On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality

Giammaria Giordano,[1] Antonio Fasulo,[1] Gemma Catolino,[2]
Fabio Palomba,[1] Filomena Ferrucci,[1] Carmine Gravino[1]
[1]Software Engineering (SeSa) Lab, Department of Computer Science - University of Salerno, Italy
[2]Jheronimus Academy of Data Science & Tilburg University, The Netherlands
giagiordano@unisa.it, a.fasulo@studenti.unisa.it, g.catolino@tilburguniversity.edu
fpalomba@unisa.it, gravino@unisa.it, fferrucci@unisa.it

***Abstract*—Source code reuse is considered one of the holy grails of modern software development. Indeed, it has been widely demonstrated that this activity decreases software development and maintenance costs while increasing its overall trustworthiness. The Object-Oriented (OO) paradigm provides different internal mechanisms to favor code reuse, i.e., specification inheritance, implementation inheritance, and delegation. While previous studies investigated how inheritance relations impact source code quality, there is still a lack of understanding of their evolutionary aspects and, more particular, of how these mechanisms may impact source code quality over time. To bridge this gap of knowledge, this paper proposes an empirical investigation into the evolution of specification inheritance, implementation inheritance, and delegation and their impact on the variability of source code quality attributes. First, we assess how the implementation of those mechanisms varies over 15 releases of three software systems. Second, we devise a statistical approach with the aim of understanding how inheritance and delegation let source code quality—as indicated by the severity of code smells—vary in either positive or negative manner. The key results of the study indicate that inheritance and delegation evolve over time, but not in a statistically significant manner. At the same time, their evolution often leads code smell severity to be reduced, hence possibly contributing to improve code maintainability.**

***Index Terms*—Software Reuse; Quality Metrics; Software Maintenance and Evolution; Empirical Software Engineering.**

I. Introduction

Software reusability refers to the development practice through which developers make use of existing code when implementing new functionalities [1], [2]. This is widely considered as a best practice, as it leads developers to save time, energy, and maintenance costs, other than relying on source code that has been previously tested [3], [4].

Contemporary Object-Oriented (OO) programming languages, *e.g.,* Java, provide developers with various mechanisms supporting code reusability: examples are design patterns [5], [6], the use of third-party libraries [7], [8], and programming abstractions [9]. These latter, in particular, have caught the attention of researchers since the rise of object-orientation and were found to be a valuable element to increase software quality and reusability [10], [11], [12], [13], [14].

When focusing on Java, there are two well-known abstraction mechanisms such as *inheritance* and *delegation* [15].

Inheritance is the process by which one class takes the property of another class: the new classes, known as derived or children classes, inherit the attributes and/or the behavior of the pre-existing classes, which are referred to as base, super, or parent classes. Delegation is, instead, the mechanism through which a class uses an object instance of another class by forwarding it messages and letting it performing actions [15].

The importance of inheritance and delegation has been remarked multiple times by the research community. In 1994, Chidamber and Kemerer [16] included in their Object-Oriented metric suite the Depth of the Inheritance Tree (DIT) metric, a measure of the number of classes that inherit from one another. Later on, various metric catalogs proposed variations of DIT as well as other inheritance metrics [17], [18], [19]. In addition, the sub-optimal adoption of inheritance and delegation mechanisms had led to the definition and investigation of reusability-specific code smells [20], [21], [22], [23]: as an example, Fowler [24] defined the *Refused Bequest* and *Middle Man* code smells, which refer to the poor use of inheritance and delegation in Object-Oriented programs that might lead to deteriorate their code quality [22], [25], [26], [27]. These studies have also led to the definition of automated code smell detection and refactoring approaches [28], [29], [30].

Still from an empirical standpoint, a number of studies targeted the role of inheritance and delegation mechanisms for monitoring software quality. In particular, researchers devoted extensive effort on the understanding of the potential impact of those mechanisms on software metrics [31], [32], [33], maintainability effort and costs [34], [35], [36], [37], design patterns [38], [39], change-proneness [40], [41], [42], [43], and source code defectiveness [44], [45], [46], [47].

While the current body of knowledge provides compelling evidence of the value of reusability mechanisms for the analysis of source code quality properties, we can still identify a noticeable research gap: as Mens and Demeyer [48] already reported in the early 2000s, the *long-term* evolution of source code quality metrics might provide a different perspective of the nature of a software project, possibly revealing complementary or even contrasting findings with respect to the studies that investigated code metrics in a fixed point of software evolution. To the best of our knowledge, Nasseri *et al.* [49] were the only researchers studying the evolution of reusability metrics. They specifically focused on the size of the inheritance hierarchies and aimed at assessing whether developers had the tendency of adding classes at different levels of the hierarchy while evolving their projects: the results reported that the growth of inheritance hierarchies is limited and typically involves up to two levels.

Stemming from the previous investigations on the matter, this paper builds on this line of research by proposing an empirical analysis of how inheritance and delegation mechanisms evolve over time as well as their effects of software quality evolution. Our interest in inheritance and delegation is due to our willingness to (1) investigate built-in abstraction mechanisms that developers are supposed to frequently use to increase the reusability of source code; and (2) bridge the gap left by previous studies, *i.e.,* an empirical understanding of the evolutionary aspects of inheritance and delegation might provide a more comprehensive understanding on the role of those mechanisms for source code quality. Our study is conducted on JAVA: while recognizing that other languages (*e.g.,* PYTHON) are becoming more popular, JAVA is still ranked in the top-three of the programming languages, according to the TIOBE index.[1] In addition, the structure of the programming language enables a more natural use of inheritance and delegation with respect to other languages [50], [51], which allows us to better understand how these mechanisms evolve and influence code quality. Finally, previous studies investigated JAVA and, therefore, our focus enables a comparison with them.

More particularly, we first mine evolutionary data pertaining to 15 releases of three open-source projects. Then, we statistically compare the number of inheritance and delegation mechanisms implemented over subsequent releases in order to assess the trend followed by the adoption of those metrics. Finally, we build a statistical model relating inheritance and delegation metrics, as well as other confounding factors, to the variation of code smell severity, in an effort of understanding the impact of reusability metrics on the likelihood of code smells to become more/less severe over time. The key results of our study report that the adoption of reusability mechanisms increases over time. Yet, when controlled for size, the increase does not appear as statistically significant. In any case, the evolution of inheritance and delegation is statistically connected to the decrease of code smell severity in most cases and, indeed, we discovered negative effects only in a few cases. To sum up, our paper offers the following three main contributions:

1) Evidence-based insights into the evolution of inheritance and delegation adoption in open-source systems, which researchers might exploit to further understand the developer's code quality practices;
2) An empirical, evolutionary exploration of the impact of inheritance and delegation mechanisms on code smell severity, which can be of the interest of researchers working in the field of code smell detection and prioritization [52], [53], [54], other than for tool vendors interested in providing developers with better monitoring tools for software quality evolution [55], [56].
3) A publicly available replication package [57] that contains both data and scripts used to conduct our experimentation and that can be used by researchers to replicate and extend the results discussed in the paper.

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

II. BACKGROUND AND RELATED WORK

This section summarizes the usage of object-oriented reusability through the mechanisms of inheritance and delegation. In addition, we overview the related literature.

A. Reuse through inheritance and delegation

In JAVA, a hierarchical dependency between two classes is established by means of two main constructs:

`'extends'`. Through the use of the keyword `'extends'`, a class `A` inherits state and behavior from a class `B`, establishing a subclass-superclass relation. The attributes defined and the methods implemented in `B` become available when calling objects of the class `A`.

`'implements'`. The adoption of this keyword allows a class `A` to inherit the methods defined within an interface `B`: in particular, a JAVA *interface* only specifies the blueprint of a class, *i.e.,* the methods that all the classes inheriting from it must provide, without providing a concrete implementation. In turn, the inheriting classes must override the acquired methods in order to specify their behavior.

These constructs enable the definition of reusability in terms of specification inheritance, implementation inheritance, and delegation [58]. The former represents the possibility to replace one object with another, combining two principles:

- **The Liskov substitution principle**, according to which if an object of type `A` can be replaced anywhere one expects a type `B` object, then `A` is a subtype of `B` [59];
- **Strict inheritance**, where a subclass `A` exhibits, without any modifications, all the behaviors and properties defined in its parents [58].

Implementation inheritance refers to the existence of a subclass that re-uses code of a parent class [58]. By default the subclass retains all the operations provided by the superclass, yet it has the possibility to override some or all of them, replacing the superclass implementation with its own.

The implementation inheritance, however, violates the encapsulation principle [58]: indeed, it does not prevent other client classes to have a direct access to the methods of the superclass, possibly causing clients to invoke those methods improperly. An encapsulation-preserving alternative to implementation inheritance is called *delegation*. This is the mechanism through which a class can delegate an operation to another class without establishing any inheritance relation.

To summarize, the use of specification inheritance, implementation inheritance, and delegation enables the reuse of portions of code in different manners. On the one hand, the reuse expressed in terms of implementation inheritance and delegation exploits the concept of superclasses. On the other hand, specification inheritance is about interfaces.

B. Related work

Software reusability is a key software engineering principle, as it allows developers to reuse pieces of code that have been previously developed and tested [1]. The research community identified benefits given using general feature provided by object oriented programming languages [60], [61], [61], [62], [63]. In the context of our work we resume previous literature that showed the benefits given by the various reusability mechanisms, but also the potential drawbacks.

Prechelt *et al.* [36] defined two controlled experiments to verify the relation between inheritance and maintenance effort, showing that keeping the inheritance depth small reduces the overall effort spent by developers while maintaining source code. These results are in line with those reported by Daly *et al.* [35], who conducted a series of controlled studies to investigate the impact of inheritance on source code maintainability. Their results indicated that the higher the depth of the inheritance tree of classes, the lower the ability of developers to maintain those classes. Albalooshi [64] corroborated these findings by showing that multiple inheritance in JAVA may result in undesirable effects on the produced software such as increased coupling, lack of cohesion, and increased software complexity; the author concluded that an improper use of inheritance might lead to major negative effects on source code reusability. Later on, Albalooshi and Mahmood [34] evaluated the implementation of the multiple inheritance mechanism on the reusability of three programming languages like JAVA, PYTHON, and C++, showing that the JAVA programming features lead developers to deteriorate source code quality—as measured by means of the Chidamber & Kemerer (CK) metrics. Along the same line of research, Goel and Bathia [37] analyzed whether multilevel inheritance impact on reusability, conducting an empirical experiment on three C++ systems and finding a negative effect of inheritance on maintainability.

While the papers discussed so far assessed inheritance properties by means of controlled studies, other researchers operationalized source code maintainability in terms of quantitative measurements. For instance, Chawla and Nath [31] assessed that the use of inheritance can have beneficial effects on coupling metrics. Similar conclusions were drawn by Chhikara *et al.* [32], who conducted a larger experimentation on the effects of inheritance on multiple CK metrics. Also Vinobha *et al.* [42] found inheritance to be associated to a higher reusability and maintainability. The three papers just discussed somehow contrasted the findings achieved by the researchers adopting controlled experiments to assess the role of inheritance on software quality, indicating the lack of a clear result on its usefulness. However, when considering the studies proposing quantitative assessments, not all of them found inheritance positive. This is the case of researchers that investigated the relation between inheritance metrics and fault-proneness. A number of papers [33], [44], [45], [46], [47] revealed that the high-values of inheritance metrics, which correspond to larger use of the inheritance mechanism, lead source code to be more fault-prone and might therefore be used as defect predictors. Similar conclusions were drawn when considering change-proneness [40], [41], [42], [43].

Last but not least, a relevant research area is the one of code smell detection and refactoring. In this respect, Fowler [24] identified sub-optimal uses of reusability mechanisms and defined code smells like (1) *Refused Bequest*, *i.e.,* subclasses that override most of the methods inherited by the superclass, (2) *Parallel Inheritance*, *i.e.,* inheritance hierarchies that grow too much over time, and (3) *Middle Man*, *i.e.,* classes that excessively use delegation. Ligu *et al.* [21] proposed a dynamic *Refused Bequest* detection approach, while Palomba *et al.* [29] exploited mining software repository techniques for detecting *Parallel Inheritance* instances. Still in terms of code smell research, a number of empirical studies focused on inheritance and delegation. They investigated the diffusion of reusability-specific code smells [26], [22] as well as their impact on defect prediction performance [25], [65], [66].

Limitations of the state of the art. On the basis of the discussion above, we can draw some conclusions. In the first place, inheritance has been the subject of an extensive amount of empirical investigations: most of the results indicate that a widespread usage of inheritance negatively affects software quality, yet there are some studies claiming the opposite. Hence, there is still some ambiguity on the actual role played by inheritance. Perhaps more importantly, most of the previous studies did not distinguish and/or individually analyze the various inheritance mechanisms: as a matter of fact, our knowledge on the effects of specification rather than implementation inheritance for software quality is still limited.

Finally, only a few code smell-related studies have considered delegation: as a consequence, no conclusive insights can be drawn when it turns to this specific reuse mechanism.

Stemming from these observations, our study can be seen as complementary. On the one hand, we aim at investigating how inheritance mechanisms evolve over time, possibly providing new, additional insights into how developers employ inheritance and delegation in practice. On the other hand, our goal is to assess the impact of different reuse mechanisms on source code quality—as measured by the severity of code smells—in an effort of complementing previous findings and possibly better understanding the relation between software reuse and code quality. The closest work to ours is the one by Nasseri *et al.* [49], who partly covered the first of our goals by proposing an evolutionary analysis of the depth of inheritance tree (DIT) metric, showing that it tends to become stable over time. With respect to this work, ours has a largest scope and investigates the impact of reuse mechanisms on code quality.

III. RESEARCH METHODOLOGY

The *goal* of the empirical study was to assess how inheritance and delegation mechanisms evolve over time and how they impact the severity of code smells during software evolution, with the *purpose* of understanding the extent to which reusability mechanisms applied by developers may provide indications on the future quality of source code. The *quality focus* was on the reusability in terms of specification

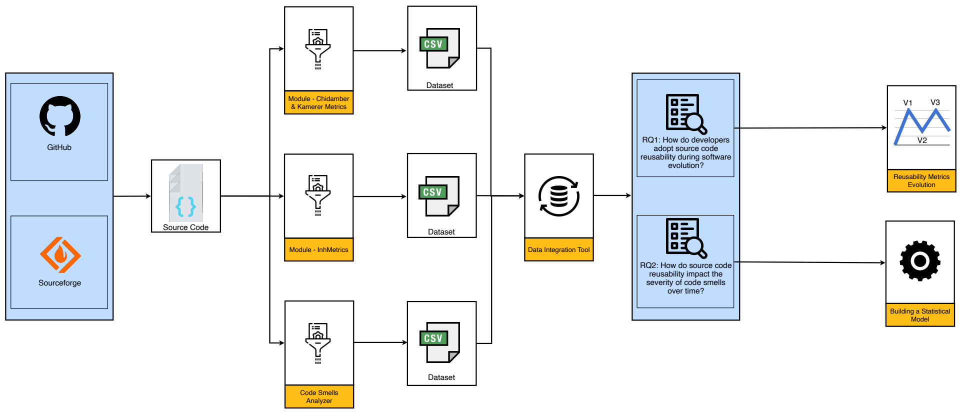

GitHub
Sourceforge
Source Code
Module - Chidamber & Kamerer Metrics
Module - InhMetrics
Code Smells Analyzer
CSV
Dataset
CSV
Dataset
CSV
Dataset
Data Integration Tool
RQ1: How do developers adopt source code reusability during software evolution?
RQ2: How do source code reusability impact the severity of code smells over time?
V1
V2
V3
Reusability Metrics Evolution
Building a Statistical Model

Fig. 1: Overview of the Methodology

inheritance, implementation inheritance, and delegation and their variability within software projects. The *perspective* was of both researchers and practitioners: the former are interested in gathering a deeper understanding of the role of inheritance and delegation for source code quality, while the latter in better monitoring the code quality looking at inheritance and delegation metrics. Our analysis was structured around two main research objectives.

We started by analyzing the evolution of inheritance and delegation. An improved understanding of the evolutionary aspects of those mechanisms would provide us with a clearer overview of how developers adopt them in practice.

RQ1. *How do developers adopt source code reusability mechanisms during software evolution?*

We targeted this research question under different angles, each of them investigating a different code reusability mechanism considered in the scope of this paper. This led to the definition of three sub-research questions:

RQ1$_1$. *How does source code reuse in terms of implementation inheritance vary in software evolution?*

RQ1$_2$. *How does reuse in terms of specification inheritance vary in software evolution?*

RQ1$_3$. *How does reuse in terms of delegation vary in software evolution?*

Once we had assessed the evolution of those mechanisms, we then proceeded with the analysis of how such evolution might impact on source code quality, as measured by the severity of code smells. Hence, we asked:

RQ2. *How do source code reusability mechanisms impact the severity of code smells over time?*

The empirical study had a statistical connotation: as further elaborated later in this section, we approached the research questions by means of statistical tests and models. In terms of reporting, we employed the guidelines by Wohlin *et al.* [67], other than following the *ACM/SIGSOFT Empirical Standards*.[2] The dataset and each script used in order to conduct the analysis of our study are available on the appendix [57]. Figure 1 shows an overview of the methodology followed in order to address our resaerch questions.

A. Context Selection

The context of the empirical study consisted of 15 releases of three JAVA systems such as JHOTDRAW, APACHE ANT, and JEDIT. Table I reports information about each of the considered releases, *i.e.,* we provide the name of the system, the release ID, its KLOC, number of classes, and link to the repository. In addition, we also report the number of uses of implementation inheritance (Inh_impl), the number of uses of specification inheritance (Inh_spec), and the number of delegations applied on each of the releases considered—these values were identified as detailed in Section III-B.

The context selection was driven by one main requirement, namely our willingness to consider a set of projects that was already selected by similar studies investigating the role of reusability mechanisms [32], [34], [42], [47], in an effort of providing results that might complement the observations done in those previous studies and enlarge our knowledge around the impact of inheritance and delegation on software quality. As such, we identified the three projects more often considered by previous research in the field.

B. Extracting Reusability Metrics

The first step of our empirical study was concerned with the computation of reusability metrics and, specifically, the adoption of specification inheritance, implementation inheritance, and delegation. To extract and quantify the presence of these mechanisms, we developed an *ad-hoc* approach—available in

[2]Available at: https://github.com/acmsigsoft/EmpiricalStandards. Given the nature of our study and the currently available empirical standards, we followed the *"General Standard"* and *"Data Science"* definitions and guidelines.

TABLE I: Descriptive statistics of the considered software systems.

System (ver. ID)	Inh_impl	Inh_spec	Delegation	classes	KLOC	Link
JHotDraw 5.2 (v_1)	299	142	57	171	2,275	https://sourceforge.net/projects/jhotdraw/files/JHotDraw/5.2/
JHotDraw 5.3 (v_2)	398	210	35	241	5,054	https://sourceforge.net/projects/jhotdraw/files/JHotDraw/5.3/
JHotDraw 6.0 (v_3)	444	183	22	328	10,285	https://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw60b1/
ANT 1.1 (v_1)	146	13	30	100	5,069	https://github.com/apache/ant/releases/tag/rel%2F1.1
ANT 1.2 (v_2)	188	14	45	166	16,332	https://github.com/apache/ant/releases/tag/rel%2F1.2
ANT 1.3 (v_3)	184	16	46	168	16,841	https://github.com/apache/ant/releases/tag/rel%2F1.3
ANT 1.4 (v_4)	235	53	59	241	39,270	https://github.com/apache/ant/releases/tag/rel%2F1.4
ANT 1.5 (v_5)	327	70	77	397	140,452	https://github.com/apache/ant/releases/tag/rel%2F1.5
ANT 1.6 (v_6)	592	75	81	513	291,882	https://github.com/apache/ant/releases/tag/rel%2F1.6.0
ANT 1.7 (v_7)	702	110	101	734	581,329	https://github.com/apache/ant/releases/tag/rel%2F1.7.0
JEdit 3.2.1 (v_1)	428	136	132	465	83,442	https://sourceforge.net/projects/jedit/files/jedit/3.2.1/
JEdit 4.0 (v_2)	458	134	153	538	116,567	https://sourceforge.net/projects/jedit/files/jedit/4.0/
JEdit 4.1 (v_3)	484	139	183	567	133,491	https://sourceforge.net/projects/jedit/files/jedit/4.1/
JEdit 4.2 (v_4)	506	149	204	701	200,019	https://sourceforge.net/projects/jedit/files/jedit/4.2/
JEdit 4.3 (v_5)	662	159	209	947	494,462	https://sourceforge.net/projects/jedit/files/jedit/4.3/

the online appendix [57]. In the following, we report on the logic and rationale of how we extracted each metric.

Implementation Inheritance (Inh_impl). For a given type (class), the metric measures the reuse expressed in terms of implementation inheritance. To better explain how to quantify such a reuse, let consider the following example. Let `A` be a class having `N` methods; let `B` be the superclass of `A` and let's assume that `B` have just one method, named `foo`. To increase the value of implementation inheritance, one of the methods of `A` must invoke `foo`. Our tool mines the source code of two subsequent releases of a class and checks for cases where the example above appears. It is important to note that, if the class `B` is a subclass of another class `C`, all methods of `C` are also considered in the computation.

Specification Inheritance (Inh_spec). For a given type (class), the metric measures the reuse expressed in terms of specification inheritance. To quantify such a reuse, our devised tool applies the following two steps:

- First, it considers all the interfaces. Suppose that the class `A` inherits from two interfaces `B` and `C`, with `C` extending another interface class `E`. In this case, the sum of the interfaces of `A` is 3.
- Second, the concept of strict inheritance must be considered. In the example discussed in the previous point, the tool considers all the extension points of class `A` and verifies that `A` does not override any methods inherited.

Delegation (Del). For a given type (class), the metric measures the reuse expressed in terms of delegation. Given a class `A`, the tool extracts all the instance variables it declares. These represent the input of a "filtering procedure" that filters out the variables that have a basic type (*e.g.,* `int`, `double`, `String`, `boolean`) or have a non-binding type to the considered project, *i.e.,* the variable is of a type coming from an external library, like the class `BottomGroup` of the `javax.swing` framework. This step allows the tool to consider only the instance variables that class `A` uses to call methods of other classes belonging to the same project. Afterwards, the tool verifies whether these remaining instance variables are actually involved in external calls, *i.e.,* they are actually used to delegate operations.

It is important to remark that we did not rely on existing metrics, like the Depth of Inheritance Tree (DIT) or the Number of Children (NoC) [16], 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 software evolution, *e.g.,* the inclusion of a new method that delegates its operations rather than 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.

*C. **RQ1**. Analyzing the variation of Delegation and Inheritance Metrics over time*

When addressing the first research question, we analyzed the distributions of the three metrics denoting the reuse—*Inh_impl*, *Inh_spec*, and *Del*—with the aim of understanding how these evolve over time. For each subsequent releases R_i and R_j, we applied non-parametric statistical tests to verify whether the distribution of each reusability metric differed between R_i and R_j. First, we applied the Mann-Whitney test [68], which is the non-parametric version of the Wilcoxon rank-sum test: the choice was due to the sample size and the non-normality of the distributions considered [69]. Second, we complemented the analysis with the application of the Cliff's Delta (δ) [70], which quantifies the effect size of the observed differences, hence providing a measure of the extent to which the reusability metrics vary over subsequent releases of the considered applications. It is important to note that, when performing the statistical analyses, we normalized the values of the reusability metrics by LOC: this was done to account for the natural evolution that software systems have in terms of size and obtain a unbiased picture of how the various reusability mechanisms are employed by developers.

The results were intended to be statistically significant at $\alpha = 0.05$. Thus, the null hypotheses tested were:

- $\text{Hn1}_{i,j}$: There is no statistically significant difference between the *Inh_impl* values of version i and *Inh_impl* values of subsequent version j.
- $\text{Hn2}_{i,j}$: There is no statistically significant difference between the *Inh_spec* values of version i and *Inh_spec* values of subsequent version j.

- Hn3$_{i,j}$: There is no statistically significant difference between the *Delegation* values of version i and *Delegation* values of subsequent version j.

where $i, j \in$ {v$_1$, v$_2$,...v$_t$}, when the system has t versions.

We analyzed this aspect using R TOOL.

D. ***RQ2***. *Analyzing the correlation between reusability mechanisms and severity of code smells*

To address **RQ2**, we defined a statistical model relating reusability metrics and other control factors to the increase/decrease of code smell severity. This implied the definition of a number of steps, that we report in the following.

Response Variable. This was represented by the severity of code smells. To compute it, we first selected the actual code smell types subject of our investigation. These were:

- *God Class*: A large class with different responsibilities that monopolizes most of the system's processing [24];
- *Spaghetti Code:* A class without structure that declares long methods without parameters [24].
- *Complex Class*: A class poorly understandable and characterized by a high cyclomatic complexity [24];
- *Class Data Should be Private*: A class exposing its attributes, violating the information hiding principle [24];

The selection of these code smells was driven by two main observations. As discussed in Section II, previous studies have established a relation between the reusability mechanisms and source code complexity (*e.g.,* [34], [64]): as such, we selected code smells that are connected to code complexity in different manners, for instance by detecting badly designed, *i.e., God Class* and *Spaghetti Code*, or too complex classes, *i.e., Complex Class*. We also included the *Class Data Should be Private* code smell: its definition suggests a poor use of code reuse principles and, for this reason, we found interesting to assess the evolution of inheritance and delegation with respect to the erosion of the code connected to this smell.

To detect instances of these code smells in our dataset we relied on a well-known code smell detector named DECOR [71], still widely used by recent literature [72], [73], [74]. It relies on the computation of code metrics that can capture the properties expressed in the definition of the smells. For instance, the *Class Data Should Be Private* smell is identified by DECOR as classes that have a number of variables with visibility `'public'` higher than 10. The detailed detection rules employed as well as the source code of the detector are available in our online appendix [57]. The use of DECOR was motivated by the fact that it represents a good compromise between execution time and detection accuracy—this compromise has been demonstrated multiple times in the past [71], [75], [76]. It is worth noting that the use of DECOR did not allow us to focus our study on other relevant code smell types, namely *Parallel Inheritance*, *Middle Man*, and *Refused Bequest*. On the one hand, the former code smell can be detected only using historical analysis [29] and, for this reason, we could not identify it using code metrics. On the other hand, *Middle Man* and *Refused Bequest* have been detected in the past using dynamic analysis [21] but, unfortunately, these approaches are neither publicly available nor easily reimplementable: to avoid the introduction of any bias due to re-implementation, we decided to exclude these smells from our empirical study. Nonetheless, the investigation of how reusability metrics impact other code smell types, identified through detectors that use historical and dynamic analysis [21], [29], [77], is part of our future work.

Once we had identified code smell instances, we proceeded with the analysis of how their severity evolved over time. To estimate the severity of code smells in the release v_i, we followed the guidelines by Marinescu [78]. In particular, DECOR classifies the presence of a code smell using a heuristic approach that combines multiple metrics: as such, a class is considered smelly *if and only if* a set of conditions are satisfied, where each condition has the form of $metric_i \geq threshold_i$. Hence, the higher the distance between the actual code metric value ($metric_i$) and the fixed threshold ($threshold_i$), the higher the severity of the code smell with respect to that specific metric. Following this reasoning, we measured the severity of code smells as follows: (1) We computed the differences between the actual metric values and the corresponding thresholds used by DECOR [71]; (2) We normalized the obtained differences in the range [0;1] using the min-max strategy [79]; and (3) We computed the final severity score as the mean of the normalized values.

As a final step, for each release pair (v$_i$, v$_{i+1}$) of a project P and for each code smell instance cs_j, we computed the difference between the severity of cs_j in v$_{i+1}$ and the one in v$_i$. If the resulting difference was higher than 0, the severity of cs_j increased: hence, we labeled the event as an *"increase"*; if the difference was negative, then we labeled the case as a *"decrease"*; otherwise, the event was labeled as *"stable"*. These three labels represented the response variable of the four models constructed, *i.e.,* we built one model for each of the code smell considered in the study.

Independent Variables. The factors that we aimed at assessing were the reusability metrics, namely the *Inh_impl*, *Inh_spec*, and *Del* metrics whose definition and computation are reported in Section III-B.

Control Variables. The variability of code smell severity may clearly depend on different aspects different from the reusability metrics we considered as independent variables. We accounted for these aspects when modeling our statistical exercise, defining a number of control variables. We first considered a set of code quality metrics, namely LOC (Lines of Code), WMC (Weighted Methods per Class), RFC (Response for a Class), LCOM (Lack of Cohesion of Methods), CBO (Coupling Between Objects), DIT (Depth of Inheritance Tree), and NoC (Number of Children). We computed these metrics with the METRICS tool.[3] As done for the response variable, we modeled these metrics by considering the difference between their values in the version v$_{i+1}$ and the ones in v$_i$, *i.e.,* for the sake of consistency, we modeled their evolution and the

[3]https://github.com/qxo/eclipse-metrics-plugin

effect it has on the evolution of code smell severity. It is worth remarking that these control variables are not used by DECOR for the detection of code smells: in other words, there is no dependency between independent and dependent variables—otherwise, this would have caused possible biases when interpreting the statistical results [80].

On the one hand, these metrics have been considered effective to assess source code quality [81], [82]. On the other hand, they estimate a variety of code quality aspects, such as size, code complexity, coupling, cohesion, and propensity to reuse—hence, perfectly fitting our goal of controlling for code quality when evaluating the evolution of code smell severity. It is worth remarking the use of DIT and NoC. These are metrics that are clearly connected to the reusability metrics we considered as independent variables. Nonetheless, we considered them with the intent of comparing their statistical power with respect to the adoption mechanisms estimated by our metrics: in other words, their employment allowed us to assess the importance of the size of the inheritance tree and the number of children in the inheritance hierarchy with respect to the general usage of inheritance and delegation—note that we assessed the presence of possible multi-collinearity due to these related metrics when performing the statistical modeling, as further described in the remainder of this section.

Running the Statistical Model. Given the nature of our categorical response variable, *i.e.,* the categories *"decrease"*, *"stable"*, and *"increase"*, we used a Multinomial Log-Linear model [83] to study the severity of the four code smells considered. This is a classification method that is applied when the dependent variable is nominal and composed of more than two levels. We built our models using R, exploiting the function `multinom` available in the package `nnet`,[4] *i.e.,* the models were fit via neural networks. When constructing the statistical models, we took into account the problem of multi-collinearity, which appears when two or more independent variables are highly correlated and can be predicted one from the other, possibly biasing the interpretation of the results. In the context of our work, we applied the guidelines proposed by Allison [84], who described how to control a model for multi-collinearity and when to ignore it. As a result, we did not remove any of the variables. This was because the standard errors of the independent variables were narrow enough not to negatively influence the interpretability of the model: in our case, for all models they were lower than 0.9——note that standard errors must be ≤ 2.5 to produce a sufficiently narrow 95% prediction interval [85].

When interpreting the results of the model, the multinom coefficients are relative to a reference category and indicate how the independent variables change the chances of the dependent variable being affected with respect to the reference category. We set such a category to *"stable"*: in this way, we could understand how the different independent variables vary, in either positive or negative manner, the likelihood of the code smell severity being stable over two releases. As an example, a negative coefficient for an independent variable of the model built when analyzing the decrease of code smell severity would suggest that for one unit increase of that variable, the chances of variation of the response variable would be increased of the amount indicated by the coefficient.

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

E. Threats to Validity

In this subsection, we discuss possible threats that could have affected our results and how we mitigated them.

Construct Validity. Threats to *construct validity* concern with the relationship between theory and observation. The main discussion point in this respect is related to the dataset exploited. We are aware that the projects selected could have influenced the extent of the analysis, yet we relied on projects that have been previously used in similar experimentations with the goal of extracting results that might have been as comparable as possible. Future investigations will extend our understanding on the evolutionary aspects of reusability mechanisms. An additional threat concerns with the data collection procedures: these were either based on well-established tools, *e.g.,* METRICS, or classical definitions of metrics, *e.g.,* the computation of reusability metrics. In any case, we made all scripts and data publicly available for the sake of verifiability.

As for the identification of code smells, we relied on DECOR [71], which is a state-of-the-art detector [76]. Its accuracy might have influenced the quality of the information reported in the dataset: while we recognize this limitation, we also point out that the choice of this detector was based on the compromise between quality and performance it ensures [71], [75], [76]. In our future research agenda, we plan to assess the impact of false positives/negatives on the achieved results.

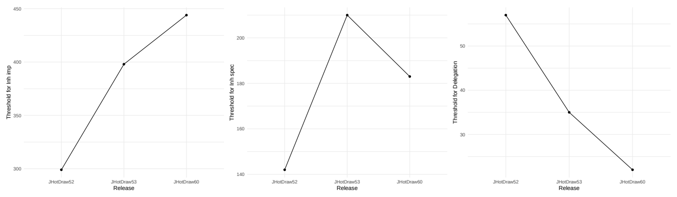
Internal Validity. Threats to *internal validity* concern factors that might have influenced our results. In the context of **RQ2**, we defined a number of control variables that could estimate, in a more appropriately manner, the effect of reusability metrics on the variation of code smell severity. The selection of those metrics was based on the analysis of the state of the art, namely on the identification of the metrics that have been previously connected to the evolution of code smells.

Conclusion Validity. A major threat to the conclusions drawn is related to the statistical methods employed. In **RQ1**, we used well-known tests widely exploited by research community, *i.e.,* Mann-Whitney and Cliff's Delta. Before using them we verified the normality of the data, which is the main requirement leading to their use. As for **RQ2**, the selection of the Multinomial Logistic Linear statistical approach [83] was driven by the fact that our response variable was categorical and composed of three levels. By definition, this statistical approach is able to handle multiclass problems with categorical and continuous independent variables, therefore fitting the problem of interest. While designing the model, we also controlled for possible multi-collinearity, hence avoiding bias in the interpretation of the results [86].

External validity. Threats in this category mainly concern with the generalization of results. We analyzed 15 releases of three open-source software systems coming from different application domains and having different characteristics (size, programming languages, number of classes, etc.). Of course, we cannot claim the generalizability of the findings to other systems; our future research agenda includes the extension of the study with more different set of systems.

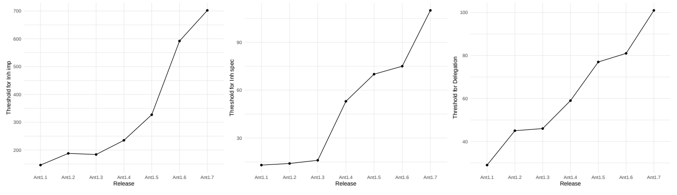
IV. Analysis of the Results

The results of the study are presented in the following.



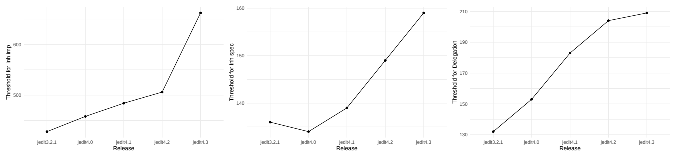
(a) (b) (c)

Fig. 2: How Inh_impl (a), Inh_spec (b), and Delegation (c) change across the considered versions of JHotDraw.



(a) (b) (c)

Fig. 3: How Inh_impl (a), Inh_spec (b), and Delegation (c) change across the considered versions of Ant.



(a) (b) (c)

Fig. 4: How Inh_impl (a), Inh_spec (b), and Delegation (c) change across the considered versions of JEdit.

A. ***RQ1*** *- How do developers adopt source code reusability mechanisms during software evolution?*

When addressing the evolution of reusability mechanisms, we first analyzed how implementation inheritance, specification inheritance, and delegation evolved over the considered projects in absolute terms and whether the difference between different releases is statistically significant. Due to space limitation, the detailed table reporting the results of Mann-Whitney and Cliff's Delta is in the online appendix [57]. Figures 2 to 4 depict the evolution of these mechanisms over the releases of the three considered projects. Looking at those figures, we found similar trends.

The adoption of implementation inheritance follows an increasing trend in three projects. While this seems to suggest that developers use more and more frequently this type of reusability mechanism, we also pointed out the case of JEdit. In this case, we observed a notable growth between versions `4.2` and `4.3`. To better understand the reason behind this finding, we manually dived into the mailing list of the project, in an effort of understanding whether the developers themselves commented on this aspect in a certain moment in time. This eventually happened on April 10^{th}, 2007. As announced in the *"jEdit-announce"* mailing list,[5] the version `4.3` was substantially revised, not only to include new features and bug fixing operations, but also to provide new APIs: these latter modifications have let developers apply consistent editing/refactoring operations of the source code, which implied the improvement of source code design and a higher adoption of inheritance mechanisms. While additional qualitative analyses would be useful to understand the specific reasons why developers increased the use of implementation inheritance while preparing the version `4.3` of the project, our findings suggest that the higher adoption was indeed due to the developer's willingness to provide other reuse mechanisms such as APIs. Nonetheless, when considering the results from a statistical standpoint, we could not identify significant variations—both the Mann-Whitney and Cliff's Delta tests did not reveal a relevant change in implementation inheritance adoption when passing from version `4.2` to version `4.3`. This is likely due to the effect of size, *i.e.,* the ratio of implementation inheritance and size was similar in both releases, even though the reuse was increased in absolute terms. A similar conclusion can be drawn when looking at the statistical tests of the other systems: there were no cases of statistically significant changes between two subsequent releases, meaning that, overall, the use of implementation inheritance does not vary too much over the evolution history when controlled for size.

Key findings for RQ1$_1$. The use of implementation inheritance increases over time, even if not in a statistically significant manner. In JEdit, the increased adoption was likely due to the developer's willingness of improving APIs.

Turning to the specification inheritance, from Figures 2-4 we could observe a similar trend with respect to the one discussed above. The adoption tends to increase over time in absolute terms, but without any statistically significant change. Hence, we can claim that the evolution is basically stable over time. A slight exception was represented by JHotDraw, where we observed a decrease adoption when passing from release `5.3` to `6`. Analyzing this case further, we could find that the JHotDraw team decided to apply substantial changes to the system, likely affecting the reusability mechanisms previously used and preferring other strategies (*e.g.,* implementation inheritance) over specification inheritance.

[5]Thread in the *"jEdit-announce"* mailing list: https://sourceforge.net/p/jedit/mailman/jedit-announce/?viewmonth=200710.

🔍 **Key findings for RQ1$_2$.** The adoption of specification inheritance is stable over time. The only exception was JHotDraw, that preferred to use different reusability mechanisms while defining a new milestone.

Finally, the reuse in terms of delegation follows a similar increasing trend in Ant and JEdit, with the exception of JHotDraw. When analyzing the evolution of the latter system more closely, we could not really derive a clear motivation behind the decreasing trend. This might potentially be connected to a progressive reluctance of designing source code for delegation, however it is important to note that, also in this case, the statistical results did not reveal significant changes. This implies that the differences observed in absolute terms are balanced by the increasing number of lines of code.

🔍 **Key findings for RQ1$_3$.** The adoption of delegations increases over time, but not in a statistically significant way. The exception is the one of JHotDraw, where the trend observed suggests a progressive reluctance to this mechanism which might be worth of studying in the future.

*B. **RQ2** - How do source code reusability mechanisms impact the severity of code smells over time?*

Table II shows the results of the statistical models built for each of the smells considered in Section III-D. In the first place, it is important to recognize that the decrease of the CK metric values—considered as control variables in our models—correlate well with the decreasing of code smell intensity. This phenomenon was somehow expected since code quality metrics have always been used as variables to predict and monitor code smells [71], [87]: as such, we can confirm the impact of these metrics on the variability of code smells.

Starting from *Spaghetti Code*, we could observe that delegation and the specification inheritance correlate well with the variability of code smell intensity. When they decrease, we found a positive correlation with the increase of the intensity of code smells. This result seems to perfectly fit the nature of this smell. One of the causes leading to the emergence of this smell and its degradation is indeed the inexperience with object-oriented design technologies: our results suggest that a decrease in the use of inheritance and implementation, which are widely considered as relevant for a successful application of object-orientation [24], leads to increase the chances of this smell being harmful. As for the implementation inheritance, we found that its decrease causes instability, *i.e.,* we found negative estimates when considering both the model build to study the increase and decrease of code smell intensity. From a practical perspective, this means that the specific uses of this mechanism may lead to different results.

Moving to *God Class*, we noticed that the increase of delegations may contribute to the decrease of intensity, while both the inheritance metrics create instability. If a class affected by this smell increases the usage of the delegation mechanism, this means that the overall number of responsibilities it manages is reduced: this may explain the reason behind the severity reduction. At the same time, an increasing use of inheritance implies exactly the opposite, with the *God Class* including more and more methods coming from its superclasses. As such, our findings suggest that an appropriate use of delegation might result in an improvement of code quality with respect to the emergence of *God Class* instances: this is also demonstrated by the way some state-of-the-art refactoring tools actually deal with this code smell: as an example, JDeodorant [88] realizes *Extract Class* refactoring operations by means of delegations, namely by moving methods from the *God Class* to other classes, letting the original class rely on those methods by means of delegation.

As for *Class Data Should Be Private*, we noticed expect for delegation, the lower the presence of inheritance, the higher the chances of the code smell severity being decreased. This result may be concerned with the encapsulation principle that characterizes this code smell. As stated by Gamma *et al.* [6] *"inheritance mechanism often breaks encapsulation, given that inheritance exposes a subclass to the details of its parent's implementation"*. A lower use of these reusability mechanisms naturally makes classes less exposed, hence reducing the risks connected to the presence of a *Class Data Should Be Private*.

Considering *Complex Class*, we noticed that these metrics can help decreasing the variability of the smell. This is, likely, the most interesting outcome of our analysis. As discussed in Section II, previous studies [34], [64], [89] have established a relation between the use of reusability mechanisms and source code complexity: with respect to those papers, our findings suggest that keeping inheritance and delegation under control may lead developers to reduce the risks of increasing complexity. In this respect, our empirical study provides an additional take on the role of reusability for software maintainability.

Finally, looking at the control factors related to inheritance, *i.e.,* DIT and NOC, we noticed that they lead to higher instability compared to our independent variables, possibly indicating that the pure adoption of inheritance and delegation might be used to better monitoring the variability of code smell severity, correlating better the phenomenon.

🔍 **Key findings for RQ2.** In most cases, delegation and inheritance metrics positively correlate to the decrease of the code smell severity. Nonetheless, in some cases, their presence causes instability.

V. Discussion and Implications

The results to our research questions targeted in this study allow us to provide a number of implications for researchers and practitioners, which we discuss in the following.

- **On the use of reusability mechanisms.** In the first part of the study, we could analyze the evolution of reusability metrics. From that, we could delineate similar trends. Developers tend to increase the use of inheritance and delegation over time: this may suggest that code reuse is indeed a relevant matter for developers, which may be worth of further investigations. Nonetheless, such an increase does

TABLE II: Results of the statistical models for each smell considered in the analysis. The table shows the value of the estimates and the significance through the asterisk.

	Spaghetti Code		**God Class**		**Class Data Should Be Private**		**Complex Class**	
Variables	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**	**Decrease**	**Increase**
Delegation	0.011***	-0.358***	1.054***	-1.363***	0.016***	0.330***	0.011***	-0.358***
Implementation Inheritance	-0.094***	-0.061***	0.048***	0.003***	-0.016***	-0.008***	-0.094***	-0.061***
Specification Inheritance	0.002***	-0.023***	0.028***	0.036***	0.022***	-0.010***	0.002***	-0.023***
DIT	0.060***	0.180***	-0.274***	0.108***	-0.133***	0.004***	0.060***	0.180***
NOC	-0.009***	0.007***	-0.053***	0.002***	0.032***	0.004***	-0.009***	0.007***
LOC	-0.000	-0.000	-0.000	-0.000	0.000**	-0.000	-0.000	-0.000
LCOM	0.910***	-0.101***	-0.177***	-3.218***	0.408***	0.230***	0.910***	-0.101***
WMC	0.0005	-0.0004	-0.002	-0.001	0.001	-0.001	0.0005	-0.0004
CBO	-0.327***	0.201***	-0.679***	0.870***	0.023***	-0.453***	-0.327***	0.201***
RFC	0.001	0.003**	0.008***	0.003	0.002	0.005***	0.001	0.003**
$*p < 0.1; **p < 0.05; ***p < 0.01$								

not turn to be statistically significant: in other terms, the increase grows along with the size of the project, hence suggesting a linear evolution. This recalls again the need for additional analyses into the specific motivations that developers have when adopting reusability mechanisms. We believe that more developer-oriented studies would increase the scientific knowledge on how programming language features and paradigms are used in practice as well as the advantages perceived by developers.

Implication 1. The research community should invest some effort in understanding the developer's perspective on the use of abstraction mechanisms.

On the relation with previous studies. Our findings partially contradict most of the investigations previously done on the matter. As already mentioned, a number of empirical investigations established a negative relation between the adoption of reusability metrics and code complexity [34], [37], [64]. However, the evolutionary nature of our study revealed something different: in most cases, the correct adoption of inheritance and delegation positively related to the decrease of code smell severity. While we are aware that further analyses would be needed to confirm our conclusions on a larger scale, the results obtained so far allow practitioners to (re-)consider reusability as a core mechanism for evolving software systems. Similarly, our findings suggest that broader and perhaps more conclusive indications into the usefulness of code metrics might be obtained by looking at how these metrics evolve over time and how they impact source code quality.

Implication 2. Inheritance and delegation mechanisms can be used by practitioners as instruments to decrease the severity of code smells. Researchers might be interested in devising novel semi-automated tools that might support practitioners in employing these abstraction mechanisms.

Implication 3. An improved understanding of the role of code metrics for source code quality can be obtained by looking at their evolution and how these impact code quality attributes. As such, the research community might consider novel empirical investigations aiming at characterizing the long-term, evolutionary impact of code metrics.

Reusability: The bright and the dark side. As a final discussion point, it is worth remarking that in some cases we observed that inheritance and delegation may lead to instability, *i.e.,* they simultaneously increase and decrease the code smell severity. This sheds light on an additional view of the problem, which the research community would be interested to face. In particular, our findings somehow suggest that there exist a few cases where the specific adoptions of reusability mechanisms can have different effects. As a matter of fact, a little knowledge is available on when inheritance and delegation are used in a way that it can be fully considered a pattern and when, instead, this becomes an *"abuse"* that may lead to undesired effects.[6] It is very likely that the difference between a proper use and an abuse is not concerned with the quantitative usage of delegation and inheritance but on their logic. Part of our future research agenda will be focused on understanding this matter, by performing qualitative analyses that may better describe the bright and the dark side of reusability.

Implication 4. Researchers should consider the definition of qualitative or mixed-method studies through which understand what are the boundaries between a correct and incorrect use of inheritance and delegation mechanisms.

VI. Conclusions

In this paper, we investigated the evolution of reusability mechanisms and their impact on the variation of code smell severity. Our results revealed that the adoption of inheritance and delegation increases over time, even though not in a statistically significant manner. At the same time, such an evolution does have an impact on code smells: our statistical approach found inheritance and delegation metrics to significantly impact the likelihood of the severity of the code smells considered to vary. Often, such a variation can be considered positive, meaning that a proper adoption of inheritance and delegation reduces the severity of code smells.

Our future research agenda includes a larger scale experimentation that could corroborate our initial findings. At the same time, we aim at assessing the role of the considered aspects from the developer's perspective, in an effort of understanding their opinions and adoption on inheritance and delegation practices in their daily development activities.

[6]https://softwareengineering.stackexchange.com/questions/12439/code-smell-inheritance-abuse

Acknowledgment

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] J. M. Bieman and J. X. Zhao, "Reuse through inheritance: A quantitative study of c++ software," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI, pp. 47–52, 1995.

[2] N. Soundarajan and S. Fridella, "Inheritance: From code reuse to reasoning reuse," in *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*. IEEE, 1998, pp. 206–215.

[3] S. Singh, S. Singh, and G. Singh, "Reusability of the software," *International journal of computer applications*, vol. 7, no. 14, pp. 38–41, 2010.

[4] B. M. Lange and T. G. Moher, "Some strategies of reuse in an object-oriented programming environment," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1989, pp. 69–73.

[5] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177–1193, 2009.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.

[7] A. Zaimi, A. Ampatzoglou, N. Triantafyllidou, A. Chatzigeorgiou, A. Mavridis, T. Chaikalis, I. Deligiannis, P. Sfetsos, and I. Stamelos, "An empirical study on the reuse of third-party libraries in open-source software development," in *Proceedings of the 7th Balkan Conference on Informatics Conference*, 2015, pp. 1–8.

[8] P. Salza, F. Palomba, D. Di Nucci, A. De Lucia, and F. Ferrucci, "Third-party libraries in mobile apps," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2341–2377, 2020.

[9] I. Sommerville, "Software engineering 9th edition," *ISBN-10*, vol. 137035152, p. 18, 2011.

[10] Y. Yokote, F. Teraoka, and M. Tokoro, "A reflective architecture for an object-oriented distributed operating system," in *Proceedings of the 1989 European Conference an Object Oriented Programming*, 1989, pp. 89–106.

[11] B. Meyer, "Reusability: The case for object-oriented design," *IEEE software*, vol. 4, no. 2, p. 50, 1987.

[12] N. Kumari and A. Saha, "Effect of refactoring on software quality," in *Proc. Conf Softw. Main t.*, 2014, pp. 37–46.

[13] A. Sane and R. Campbell, "Object-oriented state machines: Subclassing, composition, delegation, and genericity," *ACM Sigplan Notices*, vol. 30, no. 10, pp. 17–32, 1995.

[14] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of java software," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 6, pp. 10–16, 1998.

[15] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] K. M. Breesam, "Metrics for object-oriented design focusing on class inheritance metrics," in *2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'07)*. IEEE, 2007, pp. 231–237.

[18] S. Mal and K. Rajnish, "New quality inheritance metrics for object-oriented design," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 6, pp. 185–200, 2013.

[19] E. Kupiainen, M. V. Mäntylä, and J. Itkonen, "Using metrics in agile and lean software development–a systematic literature review of industrial studies," *Information and Software Technology*, vol. 62, pp. 143–163, 2015.

[20] D. Connolly Bree and M. Ó. Cinnéide, "Inheritance versus delegation: which is more energy efficient?" in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 323–329.

[21] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 392–395.

[22] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[23] H. Kegel and F. Steimann, "Systematically refactoring inheritance to delegation in java," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 431–440.

[24] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[25] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 23–31.

[26] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[27] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[28] R. Marticorena, C. López, and Y. Crespo, "Parallel inheritance hierarchy: Detection from a static view of the system," in *6th International Workshop on Object Oriented Reenginering (WOOR), Glasgow, UK*. Citeseer, 2005, p. 6.

[29] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[30] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings*. IEEE, 2002, pp. 97–106.

[31] S. Chawla and R. Nath, "Evaluating inheritance and coupling metrics," *International Journal of Engineering Trends and Technology (IJETT)*, vol. 4, no. 7, pp. 2903–2908, 2013.

[32] A. Chhikara, R. Chhillar, and S. Khatri, "Evaluating the impact of different types of inheritance on the object oriented software metrics," *International Journal of Enterprise Computing and Business Systems*, vol. 1, no. 2, pp. 1–7, 2011.

[33] F. B. e Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Proceedings of the 3rd international software metrics symposium*. IEEE, 1996, pp. 90–99.

[34] F. Albalooshi and A. Mahmood, "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*, vol. 8, no. 6, pp. 109–116, 2017.

[35] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Evaluating inheritance depth on the maintainability of object-oriented software," *Empirical Software Engineering*, vol. 1, no. 2, pp. 109–132, 1996.

[36] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy, "A controlled experiment on inheritance depth as a cost factor for code maintenance," *Journal of Systems and Software*, vol. 65, no. 2, pp. 115 – 126, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121202000535

[37] B. M. Goel and P. K. Bhatia, "Analysis of reusability of object-oriented systems using object-oriented metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1–5, 2013.

[38] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of gof design patterns on stability: a case study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 781–802, 2015.

[39] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, vol. 58, no. 3, pp. 261–269, 2001.

[40] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, A. Zaidman, and F. Ferrucci, "Improving change prediction models with code smell-related information," *Empirical Software Engineering*, vol. 25, no. 1, pp. 49–95, 2020.

[41] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen, "The ability of object-oriented metrics to predict change-proneness: a meta-analysis," *Empirical software engineering*, vol. 17, no. 3, pp. 200–242, 2012.

[42] A. Vinobha, C. Babu *et al.*, "Evaluation of reusability in aspect oriented software using inheritance metrics," in *2014 IEEE international conference on advanced communications, control and computing technologies*. IEEE, 2014, pp. 1715–1722.

[43] Y. Zhou, H. Leung, and B. Xu, "Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 607–623, 2009.

[44] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[45] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2017.

[46] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using oo metrics. an industrial case study," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE, 2002, pp. 99–107.

[47] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software quality journal*, vol. 18, no. 1, pp. 3–35, 2010.

[48] T. Mens and S. Demeyer, "Future trends in software evolution metrics," in *Proceedings of the 4th international workshop on Principles of software evolution*, 2001, pp. 83–86.

[49] E. Nasseri, S. Counsell, and M. Shepperd, "An empirical study of evolution of inheritance in java oss," in *19th Australian Conference on Software Engineering (ASWEC 2008)*. IEEE, 2008, pp. 269–278.

[50] I. D. Craig, "Inheritance and delegation," in *Object-Oriented Programming Languages: Interpretation*. Springer, 2007, pp. 83–128.

[51] E. Tempero, H. Y. Yang, and J. Noble, "What programmers do with inheritance in java," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 577–601.

[52] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[53] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 220–231.

[54] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2016.

[55] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[56] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020.

[57] omitted, "Web Appendix of the paper," https://figshare.com/s/7c046b69d8f7ea75c0c8, online.

[58] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java*, 3rd ed. USA: Prentice Hall Press, 2009.

[59] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1811–1841, 1994.

[60] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "A large-scale empirical study of java language feature usage," 2013.

[61] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, "An empirical study of refactorings and technical debt in machine learning systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 238–250.

[62] R. Khatchadourian and H. Masuhara, "Proactive empirical assessment of new language feature adoption via automated refactoring: The case of java 8 default methods," *The Art, Science, and Engineering of Programming*, vol. 2, no. 3, pp. 6–1, 2018.

[63] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and B. Ray, "An empirical study on the use and misuse of java 8 streams." in *FASE*, 2020, pp. 97–118.

[64] F. Albalooshi, "The metrics of multiple inheritance and the reusability of code–java and c++," *Journal of Advances in Mathematics and Computer Science*, pp. 1–12, 2016.

[65] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 194–218, 2017.

[66] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki, "An extensive study on smell-aware bug localization," *Journal of Systems and Software*, vol. 178, p. 110986, 2021.

[67] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[68] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[69] W. J. Conover, *Practical Non parametric Statistics*, 3rd ed. Wiley India Pvt. Limited, 2006.

[70] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[71] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[72] M. De Stefano, F. Pecorelli, F. Palomba, and A. De Lucia, "Comparing within-and cross-project machine learning algorithms for code smell detection," in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, 2021, pp. 1–6.

[73] S. Jain and A. Saha, "Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection," *Science of Computer Programming*, vol. 212, p. 102713, 2021.

[74] C. Hasantha, "A systematic review of code smell detection approaches," *Journal of Advancement in Software Engineering and Testing*, vol. 4, no. 1, 2021.

[75] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[76] E. V. de Paulo Sobrinho, A. De Lucia, and M. de Almeida Maia, "A systematic literature review on bad smells—5 w's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.

[77] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 31–34.

[78] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9–1, 2012.

[79] S. Patro and K. K. Sahu, "Normalization: A preprocessing stage," *arXiv preprint arXiv:1503.06462*, 2015.

[80] J. S. Almeida, "Predictive non-linear modeling of complex data by artificial neural networks," *Current opinion in biotechnology*, vol. 13, no. 1, pp. 72–76, 2002.

[81] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo, "An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite," *Empirical Software Engineering*, vol. 10, no. 1, pp. 81–104, 2005.

[82] D. A. Tamburri, F. Palomba, and R. Kazman, "Success and failure in software engineering: A followup systematic literature review," *IEEE Transactions on Engineering Management*, 2020.

[83] H. Theil, "A multinomial extension of the linear logit model," *International economic review*, vol. 10, no. 3, pp. 251–259, 1969.

[84] P. Allison, "When can you safely ignore multicollinearity," *Statistical horizons*, vol. 5, no. 1, pp. 1–2, 2012.

[85] D. N. McCloskey and S. T. Ziliak, "The standard error of regressions," *Journal of economic literature*, vol. 34, no. 1, pp. 97–114, 1996.

[86] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Quality & quantity*, vol. 41, no. 5, pp. 673–690, 2007.

[87] P. Tourani, B. Adams, and A. Serebrenik, "Code of conduct in open source projects," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 24–33.

[88] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[89] G. Seront, M. Lopez, V. Paulus, and N. Habra, "On the relationship between cyclomatic complexity and the degree of object orientation," in *Proc. of QAOOSE Workshop, ECOOP, Glasgow*, 2005, pp. 109–117.