Blessing or Curse? Investigating Test Code Maintenance through Inheritance and Interface

1st Dong Jae Kim
DePaul University
Chicago, Illinois, United States
djaekim086@gmail.com

2nd Tse-Hsun (Peter) Chen
Concordia University
Montreal, Quebec, Canada
peterc@encs.concordia.ca

2024 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3503-9568-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICSME58944.2024.00031

***Abstract*—Since the advent of object-oriented programming languages, inheritance and interface have been fundamental concepts in software design principles, facilitating code reuse and extensibility in software systems. Despite their potential benefits, inheritance, and interface remain underexplored in software test code. Currently, there is a limited established standard for how inheritance and interface may impact test reusability, extensibility, and maintainability, nor for understanding the potential design challenges that may arise from improper usage. Addressing these research gaps is crucial for optimizing test maintainability and software quality. In this paper, we address this gap in empirical research by conducting the first comprehensive study on the prevalence and maintenance of inheritance and interface within test code. To accomplish this goal, we use RefactoringMiner's AST differencing API to detect inheritance and interface changes in modified test classes within the software evolution commit history by studying 12 open-source Java systems. Our key findings are as follows: (1) Among the 23,651 commits that modify test classes, 4,429 (18%) involve changes to their inheritance relationships, whereas a significantly smaller subset, 233 (1%), pertain to changes in their interface relationships. (2) 59.5% of test classes already incorporate inheritance when initially created, while 40% of test classes incorporate interfaces. (3) We manually categorized the use of inheritance and interfaces and their impact on test maintainability to provide valuable insights for developers. In summary, this study takes the first step in exploring how the use of inheritance and interfaces in test code affects software reusability and extensibility, offering meaningful insights for both developers and researchers**

I. Introduction

In the rapidly evolving software industry, customers increasingly demand new features alongside reliable and high-quality products. To meet these demands, code reusability has become crucial for enabling developers to extend functionality, eliminate redundancies, and enhance maintainability.

To achieve reusability, object-oriented programming languages like Java have introduced polymorphism, allowing objects to be instances of multiple classes [14]. Polymorphism is primarily achieved through subtyping using inheritance and interfaces. Inheritance allows a subclass to inherit properties and behaviors from its superclass [13], while interfaces ensure adherence to specified contracts defined in the superclass. These mechanisms enable developers to write code once and apply it across various contexts, thereby enhancing productivity in software development.

Despite the advantage of reusability, the use of inheritance has raised concerns regarding potential technical debt in the industry. Inheritance can create tight coupling between classes [21], reducing flexibility [23]. Empirical studies in academia have shown a strong correlation between inheritance and increased defect proneness in software systems [8, 11, 27, 18]. As a result, the inheritance-related metric is often used in defect prediction and vulnerability detection models [32, 4, 26, 3]. However, utilization of inheritance and interface remain underexplored in software test code, despite the significant time (25%) developers spend on testing [25]. Initial consensus indicates that developers should caution against using inheritance in test code practices [10, 24, 6]. Yet, there are limited insights into how inheritance and interface impact test reusability, extensibility, and maintainability, nor for understanding the potential design challenges that may arise from improper usage. Addressing these research gaps is crucial for optimizing test maintainability and software quality. In this study, we take the initial step toward understanding test reusability and extensibility from the perspective of inheritance and interfaces in test code within open-source software systems. Our approach involves mining code changes by using the state-of-the-art API differencing tool, RefactoringMiner 2.0 [29], such as additions, removals, and replacements of `extends` and `implements` inheritance and interface. We analyze these developer-driven changes to assess the impact of inheritance and interface usage on test maintainability in real software development by addressing the following research questions:

RQ1: Do inheritance and interface play prominent roles in the initial development of test code and how does it compare to source code? At the beginning of test evolution, 40% of the test classes were initially committed into the repository without inheritance or interfaces but were added later. In comparison, the source code is more stable, with only 16% of classes showing changes in inheritance or interfaces. This indicates that inheritance and interfaces play a more significant role in the maintenance and evolution of test code compared to source code. Despite controversial ideas behind inheritance, its use may be prevalent in the test code.

RQ2: How does test code reusability change with inheritance and interface modifications? Inheritance modifications significantly impact test reusability; adding an `extends` in-

2576-3148/24/$31.00 ©2024 IEEE
DOI 10.1109/ICSME58944.2024.00031

creases the number of inherited test methods (average increase of 8), while removal reduces it (average decrease of 10). In contrast, changes to the interface have a minimal effect on test reusability. Hence, developers should carefully manage inheritance changes, as changes to inheritance can significantly modify test reusability.

RQ3: How do inheritance and interfaces contribute to test design and maintainability? We conduct a manual analysis on developer-driven inheritance and interface modification to determine how test inheritance and interface may impact test code design. We develop a manual categorization of use cases in which developers may utilize inheritance and interface usages to achieve test reusability and extensibility.

RQ4: What potential issues arise from using inheritance in test maintainability? We document design issues caused by inheritance that impact test execution and comprehension, discussing their manifestations in software systems. Finally, we have made our replication package, containing all datasets and code, available [2].

Paper organization. Section II discusses background and motivations. Section III discusses our experimental design. Section IV presents our quantitative analysis results, and Section V presents our manual categorization results. Section VI summarizes the implication of our findings. Section VII discusses related work. Section VIII discusses threats to validity. Section IX concludes the paper.

II. Promise and Perils of Inheritance and Interfaces in Test Code

Object-oriented programming languages like Java allow developers to declare keywords like `extends` and `implements` to achieve reusability across classes. For instance, with `A extends B`, class A gains access to all the concrete implementations of methods, attributes, and fields in class B. Conversely, with `A implements B`, interface B acts as a contract for a class, specifying the required methods that all implementing classes must define without dictating the actual code for those methods. Consequently, implementing classes must provide new definitions for these methods to determine their unique behavior, facilitating code reuse.

We conjecture that test developers commonly adopt inheritance and interface usages within the test code. Hence, we investigated the prevalence of inheritance and interface usage in test code by mining open-source projects on GitHub. We began with the Java-med dataset [1], consisting of the top 1,000 most-starred Java systems from GitHub. From the list of java classes in the repository, we checked (i) whether the test class contains inheritance or interface usage and (ii) whether a class is a test class. To determine whether a test class contains inheritance or interface usage, we searched for the `extends` and `implements` keywords in the class modifiers. To check if a class is a test class, we examined if the filename's `Prefix/Suffix` contains the *"T/test"* keyword. We omitted systems that do not have test classes. As a result, among the 1,000 repositories initially considered, we identified 583 repositories that contained test classes. Within these repositories, we found that 42% have at least one inheritance usage, and 25% have at least one interface usage. These findings indicate inheritance and interface usage are prevalent in test code. Based on this analysis, we believe that inheritance and interfaces may play a significant role in test code maintenance.

III. Experiment Design and Preliminary Analysis

In this section, we first introduce the systems studied. Then, we discuss our experimental design for tracking inheritance and interface changes in software evolution to understand their prevalence in the test code. Figure 1 shows the overview of our experiment design. In summary, we first detect all commits that undergo inheritance and interface changes from the beginning of software history until the end of 2021 (Section III-A). Then, we perform a version-by-version comparison to map the inheritance and interface changes for the modified Java classes to get the complete evolution of inheritance and interface changes (Section III-B).

TABLE I: An overview of the studied systems, and the changes in test and source code from the beginning of history to the end of 2021.

	Test		Src	
Systems	Test LOC	Test Class	Src LOC	Src Class
cayenne	32K→33K	382→441	112K→260K	2K→4K
commons-collections	2K→37K	5→230	2K→30K	5→360
cucumber-jvm	26→24K	1→187	26→26K	1→598
cxf	70K→228K	618→2K	182K→457K	2K→6K
httpcomponents-client	2K→70K	11→618	69K→70K	618→686
iotdb	20→70K	1→392	4K→241K	42→2K
jclouds	20→166K	1→2K	20→233K	1→4K
kylin	20→9K	1→129	20→58K	1→593
maven	20→19K	1→196	20→73K	1→841
ranger	102→8K	4→49	37K→223K	319→1K
ratis	313→3K	7→18	313→53K	7→608
wicket	313→56K	7→639	313→160K	7→3K
Total	106K→723K	1K→6.9K	406K→1884K	1K→23.6K

Studied Systems. Table I provides an overview of the systems we studied. To select these systems, we began with the initial 583 repositories identified in Section II having at least one test class. From these repositories, we refined our selection based on specific criteria to ensure the quality and relevance of the systems examined. Firstly, systems must use inheritance and interface. Secondly, as we investigated test evolution, we established additional requirements: we discarded projects that are below 90 percentile in terms of size (i.e., lines of code), repository popularity (i.e., stars) and the number of commits. We also ensured that the repositories were not forks. Through this process, we selected 12 systems for our in-depth analysis. These systems include Cayenne, Commons-Collections, Cucumber-JVM, CXF, Httpcomponent-client, Iotdb, Jclouds, Kylin, Maven, Ranger, Ratis, and Wicket. Since we aim to provide detailed quantitative and manual categorization results, we focused on a small sample of high-quality systems.

A. Detecting Inheritance and Interface Changes

To gain preliminary insights into how inheritance and interfaces may impact test code maintenance during software

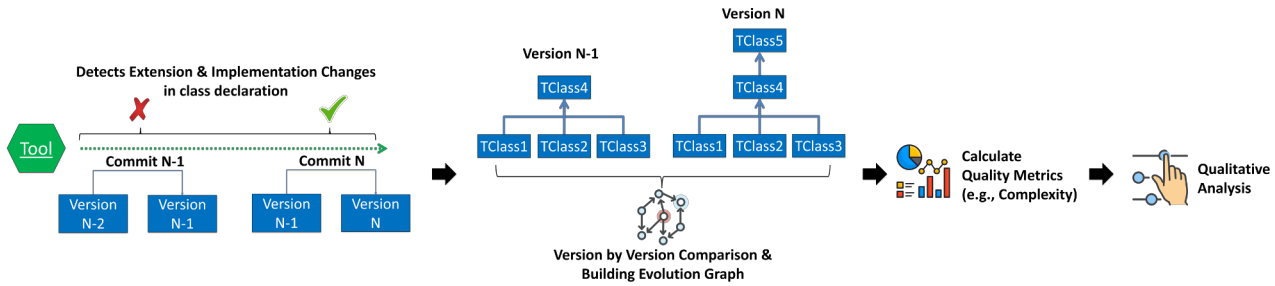

Fig. 1: The overview of the end-to-end process tracking the evolution of inheritance and interface in test and source code. *TClass* represents test class.

evolution, we began by identifying inheritance and interface changes between consecutive commits. Specifically, we develop to a tool to detect inheritance and interface addition, deletion, and replacement changes. For this task, we utilized RefactoringMiner to detect inheritance changes at the commit level [29].

We have chosen RefactoringMiner for its high precision (96.6%) and recall (94%) in comparison to other refactoring tools and abstract syntax tree (AST) differencing tools [29]. Additionally, RefactoringMiner serves as an AST differencing tool at its core, providing an API to compare AST diffs across commits, making it adaptable for identifying both semantic-preserving changes (e.g., refactoring) and non-semantic-preserving operations (e.g., addition, deletion, or replacement). Hence, we did not modify RefactoringMiner's algorithm but used its internal API to analyze similarities and differences in AST changes across commits. This capability is pivotal for determining whether changes such as `A extends B` to `A extends C` are semantically preserving. For example, if classes B and C represent distinct classes, the change is not semantically preserving. However, this change may also represent a rename from B to C, which does preserve the semantic integrity of the code. Below, we elaborate on our extension of RefactoringMiner to detect inheritance and interface changes and outline our analysis steps performed across commits.

Using RefactoringMiner to Detect Inheritance and Interface Changes. We use RefactorMiner to detect and analyze all inheritance and interface changes across commit history. The current version of RefactoringMiner detects the following class-level refactoring operations, we denote it as ***Type 1***:

1) **`Add/Remove`** Class Modifier **`Abstraction`**
2) **`Extract - SuperClass/SubClass/Interface/Class`**

However, ***Type 1*** refactoring operations may miss many inheritance and interface changes. For example, Add/Remove `extends` can occur without any of extract subclass/superclass/interface refactoring when the class already exists. Moreover, for Add/Remove Abstraction, not all SuperClass require Abstract modifiers. Hence, we use RefactoringMiner to detect all potential inheritance and interface changes. Specifically, we use RefactoringMiner's AST differencing API to detect the following additional types of inheritance and interface declarations modification referred to as ***Type 2 (AST)*** changes, as listed below:

1) **`Add/Remove/Replace`** Class **`Extends`**
2) **`Add/Remove/Replace`** Class **`Implements`**

More formally, to detect ***Type 2 (AST)*** changes across commit history, we first rely on RefactoringMiner to match class-level program elements that have the same TypeName (e.g., equal Class Type and Qualified Name) across two consecutive commits. We then use RefactoringMiner's AST differencing API to detect inheritance and interface changes among the matched program elements. Here, I_p represents the set of inheritance and interface type declarations of the program element in the parent commit, and I_c represents the set of inheritance and interface type declarations of the matched program element in the child commit. The added inheritance type declarations are computed as $I^+ = I_c \setminus (I_p \cap I_c)$. The removed inheritance type declarations are computed as $I^- = I_p \setminus (I_p \cap I_c)$. The replaced inheritance type declarations are computed as $I^{\sim} = (I^- \cap I^+)$. We then run RefactoringMiner to all studied systems over their entire software history until the end of 2021. We find that incorporating ***Type 2 (AST)*** changes detects 68% more inheritance and interface changes that were previously omitted by ***Type 1*** changes.

TABLE II: Statistics on the changes of inheritance and interface, comparing both the source code and test code, where (+) is addition, (-) is removal and $\sim$ is replacement changes.

Type	Extension (+)	Extension (-)	Extension ($\sim$)	Implementation (+)	Implementation (-)	Implementation ($\sim$)	Total # Changes	# Evolution Graph
Test	329	1255	2872	90	70	80		
	Total 4,429			*Total 233*			*4,696*	3,043
Source	930	563	6451	2132	1428	2592		
	Total 7,944			*Total 6,152*			*14,096*	6,721

Preliminary Analysis: Detecting Inheritance and Interface Changes Across Commits. Table II shows the ***Type 2 (AST)*** changes that we detected in each system's entire software evolution history up until the end of 2021. We considered such changes in both source code and test code for comparison, since prior works may only consider source code [8, 11, 27, 18]. Following [30], to determine that a class

is a test class, we ensure that class's `Prefix/Suffix` in its name is *"T/test"*. For each studied system, we present the change at three different levels for the test & source code, i.e., add (+), remove (-), replacement (∼). More intuitively, replaced inheritance occurs when `TestClassA extends B` is replaced by `TestClassA extends C`. This would be counted as one replacement (∼), but not one add (+) and one remove (-).

As shown in Table II, we have identified 4,429 inheritance and 233 interface changes within the test code. This indicates inheritance is widely adopted by test developers over the interface in test code. To understand the prevalence of inheritance and interface-related changes that occur in test code, we compare them with the number of commits that modify test classes, totaling 23,651. We observe that out of 23,651 commits that modify the test classes, 4,429/23,651 (18%) change their inheritance relationship, whereas much less, 233/23,651 (1%) change their interface relationship. Hence, we observed a substantial dependency on inheritance usage and relatively less reliance on interface for test maintainability during software evolution. Specifically, among the 4,429 inheritance changes, the majority, 2,844/4,429 (64%), entailed the replacement of inheritance, while 1,260/4,429 (28%) involved the addition of inheritance, and 325/4,429 (7%) led to the removal of inheritance in the test code. Shifting our focus to interface changes within the test code, we also identified 233 interface changes. Among these changes, the majority, 90/233 (64%), involved the replacement of interface, while 75/233 (28%) consisted of the addition of interface, and 68/233 (7%) consisted of the removal of interface in the test code. The high frequency of addition and replacement inheritance and interface changes indicates that test developers may use inheritance to design and maintain their test code. ***Nevertheless, the non-negligible removal of inheritance and interface implies that developers may encounter challenges over time, hence removing them in the test. This observation motivated our study to investigate further and manually categorize how inheritance and interface are used in the test code.***

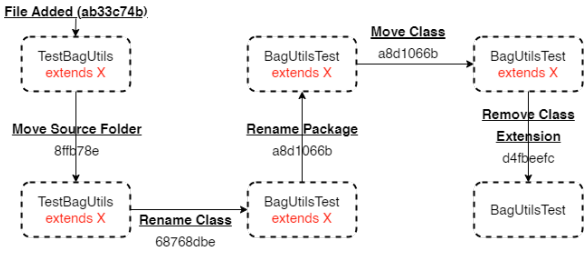

Fig. 2: An example of class evolution graph.

B. Tracking the Evolution of Inheritance and Interface Changes

Upon detecting inheritance and interface changes in one commit, we continue to track the evolution in the commit history to detect the history of changes for all modified Java class. Detecting the complete evolution of inheritance and interface can provide preliminary insights into how they influence test maintainability. Tracking the evolution requires our approach to match program elements in every two consecutive commits. For example, class that was moved and renamed several times should belong to the same evolutionary history. Below, we further explain our commit-by-commit mapping.

Building Java Code Evolution Graph. Refactoring changes, such as *Rename*, can affect the tracking of inheritance and interface evolution. Therefore, to improve the accuracy of tracking the evolution changes, we apply RefactoringMiner to detect class-level refactoring, such as *rename/move package*, *move/rename class*, and extract *superclass/subclass/class*. These refactorings give traces of the complete history of modified Java classes, which is critical for mapping Java classes' history from class creation and modification to persistence in its entire commit history. In particular, we build a class evolution graph, where N_p is the node representing the class signature before class-level refactoring, N_c is the node representing the class signature after class-level refactoring, and E_{pc} represents the edge between N_p and N_c. Namely, there is an edge E_{pc} if the two nodes have the same fully qualified names or their fully qualified name belong to the class-level refactoring changes mentioned earlier. Figure 2 shows an example of a graph that tracks code evolutions for class *TestBagUtils* for *Apache/Commons-Collections*. The test code undergoes several *rename/move* refactorings, and finally, inheritance type declaration changes. Refactorings such as extract superclass are less trivial to track as both the extracted superclass and the original class may undergo further changes. Hence, we consider both edge between $N_p \rightarrow N_{c\_superclass}$ and $N_p \rightarrow N_{c\_subclass}$ as evolution on node N_p. Hence, each class may undergo several inheritance and interface changes in the evolution.

Preliminary Analysis: Detecting Code Evolution. We conduct a preliminary analysis of the test and source code using our code evolution graph. Figure 2 provides an overview of the evolution graph for a class that undergoes multiple inheritance and interface changes during software evolution, and Table II shows their frequency in test & source class. As shown in Table II, after mapping inheritance and interface changes into the code evolution graph, we obtained 3,043 test classes and 6,721 source classes that undergo inheritance and interface changes. In other words, the number decreases as some changes may come from similar evolution history within the test code as indicated in Figure 2. ***This shows that test and source classes may undergo multiple inheritance or interface changes throughout history, which indicates their importance in test maintainability. In Section IV, we explore possible factors behind these changes.***

IV. Quantitative Analysis of Inheritance and Interface Usage in Test Code

In this section, we perform a quantitative analysis to understand the prevalence and evolution of inheritance and interface utilization in the test code.

RQ1: Do inheritance and interface play prominent roles in the initial development of test code and how does it compare to source code?

Motivation. To understand the importance of inheritance and interface in test development and maintenance, it is crucial to determine if inheritance and interface are present from the start or if it is introduced later in software development. If the prevalence of inheritance and interface is low at the early stages of software, and predominant later on, it may suggest that developers frequently rely on inheritance and interface to maintain test code.

TABLE III: For test and source code that undergoes inheritance and interface changes in evolution, we measure the prevalence of the initial version of the test and source Class vs. the snapshot analyzed at the end of 2021.

	# Test Class	# Source Class
*Inheritance or Interface is **present** from the beginning of code creation.*		
Inheritance	1,615 (53%)	2,985 (44%)
Inheritance & Interface	142 (4%)	964 (14%)
Interface	89 (2.5%)	1,720 (25%)
*Inheritance or Interface is **added later** on during software evolution.*		
	1,197 (40%)	1,052 (16%)
	Total = 3,043	Total = 6,721

Approach. We study the prevalence of inheritance and interface during test class creation by examining the evolution graph. Take Figure 2 as an example, we trace the evolution graph to its earliest commit, 8ffb78e, and retrieve its parent commit (ab33c74b), representing the initial test code state before adding any inheritance and interface changes. At commit ab33c74b, we use the *Tree-Sitter* [28] tool to parse the test class, identifying `extends` and `implements` keywords added at the start of test creation. In our analysis of inheritance frequency, we exclude relationships associated with *TestCase* and *Assert*, as they are integral parts of the JUnit Test Framework and do not indicate test class written by test developers.

Result. ***Inheritance and interface is initially present in 59.5% of the test code upon its creation. This percentage is even higher, at 83%, for source code.*** We observed that 59.5% changes in inheritance and interface occur in test classes already having inheritance and interface. This percentage is even higher, at 83%, for changes within the source code. We also observed that only 2.5% of test classes undergoing interface changes already have interfaces upon creation and a much higher percentage, 25%, of source code includes interfaces in the first version of the studied systems. In summary, inheritance undergoes more frequent changes in source code than test code when inheritance and interface already exist from the initial creation of the class.

40% of test classes without inheritance eventually add inheritance during software evolution, while source classes show a lower percentage, with only 16%. We observe that many test classes that do not have inheritance and interface, (a) 10/1,197 (1%) adopt only interface, (b) 10/1,197 (1%) adopt both inheritance and interface, and (c) 1,177/1,197 (98%) adopt inheritance in test code. Conversely, for source classes that do not have inheritance/interface, (a) 843/1,052 (80%) adopt only interface, (b) 51/1,052 (4.8%) adopt both inheritance and interface, and (c) 158/1,052 (15%) adopt inheritance in test code. In summary, although inheritance and interface play a lesser role in the initial design of test code than the source code, test developers eventually adopt more inheritance than source code. Moreover, the source code adopts more interfaces than the test code.

Developers adopt inheritance and interfaces into their test code more frequently over time. Initially, 40% of test classes lacked inheritance or interfaces, but these were added later. In comparison, the source code is more stable, with only 16% of classes showing changes in inheritance or interfaces. This indicates that inheritance and interfaces play a more significant role in the maintenance and evolution of test code compared to source code.

RQ2: How does test code reusability change with inheritance and interface modifications?

Motivation. As observed in RQ1, developers frequently rely on inheritance and interfaces to develop and maintain their test code. Thus, in this RQ, we further investigate how these changes affect test reusability. Understanding this impact is crucial, as it shows initial evidence that developers use test maintenance by reducing redundancy and improving extensibility.

Approach. To measure test reusability, we define three metrics: (i) changes in the depth test hierarchy and the changes in the number of methods that are either (ii) inherited or (iii) overridden. Firstly, to measure the depth of the test hierarchy tree (`DIT`), we use the metric proposed by [20]. We developed a static analysis tool that recursively visits superclasses using the *Visitor Pattern* to measure the hierarchy for a test class. If the superclass is an interface, we recursively visit its superclasses since it can further *extend* other object classes. We continue recursion until the terminating condition, Java's root *Object()* class, is met. We define depth as the longest number of classes visited to reach from the leaf class to the root (e.g., *Object()*). Secondly, our static parser analyzes whether the test method is inherited from the superclass or overridden by the child classes. We consider a method to be overridden if and only if (1) it has the same signature, i.e., the same method name, the same number of parameters, and is not static; (2) the method is a subtype of a supertype method; and (3) the type erasure of the parameter is equal for generic types [16]. Once overridden methods are detected, identifying inherited methods becomes straightforward. Any method signature statically present in the *superclass* but is not overridden is categorized as inherited. Subsequently, we apply our static parser to the results from RefactoringMiner (Section III-A). For exmaple, for each Java class undergoing inheritance and interface changes, we calculate the delta between the depth

TABLE IV: Impact of inheritance and interface changes on test reusability features, such as on the depth of test class hierarchy tree, number of inherited and overridden test methods.

Type of Code Changes		Test Hierarchy Changes				Number of Inherited Method Changes				Number of overridden Method Changes			
		mean	median	min	max	mean	median	min	max	mean	median	min	max
Extension	Addition	1.31	1.0	0	4	8.57	4.0	0	69	0.97	0	0	7
	Removal	-1.30	-1.0	-3	-1	-9.55	-4.0	-38	0	-0.68	0	-6	
	Replacement	0.07	0	-3	4	0.03	0	-43	40	0.03	0	-38	7
Implementation	Addition	0.33	0	0	2	0.41	0	0	8	0.67	0	0	6
	Removal	-1.0	-1.0	-1	-1	0	0	0	0	-1.0	-1.0	-1	-1
	Replacement	0.03	0	-1	1	-0.03	0	-1	0	0.02	0	0	1

of hierarchy, number of inherited and overridden before and after the code changes. Namely, changes in the depth of the test hierarchy for a `TestClass1` from 4 to 3 are indicated by the delta of -1, indicating a decrease in the hierarchy.

Result. ***Inheritance generally increases the depth of hierarchy by an average of 1, it has a much greater impact on the number of inherited test methods, with an average increase of 8 and instances reaching up to 69, highlighting the significant role of inheritance in test reusability.*** Figure IV shows statistics of how changes in inheritance and interfaces impact reusability in test code, specifically focusing on the depth of the test class hierarchy tree and the number of inherited and overridden test methods. We analyze three types of code changes, such as `add/remove/replace class extension/implementation`. In the test code, adding an `extends` increases the `DIT` with an average of 1.31, while removing an `extends` decreases the `DIT` by an average of -1.30. Replacing an `extends` can lead to both increases and decreases in the `DIT`. Generally, changes in inheritance tend to increase the depth of the hierarchy. For instance, in the studied system *commons-collections system (9752389b)*, replacing an abstract class *TestAbstractIntArrayList* increases the inheritance depth from 1 to 3. A similar trend is observed in changes to interface changes. Adding an `implements` results in a slight increase in hierarchy depth (mean of 0.33), whereas removing an `implements` may lead to a decrease in hierarchy depth (mean of -1.0). Interestingly, some projects may not show any change in depth despite modifications such as adding an `extends`. This is often due to external libraries (e.g., Apache Directory with AbstractLdapTestUnit) that provide abstract classes for reusable functionality. Test code inherited from such APIs cannot be statically determined by our parser, highlighting the limitations in accurately measuring the impact of inheritance changes. Nevertheless, it still demonstrates the prevalence of inheritance and interface usage in promoting reusability in test code.

Developers should carefully manage inheritance changes, as a modification to inheritance can significantly impact test reusability. While the depth of the hierarchy may provide an initial indication of test reusability, we also analyze reusability from the perspective of test methods that become inherited and overridden due to such modifications. Understanding inherited or overridden test methods may show that developers introduce test hierarchy to reuse or extend test methods that may help test source code functionality. While adding an `extends` may increase the depth of hierarchy (e.g., mean of 1), there is a much greater impact on the number of inherited test methods. For example, adding an `extends` results in an average of 8 inherited test methods, with some instances as high as 69 test methods. In contrast, the average number of inherited test methods decreases significantly (e.g., mean of -43) when an `extends` is removed, suggesting the important role of inheritance in test method reusability and extensibility. More interestingly, replacing an `extends` has a stable mean in the number of inherited test methods (mean of 0.03). However, we observe extreme cases where the number of impacted inherited test methods can be as high as adding 40 additional test methods or removing 43 test methods. In contrast, changes to `implements` are relatively stable, suggesting less prevalence of interface utilization in the test code. Finally, we provide statistics on changes in the number of overridden test methods. While overridden test methods generally follow similar trends as inherited counterparts, the impact is less pronounced. Adding an `extends` increases overridden test methods by an average of 0.97, with a maximum of 7, whereas removing an `extends` leads to an average decrease of 0.68 overridden methods. Replacing an `extends` shows minimal change, with an average of 0.03 overridden methods, although it may range from -38 to 7 methods. In summary, our results show that modifications to the inheritance hierarchy can substantially impact the reusability of the test method, depending on whether they involve adding or removing inheritance relationships.

> Inheritance changes promote test reusability by increasing hierarchy depth and the number of inherited and overridden tests. In contrast, interface changes have a more limited effect. However, developers should carefully manage inheritance changes, as modification to inheritance can significantly impact test reusability.

V. Manual Categorization

Motivation. We conducted a manual analysis to uncover how developers use inheritance and interfaces to enhance test case design. Our goal is to develop a catalog on how inheritance and interfaces enhance test design and maintainability, which

can inform researchers, practitioners, and testing framework designers on how to improve test quality.

Approach. Our manual categorization comprises the following phases: (1) We used stratified random sampling [15], with a 95% confidence level and a 5% confidence interval, to obtain 226 samples. (2) To create a categorization, the first two authors (A1 and A2) independently derived an initial list of category for test inheritance changes by manually inspecting relevant commit messages, test source code, and bug reports. (3) Authors A1 and A2 then unified their lists and compared their category for each inheritance and interface change until reaching a consensus. The inter-rater agreement for the coding process had a Cohen's kappa of 0.75, indicating a very moderate level of agreement [7]. We divide our manual categorization into two research questions (RQs): RQ3 examines use case of inheritance and interfaces in test code, while RQ4 investigates the design issue that may arise from inheritance usage.

RQ3: How do inheritance and interfaces impact test maintainability?

Test Code Reusability (60%). Developers frequently utilize inheritance to enhance test code reusability in software testing. This approach focuses on reusing utility functions and test fixtures to streamline the initialization of test environments. For instance, in *Cxf - 9c9d5c4b*, a developer created a superclass named *AbstractSTSTokenTest.java*, which encapsulated reusable components such as fixtures (e.g., @Before or @BeforeClass) and utility functions related to web service startup. This design facilitates efficient test development across various scenarios by allowing two subclasses in the same commit to reuse functionality. Similarly, in systems like *Commons-collections - 15cf438*, developers use inheritance (e.g., extending a class with *AbstractTestSortedMap*) to achieve comprehensive test coverage. This system often involves multiple algorithms with shared source code functionalities, necessitating unified testing. For example, testing list data structures requires handling edge cases common to all lists (e.g., `listEquals`). Thus, adopting inheritance not only simplifies test case management but also enhances coverage through effective reuse.

Bug Handling and Prevention (9.3%). Developers may utilize inheritance to effectively address bugs arising from inconsistencies in the test environment, such as failures to establish necessary pre-conditions (e.g., initialization of file storage or cluster servers) before executing tests. For instance, in *Kylin - ae503d9*, a test failed due to an improperly configured environment where the required metadata was not set up by the test code. Similarly, in the studied system *Cxf - f1953fce*, test failures occurred because the test environment was not properly reset after executions, affecting subsequent test cases. To mitigate these issues, developers use inheritance by extending a base class to reuse methods that establish the necessary pre-conditions (e.g., fixtures), thereby resolving the associated bugs.

Test Code Extensibility (5.9%). Developers utilize inheritance to implement the *Template Method* design pattern, defining the testing boilerplate in a superclass while allowing subclasses to make minor changes in test behavior. For example, *Commons-collections* includes different implementations of the *Iterator* class (e.g., *ProxyIterator.java*, *FilterIterator.java*). In *Commons-collections - d3a61e7*, to test various implementations of the *Iterator* class, developers use inheritance to create abstract methods (e.g., makeFullIterator) in the superclass. This approach allows subclasses to implement concrete versions of these abstract methods, reflecting the specific class under test. Thus, inheritance facilitates developers to create new tests as new source code behavior is added.

Source Code Stubbing (1%). Developers may use inheritance or interfaces to stub dependencies in the source code under test by creating a nested test class that inherits the necessary methods from the source code. This allows the test case to utilize the nested class, which instantiates the dependent source code to meet the preconditions for testing. For instance, in *ResponseIOExceptionTest.java* from the studied system *Wicket - f75b829*, developers use the nested class within the test class to implement code dependencies. The study by Wang et al. [31] also highlights the widespread use of inheritance in nested test classes for stubbing behaviors of the source class. Their research primarily focuses on refactoring to replace inheritance-based stubbing, aiming to decouple source code from test code for better maintainability. However, our analysis reveals limited use of Mockito, with many developers preferring delegation over inheritance when implementing stubbing behaviors.

Test Code Selection and Organization (1%). Marker interfaces are empty interfaces that provide type information to the JVM at runtime, enabling specific actions based on this information [9]. In the studied system *Commons-collections - 787edf0*, we discovered an interesting use of the marker interface to control test case execution. Developers created a test class that extended an abstract class and implemented a nested interface representing the marker interface. This approach allowed the test code to inherit test cases from the abstract class and determine the instantiating type of the current test class, corresponding to the interface. Depending on this type, the test code could control whether a test should be executed. While uncommon, we found that developers use marker interfaces to categorize and selectively execute tests within the testing framework or tool.

Junit Migration (9.4%). Test framework (e.g., JUnit) migration is another core use for changing test inheritance. For example, *Junit5* ships with new annotation-based features, such as @ParameterizedTest and @ExtendWith, and a new interface for dealing with the test-fixture lifecycle that removes the need for test inheritance. Before @ParameterizedTest, *JUnit4* requires a child class to define the test arguments and a base class to define parameterizable @Test cases. With *JUnit5*, developers can now utilize @ValueSource/@MethodSource to inject arguments to the test cases directly instead of adding bloated dependencies with another class. Secondly, with @ExtendWith annotations, while it removes the need to extend the base class to configure test-

fixture, we find that logically, there is no difference from using inheritance as they may accomplish the same functionality. However, it may improve code readability and ease of code extensibility.

We identified various uses of inheritance and interfaces in real-world test code design, with a strong emphasis on improving test reusability. Additionally, we observed how developers may use interfaces to support test development and selection.

RQ4: What potential issues arise from using inheritance in test maintainability?

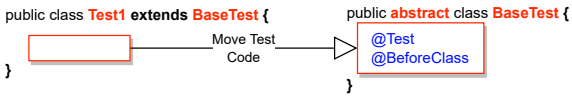

Fig. 3: Issue 1: Empty Inheritance. In `Test1`, we see an empty class because the code moved to the `BaseTest`. However, it may cause other subclasses depending on `Test1` to create arbitrary inheritance relationship.
i

Issues with obsolete test code (1%). While inheritance enhances test code reusability, it can also lead to maintainability issues. We have observed that developers may unintentionally extend a class that turns out to be empty, thus undermining the intended reusability achieved through inheritance. For instance in the studied system *Jclouds* (*commit 180265fe*), developers eliminated the inheritance (e.g., *Remove Class Extension*) as it did not contain any code. To gain a deeper understanding of the motivations behind adding such empty code segments, we analyze its commit history and its code changes. In our examination, as depicted in Figure 3, we identified that in Jclouds (*commit 72ba1639*), developers extracted a *BaseTest* and transferred code from *Test1* into this new class for code reusability. However, it appeared that all of the code was relocated to *BaseTest*, leaving the original test code as an empty and unnecessary intermediary step in the test code hierarchy. This empty step served no purpose in the testing process.

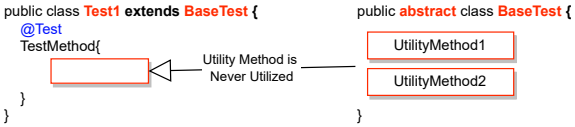

Fig. 4: Issue 2: Under-utilized Inheritance. `Test1` inherits utility methods from `BaseTest`, yet none of them are actually utilized.

Moreover, we came across instances where test code extends a *BaseTest*, yet none of the inherited methods are utilized by the *ChildTest*. As depicted in Figure 4, in Cayenne (*de8123a2*), developers may view the *BaseTest* as unnecessary and remove it from inheritance hierarchy. While this issue shares similarities with the category of empty inheritance, it presents a more significant challenge in terms of code comprehension. Specifically, it may not be immediately evident whether the code in *BaseTest* is being used in the *ChildTest*. ***Hence, there is a need to monitor changes to inheritance to prevent the adoption of bad practices.***

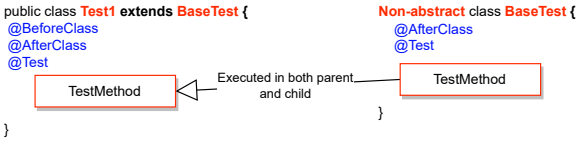

Fig. 5: Issue 3: Duplicate Test Execution. In *TestMethodName* is executed both in *Test1* and *Test2*.

Issues with test runtime (1%). Inheritance enables test code reusability, facilitating a straightforward way to test for common source code functionality. However, it may also introduce test execution duplication, potentially increasing execution time. In Figure 5, we provide an example from the Cxf project (commit *d95ed565*), where a test class extends a base class that contains the test cases. Although the intention is to reuse test cases from the superclass within the subclass, the absence of the *Abstract* modifier allows the superclass's test cases to be executed both in itself and in the subclass. While this issue may not lead to software faults, it may lead to prolonged execution time which may increase the cost to test the software system. ***These findings emphasize the need for careful consideration when using inheritance in testing to avoid unnecessary duplication and inefficiency.***

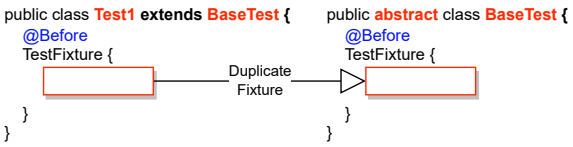

Fig. 6: Issue 4: Obstructing Code Comprehension. Developers add duplicate fixture in both *Test1* and *Test2*.

Issues with test comprehension (1%). Incorporating inheritance in test code can hinder code comprehension. For instance, in Figure 6 (Iotdb project, commit *855675f4*), developers created a test fixture in the subclass with identical names and functionality as in the superclass. Although the commit message mentions *"Fix unclosed file"*, the file was already closed by the fixture method inherited from the superclass. Unfortunately, the developers overlooked the presence of the inherited method and added a redundant fixture in the subclass. While the new method overrides the inherited one and does not result in duplicate test execution, it highlights that using inheritance can negatively impact code comprehension. Moreover, in CXF d47f3a58 we observe that the subclass improperly overrides the fixture from the BaseTest, i.e., uses `super` keyword to reuse functionality but forgets clear state.

To fix this, the developers fix test flakiness by adding clear() in a subclass. Based on this scenario, although overriding tests can provide developers with more flexibility, it also increases the possibility of unintended consequences. As previously reported in Maven-Surefire [10], experts recommend completely avoiding inheritance in testing to minimize bugs. However, this may not always be feasible, e.g., labor-intensive, due to the prevalence of test code inheritance in practice. Moreover, in some projects, like *Jclouds*, where inheritance has a very high depth, possibly up to a depth of 10, it can become challenging to discern which tests are overridden or inherited, which can impede effective test code writing. ***Hence, there is an opportunity to analyze the relationship between inherited methods and their purpose in test design to determine if the inheritance is necessary.***

While there are advantages to using inheritance, it also presents potential issues. The challenge lies in comprehending these advantages and disadvantages of inheritance, which involve facilitating extensible test code while carrying the risk of introducing problems.

VI. Implications and Future Work

Based on our empirical findings, we present actionable implications and potential future work.

Implications for Test Developers

D1: Developers need clearer guidance on the effective use of inheritance and interface in test code. Our findings in RQ1 suggest the value of inheritance and implementation in guiding early test case design. Our RQ3 unveils motivations for these changes, primarily driven by the desire to enhance test code design. Despite ongoing controversy and ambiguity regarding interfaces [8, 11, 27, 18, 32, 4, 26, 3, 23, 21, 22], open-source projects demonstrate the prevalence of inheritance and interface design, and it proves beneficial in specific use cases to aid test code design. Hence, this underscores the need to provide clear guidance to developers on effective uses of inheritance and interface in the test code. Our derived motivations behind their uses may assist developers in making informed decisions.

D2: Developers should employ automated tools to address obsolete code resulting from inheritance during software evolution. In RQ3, we identified various issues stemming from the use of inheritance in test code. For instance, we discovered instances of empty inheritance, considered as obsolete tests, which can significantly increase code complexity. Similarly, developers may occasionally eliminate underutilized inheritance, where inheritance is implemented for reusability but remains unused. While these issues may initially appear problematic, as discussed in RQ3, they are artifacts of software evolution caused by developers. Obsolete inheritance resulted from refactoring activities that involve extracting a superclass and pulling up methods and attributes for the sake of reusability. On the other hand, underutilized inheritance reflects developers' initial intentions to improve test code with new features, which, unfortunately, end up going unused during the software evolution process. We do not consider these issues to be severe, as they do not significantly impact the effectiveness of testing. Furthermore, these issues can be easily mitigated through the use of automatic tools while maintaining the benefits of inheritance in the creation of effective test cases.

D3: Developers should consider improving readability when using inheritance for reusability. As our finding in RQ3 reveal, many developers resort to test inheritance to increase code coverage. For instance, in studied systems like *Commons-Collections* we observe the advantages of utilizing test inheritance. This system deals with various collection handling algorithms, each having distinct implementations. However, they often share common functionalities that require thorough testing, and test inheritance provides a convenient means to accomplish this. In scenario like this, test inheritance is proves beneficial. However, there may be potential issues with inheriting test cases. For example, these test cases are silently executed, sometimes without developers' awareness. In RQ3, we see case that developers accidentally omitted the *Abstract* modifier from the base test case. This omission enabled *Junit* to instantiate the test class and executes its test case both in the superclass and subclass, leading duplicate execution. One mitigation strategy on raising awareness about inherited test cases is to improve the readability. When using inheritance, instead of allowing it to be silently executed, we could explicit allow the test case call superclass's test case, `super.TestCase()`. This practice serves as a form of code documentation that enhances readability. This presents a trade-offs that developers should carefully consider for the sake of long-term maintainability.

Implications for Researchers

R1: Future research in test case minimization should consider test inheritance. As highlighted by our analysis in RQ3, test inheritance, while offering benefits, can lead to instances of duplicate test execution. In systems that extensively rely on test inheritance, it can be challenging for developers to entirely eliminate such scenarios. Therefore, future research can apply program analysis techniques to identify and manage cases of duplicated test execution stemming from test case inheritance. Furthermore, there are promising research directions that extend beyond detection. Researchers should explore reduction and minimization techniques specifically tailored to the context of inheritance. These reduction technique poses a significant challenge since inheritance impacts many classes, and their removal may require further refactoring to preserve behavior of the test code. Consequently, we recommend that future advancements in test case reduction techniques take into account the nuances of test inheritance and devise strategies to mitigate redundancy within this unique context.

R2: There exist widespread refactoring that replace inheritance with annotation. Future automated tool support may consider this unique refactoring. In RQ3 we find several instances where developers uses annotations (e.g.,

@ExtendWith) from testing frameworks like *Junit* to replace test inheritance. @ExtendWith is annotation introduced by *Junit5* to extend the behavior of test classes or methods, i.e., adding conditional testing to the test code, when certain JVM or operation system is absent. Using @ExtendWith may provide benefit over inheritance due to better separation of concerns in the test code. For example, each @ExtendWith can focus on specific aspect of testing, which makes the test code more maintainable. This is beneficial for cases we find in RQ3, where inheritance contain many different aspect of the test automation (e.g., as you cannot create multiple inheritance), such as reusing test cases, utility method (e.g., performance testing and mocking) and fixture. Hence, there needs to be further research on when it is beneficial to use annotation or inheritance.

VII. Related Works

Inheritance Evolution and Maintenance. Many works investigated the evolution of inheritance in source code. For example, Shaheen and du Bousquet [19] studied the relationship between inheritance and the number of methods to test. They claim that testing should be more expensive if the inheritance depth is high, as the inherited method should be re-tested. Nasseri et al. [11] studied whether inheritance evolves breadth-wise or depth-wise, and developers consider depth-wise as hard to maintain and prefer breadth-wise inheritance. Nasseri et al. [12] studied the evolution of inheritance from the perspective of class re-location to understand what motivates their move and try to give insights on potential maintenance challenges. Giordano et al. [5] studied the evolution and impact of delegation and inheritance on code quality. They find that their evolution often leads to code smell severity being reduced and improved maintainability. While these works investigate inheritance in source code, we focus on studying the benefits and issues associated with inheritance from test code perspective.

Inheritance Maintenance in Test Code. Limited works investigated the evolution and maintenance of test inheritance.The work by Wang et al. [31] conjectured that despite the existence of powerful mocking frameworks, developers often turn to inheritance to mock source code under test. Hence, they proposed a tool to refactor mocking via inheritance with a mocking framework. Our work is different in fact that we aim to derive how inheritance can be of benefit to developers, as well as their negative impact on maintainability. Peng et al. [17] studied the impact of code dependencies on continuous integration. They found that inheritance causes the majority of dependency in test cases and proposed test dependency-related smells. Different from our work, they emphasize test dependencies and little on the impact of test code reusability through inheritance.

Quality Issue in Inheritance. Many prior studies extensively studied the quality issue, such as the change/defect proneness of using inheritance [8, 11, 27, 18, 32, 4, 26, 3]. For example, researchers found that ineffective use of inheritance is correlated to software quality issues and maintenance difficulties [8, 11, 27, 18]. Prior studies even used inheritance as a proxy to measure software complexity and to predict software defects in industry systems [32, 4, 26, 3]. Unfortunately, these researches on inheritance primarily analyzed the source code, largely overlooking inheritance in the test code. Hence, whether inheritance impacts software maintainability remains unclear. In our work, we do not dismiss use of inheritance. Instead, we comprehensively discuss both its advantages and disadvantages, taking into account requirements and motivations of developers.

VIII. Threats to validity

Internal Validity. Our findings depend on the accuracy of RefactoringMiner to detect inheritance changes. We mitigate this threat by validating the tool thoroughly during our manual analysis in RQ3. We observed that the extension of RefactoringMiner detects inheritance and interface changes with 100% precision with 0 false positives.

External Validity. Our studied systems are all open source and implemented in Java, so the result may not be generalized to other systems. However, our studied projects are high-quality and used in many commercial settings. The usage patterns we derived in RQ3 also encompass many critical aspects of test code execution and maintenance, which should also exist in closed-source projects. Hence, we believe that the findings will be similar in closed-source projects or other programming languages that use inheritance. Nevertheless, to minimize the threat, we follow a set of criteria to select popular systems on GitHub, large in scale, and actively maintained, and frequently used in commercial settings. Future studies are encouraged to replicate our experiment on other systems implemented in different programming languages.

Construct Validity. In RQ3, we conduct a manual analysis to understand how developers use inheritance and interface on test maintenance. We perform the study on a statistically significant sample using a 95% confidence level and a 5% confidence interval. To reduce the biases in our manual categorization, two of the authors independently studied the sample and compared the results. Any discrepancy is discussed until a consensus is reached. We computed Cohen's Kappa and found that the level of agreement is substantial between the two authors (0.75).

IX. Conclusion

This paper presents the first empirical study on inheritance and interface in tests to fill the knowledge gap regarding the evolution and maintenance of tests since prior studies focused mainly on source code inheritance/interface. Our study reveals many actionable implications and research directions: 1) We observe that 59.5% of test classes already incorporate inheritance when initially created, while 40% of test classes incorporate interfaces. Notably, interfaces are less likely to undergo modification over time compared to inheritance. 2) Test code and source code undergoes many inheritance/interface changes during evolution, and test code takes longer time

to change inheritance/interface 3) Such changes significantly increase code complexity. 4) Despite the controversial nature of inheritance, it still offers extensibility in test design. Finally, we report three issues, underscoring the need for careful inheritance use in specific cases.

Data Availability

We have made our replication package available, which contains all the datasets and code available [2].

References

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[2] Anonymized Author. replication_package, 2014. URL https://anonymous.4open.science/r/Blessing-or-Curse-Investigating-Test-Code-Maintenance-via-Inheritance-and-Interface-F174/README.md.

[3] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[4] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[5] Giammaria Giordano, Antonio Fasulo, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. On the evolution of inheritance and delegation mechanisms and their impact on code quality. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 947–958. IEEE, 2022.

[6] Petri Kainulainen. Three reasons why we should not use inheritance in our tests, 2014. URL https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/.

[7] Tarald O Kvålseth. Note on cohen's kappa. *Psychological reports*, 65(1):223–226, 1989.

[8] Cristina Marinescu and Mihai Codoban. Should we beware the inheritance? an empirical study on the evolution of seven open source systems. In *2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA)*, pages 246–253. IEEE, 2014.

[9] Marker. Marker interface, November 2022. URL https://www.baeldung.com/java-marker-interfaces.

[10] Maven. Maven apache pony mail, 2023. URL https://lists.apache.org/thread/cpm046p745j7nj0dvw9mtxfmthkgobp6.

[11] Emal Nasseri, Steve Counsell, and M Shepperd. An empirical study of evolution of inheritance in java oss. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 269–278. IEEE, 2008.

[12] Emal Nasseri, Steve Counsell, and M Shepperd. Class movement and re-location: An empirical study of java inheritance evolution. *Journal of Systems and Software*, 83(2):303–315, 2010.

[13] Oracle. Inheritance, 2022. URL https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html.

[14] Oracle. Polymorphism, 2022. URL https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html.

[15] Van L Parsons. Stratified sampling. *Wiley StatsRef: Statistics Reference Online*, pages 1–11, 2014.

[16] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46: 1155–1179, 2015. doi: 10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

[17] Zi Peng, Tse-Hsun Chen, and Jinqiu Yang. Revisiting test impact analysis in continuous testing from the perspective of code dependencies. *IEEE Transactions on Software Engineering*, 2020.

[18] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software*, 65(2):115–126, 2003.

[19] Muhammad Rabee Shaheen and Lydie du Bousquet. Relation between depth of inheritance tree and number of methods to test. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 161–170. IEEE, 2008.

[20] Frederick T Sheldon, Kshamta Jerath, and Hong Chung. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(3):147–160, 2002.

[21] Stackoverflow. why inheritance is strongly coupled where as composition is loosely coupled in java?, 2013. URL https://stackoverflow.com/questions/19146979/why-inheritance-is-strongly-coupled-where-as-composition-is-loosely-coupled-in-j.

[22] Stackoverflow. Prefer composition over inheritance?, 2013. URL https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance.

[23] Stackoverflow. How can i resolve this redundancy caused by inheritance and nested class?, 2020. URL https://stackoverflow.com/questions/56063575/how-can-i-resolve-this-redundancy-caused-by-inheritance-and-nested-class.

[24] Stackoverflow. Should i use inherited tests?, 2023. URL https://stackoverflow.com/questions/59312507/should-i-use-inherited-tests.

[25] Philipp Straubinger and Gordon Fraser. A survey on what developers think about testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 80–90. IEEE, 2023.

[26] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.

[27] Amjed Tahir, Steve Counsell, and Stephen G MacDonell.

An empirical study into the relationship between class features and test smells. in 2016 23rd asia-pacific software engineering conference (apsec), 2016.

[28] Tree-sitter. Tree sitter, November 2023. URL https://tree-sitter.github.io/tree-sitter/.

[29] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[30] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. Methods2test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 299–303, 2022.

[31] Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. An automatic refactoring framework for replacing test-production inheritance by mocking mechanism. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 540–552, 2021.

[32] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third international conference on software testing, verification and validation*, pages 421–428. IEEE, 2010.