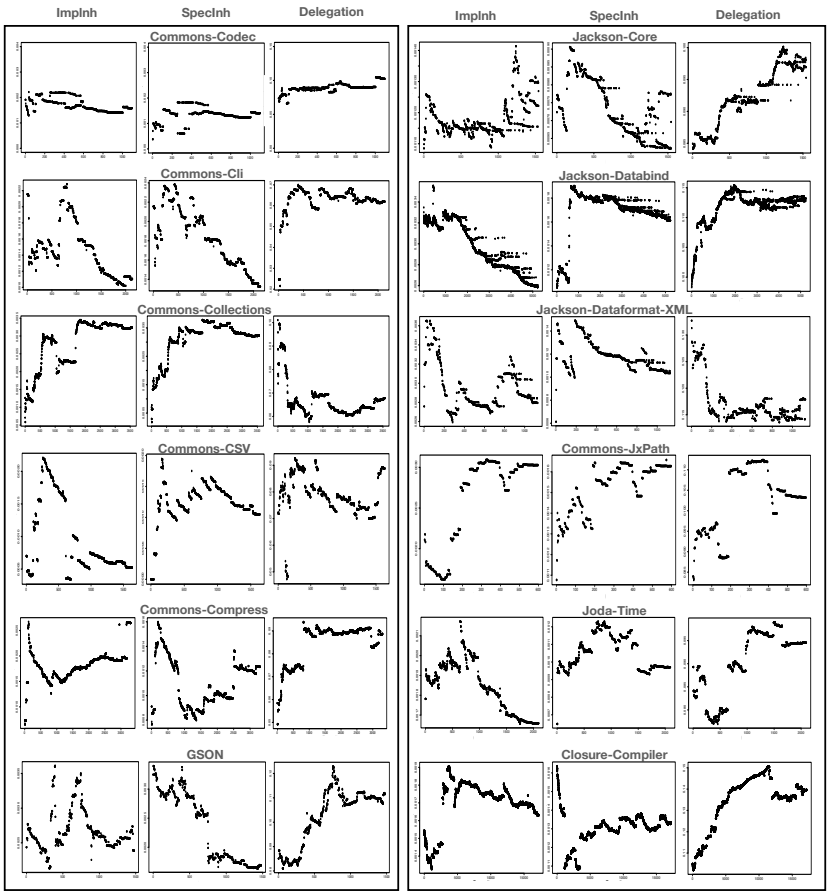

Fig. 2: **RQ$_1$.** Adoption of Reusability Mechanisms Over Time.

by lines of code—in other terms, the figure shows the amount of implementation inheritance, specification inheritance, and delegation mechanisms applied per line of code over the evolution history of the considered projects. These trends were used to interpret the results and address the specific sub-research questions defined in the context of **RQ$_1$**.

4.1.1 RQ$_{1.1}$ - Variation of Implementation Inheritance Over time.

As for the implementation inheritance, the trends in Figure 2 do not always follow a common tendency among the projects.

Increasing - Decreasing Pattern. As shown in Figure 3, we discovered an initial increasing trend in adopting implementation inheritance in seven projects, i.e., Closure-Compiler, Commons-Cli, Commons-CSV, GSON,

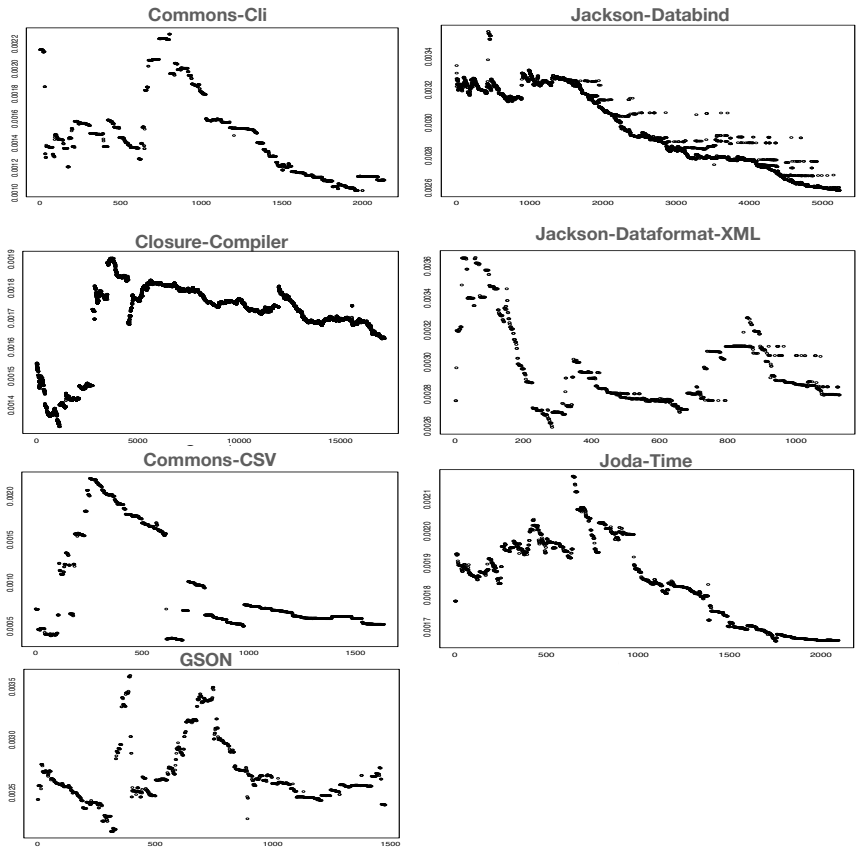

Fig. 3: Increasing - Decreasing Pattern.

Jackson-Databind, Jackson-Dataformat-XML, and Joda-Time, followed by a decreasing usage.

While the shape of the curves varies from case to case, we can still see a common pattern. When we look more closely at these cases, we can identify a similar behavior among the developers of those systems. In all the cases, the adoption of implementation inheritance quickly increased during the first commits, suggesting that developers approached the design of the systems to take reusability into account. Nonetheless, the trend quickly decreased after a while, leading implementation inheritance to be used less and less over time.

This trend leads us to formulate two observations. Firstly, the decline in adoption following a peak could be indicative of a phenomenon known as "design erosion" in the literature [105]. Regardless of the intentions of developers and designers, software design tends to degrade over time due to ongoing changes and increasing complexity, as highlighted by Lehman's laws [60]. This

erosion can also be attributed to inadequate utilization of software quality measures, as emphasized in previous research [30,107,108]. Our findings seem to suggest implementation inheritance is not exempt from this trend, and its adoption is likely to decrease over time.

In the second place, the *"increasing-decreasing"* trend might have clear implications on how reuse mechanisms should be considered within prediction approaches, e.g., defect prediction. Indeed, the employment of implementation inheritance should be carefully considered, and perhaps the usage trend might even lead to the definition of novel feature selection procedures that monitor the way developers are using certain programming constructs to inform the model of the most promising features to consider in that evolution moment.

Steady-Increasing Pattern. Looking at Figure 2, we can identify three less common usage patterns. In particular, two projects, namely Commons-Collections (3rd row) and Commons-JxPath (4th row), appear to exhibit a *"steady-increasing"* trend. The nature of these projects seems to offer a natural explanation for this trend. The former project provides a framework to use efficient data structures in Java, while the latter implements an interpreter of the XPath expression language. Both projects are structured so that most of the source code relies on a core set of classes. For instance, in the Commons-Collections project, classes within the `list` package establish the foundation for creating various advanced element lists. This seems encouraging developers to employ reuse mechanisms like implementation inheritance.

Stable Pattern. Two other projects, namely Commons-Codec (1st row) and Jackson-Core (1st row) of Figure 2, follow mostly a *"stable"* trend. In both cases, the amount of implementation inheritance uses remains constant throughout the evolution. We analyzed the repositories of those projects deeper to better understand this trend. While we could not identify any specific tool or verification procedure conducted by developers to keep reusability under control, we could observe that most of the commits performed over the last years were *peripheral* [5], namely, they pertained to packages of the systems other than core. This may explain the observed trend: developers did not modify any central part of those systems, leaving the original design stable and avoiding an excessive effect of design erosion.

Decreasing - Increasing Pattern. Finally, the Commons-Compress project (5th row in Figure 2) exhibited an anomalous trend which we coined *"decreasing-increasing"*. After a greater adoption of implementation inheritance, the trend steadily decreased before increasing again, but at a lower rate. Also, in this case, we manually dived into the repository in search of possible explanations. We discovered that after the release of the second version of the project in 2010 (release `commons-compress-1.1`), the release engineering process of the system changed, passing from annual to monthly releases. This switch caused a substantial rework of the original architecture, replacing existing code with third-party libraries. Consequently, the overall amount of implementation inheritance uses suddenly decreased in favor of other code

reuse mechanisms. Afterward, the developers of the system kept the implementation inheritance under control, leading to an increasing usage trend.

4.1.2 $RQ_{1.2}$ - Variation of Specification Inheritance Over time

When considering the specification inheritance, the usage patterns identified in $\mathbf{RQ}_{1.1}$ still hold. In particular, we observed the same *"increasing-decreasing"* trend in Commons-Cli, while in Commons-Codec a *"stable"* trend. These findings seem to suggest the existence of a possible strict (cor)relation between implementation and specification inheritance throughout the evolution of software systems, which might depend on the willingness of developers to take (or not) code reusability into account when evolving source code. Part of our future research agenda will consider the effects of this co-evolution of metrics on software quality.

4.1.3 $RQ_{1.3}$ - Variation of Delegation Over time

Regarding the delegation, we could observe similar usage patterns discussed above. Nonetheless, we could also discover situations where the evolution of delegation followed an opposite trend with respect to implementation and specification inheritance ones. This is, for instance, the case of Commons-Collections. Indeed, starting from a high adoption during the first development phases, the amount of delegation used kept decreasing till reaching a stable level. This result was, however, somehow expected as inheritance and delegation are alternatives to each other [15] and, therefore, an increasing use of one may lead to a decreasing use of the other. Similar results were observed when analyzing other projects, e.g., Closure-Compiler Jackson-Core and Compress.

The apparent synergy between inheritance and delegation could offer an opportunity for source code quality predictive models. These models could decide which metrics to focus on at different stages of development. In this way, the models could rely on metrics that can best represent the current state of the system under analysis, potentially improving their predictive capabilities.

Key findings for RQ_1.

In 7 projects out of 12, the use of implementation and specification inheritance followed an *"increasing-decreasing"* trend, with design erosion being the most likely cause behind this result. Other usage patterns were less common and dependent on the specific scope of the projects. We could also confirm the orthogonality of delegation, which followed an opposed trend with respect to both implementation and specification inheritance in four of the considered systems. The results achieved in $\mathbf{RQ}_1$ may have interesting applications in the context of predictive software quality analytics, whose models might be informed by the usage trends on which metrics should be better to use in specific moments of software evolution.

Project	Discarded Variables
Commons-Codec	RFC, NOC
Commons Cli	DIT, NOC, InhImp
Commons-Collections	WMC
Commons-CSV	RFC
Commons-Compress	RFC
Gson	RFC
Jackson-Core	WMC, RFC, DIT, InhImp
Jackson-Databind	RFC
Jackson-Dataformat-XML	WMC, RFC, DIT
Commons-JxPath	DIT
Joda-Time	WMC, RFC, DIT
Closure-Compiler	RFC

Table 2: $\mathbf{RQ}_2$**.** Variables removed because of multi-collinearity.

4.2 $\mathbf{RQ}_2$ - The Impact of Reusability Metrics on Defect-Proneness

In this sub-section, we report the results when studying the impact of reusability metrics on the defect-proneness of source code.

Multi-collinearity analysis. Before discussing the results of the statistical model, it is worth reporting the outcome of the multi-collinearity analysis—which was performed to make sure that no correlated variables were employed within the statistical model and could bias the interpretation of the results (see Section 3). Table 2 lists the variables removed after the application of the correlation analysis. In the first place, we found that RFC was the metric most often removed: in all the cases, it was correlated with LOC and, therefore, we preferred keeping LOC because of its highest degree of interpretability. Secondly, in three projects, i.e., Commons-Collections,

Jackson-Core, and Joda-Time, the WMC metric was removed, again for its correlation with LOC. We also discovered correlations between DIT and NOC in two projects such as Commons-Codec and Commons-Cli: we kept NOC, namely the metric reporting the number of immediate subclasses of a class. In the cases of Jackson-Dataformat-XML and Joda-Time, we found a correlation between DIT and specification inheritance: as the latter was one of the independent variables, we preferred keeping it. Finally, we identified correlations between specification and implementation inheritance in the projects Commons-Cli and Jackson-Core—these correlations could be already hypothesized looking at the trends observed in the context of $\mathbf{RQ}_1$: in these two cases, we were obliged to remove one of the independent variables and decided to opt for implementation inheritance.

	Com.-Codec N=2,134		Com.-Cli N=1,099		Com.-Col. N=3,560		Com.-CSV N=1,634		Comp. N=3,305		Gson N=1,478	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC	-10.098	2.280	-0.691	2.416			-3.539	0.627	-5.248***	-1.903	3.261	-1.305
	(7.495)	(10.981)	(3.434)	(3.602)			(2.559)	(5.136)	(0.033)	(4.387)	(30.899)	(25.907)
DiffNOC					-0.038	0.004	-4.413***	1.052***	10.188***	-5.653***	-0.159	1.536***
					(0.050)	(0.013)	(0.156)	(0.176)	(0.002)	(0.051)	(0.275)	(0.415)
DiffLCOM	0.092	0.054	0.166	-0.744***	0.422	0.476	-0.056	-0.040	-0.066	0.046	0.013	-0.159
	(0.140)	(0.261)	(0.256)	(0.244)	(0.808)	(22.615)	(0.066)	(0.130)	(0.335)	(0.125)	(1.242)	(1.157)
DiffDIT	11.927***	−0.183***	0.012	-0.0003	0.012	-0.0003	-4.526***	0.661***	12.511***	-5.151***	0.696	1.896**
	(0.269)	(0.033)	(5.830)	(0.023)	(5.830)	(0.023)	(0.238)	(0.169)	(0.002)	(0.125)	(0.466)	(0.798)
DiffCBO	-5.434	-9.729***	-0.645	-5.947	-0.878	-0.495***	-0.994	-3.484	-4.163***	1.467	-17.977	-3.472
	(5.898)	(0.243	(3.617)	(3.821)	(58.069)	(0.103)	(4.485)	(8.728)	0.021)	(2.717)	(12.462)	(12.553)
DiffRFC	4.123	-0.030	-0.014	4.123	1.013	1.924						
	(5.784)	(1.106)	(1.027)	(5.784)	(6.087)	(14.548)						
DiffLOC	0.005	0.075	0.002	0.056	-0.611	-0.099	0.175	0.045	0.149*	0.024	0.115	0.689
	(0.302)	(0.346)	(0.139)	(0.200)	(1.053)	(11.153)	0.121)	(0.236)	(0.090)	(0.104)	(1.273)	(1.373)
DiffDelegations	0.058	-0.060	0.017	0.001	-0.069	0.004	0.031	-0.058	0.013	-0.003	0.068	-0.018
	(0.049)	(0.077	(0.022)	(0.025	(0.137)	(0.654)	(0.076)	(0.147)	(0.017)	(0.013)	(0.059)	(0.078)
DiffSpecInh	-1.791	-1.510	0.070	1.382***	1.013	1.924	-0.571	-1.495	-0.187	-0.148	-0.356	0.226
	(1.685)	(3.395	(0.618)	(0.542)	(6.087)	(14.548)	(5.267)	(10.178)	(0.862)	(0.637)	(3.632)	(1.865)
DiffimpInh	-0.060	0.046			0.767	0.771	-1.141	-0.094	-0.337	0.154	0.047	0.267
	(0.940)	(2.134			(3.457)	(13.557)	(2.923)	(4.432)	(0.488)	(0.383)	(2.105)	(1.713)
Churns	-0.002	-0.002	-0.002	-0.005	-0.026	-0.100	-0.004	-0.009	-0.003	-0.0003	-0.015	-0.007
	(0.003)	0.004)	(0.002)	(0.004)	(0.038)	(0.127)	(0.007)	(0.013)	(0.002)	(0.001)	(0.014)	(0.009)
Constant	-4.762***	-4.717***	-3.472***	-3.448***	6.429***	-6.230***	4.605***	4.520***	-4.236***	-4.327***	-4.910***	-5.051***
	(0.248)	(0.246)	(0.187)	(0.187)	(0.536)	(0.527)	(0.265)	0.264)	(0.160)	(0.159)	(0.369)	(0.372)
	Jack.-Core N=1,543		**Jack.-Datab. N=5,228**		**Jack.-XML N=1,128**		**Com.-JxPath N=598**		**Joda-Time N=2,094**		**Clo.-Compiler N=17,171**	
	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
DiffWMC			-2.330	3.344**			-14.452***	41.577***			-8.302***	-3.841***
			(1.546)	(1.619)			(2.914)	(3.503)			(0.006)	(0.004)
DiffNOC	-37.085***	-93.807***	-58.836***	-152.598***	-0.066***	-0.235***	4.203***	-1.798***	-1.001***	-0.723***	0.617***	2.750***
	(0.359)	(0.372)	(0.037)	(0.043)	(0.020)	(0.007)	(0.113)	(0.040)	(0.007)	(0.021)	(0.0003)	(0.0003)
DiffLCOM	-0.024	-0.008	0.155***	0.179***	-0.016	-0.420	0.217	-1.167	-0.835	-0.255	0.076	0.034
	(0.029)	(0.031)	(0.049)	(0.045)	(0.244)	(0.478)	(1.161)	(1.509)	(0.867)	(2.234)	(0.086)	(0.069)
DiffDIT			-70.763***	-124.104***							0.391 ***	-0.665***
			(0.028)	(0.029)							(0.0003)	(0.0002)
DiffCBO	-2.840	-8.386	1.062	2.261*	-1.162	18.673*	-11.521	-28.867***	5.642***	-7.746***	-7.895***	3.860***
	(5.329)	(5.859)	(0.979)	(1.194)	(3.528)	(10.142)	(12.928)	(9.482)	(0.319)	(0.028)	(0.003)	(0.002)
DiffRFC							8.152	-47.303***				
							(7.483)	(8.690)				
DiffLOC	-0.039	0.039	-0.045	-0.013	-0.009	0.077	-3.004*	5.674***	0.645	1.564	0.615***	0.035
	(0.053)	(0.047)	(0.147)	(0.141)	(0.228)	(0.639)	(1.736)	(1.428)	(0.699)	(2.364)	(0.153)	(0.116)
DiffDelegations	0.001	0.003	-0.005	-0.010***	0.027	0.005	0.201*	0.399***	0.032	-0.074	-0.003	0.002
	(0.003)	(0.003)	(0.003)	(0.003)	(0.043)	(0.066)	(0.104)	(0.100)	(0.051)	(0.117)	(0.002)	(0.002)
DiffSpecInh	-0.371	0.491	0.109	-0.065	-0.489	1.159	-4.686*	0.161	-1.633	-6.170***	0.002	-0.048
	(0.439)	(0.407)	(0.095)	(0.100)	(2.984)	(5.170)	(2.742)	(1.719)	(3.875)	(0.250)	(0.278)	(0.268)
DiffimpInh			0.018	0.341***	-0.461	-0.430	-2.314	-15.482***	-1.313	1.200	0.133	-0.034
			(0.099)	(0.090)	(1.068)	(2.108)	(1.718)	(4.800)	(2.317)	(4.509)	(0.118)	(0.096)
Churns	-0.001	-0.004	-0.001	-0.004**	0.002	-8.872***	-0.015*	-0.026***	-0.006(0.006)	-0.019	-0.001	-0.00001
	(0.002)	(0.004)	(0.001)	(0.002)	(0.006)	(0.001)	(0.008)	(0.007)		(0.013)	(0.0004)	(0.0001)
Constant	-4.183***	-4.082***	-4.048***	-4.043***	-5.312***	-5.346***	-3.345***	-3.323***	-4.545***	-4.462***	-4.624***	-4.654***
	(0.237)	(0.237)	(0.118)	(0.127)	(0.440)	(0.518)	(0.262)	(0.264)	(0.243)	(0.251)	(0.081)	(0.080)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

Table 3: $\mathbf{RQ}_2$**.** Results of the statistical model.

Statistical model explanation. Table 3 shows the results of the statistical models built in $\mathbf{RQ}_2$. The independent variables and control variables are reported on the rows, while the various considered systems are reported on the columns—empty cells indicate that a certain variable was removed from the analysis of a specific system as a consequence of the multi-collinearity analysis, while the number of observations (the commits analyzed) for each project is reported in the header of each column. The statistical codes report the p-value for each variable and each project and were used to interpret the results obtained. According to the description reported in the last row of Table 3, a higher amount of '*' implies a higher statistical relevance of a variable with respect to decrease (↓) or increase (↑) of the likelihood to affect the defect-proneness of source code.

Statistical model analysis. Looking at the table, various considerations can be drawn. First and foremost, in 10 out of the total 12 projects we found at least one of the inheritance metrics to be a statistically significant factor to explain the defect-proneness of the considered systems. The NOC metric, in particular, is the one being relevant in more systems. On 8 projects the metric was observed to explain both the increase and decrease of defect-proneness.

To understand how the metric affects the phenomenon of interest, we analyzed the sign of the coefficients. Specifically, the coefficients of a *Multinomial Log-Linear* model relate to a reference category and indicate how the variables change the chances of the dependent variable being affected with respect to the reference category—which was set to *"stable"* in our case. As for the columns "↓" of Table 3, this means that a negative coefficient for a variable X suggests that for one unit increase of X, the chances that the defect-proneness of source code varies toward a decrease are estimated in the amount indicated by the coefficient, i.e., the higher the coefficient the higher the chance that the variable contributes to decrease the defect-proneness of source code. On the contrary, a positive coefficient implies that for one unit increase of X, the chances that the defect-proneness of source code varies toward the stability are estimated in the amount indicated by the coefficient, i.e., the higher the coefficient the higher the chance of defect-proneness being stable over time. Similarly, in the case of the columns "↑", a negative coefficient for X implies that the chances that the defect-proneness of source code varies toward the stability are estimated in the amount indicated by the coefficient, i.e., the higher the coefficient the higher the chance of defect-proneness being stable over time. A positive coefficient would instead indicate that the chances of defect-proneness increasing are estimated in the amount indicated by the coefficient, i.e., the higher the coefficient the higher the defect-proneness of source code.

According to this interpretation, the signs of the coefficients for NOC over the various projects did not report a common pattern. For example, in Commons-compress (5th column, 1st row of Table 3) we observed a positive coefficient of the variable for "↓" and a negative coefficient for "↑", meaning that the variable statistically influences the stability of defect-proneness over time. On the contrary, on the Closure-compiler project (6th column, 2nd

row of Table 3) the coefficients are positive for both "↓" and "↑", meaning that the variable tends to influence the increase of defect-proneness, overall. As such, we could not delineate a common behavior for NOC. Likely, its impact depends on the peculiarities of the development process in place in the different projects rather than on more general aspects.

As for the independent variables considered in our study, namely inheritance and delegation, the discussion is similar. On the one hand, the impact of these metrics is limited to a few projects, suggesting that the defect-proneness of source code is only partially dependent on reusability metrics. On the other hand, the coefficients of the metrics vary without a common pattern. As an example, the coefficient for specification inheritance was positive for "↑" in Commons-Cli and negative in Joda-Time. On the same line, implementation inheritance had a slightly positive coefficient for "↑" in Jackson-Databind, while a negative coefficient in JxPath. As for the delegation, this turned to be statistically relevant on just two projects, i.e., Jackson-Databind and JxPath without a consistent sign. Hence, we could conclude that the reusability metrics themselves have a limited connection to defect-proneness. Other indicators, like the structure of the hierarchies computed by NOC, seem to have more statistical power. As such, it is not the amount of reusability mechanisms used by developers to influence the defect-proneness of source code, but rather the way these mechanisms are used in the specific cases. This result has two main implications. First, we could not identify a drawback in the use of inheritance and delegation with respect to software reliability: hence, the application of reusability mechanisms is not per se something to avoid. However, this result represents a call to researchers in software quality, who are required to devise novel quality checkers and/or empirical investigations to monitor the way code reuse is implemented and how it may negatively affect the defect-proneness of source code.

Another valuable consideration can be drawn when considering the control variables. According to our results, none of them seems to be statistically impactful on defect-proneness. We believe this is a relevant result for the software maintenance and evolution research community as a whole. Code quality metrics have been indeed often used to estimate and/or predict defects: our results indicate the lack of statistical significance and possibly imply that the set of metrics considered within defect prediction models should be reconsidered - in this sense, we corroborate previous findings on the limited value of the Chidamber-Kemerer metric suite for defect prediction [47, 53, 82] as well as further stimulate the research on alternative predictors [11, 28, 76, 79].

Key findings for RQ_2.

Our findings suggest that the use itself of inheritance and delegation does not influence the defect-proneness of source code. Rather, the specific adoption, e.g., how developers structure the hierarchy of the systems being developed, tends to influence more the likelihood of source code being defective. Furthermore, we found a limited connection between code quality metrics and defect-proneness, possibly revealing that previous research on the relation between metrics and defects should be reconsidered.

4.3 $\mathbf{RQ_3}$. On the impact of reusability mechanisms in code churns

Table 4 reports the statistical results obtained when building a *Generalized Linear* model on the data collected for $\mathbf{RQ_3}$. Differently from $\mathbf{RQ_2}$, the dependent variable was the code churn, namely a numerical variable.

Statistical model explanation. The statistical model output a single coefficient for each independent variable: this coefficient corresponds to the impact of a one-unit increase on the amount of code churn. Also in this case, the statistically significant coefficients are highlighted with a '*' symbol - a higher amount of '*' implies a higher statistical relevance of a variable with respect to the code churn computed on a defect-fixing commit i. The variables discarded through the multi-collinearity are the same as $\mathbf{RQ_2}$.

Statistical model analysis. Looking at the table, we can draw various conclusions. As expected, the LOC metric was found to be statistically significant in 9 systems out of 12. The coefficients are also relatively high in all cases, meaning that larger classes are typically harder to maintain - in this respect, we could corroborate previous findings in literature [46,92]. The CBO metric, which computes the coupling between objects, was also statistically significant in nine projects, confirming that developers spend more effort in fixing defects pertaining to highly-coupled classes [59]. Other code quality metrics were not statistically significant. So, in conclusion of this first point of discussion, we could report that, besides LOC and CBO, the role of code metrics to estimate the maintenance effort seems to be limited. Once again, this finding is of the interest of the software maintenance and evolution research community, which might be called to define novel metrics and/or instruments to monitor maintenance effort over time.

Turning the focus on our independent variables, we could find similar conclusions as in $\mathbf{RQ_2}$ when considering inheritance. Both specification and implementation inheritance were indeed most not statistically significant, with some exceptions. The former was relevant for the projects Commons-Cli, Jackson-Dabind, and Joda-Time. However, the sign of the coefficients revealed that the metric was statistically related to the increase of code churn only in the case of Commons-Cli. By analyzing this case further and relating the statistical result with the trend analysis conducted in $\mathbf{RQ_1}$, we could better

	Com.-Codec N=2,134	Com.-Cli N=1,099	Com.-Col. N=3,560	Com.-CSV N=1,634	Comp. N=3,305	Gson N=1,478
DiffWMC	163.951	26.263	-20.375	-20.375	-1,988.919***	-377.039
	(105.295)	(227.457)	(56.965)	(56.965)	(210.722)	(269.203)
DiffNOC			10,213.080***	-132.074	-10,740.970***	17,827.570***
			(2,341.143)	(1,357.614)	(1,699.369)	(3,288.230)
DiffLCOM	1.383	-12.154	7.799*	10.673***	26.285***	-15.488
	(2.713)	(16.169)	(4.680)	(1.905)	(6.021)	(12.955)
DiffDIT	3,341.228***	-2,378.489	-1,787.167		52,852.530***	-6,958.231**
	(903.732)	(1,497.270)	(1,192.199)		(2,813.141)	(2,826.673)
DiffCBO	1,021.357***	-108.063	6,717.225***	-56.282	5,529.115***	1,916.428***
	(150.161)	(134.420)	(652.191)	(94.958)	(307.003)	(145.944)
DiffRFC		192.611***	4.682			
		(57.094)	(60.012)			
DiffLOC	1.293	-9.992*	-58.840***	2.994	5.769	46.604***
	(2.158)	(5.254)	(10.760)	(2.693)	(5.471)	(10.894)
Delegation	0.017	-0.697***	-0.003	0.165	-0.080***	-0.119**
	(0.045)	(0.229)	(0.057)	(0.124)	(0.022)	(0.050)
SpecInh	-1.217	39.595***	1.143	-6.889	-0.710	1.415
	(3.145)	(11.915)	(1.211)	(6.984)	(1.950)	(1.301)
ImpInh	-0.131	-1.026	-0.386		3.653***	2.080
	(1.747)	(0.870)	(4.387)		(1.093)	(2.226)
BugDecrease	1.433	-74.159	-106.139	-1.272	-26.208	-6.128
	(37.641)	(102.978)	(620.995)	(69.685)	(85.330)	(90.985)
BugIncrease	-20.191	-70.799	-111.854	-14.892	-15.856	-14.031
	(37.620)	(101.593)	(620.996)	(69.652)	(86.311)	(96.408)
Constant	52.948***	126.288**	103.349	11.161	91.271***	111.100*
	(15.918)	(51.413)	(77.155)	(20.003)	(25.693)	(63.054)
	Jack.-Core N=1,543	**Jack.-Datab. N=5,228**	**Jack.-XML N=1,128**	**Com.-JxPath N=598**	**Joda-Time N=2,094**	**Clo.-Compiler N=17,171**
DiffWMC		853.627***		-889.089		-35,485.190 ***
		(71.347)		(1,208.343)		(1,024.124)
DiffNOC	21,588.520***	22,830.430***	333.786	24,786.920***	54,104.760***	204,776.100***
	(1,212.595)	(1,564.318)	(509.649)	(5,760.288)	(3,864.815)	(25,377.820)
DiffLCOM	1.241	-28.862***	-8.269***	21.501***	189.720***	454.243***
	(0.765)	(1.208)	(0.992)	(5.505)	(23.536)	(9.841)
DiffDIT		50,782.460***				305,846.900***
		(1,712.723)				(27,225.780)
DiffCBO	1,682.687***	3,147.449***	239.229***	3,504.462***	-31,929.670***	-472.842
	(131.899)	(74.440)	(19.892)	(363.997)	(2,814.595)	(643.145)
DiffRFC				1,358.532***		
				(363.735)		
DiffLOC	28.585***	-8.353***	8.568***	-158.255***	344.715***	1,028.922***
	(1.371)	(3.029)	(0.946)	(12.875)	(42.978)	(38.580)
Delegation	-0.012	0.010*	-0.096**	-0.598**	-0.580***	-0.014*
	(0.017)	(0.006)	(0.044)	(0.294)	(0.107)	(0.008)
SpecInh	2.701	-1.281***	-2.847	6.773	-156.445***	0.868
	(3.677)	(0.305)	(2.324)	(14.133)	(22.026)	(0.737)
ImpInh		-0.140	3.001**	0.942	179.745***	0.406
		(0.499)	(1.408)	(3.559)	(20.294)	(0.407)
BugDecrease	83.885*	28.236	19.831	-41.147	-625.949	-73.094
	(48.156)	(19.492)	(23.738)	(149.800)	(602.612)	(91.940)
BugIncrease	-51.820	-7.043	-20.624	14.086	-690.321	-52.403
	(48.181)	(20.359)	(26.005)	(149.626)	(618.444)	(91.954)
Constant	25.537	182.664***	57.931***	1,561.449***	-6,729.363***	89.824
	(47.505)	(57.987)	(9.413)	(387.616)	(706.442)	(76.824)

Significance codes: *p<0.1; **p<0.05; ***p<0.01.

Table 4: **RQ$_3$.** Results of the statistical model.

understand the reason behind this correlation. Most of the defects available for Commons-Cli were introduced and fixed after the design erosion discussed in **RQ**$_1$. It is therefore reasonable to believe that it was the lack or the decrease in the use of inheritance mechanisms which caused a higher maintenance effort when fixing defects. This interpretation is in line with what observed on the other systems, i.e., Jackson-Dabind and Joda-Time, where the specification inheritance was negatively correlated to maintenance effort, meaning that this was a significant factor to reduce the code churn required to fix defects.

Implementation inheritance was found to be statistically relevant in just two cases, i.e., on Jackson-Databind and JxPath. While in the former case the coefficient was close to zero—indicating little to no correlation to the dependent variable—, it was of -15.482 in the second case. Hence, also in this case we could conclude that this metric was negatively correlated to the maintenance effort. Enlarging the discussion to the other inheritance metrics subject of the study, namely NOC and DIT, we could discover similar results as $\mathbf{RQ}_2$. Both NOC and DIT were positively correlated to the dependent variable and the coefficients were relatively large in all cases: these results imply that the structure of hierarchies might strongly influence the maintenance effort to fix defects, hence corroborating the results obtained in our previous research question, other than the results of empirical studies reporting how NOC and DIT could worsen software maintainability [25, 26, 81].

As for delegation, the coefficients were mostly negative, even if relatively small. Hence, we could conclude that there exist a small negative correlation between the metric and maintenance effort, which implies that the use of delegation may decrease the overall amount of code churn required to fix defects.

Key findings for $\mathbf{RQ}_3$.

Reusability metrics mostly reduce the effort required to fix defects, as measure by code churn. Also in this case, we found that the structure of the hierarchies might affect more maintenance effort than the mere use of inheritance. Finally, the lines of code and coupling between classes represent factors that strongly influence the maintenance effort.

5 Discussion and Implications

The results of our study revealed a number of insights which are worth to further discuss. This section elaborates on the analyses conducted and discusses the key implications of our findings for researchers and practitioners.

5.1 Further Discussion and Analyses

In this respect, there are three main points to discuss.

Relation to Existing Literature. In the first place, it is worth discussing the way our findings relate to previous research on the matter. As discussed already in Section 2, various empirical studies have linked implementation and specification inheritance to source code quality. Some of them, like the works by Mahmood [2] and Goel and Bathia [41], discovered negative correlations between the use of those reuse instruments and source code quality. Our results could not corroborate those observations: according to our analyses, indeed, implementation and specification inheritance are mostly correlated

with positive improvements of source code. As such, we could instead confirm the "common wisdom" for which a higher degree of reusability leads to a higher maintainability of source code [15]. At the same time, we could extend the set of observations conducted on implementation and specification inheritance with respect to our previous work [38]: not only those mechanisms tend to decrease the severity of code smells over time, but also other desirable software maintenance properties, like defect-proneness and effort to fix defects. Last but not least, the statistical results provide additional insights to the body of knowledge on software evolution and maintenance effort estimation. In the former case, our commit-level analysis could provide finer-grained information on how the adoption of the three considered code reuse mechanisms evolves over time. In the latter case, instead, the results of our $\mathbf{RQ}_3$ unveiled the actual relation between code reuse and corrective maintenance—this represents a premier of our study.

Making Sense of the Statistical Data. By definition, our empirical study had a statistical connotation and aimed at analyzing patterns and correlations extracted through the mining of software repositories. As such, the relation between code reuse and defect-proneness has been observed quantitatively. The nature of such an analysis naturally brings some considerations about the reliability of the conclusions provided. In particular, the independent variables in our statistical exercise were computed by means of metrics accounting for their adoption and were assessed against defect-proneness through statistical correlations. The relations unveiled might therefore due to spurious correlations among metrics rather than being the result of causal inference. To account for this potential threat to validity and strengthen the conclusions of the study, we conducted an additional qualitative analysis aimed at assessing the relation between code reuse and defects. In particular, starting from the dataset considered in the study, we (1) computed the number of cases in which defect-inducing and defect-fixing commits involved the variation of inheritance and delegation metrics and (2) manually analyzed those cases to better understand the way these metrics can affect defect-proneness of source code. Such an analysis allowed us to verify more closely which kind of modifications have been applied by developers in terms of inheritance and delegation and how these led to the variations of defect-proneness. The analysis was led by the first author of the paper, who selected the relevant commits and analyzed the diffs between these and their predecessor. To support the manual investigation, the inspector employed automated static analysis tools such as RefactoringMiner [104] and SonarQube [61]—these tools were used to the sole scope of extracting additional information on the code changes applied within the commits.
Such an additional analysis first revealed that in a non-negligible amount of cases, i.e., in about 50% of the defect-fixing commits, the changes applied by developers included modifications that impacted inheritance and delegation metrics. Perhaps more importantly, those modifications were instrumental to accommodate the defect-fixing activities. For instance,

let consider the commit `40689aa` of the project JXPath. This commit addressed a defect concerning the evaluation of strings as boolean expressions. To fix it, the developer moved methods from the subclasses `CoreOperationEqual` and `CoreOperationNotEqual` to the abstract superclass `CoreOperationCompare`, and add a parameter in the super method of the subclass `CoreOperationNotEqual`. These operations had the effect of modifying the implementation inheritance relations of the `CoreOperation` hierarchy. This example well shows how code reuse is employed in practice to reduce the overall complexity of the system and possibly reduce defect-proneness. Indeed, the developer exploited code reuse to let propogate the fix to all subclasses that would have possibly been affected by the string evaluation defect, hence reducing defect-proneness while improving software maintainability. We observed similar cases in the dataset, particularly in 75% of the commits where inheritance and delegation metrics varied as a consequence of defect-fixing activities - for the sake of completeness, we report the details of this qualitative investigation in our online appendix [39]

On Metrics and Their Relation to Defect-Proneness. The last point to further discuss is concerned with the role of the considered metrics with respect to their relation to defect-proneness. In this respect, two observations should be made. In the first place, we discovered that our inheritance and delegation metrics, coming from the operationalization of the reusability mechanisms used by developers, have a relatively low impact on defect-proneness. In the second place, we found out that the control variables of our statistical analysis, namely the metrics pertaining to the Chidamber & Kemerer [22] metric suite, have also a limited connection to defect-proneness. Both findings are somehow surprising: these metrics were indeed experimented in plenty of studies on source code quality and researchers have been often analyzing the extent to which they can support the monitoring and prediction of defect-proneness of source code [9,42].

To provide further, more actionable insights into our findings and better understand the extent to which our statistical analysis would be actually corroborated when considering the impact of code quality metrics on defect prediction, we conducted an additional analysis where we (i) built a defect prediction model and (ii) assessed whether the findings obtained in the context of $\mathbf{RQ}_2$ might have been confirmed.

More specifically, given that our analysis granularity level was the commit and that we needed to account for the time relations between commits, we focused on the so-called *just-in-time* defect prediction [56], that is, the creation of defect prediction models able to assess the defectiveness of individual code commits based on the data collected through the analysis of previous commits.

To make our analysis as precise and sound as possible, we conducted a partial replication of the work by Pascarella et al. [77], who experimented with a large set of features composed of 24 process, product, and developer-oriented metrics to capture the defectiveness of code commits. As product metrics, the

original authors used the metrics also employed within our study. Through this replication, we could therefore assess the role of these metrics when considering their contribution to defect prediction, other than comparing such a contribution with respect to additional metrics typically used in defect prediction, hence enlarging our overview on the value of the considered metrics. While Pascarella et al. [77] mainly focused on a variant of the problem of just-in-time defect prediction aiming at predicting defective files within commits rather than defective commits, they also compared against a standard just-in-time defect prediction model, hence enabling an analysis at commit-level. The reason for relying on this work was threefold. In the first place, Pascarella et al. [77] released an online appendix with all the scripts used in their study and documentation that enables the exact replication of their work: as such, we avoided possible bias due to the re-implementation of the defect prediction model. Second, one of the authors of the work by Pascarella et al. [77] is also a co-author of this submission: as a consequence, we could exploit his knowledge in case of replication issues. Third, Pascarella et al. [77] took into account a large amount of metrics having different nature and coming from previous literature on defect prediction [56,83]: as such, we could conduct a larger and sound experimentation of how quality metrics affect the performance of just-in-time defect prediction. To conduct our analysis, we performed the following steps:

- For each project considered in our study, we mined all the commits to compute the 24 process, product, and developer-oriented metrics. Since the metrics were computed on the files modified within the considered commits, we aggregated them to have a unique commit-level value for each metric. This was done using the "group by" operation, considering the commit hash as the primary key, and applying the mean and median over all the metrics;
- We merged the information collected with the one available in our dataset: for each project and for each commit, we combined the 24 process, product, and developer-oriented metrics with the inheritance and delegation metrics;
- We trained and tested a *Random Forest* classifier, i.e., the best classifier identified in the work by Pascarella et al. [77], by applying a Time Series Split validation. This is a time-aware variant of the cross-fold validation that (i) divides the dataset into K (in our case, K = 10) folds and (ii) in the k^{th} split, it returns first k folds as train set and the $(k+1)^{th}$ fold as test set.[6]
- This validation can be applied when the time order may impact the results and avoid training the model using future commits to predict the defectiveness of past commits. The performance of the model was assessed through multiple evaluation metrics such as precision, recall, F-Measure, and AUC-ROC.

[6] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html`

We investigated two predictive configurations. In the first one, we devised a *binary* defect prediction model that predicts a commit as defective or not, i.e., the standard defect prediction scenario. In the second configuration, we devised a *multi-class* defect prediction model able to assess how the source code defectiveness varies over the evolution of the project, i.e., a defect prediction scenario where the task is to foresee the defectiveness trend in terms of increase, decrease, or stability of the number of defects within a software project. This latter scenario is closer to the research methods employed in our study and was set up with the aim of embedding additional evolutionary considerations within the defect prediction model and investigating the contribution of code quality metrics to assess the overall defectiveness of a software project. From a more technical perspective, the model was devised to assign a commit to a categorical variable within the set {*'Increased'*, *'Decreased'*, *'Stable'*}, namely the same variables used within the Multinomial Log-Linear statistical model built to address $\mathbf{RQ}_2$.
For both predictive scenarios, we ran the model twice: the first time relying on all the metrics and the second time relying on all metrics but those concerned with inheritance and delegation. This was done in an effort to more closely monitor the impact of the main variables of our work, i.e., inheritance and delegation metrics, by quantifying the accuracy gain/drift achieved when considering them as features of the defect prediction models. In addition, we also computed the feature importance to verify which metrics were most relevant for the experimented models.
In terms of results, we could draw multiple considerations. When considering the binary defect prediction scenario, the performance achieved was close to 94% in terms of F-Measure both when considering the models with and without inheritance and delegation metrics. On the one hand, this result seems to indicate that the overall defect prediction capabilities cannot be improved through the use of reusability metrics, hence confirming the results of $\mathbf{RQ}_2$, i.e., inheritance and delegation metrics have a limited connection to defect proneness. On the other hand, it is worth observing that improving over an F-Measure of 94% is always particularly tough: in this sense, the contribution given by inheritance and delegation metrics may be somehow "hidden" by the high performance of the defect prediction model. As a consequence, a more reasonable way to assess the contribution of reusability metrics was to assess the feature importance of the metrics considered by the model relying on inheritance and delegation indicators. Through this analysis, we discovered that (1) the *Random Forest* classifier never selects *specification* and *implementation* inheritance among the top-20 features to use for predicting defective commits in the considered projects; (2) the amount of delegations was in the top-15 features employed by the model in all the projects; (3) the specification and implementation inheritance metrics had limited predictive power, with other inheritance metrics such as NOC and DIT having a slightly higher impact on the predictions. These findings were perfectly in line with the observations reported in $\mathbf{RQ}_2$: we could indeed further corroborate that the defect-proneness of source code is only partially dependent

on reusability metrics and that, instead, the way developers structure hierarchies might impact defects more than the specific reusability mechanisms employed.

In addition, our $\mathbf{RQ}_2$ revealed that the control variables used in our statistical analysis, i.e., the Chidamber-Kemerer metrics, were not statistically impactful on defect proneness. The defect prediction investigation confirmed these findings as well. Indeed, the feature importance analysis constantly reported process metrics such as the entropy of changes [45], the scattering of code changes [28], and commit date [83] as the most impactful features. In the first place, our findings corroborate previous research showing that process metrics can better predict defects with respect to traditional code quality attributes [83] and, as a consequence, provides additional support to the research field involved in the definition of process and developer-oriented metrics for defect prediction. Secondly, our research outlines that the use of code quality metrics, including the inheritance and delegation ones, to assess the defectiveness of source code may result in suboptimal recommendations for developers and, for this reason, these metrics should be used for different purposes and/or for different use cases: for instance, our previous work [38] revealed that quality, inheritance, and delegation metrics can positively contribute to the evolutionary analysis of code smells.

A similar discussion could be done when considering the multi-class prediction model. Also in this case, we found that the models relying and not on reusability metrics had similar performance in terms of F-Measure (94%), with inheritance and delegation metrics that were selected by the *Random Forest* classifier for all projects. While they had a lower predictive power than NOC and DIT, we found that both inheritance and delegation metrics were more impactful than cohesion, coupling, and complexity metrics, e.g., LCOM, CBO, WMC. As such, we could further corroborate that quality, inheritance, and delegation metrics have a limited connection to defect proneness. Similarly to the previous experiment, the entropy of changes [45], the scattering of code changes [28], and commit date [83] were the most important characteristics to predict defective commits, hence suggesting that evolutionary considerations on the defect proneness of source code should be made through the analysis of historical information coming from the the complexity of the development process.

All in all, our findings corroborated the negative results obtained by previous researchers who experimented with code quality metrics in defect prediction [47, 53, 82]. While this is already worrisome for the entire software maintenance and evolution research community, our findings should be considered as even more worrisome because of the granularity of the analysis conducted. We indeed elaborated on the change history information of software projects, analyzing how code quality metrics were related to defect-proneness throughout the evolution of the considered projects, discovering that none of them was statistically correlated to the variation of defect-proneness. As such, our results represent an additional alarm signal for the research community. Our future research agenda includes experimentations aiming at elaborating on

code quality metrics and their actual relation to software maintenance. For the sake of completeness, we report the details of this analysis in our online appendix [39]

5.2 Implications of the Study

On the basis of the results achieved and the additional discussion points elaborated in the previous section, we identified a number of implications for researchers and practitioners.

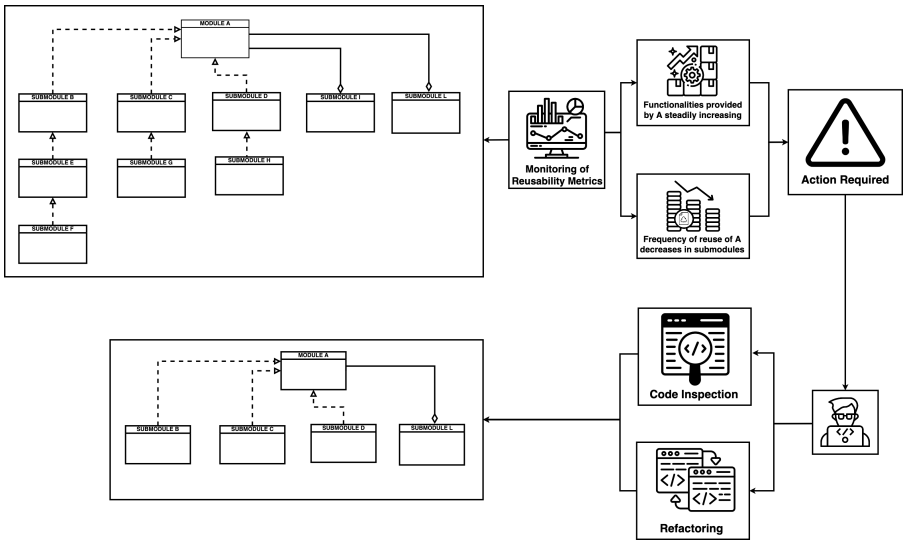

Fig. 4: Use case scenario in which the monitoring of reusability metrics might be exploited.

Monitoring Usage Trends to Improve Software Quality. The usage trends elicited in the context of $\mathbf{RQ}_1$ revealed various forms in which code reusability mechanisms are employed throughout software evolution, while the results obtained in $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ - and the additional qualitative analysis discussed in Section 5.1 - pointed out the benefits reusability may have to reduce both risks connected to poor software reliability and effort required for corrective maintenance activities.

Altogether, these findings seem to suggest that an advanced knowledge on how to improve software quality might be obtained by exploiting precious pieces of information coming from the analysis of the change history of software projects. For instance, we envision the definition of monitoring techniques that, by exploiting the way developers use to adopt code reusability

mechanisms, may recommend the most appropriate actions to conduct while performing corrective maintenance. Similarly, we can envision the definition of novel approaches based on nudge-theory [14] to stimulate developers toward the more frequent or most appropriate adoption of code reuse to reduce the overall defect-proneness of source code. To make our conjectures more tangible, let us consider the scenario depicted in Figure 4, which represents the way we envision a monitoring system may support developers during software maintenance and evolution. More specifically, suppose that a system 'S' contains a module 'A' having (1) multiple submodules, i.e., 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I' and 'L' in Figure 4, each either directly or indirectly inheriting from 'A'; (2) some operations through which the submodules delegate operations to 'A'. In such a scenario, a regular monitoring of reusability metrics or the prediction of usage trends may allow the developer to observe or predict the way the inheritance and delegation relations vary over time, possibly detecting or even preventing the increasing complexity affecting 'A' and its submodules, other than the presence of suboptimal design decisions that would require some refactoring actions.

For instance, suppose that in the scenario proposed in Figure 4 a monitoring system realizes that the amount of functionalities provided by 'A' is steadily increasing, with the frequency of 'A' being reused decreasing in the submodules—this case may indicate that the system is in the descending path of a *'increasing-decreasing'* implementation inheritance pattern identified in $\mathbf{RQ}_1$. This may indicate a suboptimal use of inheritance and delegation: 'A' offers more services, but the submodules inheriting from it do not fully exploit them, suggesting that they are not properly exploiting the inheritance mechanism—note that a similar scenario has been associated with multiple risks for software reliability, including an increasing change- and defect-proneness [74] and a higher likelihood of the system being maliciously attacked because of the suboptimal visibility granted to fields and operations [98]. By monitoring reusability metrics, multiple insights may be provided. On the one hand, developers may be informed of the evolution of reusability metrics and exploit such an information to schedule quality assurance sessions aiming at reducing quality and security concerns, e.g., code review targeting 'A' and the way the submodules interact with it. On the other hand, automated instruments might exploit reusability metrics to recommend refactoring actions aiming at simplifying the hierarchy: for instance, the situation described above, i.e., submodules not fully exploiting the features of 'A', may suggest the presence of a *Refused Bequest* smell [35], whose refactoring may either consist of defining a new superclass only containing the fields and operations that are actually needed to the submodules, i.e., *Extract Superclass* refactoring, or replacing the inheritance mechanism with delegations, *Replace Inheritance with Delegation* refactoring.

On the basis on the considerations above, the multifaceted ways our findings can be exploited therefore represent a call for researchers in the field of software quality and software maintenance and evolution.

Code Reuse and Its Adoption: Two Sides of the Same Coin. Our empirical investigation ($\mathbf{RQ}_2$ and $\mathbf{RQ}_3$) revealed a dichotomy between the concept of code reusability and its actual application. In particular, we found that while reusability itself is a useful instrument to improve software quality and reduce maintenance effort, an inappropriate adoption of these mechanisms may have negative implications. This is indeed the case observed with DIT and NOC in our statistical exercise, two well-known metrics that measure the extent of the hierarchical relations among classes. We found that increases in terms of hierarchical relations lead to negative variations of the defect-proneness of software artifacts. As such, we argue the need for further research, especially in terms of software refactoring optimization. Researchers are indeed called to better investigate the reasons behind the misuse of inheritance and delegation mechanisms and when and why these can deteriorate software quality. These investigations would be instrumental to the definition of novel refactoring techniques that may support developers while optimizing hierarchies of classes.
At the same time, our findings provide two key implications for practitioners. On the one hand, an improved knowledge of the usage patterns might be beneficial to understand the way code reusability evolves in their own projects: practitioners would therefore put in place monitoring instruments to verify the evolution of inheritance and delegation uses and assess how the usage trends co-evolves with software quality. On the other hand, our results might be exploited by practitioners to reason on the use and misuse of inheritance and delegation mechanisms, other than on how the creation of complex hierarchies might possibly worsen source code quality and increase corrective maintenance effort.

Prediction of Code Quality Properties: The Road Ahead. Another aspect to consider is the one concerned with the prediction of code quality properties. In this respect, the findings coming from our research questions altogether contribute to increase the research community awareness with respect to the need for novel code quality prediction techniques and tools. First, the traditional code quality metrics employed in prediction models have little to no correlation to defect-proneness. Second, code reusability mechanisms might potentially boost the code quality analysis and possibly being used within predictive modeling techniques. In addition, the usage trends can be exploited to recommend which of the features would be more worth to use in specific moments of the evolution. All these aspects, emerged from our analyses, represent future perspectives that our research community would like to further investigate. We envision multiple experimentations aiming at revisiting previous findings obtained in literature to account for the evolutionary nature of software - the research method employed in our study, which took the change history information into account, may indeed be generalized to understand how different code quality metrics evolve over time and how they impact software quality. In our opinion, analyses of this type would potentially lead to revolutionize

code quality as we know, revealing insights driven by the actual adoption of code metrics by developers.
At the same time, we envision novel techniques that, by analyzing the evolutionary development context, may feed predictive models with the most relevant metrics to predict source code quality. Also in this case, we believe that an evolution- and context-aware view of predictive software maintenance might potentially substantially boost the support that we, as researchers, may provide to practitioners.
These observations represent the road ahead of software quality prediction models and are part of our future research agenda on the matter.

On the Teaching of Reusability Mechanisms. From an educational perspective, our findings provided multiple insights that may be useful to guide or tune the teaching of reusability mechanisms. In the first place, the findings coming from $\mathbf{RQ}_1$ reported that inheritance and delegation instruments typically follow four well-defined adoption patterns, each of them having implications on source code quality and being motivated by contextual development factors. For instance, we observed that a *"decreasing-increasing"* pattern in terms of inheritance adoption might be motivated by the substantial rework required to include third-party libraries or adapt the architecture of the system being developed and may naturally favor these complex modifications. As a consequence, teaching the *contextual circumstances* making these patterns instrumental for software maintenance and evolution tasks may potentially increase the awareness of the next generation of software engineers toward the adoption of reusability mechanisms, other than increasing their willingness to actually employ them in practice. In other terms, rather than teaching reusability on its own, our findings suggest that an improved way of teaching those principles might involve more complex scenarios where students are exposed to contextual situations requiring them to understand the benefits and drawbacks of reusability, other than the impact that reusability may have on other evolutionary tasks.
Also, $\mathbf{RQ}_2$ showed that the defect-proneness of source code is not influenced by the reliance on inheritance and delegation mechanisms, but rather by the specific adoption of these mechanisms. In our opinion, this is a key finding from the educational perspective: we argue that case-based learning [32] might be a notable advance to let students reason on the effects that reusability may have in specific use cases, hence having a tangible and concrete understanding of the implications of reusability for software quality. In this sense, the use of gamification [16] might further stimulate the capabilities of students to distinguish when and why reusability may represent a valuable tool to improve software quality and reduce risks to software reliability. On a similar note, the results of $\mathbf{RQ}_3$ indicated that the adoption of inheritance and delegation may reduce the effort required to fix defects. Also in this case, the use of case-based learning and gamification may allow students to work on specific, ad-hoc use cases where they are required to fix

defects through the use of reusability mechanisms and assess the impact of their actions on software quality and reliability.
It is our hope that the insights of our study can be of inspiration for educators, who may partially redesign their courses to account for our findings, and software engineering education researchers, who may further investigate the way teaching reusability differently impacts the students' abilities to use inheritance and delegation instruments in practice.

6 Threats To Validity

A number of potential threats might have biased the study. This section discusses them and reports the mitigation strategies applied.

Threats to Construct Validity. Threats in this category refer to a possible mismatch between theory and observation. In this respect, the selection of the dataset represents a crucial point for which there are various observations and remarks to make. We used DEFECTS4J (version 2.0.0), which has been already widely used by the research community in several previous studies (e.g., [52,78,94]) and that reduced possible bias due to the presence of uncontrolled conditions, e.g., tangled changes [48], allowing us to investigate the impact of reuse mechanisms on defect-proneness and maintenance effort more precisely.

As for the defects considered, the GIT repositories of the considered projects may contain more issues than those reported in DEFECTS4J. However, there are two observations to make in this respect. First, a notable amount of these issues do not actually pertain to defects but to other maintenance and evolution tasks. For instance, let us consider the case of the COMMONS-COLLECTIONS project, i.e., the project having the least amount of defects in our study. According to the issue tracker,[7] the project has a total of 787 issues (filtering by Type='All' and Status='All'): of those, only 374 pertain to defects (filtering by Type='Bug' and Status='All'), while the remaining 413 issues refer to enhancements, implementation of new features, and other evolutionary tasks. As such, the set of candidate defects that we might have considered is much lower in size with respect to the raw data reported on the issue trackers. In the second place, a number of issues do not report reliable information. Still taking the COMMONS-COLLECTIONS project as an example, we noticed that 159 of the issues marked as 'Closed' or 'Resolved' (filtering by Type='Bug' and Status='Resolved, Closed') report the strings *"Invalid"*, *"Not a Bug"*, *"Won't Fix"*, *"Cannot Reproduce"*, and *"Duplicate"* as actual resolution, hence indicating that these defects were false positives, not taken into account by the developers, or already addressed as part of duplicated issue reports. As a conclusion, we found out that issue trackers contain a non-negligible amount of noise that would require substantial filtering and data quality procedures, which is indeed what DEFECTS4J guarantees.

[7] The COMMONS-COLLECTIONS issue tracker: `https://issues.apache.org/jira/projects/COLLECTIONS/issues/`.

Still reasoning on the number of issues reported on the issue trackers of the considered systems, it is worth remarking that the candidate set of defects was limited by the types of defects and the types of fixes performed. We should distinguish multiple cases. First, some defects may not pertain to production code, e.g., test code defects, or might relate to the update of third-party libraries or configuration files. As explained in Section 3.1, these defects were not considered by DEFECTS4J and, as a consequence, by our work. However, these defects would have not created any noise for our analysis: indeed, our work aims at understanding how reusability metrics affect the defect proneness of the production code and, for this reason, all the defects that are not related to production code cannot affect our measurements. Second, some defects might not be verifiable or not traceable, even though they relate to the production code. As for the former, they might either represent true defects that developers did not have enough time to deal with or false positives, namely defects that developers ignored and that were marked as 'Resolved' or left opened in the issue tracker without any further action: considering these defects in our analysis would have caused some degree of uncertainty in terms of number of defects considered and, for this reason, we would have likely introduced some bias. As for the latter, these are defects that we could not trace back in the history of the considered projects and, as such, we could not technically analyze without approximation or heuristics that would have, again, introduced some degree of uncertainty. Last but not least, the candidate set of defects might have been limited by the types of fixing activities: DEFECTS4J indeed discards defects whose fixes were performed along with other maintenance and evolution activities, e.g., tangled changes. Among the various cases discussed, this latter was the most critical in our case, as it refers to real defects that were not considered in the scope of the analysis and that might have biased the computation of the number of defects in the change history of the projects considered. A systematic assessment of the noise caused by these missing defects would have required the definition of dedicated data quality protocols through which we could have (i) systematically classified real defects among those not considered by DEFECTS4J; (ii) analyzed the corresponding fixes to understand their nature; and (iii) assessed the extent to which our findings varied when considering the newly classified defects. To the best of our knowledge, the current literature does not offer any (semi-)automated instrument to perform a similar assessment nor guidelines to follow. We deemed the research investigation and methods required to perform such a systematic assessment as out of scope. Nonetheless, to partially analyze the potential noise given by those missing defects, we have attempted to estimate the noise of our analysis in the case of the COMMONS-COLLECTIONS project through a simple, likely suboptimal approach based on text mining and manual analysis. We first (i) mined the summary of each of the 215 marked as 'Closed' or 'Resolved' defects having as resolution the string *"Fixed"*, and (ii) used a keyword-based approach to classify those issues according to their type. More specifically, we classified an issue as 'test-related' if the summary contained the keyword *"test"*, as 'documentation-related' if it contained keywords such as *"JavaDoc"*

and *"comment"*, and as 'configuration-related' if it contained keywords such as *"JDK"*, *"compil*"*, *"build"*, and *"CI"*. In this way, we could estimate the amount of issues whose fixes did not modify the production code, hence covering the first case described above. Afterwards, we manually went through the summaries of the remaining issues to assess how many of them revolved around modifications that were not verifiable, not traceable, or that performed modifications other than defect fixes—hence covering the other possible cases of noise. As a result, we discovered that 181 issues were not considered within Defects4J. Among them, 1% referred to Continuous Integration concerns, 7% to JDK compilation issues, 13% to test code defects, e.g., flaky tests, and 17% to documentation issues, e.g., unclear JavaDoc comments. Hence, 69 of them (38%) of the discarded defects did not concern production code. From the subsequent manual analysis, we discovered that 21 were untraceable (19%), while 84 were issues raised by specific users that the maintainers of the system solved by recommending configuration changes, hence not making any change to the system itself (46%). The remaining 7 defects were not correctly classified by the keyword-based approach and pertaining to documentation or configuration issues - in these cases, the summaries reported keywords different from those used by the classifier, e.g., *"typo"*. Perhaps more interestingly, we found that 34 defects matched the requirements of Defects4J: yet, six were reported between November 2020 and June 2023, namely after the release of Defects4J 2.0.0 (issued on September 15, 2020), while 24 were part of the defects deprecated by Defects4J. As such, the set of defects actually analyzable was four, which is exactly the number of defects we analyzed. While such an additional analysis was not performed on all the considered systems, it let us provide some insights on the noise possibly affecting our results. While we acknowledge that our study took into account only a subset of defects having specific properties, it actually contains most of the real defects that should be taken into account. The noise caused by the presence of additional issues on the issue trackers is likely to be limited, as most defects and corresponding fixes are not related to production code. In conclusion, we argue that our conservative approach in terms of defect selection, i.e., that of relying on the defects pointed out by Defects4J, represents the best option to properly measure the extent to which reusability mechanisms impact the defect proneness of source code. As a side result of our additional analysis, we could also further corroborate the validity of Defects4J - which we consider as a valuable outcome for our research community.

A second threat to validity relates to the selection of the metric used to operationalize maintenance effort. We used code churn [71]: we are aware that this metric can only proxy the actual effort spent when maintaining source code, yet this choice is required in our case because of the unavailability of precise data regarding the maintenance effort in our dataset. Nonetheless, proxy measurements are still used and considered in the field[87]. The tool we used to extract metrics, e.g., reusability or CK metrics, represents another potential threat to validity. We used tools already validated and used by the research community [38,97]. Finally, as mentioned in Section 3.1, in Defects4J a sin-

gle bug can be introduced by multiple factors, but its resolution will always occur within a JAVA file. Thus, to avoid possible threats to contraction validity, we discard commits that introduced defects caused by issues not involving source code. This allowed us to only focus on defects introduced and resolved through changes to the source files.

Threats to Internal Validity. These threats refer to factors that might have impacted the results of the study. In our context, these might be connected to the selection of the metrics used to build the statistical models. On the one hand, we were interested in understanding the role of reusability metrics and, for this reason, we operationalized implementation and specification inheritance, other than delegation, following their exact definition. On the other hand, we used control variables previously shown to be significantly correlated to source code quality [100,99,21,26]. Through these actions, we could rely on a set of independent variables and control metrics which come from either our working hypotheses or the state of the art.

Threats to Conclusion Validity. Threats related to this category refer to the selection and the use of the statistical test. When addressing $\mathbf{RQ}_2$ we modeled the problem using a *Multinomial Logistic Linear* model [103], while we built a *Generalized Linear* model [33] in the context of $\mathbf{RQ}_3$. These choices come from the nature of our response variables, i.e., multiclass and continuous, respectively. Moreover, the research community used these types of model in similar contexts [18,38,57]. The empirical analysis conducted in this study had a quantitative connotation and, in particular, we sought to understand the relation between code reusability and defects through statistical modeling. Nonetheless, we are aware that more qualitative investigations aiming at linking the root cause of defects with the reuse mechanisms might potentially reveal further insights into the matter. While a more complete overview of this type is part of our future research agenda, in the context of this work we already provided some preliminary insights through the manual analysis discussed in Section 5. Such an analysis was in line with the statistical conclusions drawn when addressing $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, increasing our confidence in the results reported in the paper.

Threats to External validity. As for the generalizability of the results, the main threat might be connected to the target of our work. In particular, we focused on 12 JAVA projects having more than 44,900 commits and coming from the DEFECTS4J dataset. As such, our work was based on the analyses conducted on a *sample*: our generalization strategy can be identified within the *sample-based generalization* strategies proposed by Wieringa and Daneva [109]. In particular, among those strategies, the "statistical learning" seems to be the most appropriate. Wieringa and Daneva [109] reported that the *"descriptions of statistical sample phenomena can be used to predict similar phenomena in new samples. [...]. The goal is not to generalize to a population, but to generalize to the next few cases"*. This strategy is basically in line with the *generalizing by similarity* principle described by Ghaisas et al. [37]. When contextualizing those strategies in our case, it is likely that similar results

might be obtained in projects having similar characteristics with respect to those analyzed in our work (see Table 1). Therefore, we cannot claim the generalizability of our findings to projects having different properties or even written in different programming languages. Replications in these contexts would still be desirable and already part of our future research agenda.

7 Conclusion

In this paper, we empirically assessed the evolution of reusability metrics and their impact on defect-proneness and maintenance effort to fix defects. To conduct our analysis, we focused on two specific reusability metrics such as inheritance and delegation. Our empirical study was conducted on the projects available in DEFECTS4J, a well-known dataset reporting a set of JAVA projects along with their own defects. Notably, we conducted the study using a commit-level granularity, in an effort of providing finer-grained observations into the relevance of reusability mechanisms for handling defects.

In the first place, the results let emerge five usage patterns through which specification inheritance, implementation inheritance, and delegation are used throughout software evolution. Secondly, we discovered that the reusability mechanisms are, overall, associated to a decrease of defect-proneness and maintenance effort. At the same time, we found out that other inheritance metrics, like NOC and DIT, relate more to the dependent variables, hence suggesting that it is not the reuse itself that influences defects, but rather the way these mechanisms are used by developers to create hierarchies. These findings raised a number of implications for researchers and practitioners, especially with respect to the need for (1) novel code quality checkers that might monitor how developers adopt reuse mechanisms and how these impact on source code quality; (2) revising previously proposed code quality prediction models on the basis of how code reuse evolves over time.

To sum up, this article proposed the following contributions:

1. The first large-scale empirical study conducted at commit-level to understand how reusability mechanisms are employed by developers over time;
2. Statistical insights into the relation between three code reuse mechanisms, i.e., implementation inheritance, specification inheritance, and delegation, and defect-proneness of source code, both considering the likelihood of code being defective and the effort required to fix defects;
3. A publicly available replication package [39], which releases data and scripts used to conduct this study and that can be used by fellow researchers to replicate the study and build on top of our findings.

Our future research agenda will be devoted to the replication of the analyses conducted on different datasets—including projects written in different programming languages—and considering a larger amount of code reuse mechanisms, e.g., design patterns. In addition, we plan to conduct qualitative investigations to corroborate the findings of the study. Last but not least, we

will work toward the definition of novel code quality monitoring systems and prediction models that exploit the results of our empirical study to improve the support provided to practitioners.

Declaration of Interest

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

Data Availability Statement

The manuscript has data included as electronic supplementary material. In particular: datasets generated and analyzed during the current study, detailed results, as well as scripts and additional resources useful for reproducing the study are available as part of our online appendix `https://giammariagiordano.github.io/On_the_Adoption_and_Effects_of_Source_Code_Reuse_on_Defect_Proneness_and_Maintenance_Effort/`.

Credits

Giammaria Giordano: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Gerardo Festa**: Data Curation, Validation, Writing - Original Draft, Visualization. **Gemma Catolino**: Supervision, Resources, Writing - Review & Editing. **Fabio Palomba**: Supervision, Resources, Writing - Review & Editing. **Filomena Ferrucci**: Supervision, Resources, Writing - Review & Editing. **Carmine Gravino**: Supervision, Resources, Writing - Review & Editing.

Acknowledgements Gemma is partially supported by the European Commission grant no. 825040 (RADON). Fabio is supported by the Swiss National Science Foundation through the SNF Project No. PZ00P2 186090 (TED).

References

1. e Abreu, F.B., Melo, W.: Evaluating the impact of object-oriented design on software quality. In: Proceedings of the 3rd international software metrics symposium, pp. 90–99. IEEE (1996)
2. Albalooshi, F., Mahmood, A.: A comparative study on the effect of multiple inheritance mechanism in java, c++, and python on complexity and reusability of code. International Journal of Advanced Computer Science and Applications **8**(6), 109–116 (2017)
3. Allison, P.: When can you safely ignore multicollinearity. Statistical horizons **5**(1), 1–2 (2012)

4. Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., Avgeriou, P.: The effect of gof design patterns on stability: a case study. IEEE Transactions on Software Engineering **41**(8), 781–802 (2015)
5. Amrit, C., Van Hillegersberg, J.: Exploring the impact of soclo-technlcal core-periphery structures in open source software development. journal of information technology **25**(2), 216–229 (2010)
6. Anbalagan, P., Vouk, M.: On predicting the time taken to correct bug reports in open source projects. In: IEEE International Conference on Software Maintenance, pp. 523–526 (2009)
7. Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering **21**(3), 1143–1191 (2016)
8. Arnold, K., Gosling, J., Holmes, D.: The Java programming language. Addison Wesley Professional (2005)
9. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering **22**(10), 751–761 (1996)
10. Bieman, J.M., Zhao, J.X.: Reuse through inheritance: A quantitative study of c++ software. ACM SIGSOFT Software Engineering Notes **20**(SI), 47–52 (1995)
11. Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P.: Don't touch my code! examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 4–14 (2011)
12. Bougie, G., Treude, C., German, D.M., Storey, M.A.: A comparative exploration of freebsd bug lifetimes. In: IEEE Working Conference on Mining Software Repositories (MSR), pp. 106–109. IEEE (2010)
13. Breesam, K.M.: Metrics for object-oriented design focusing on class inheritance metrics. In: Inter. conference on dependability of computer systems (DepCoS-RELCOMEX'07), pp. 231–237. IEEE (2007)
14. Brown, C.: Digital nudges for encouraging developer actions. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 202–205. IEEE (2019)
15. Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd edn. Prentice Hall, USA (2009)
16. Caponetto, I., Earp, J., Ott, M.: Gamification and education: A literature review. In: European Conference on Games Based Learning, vol. 1, p. 50. Academic Conferences International Limited (2014)
17. Catolino, G., Palomba, F., Fontana, F.A., De Lucia, A., Zaidman, A., Ferrucci, F.: Improving change prediction models with code smell-related information. Empirical Software Engineering **25**(1), 49–95 (2020)
18. Catolino, G., Palomba, F., Tamburri, D.A., Serebrenik, A.: Understanding community smells variability: A statistical approach. In: International Conference on Software Engineering: Software Engineering in Society, p. 77–86 (2021)
19. Chawla, S., Nath, R.: Evaluating inheritance and coupling metrics. International Journal of Engineering Trends and Technology (IJETT) **4**(7), 2903–2908 (2013)
20. Cherkaoui, O., Obaid, A., Serhouchni, A., Simoni, N.: Qos metrics tool using management by delegation. In: IEEE Network Operations and Management Symposium, vol. 3, pp. 836–839. IEEE (1998)
21. Chhikara, A., Chhillar, R., Khatri, S.: Evaluating the impact of different types of inheritance on the object oriented software metrics. International Journal of Enterprise Computing and Business Systems **1**(2), 1–7 (2011)
22. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
23. Craig, I.D.: Inheritance and delegation. In: Object-Oriented Programming Languages: Interpretation, pp. 83–128. Springer (2007)
24. Dalla Palma, S., Di Nucci, D., Palomba, F., Tamburri, D.A.: Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Transactions on Softw. Engineer. pp. 1–1 (2021)

25. Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M.: The effect of inheritance on the maintainability of object-oriented software: an empirical study. In: Proceedings of International Conference on Software Maintenance, pp. 20–29. IEEE (1995)
26. Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M.: Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering **1**(2), 109–132 (1996)
27. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Design pattern recovery through visual language parsing and source code analysis. Journal of Systems and Software **82**(7), 1177–1193 (2009)
28. Di Nucci, D., Palomba, F., De Rosa, G., Bavota, G., Oliveto, R., De Lucia, A.: A developer centered bug prediction model. IEEE Transactions on Software Engineering **44**(1), 5–24 (2017)
29. Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A.: Detecting code smells using machine learning techniques: are we there yet? In: International conference on software analysis, evolution and reengineering (SANER), pp. 612–621. IEEE (2018)
30. Do, L.N.Q., Wright, J., Ali, K.: Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. IEEE Transactions on Software Engineering (2020)
31. Durieux, T., Martinez, M., Monperrus, M., Sommerard, R., Xuan, J.: Automatic repair of real bugs: An experience report on the defects4j dataset (2015)
32. Eshach, H., Bitterman, H.: From case-based reasoning to problem-based learning. Academic Medicine **78**(5), 491–496 (2003)
33. Faraway, J.J.: Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models. Chapman and Hall/CRC (2016)
34. Fontana, F.A., Maggioni, S., Raibulet, C.: Design patterns: a survey on their micro-structures. Journal of Software: Evolution and Process **25**(1), 27–52 (2013)
35. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
36. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Abstraction and reuse of object-oriented design. In: European Conference on Object-Oriented Programming, pp. 406–431. Springer (1993)
37. Ghaisas, S., Rose, P., Daneva, M., Sikkel, K., Wieringa, R.J.: Generalizing by similarity: Lessons learnt from industrial case studies. In: 2013 1st International Workshop on Conducting Empirical Studies in Industry (CESI), pp. 37–42. IEEE (2013)
38. Giordano, G., Fasulo, A., Catolino, G., Palomba, F., Ferrucci, F., Gravino, C.: On the evolution of inheritance and delegation mechanisms and their impact on code quality. In: IEEE Inter. Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1–12 (2022)
39. Giordano, G., Festa, G., Catolino, G., Palomba, F., Ferrucci, F., Gravino, C.: Web Appendix of the paper. `https://giammariagiordano.github.io/On_the_Adoption_and_Effects_of_Source_Code_Reuse_on_Defect_Proneness_and_Maintenance_Effort/`. Online
40. Giordano, G., Festa, G., Catolino, G., Palomba, F., Ferrucci, F., Gravino, C.: On the adoption and effects of source code reuse on defect proneness and maintenance effort. arXiv preprint arXiv:2208.07471 (2022)
41. Goel, B.M., Bhatia, P.K.: Analysis of reusability of object-oriented systems using object-oriented metrics. ACM SIGSOFT Software Engineering Notes **38**(4), 1–5 (2013)
42. Gyimóthy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software engineering **31**(10), 897–910 (2005)
43. Haefliger, S., Von Krogh, G., Spaeth, S.: Code reuse in open source software. Management science **54**(1), 180–193 (2008)
44. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering **38**(6), 1276–1304 (2011)
45. Hassan, A.E.: Predicting faults using the complexity of code changes. In: 2009 IEEE 31st international conference on software engineering, pp. 78–88. IEEE (2009)

46. Hayes, J.H., Patel, S.C., Zhao, L.: A metrics-based software maintenance effort model. In: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., pp. 254–258. IEEE (2004)
47. He, P., Li, B., Liu, X., Chen, J., Ma, Y.: An empirical study on software defect prediction with a simplified metric set. Information and Software Technology **59**, 170–190 (2015)
48. Herzig, K., Just, S., Zeller, A.: The impact of tangled code changes on defect prediction models. Empirical Software Engineering **21**(2), 303–336 (2016)
49. Hosseini, S., Turhan, B., Gunarathna, D.: A systematic literature review and meta-analysis on cross project defect prediction. IEEE Transactions on Software Engineering **45**(2), 111–147 (2017)
50. Huston, B.: The effects of design pattern application on metric scores. Journal of Systems and Software **58**(3), 261–269 (2001)
51. Jalender, B., Govardhan, A., Premchand, P.: Designing code level reusable software components. International Journal of Software Engineering & Applications **3**(1), 219 (2012)
52. Jiang, J., Xiong, Y., Xia, X.: A manual inspection of defects4j bugs and its implications for automatic program repair. Sci. China Inf. Sci. **62**(10), 200102:1–200102:16 (2019)
53. Jureczko, M.: Significance of different software metrics in defect prediction. Software Engineering: An International Journal **1**(1), 86–95 (2011)
54. Jureczko, M., Madeyski, L.: Towards identifying software project clusters with regard to defect prediction. In: International conference on predictive models in software engineering, pp. 1–10 (2010)
55. Jureczko, M., Spinellis, D.: Using object-oriented design metrics to predict software defects. Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej pp. 69–81 (2010)
56. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2012)
57. Lambiase, S., Catolino, G., Tamburri, D.A., Serebrenik, A., Palomba, F., Ferrucci, F.: Good fences make good neighbours? on the impact of cultural and geographical dispersion on community smells. In: IEEE/ACM International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), p. to appear. ACM (2022)
58. Lange, B.M., Moher, T.G.: Some strategies of reuse in an object-oriented programming environment. In: Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 69–73 (1989)
59. Leach, R.J.: Software metrics and software maintenance. Journal of Software Maintenance: Research and Practice **2**(2), 133–142 (1990)
60. Lehman, M.M.: Laws of software evolution revisited. In: European Workshop on Software Process Technology, pp. 108–124. Springer (1996)
61. Lenarduzzi, V., Pecorelli, F., Saarimaki, N., Lujan, S., Palomba, F.: A critical comparison on six static analysis tools: detection, agreement, and precision. Journal of Systems and Software p. 111575 (2022)
62. Lieberman, M.G., Morris, J.D.: The precise effect of multicollinearity on classification prediction. Multiple Linear Regression Viewpoints **40**(1), 5–10 (2014)
63. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(6), 1811–1841 (1994)
64. Liu, J., Zhou, Y., Yang, Y., Lu, H., Xu, B.: Code churn: A neglected metric in effort-aware just-in-time defect prediction. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 11–19 (2017)
65. Mal, S., Rajnish, K.: New quality inheritance metrics for object-oriented design. International Journal of Software Engineering and Its Applications **7**(6), 185–200 (2013)
66. Mantyla, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: International Conference on Software Maintenance (ICSM), pp. 381–384. IEEE (2003)
67. Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M.: Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empirical Software Engineering **22**(4), 1936–1964 (2017)

68. McIntosh, S., Adams, B., Nguyen, T.H., Kamei, Y., Hassan, A.E.: An empirical study of build maintenance effort. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 141–150. IEEE (2011)
69. Mishra, R., Sureka, A.: Mining peer code review system for computing effort and contribution metrics for patch reviewers. In: IEEE Workshop on mining unstructured data, pp. 11–15. IEEE (2014)
70. Munro, M.J.: Product metrics for automatic identification of" bad smell" design problems in java source-code. In: IEEE International Software Metrics Symposium (METRICS'05), pp. 15–15. IEEE (2005)
71. Munson, J.C., Elbaum, S.G.: Code churn: A measure for estimating the impact of code change. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), pp. 24–31. IEEE (1998)
72. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: International conference on Software engineering, pp. 284–292 (2005)
73. O'brien, R.M.: A caution regarding rules of thumb for variance inflation factors. Quality & quantity **41**(5), 673–690 (2007)
74. Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering **23**(3), 1188–1221 (2018)
75. Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., De Lucia, A.: Mining version histories for detecting code smells. IEEE Transactions on Software Engineering **41**(5), 462–489 (2014)
76. Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R.: Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering **45**(2), 194–218 (2017)
77. Pascarella, L., Palomba, F., Bacchelli, A.: Fine-grained just-in-time defect prediction. Journal of Systems and Software **150**, 22–36 (2019)
78. Perera, A.: Using defect prediction to improve the bug detection capability of search-based software testing. In: IEEE/ACM Inter. Conf. on Automated Software Engineering (ASE), pp. 1170–1174 (2020)
79. Posnett, D., D'Souza, R., Devanbu, P., Filkov, V.: Dual ecological measures of focus in software development. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 452–461. IEEE (2013)
80. Prechelt, L., Unger, B., Philippsen, M., Tichy, W.: A controlled experiment on inheritance depth as a cost factor for code maintenance. Journal of Systems and Software **65**(2), 115 – 126 (2003)
81. Prechelt, L., Unger, B., Philippsen, M., Tichy, W.: A controlled experiment on inheritance depth as a cost factor for code maintenance. Journal of Systems and Software **65**(2), 115–126 (2003)
82. Radjenović, D., Heričko, M., Torkar, R., Živkovič, A.: Software fault prediction metrics: A systematic literature review. Information and software technology **55**(8), 1397–1418 (2013)
83. Rahman, F., Devanbu, P.: How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 432–441. IEEE (2013)
84. Rajnish, K., Bhattacherjee, V.: Class inheritance metrics-an analytical and empirical approach. INFOCOMP Journal of Computer Science **7**(3), 25–34 (2008)
85. Salza, P., Palomba, F., Di Nucci, D., De Lucia, A., Ferrucci, F.: Third-party libraries in mobile apps. Empirical Software Engineering **25**(3), 2341–2377 (2020)
86. Sharma, A., Grover, P., Kumar, R.: Reusability assessment for software components. ACM SIGSOFT Software Engineering Notes **34**(2), 1–6 (2009)
87. Shihab, E., Kamei, Y., Adams, B., Hassan, A.E.: Is lines of code a good measure of effort in effort-aware models? Information and Software Technology **55**(11), 1981–1993 (2013). DOI https://doi.org/10.1016/j.infsof.2013.06.002. URL `https://www.sciencedirect.com/science/article/pii/S0950584913001316`
88. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE transactions on software engineering **37**(6), 772–787 (2010)

89. Singh, P.D., Chug, A.: Software defect prediction analysis using machine learning algorithms. In: Inter. Conf. on Cloud Computing, Data Science & Engineering-Confluence, pp. 775–781. IEEE (2017)
90. Singh, S., Singh, S., Singh, G.: Reusability of the software. Inter. journal of computer applications **7**(14), 38–41 (2010)
91. Singh, Y., Kaur, A., Malhotra, R.: Empirical validation of object-oriented metrics for predicting fault proneness models. Software quality journal **18**(1), 3–35 (2010)
92. Sjøberg, D.I., Yamashita, A., Anda, B.C., Mockus, A., Dybå, T.: Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering **39**(8), 1144–1156 (2012)
93. Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., de Almeida Maia, M.: Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 130–140. IEEE (2018)
94. Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., de Almeida Maia, M.: Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: International Conference on Software Analysis, Evolution and Reengineering, SANER, pp. 130–140. IEEE Computer Society (2018)
95. Sommerville, I.: Software engineering 9th edition. ISBN-10 **137035152**, 18 (2011)
96. Soundarajan, N., Fridella, S.: Inheritance: From code reuse to reasoning reuse. In: International Conference on Software Reuse (Cat. No. 98TB100203), pp. 206–215. IEEE (1998)
97. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: Python framework for mining software repositories. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 908–911 (2018)
98. Spooner, D.L., et al.: The impact of inheritance on security in object-oriented database systems. In: DBSec, pp. 141–150. Citeseer (1988)
99. Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., Russo, B.: An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. Empirical Software Engineering **10**(1), 81–104 (2005)
100. Tamburri, D.A., Palomba, F., Kazman, R.: Success and failure in software engineering: A followup systematic literature review. IEEE Transactions on Engineering Management (2020)
101. Taylor, R.: Interpretation of the correlation coefficient: a basic review. Journal of diagnostic medical sonography **6**(1), 35–39 (1990)
102. Tempero, E., Yang, H.Y., Noble, J.: What programmers do with inheritance in java. In: European Conference on Object-Oriented Programming, pp. 577–601. Springer (2013)
103. Theil, H.: A multinomial extension of the linear logit model. International economic review **10**(3), 251–259 (1969)
104. Tsantalis, N., Ketkar, A., Dig, D.: Refactoringminer 2.0. IEEE Transactions on Software Engineering (2020)
105. Van Gurp, J., Bosch, J.: Design erosion: problems and causes. Journal of systems and software **61**(2), 105–119 (2002)
106. VanHilst, M., Fernandez, E.B.: Reverse engineering to detect security patterns in code. In: International Workshop on Software Patterns and Quality. Information Processing Society of Japan. Citeseer (2007)
107. Vassallo, C., Palomba, F., Bacchelli, A., Gall, H.C.: Continuous code quality: are we (really) doing that? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 790–795 (2018)
108. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H.C., Zaidman, A.: How developers engage with static analysis tools in different contexts. Empirical Software Engineering **25**(2), 1419–1457 (2020)
109. Wieringa, R., Daneva, M.: Six strategies for generalizing software engineering theories. Science of computer programming **101**, 136–152 (2015)
110. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
111. Wu, H., Shi, L., Chen, C., Wang, Q., Boehm, B.: Maintenance effort estimation for open source software: A systematic literature review. In: IEEE international conference on software maintenance and evolution (ICSME), pp. 32–43 (2016)

112. Yu, P., Systa, T., Muller, H.: Predicting fault-proneness using oo metrics. an industrial case study. In: European Conference on Software Maintenance and Reengineering, pp. 99–107. IEEE (2002)
113. Zaimi, A., Ampatzoglou, A., Triantafyllidou, N., Chatzigeorgiou, A., Mavridis, A., Chaikalis, T., Deligiannis, I., Sfetsos, P., Stamelos, I.: An empirical study on the reuse of third-party libraries in open-source software development. In: Balkan Conference on Informatics Conference, pp. 1–8 (2015)
114. Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., Liu, Y.: Research on third-party libraries in android apps: A taxonomy and systematic literature review. IEEE Transactions on Software Engineering (2021)
115. Zhang, C., Budgen, D.: A survey of experienced user perceptions about software design patterns. Information and Software Technology **55**(5), 822–835 (2013)