A First Look at the Inheritance-Induced Redundant Test Execution

Dong Jae Kim
Software PEformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Quebec, Canada
k_dongja@encs.concordia.ca

Tse-Hsun (Peter) Chen
Software PEformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Quebec, Canada
peterc@encs.concordia.ca

Jinqiu Yang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
jinqiu.yang@concordia.ca

Abstract

Inheritance, a fundamental aspect of object-oriented design, has been leveraged to enhance code reuse and facilitate efficient software development. However, alongside its benefits, inheritance can introduce tight coupling and complex relationships between classes, posing challenges for software maintenance. Although there are many studies on inheritance in source code, there is limited study on using inheritance in test code. In this paper, we take the first step by studying inheritance in test code, with a focus on redundant test executions caused by inherited test cases. We empirically study the prevalence of test inheritance and its characteristics. We also propose a hybrid approach that combines static and dynamic analysis to identify and locate inheritance-induced redundant test cases. Our findings reveal that (1) inheritance is widely utilized in the test code, (2) inheritance-induced redundant test executions are prevalent, accounting for 13% of all execution test cases, (3) bypassing these redundancies can help reduce 14% of the test execution time, and finally, (4) our study highlights the need for careful refactoring decisions to minimize redundant test cases and identifies the need for further research on test code quality.

Keywords

Software Testing, Software Evolution, Software Maintenance

ACM Reference Format:
Dong Jae Kim, Tse-Hsun (Peter) Chen, and Jinqiu Yang. 2024. A First Look at the Inheritance-Induced Redundant Test Execution. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639150

1 Introduction

In a highly evolving software market, customers expect new features delivered on time alongside reliable and high-quality products [1]. To reduce maintenance cost and improve productivity, code reuse plays a pivotal role. Through code reuse, developers can take the advantage of existing functionality and achieve faster development while maintaining code quality. Particularly, one of the main advantage of inheritance, a fundamental aspect of object-oriented design, is to facilitate code reuse [34, 35]. Inheritance offers a simple way for a Class A to reuse a feature defined in Class B by utilize the `extends` keyword, such as Class A extends Class B.

While inheritance provides many benefits in reducing implementation and maintenance overhead, using inheritance ineffectively can also create tight coupling between classes [44], causing non-flexibility and overly redundant code [45]. Many researchers found that ineffective use of inheritance is correlated to software quality issues and maintenance difficulties [29, 32, 41, 47]. Prior studies even used inheritance as a proxy to measure software complexity and to predict software defects in industry systems [6, 7, 46, 54].

Existing studies on inheritance primarily focused on the source code [6, 7, 29, 32, 41, 46, 47, 54]. However, there has been limited investigation into the impact of inheritance in the test code; especially inherited test cases. Our preliminary analysis reveals that 40% of 503 sampled open-source software systems use inheritance in test classes, indicating a significant adoption of inheritance in software testing. One potential benefit of test inheritance, as found by Wang et al. [49], is that developers often turn to inheritance to mock the source code under test. Another benefit of inheritance is test code reusability, which can improve coverage and help test maintenance [5].

Despite the potential benefits of test case inheritance, using inheritance in test code can also lead to overly complicated code as software systems become more complex. A study by Peng et al. [40] showed that test case inheritance causes most code dependencies, which can over-complicate test case design and maintenance. Moreover, some practitioners view inheritance as poor practice and should be refactored [43], i.e., "*Prefer composition over inheritance and interfaces*" [45] or "*It is a bad idea to use inheritance in test*" [17]. In addition to test case design, one issue with test inheritance is that it can result in multiple *subclasses* inheriting identical test cases from the same *superclass*. Such inherited and identical test cases are redundant and can cause test execution overhead, which further extends the already time-consuming testing process.

In this paper, we aim to study the impact of inheritance in the test code by focusing on the redundant test executions caused by inherited test cases. We develop a hybrid approach that combines static and dynamic analysis to study and detect inheritance-induced redundant test executions. First, we apply static analysis to analyze the source code and extract the inheritance hierarchy in test classes. Then, we extract test cases candidates that potentially cause redundancies. Finally, we apply dynamic analysis, which involves source code coverage and test oracle analysis, to detect whether these candidates are truly redundant. We conduct our study on 15

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639150

open-source Java systems and found that (1) 13% of the total executable test cases are in fact redundant, and (2) such redundant tests take 14% of total test execution time. Finally, we shed light on (3) the challenges of addressing redundancy, precisely the difficulty in preserving code coverage while removing redundancy. This paper makes the following contributions below:

- We are among the first to show high inheritance usage in test code (over 40% of analyzed systems).
- We observe that 13% of executed test cases are introduced through inheritance, accounting for 14% of the total test execution time.
- Our hybrid approach combines static and dynamic analysis to identify inheritance-induced redundant test executions, providing developers with tools to detect and simply bypass the bottlenecks in test executions.
- We find that eliminating redundancy poses challenges in preserving code coverage, as inherited test cases can be redundant in certain *subclass* but non-redundant in the rest.
- We release the source code of our tool and the dataset[1] of our experiments to help other researchers replicate and extend our study.

Paper organization. Section 2 discusses motivations. Section 3 discusses our methodology. Section 4 presents ours research questions. Section 5 summarizes the implication of our findings. Section 6 discusses related work. Section 7 discusses threats to validity. Section 8 concludes the paper.

2 Motivation

Existing studies on inheritance have primarily focused on the source code [6, 7, 29, 32, 41, 46, 47, 54]. However, there has been limited investigation into the impact of inheritance in the test code. We conjecture that developers regularly use test code inheritance in practice. To verify whether developers use inheritance in test code, we analyze the number of test code inheritance that is attributed to various software systems, by mining hundreds of open-source Java Repositories, similar to technique the employed by [22]. We start with the Java-med dataset from Alon et al. [4], which consists of 1,000 top-starred Java systems from GitHub. We utilize *Spoon*, a static analysis tool, to create the source/test code model for the entire repository [38]. From the list of `Java` files in the repository, we check (i) whether a file is a `test` file and (ii) whether the `test` file is part of the inheritance tree. To measure the prevalence of inheritance, we choose to use the inheritance tree rather than simply counting inheritance usage (e.g., `extends`). The inheritance tree offers a more comprehensive view, representing the hierarchical structures of test classes and the holistic relationships among classes in the repository.

Table 1: Analyzed repository characteristics.

Repository Inheritance Characteristics	#Repositories
Has at least one inheritance tree in the test code	202
No inheritance tree in the test code	301

[1]https://github.com/djaekim/ICSE2024.git

To check if a file is a `test`, we examine if the name's *Prefix* or *Suffix* contains the *"T|test"* keyword. To determine whether a `test` file is a component of the inheritance tree, we search for `extends` keywords and iteratively traverse upward through its `superclass` until we reach the root node of the inheritance tree. During traversal, a test class itself may be a `superclass` for other test classes. Hence, we also traverse through all the `subclasses` until reaching a leaf class. We omit systems that do not have test classes. Accordingly, from the 1000 repositories, we are left with 503 repositories as shown in Table 1. We find that the ratio of the repositories that have at least one inheritance hierarchy tree in the test code is 40% among 503 studied repositories. The finding suggests that a significant number of repositories in fact adopt inheritance in software testing. This percentage is a staggering proportion considering that many practitioners view inheritance as poor practice and should be refactored, i.e., *"Prefer composition over inheritance and interfaces"* [43] or *"It is a bad idea to use inheritance in test"* [17, 45]. More interestingly, there is a moderate to strong correlation (i.e., 0.61) between the number of test files in the repository and the number of test inheritance hierarchy [2]. The finding indicates that as the software becomes complex, developers may be more inclined to use test inheritance. Based on this analysis, we believe that inheritance plays a significant role in the design of test code.

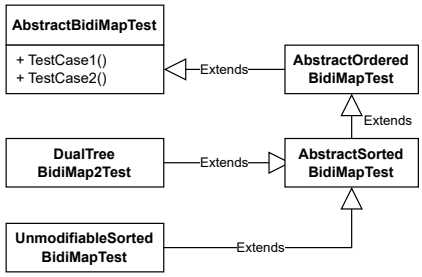

Figure 1: Developers re-use test through inheritance to ease maintenance.

Figure 1 shows a real-life scenario of test inheritance in *Commons-collections*, where the developer uses inheritance for test code reuse and easing maintenance. For example, test cases 1 and 2 from *AbstractBidiMapTest* cover the coverage of different source code functionality, *DualTreeBidiMap* and *UnmodifiableSortedBidiMap*. Despite the aforementioned advantages, the prevalence of inheritance in test code shows a potential challenge: the proliferation of redundancies within test cases. For example, as shown in Figure 1, not only is the inheritance deep, but the two test cases declared in the root class can also be inherited throughout the hierarchy. The impacted subclasses with the `abstract` modifier, i.e., *AbstractSortedBidiMapTest*, *AbstractOrderedBidiMapTest*, do not execute the test cases. However, the concrete *subclasses*, i.e., *DualTreeBidiMap2Test* and *UnmodifiableSortedBidiMapTest*, in the leaf position will eventually execute the test cases, as the testing frameworks (e.g., JUnit) will instantiate them during testing, which leads to potential redundant test case execution, i.e., proliferation. While such redundancies may have no consequences (e.g., performance overhead) in the source code, their presence in test code represents a considerably detrimental practice.

Unlike source code, where redundant code may remain dormant and unexecuted, test cases annotated with the `@Test` annotations are automatically executed within the inheriting child classes. As shown in Figure 1, while such a test case aims to help maintenance and code reuse, it may also lead to redundancies if the coverage and test oracle remain the same. This fundamental distinction forms the basis of our research focus, which aims to identify redundant test cases lacking fault-locating capabilities while contributing to an increase in test execution time. Our investigation seeks to shed light on these intricacies of inheritance-induced redundancies in test code.

Based on the aforementioned discussion about redundant test cases, we focus on two important definitions: (1) ***Redundancy-Inducing Inheritance:*** Inheritance is considered redundancy inducing if it declares test cases and has impacting *subclasses*, and (2) ***Inheritance-Induced Redundant Test Candidates:*** Test cases are classified as potentially redundant if they are executed more than once due to inheritance, although they may or may not be truly redundant. In Figure 1, the example represents (1). The root class, *AbstractBidiMapTest*, contains test cases and has many impacting *subclasses*. Its test cases are hence (2), as they are executed multiple times in the *subclasses*. In the subsequent section, we present our technique for detecting ***Inheritance-Induced Redundant Test Executions***, given (1) and (2).

3 Our Technique for Identifying Inheritance-Induced Redundant Test Executions

Due to the complexity of inheritance trees and the scale of modern software, developers may not always be aware of redundant test executions caused by inheritance. In this section, we present our technique to detect redundant test executions. Figure 2 summarizes the overview of our technique, which consists of three main parts: static analysis to detect (1) ***Redundancy-Inducing Inheritance*** and (2) ***Inheritance-Induced Redundant Test Candidates***, and (3) dynamic analysis to identify ***Inheritance-Induced Redundant Test Executions***. Performing static analysis before dynamic analysis reduces the cost of the latter since not all test cases are *Inheritance-Induced Redundant Test Executions*. Therefore, we perform dynamic analysis on a subset of the total test cases, specifically on the ***Inheritance-Induced Redundant Test Candidates***.

3.1 Statically Detecting *Redundancy-Inducing Inheritance*

In this section, we describe how we extract redundancy-inducing test inheritance. We use *Spoon*, a static analysis tool [39].

3.1.1 Identifying Test Cases. In our detection of redundant test cases, our first step is to identify the relevant test classes from the *.java* files. Our studied systems use *JUnit4+* testing frameworks to design test cases and *Maven* to execute the test cases. Hence, we first search for all potential test cases that exist within the studied systems. We look for methods that are annotated using the @Test annotation. However, not all test cases written in the test code are executed during regression testing. A test case can either be disabled to prevent flaky test or excluded in the *Maven* build, specified in the `pom.xml`, according to development needs [25]. Hence, we execute our test cases using *Maven* and filter out *skipped* test cases.

3.1.2 Extracting Inheritance Hierarchy. We then determine whether the identified test classes form an inheritance tree, i.e., `extends superclass`. For every test class, we use the *Vistor* Pattern to recursively visit its `superclass` until the terminating condition is met, such as Java's root Object() class or a class from external libraries. When traversing upstream, a test class itself may be a `superclass` for other test classes. Hence, we also traverse through all the `subclass` until reaching a `leaf` class. Once we have traversed all reachable test classes, we generate a comprehensive tree hierarchy denoted as $Tree$, containing crucial information. We represent nodes 1) N_cl as test classes, and nodes 2) N_mt as test cases. We represent edges 1) E_cl as a directed edge between two N_cl (e.g., node1 and node2), where the node1 is the subclass of node2, and 2) E_mt is the non-directed edge between N_cl and N_mt, where N_mt is a test case of test class, N_cl. As shown in the Figure 2, this step is described by transformation from *source code* to *tree-preprocessed*.

3.1.3 Identifying Inherited Test Cases. In Section 3.1.2, we assigned test case nodes, N_mt, to its corresponding test class, N_cl. Based on this *Tree*, we now extract the following test case types: unique methods, overridden methods, and inherited methods, which are annotated as U_mt, O_mt, and I_mt. To elaborate, U_mt are methods that are unique to the class, O_mt are methods that override the parent method, and I_mt are methods inherited from its `superclass`. More formally, we consider a method to be O_mt, if and only if (1) they have the same signature, i.e., the same method name, the same number of parameters, and are not static, (2) the method is a subtype of a supertype method and (3) type erasure of the parameter is equal for generic types [38]. Once O_mt is detected, identifying U_mt and I_mt becomes straightforward. Any method signature statically present in N_cl is classified as U_mt for that particular class, while any method inherited from the *superclass* is categorized as I_mt. Following the definitions (1-3), we generate a post-processed inheritance hierarchy, termed $Tree\_post$, where test cases are accurately annotated. This transformed *Tree* is now ready for further analysis of redundant test cases. Refer to Figure 2 for a visual representation of this transformation from *tree-preprocessed* to *tree-postprocessed*.

3.2 Statically Detecting *Inheritance-Induced Redundant Test Candidates*

As discussed in Section 2, our focus is to detect redundant test case execution caused by inheritance. For this, we first perform static analysis to identify potential redundant candidates. In particular, we analyze the $Tree\_post$ generated in prior Section 3.1.3 in the following ways: 1) We first determine the test classes, T_cl, that are `non-leaf` class in the $Tree\_post$ and have executable test cases, T_mt. Such T_cl indicate potential sources of inherited test cases, 2) We then determine whether T_cl contain more than one `subclasses`, and if the resulting `subclasses` is a `non-abstract` class that can execute the inherited test case. Then, such test cases are executed more than once and are potentially redundant. More formally, if a test case T_mt1 is inherited by both `non-abstract` *subclassA* and *sublassB*, then such T_mt1 has the potential of being a redundant candidate.

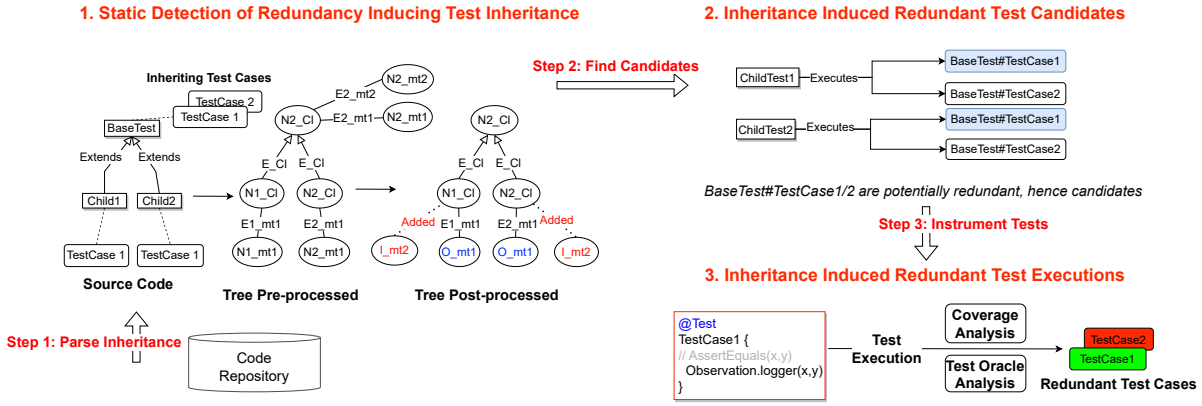

Figure 2: Overview of the End-to-End Process for Finding Redundant Test Cases.

3.3 Detecting *Inheritance-Induced Redundant Test Executions* through Dynamic Analysis

Whether *T_mt*1 is a truly redundant test case cannot be fully determined by static analysis discussed in Section 3.2. For example, a *subclassA* and *subclassB* may both inherit *T_mt*1, but through a program dependency, `subclasses` can alter the state of the *T_mt*1 (e.g., by setting environment variables). In this case, *T_mt*1 can no longer be considered redundant as its behavior may be intentionally written by the developer to achieve partial code reuse by subclassing. However, it is challenging to statically determine the ground truth of program dependency to guarantee that *T_mt*1 is redundant. Hence, we resort to dynamic analysis by executing the test cases to collect execution trace information. Specifically, we instrument the test cases prior to execution to collect (i) source code coverage and (ii) test oracle information. Our intuition is that if the *T_mt*1 that is executed in both *subclassA* and *subclassB* has the same code coverage and test oracle, then *T_mt*1 is truly redundant. This technique draws inspiration from prior works in test case amplification [8, 15] and test case reduction [9, 27, 30, 31, 48, 53], where code coverage and oracles are considered to validate test case quality. Below, we discuss the detail of our dynamic analysis.

3.3.1 Extracting Code Coverage. To detect redundant tests, we conduct code coverage analysis on *Inheritance-Induced Redundant Test Candidates*. We use *JaCoCo* to collect coverage data during test execution. Below, we outline the process for executing 1) inherited test cases and 2) instrumenting *JaCoCo*.

Executing Inherited Tests. In our analysis, we run test cases one-by-one to avoid accumulation of dirty states that may affect the test result, i.e., coverage and oracle. Hence, we re-initialize the JVM and reset the environment for every test execution. However, executing test cases in *Maven* is non-trivial for inherited test cases. Such inherited test cases are not directly present in the test code, yet may be executed many times through inheritance (e.g., a test case in the superclass is inherited by multiple subclass). To execute the inherited test case, we use *Maven-Surefire*'s command-line options called `-DTest` with specified test cases such as *=subclass#inherited_method*. Here, `#inherited_method` refers to the test case inherited from its `superclass`, while `subclass` represents the class that inherits the test case. In addition, we observed that running a single test case in a multi-module project may only rely on a compilation of a specific subset of modules rather than on the entire project. Hence, we improve test execution time to speed up the experiment by leveraging the *-pl* option with a comma-separated list of modules to remove the compilation of unnecessary modules. However, we may miss some dependent modules when using the `-pl` option. Hence, we use the options `-am` to build all the dependent modules of the specified modules. For example, if *module_A* depends on *module_B*, using `-am` will build both modules. Finally, *Maven* may run numerous static analyses in the default build, such as *License check* and *CheckStyle*, which are not required to execute test cases. We also remove these to improve test execution time. Finally, we run these options in the root directory to successfully execute a single inherited test case in a multi-module project.

Collecting Code Coverage. We use *JaCoCo* to generate the code coverage report at three levels, i.e., instruction coverage, branch coverage, and line coverage. *JaCoCo* is one of the most popular code coverage tools that instruments bytecode to trace test execution [20]. We integrate *JaCoCo* as a *Maven* plugin by configuring the *pom.xml* of the studied projects. While integration is simple for most *Maven* projects, for the multi-module *Maven* project, the coverage report is only limited to classes within the module, and will not be shown for test cases covering classes outside of the modules (e.g., integration test). Therefore, as some test cases cover multiple modules, we add an extra *report-aggregate* goal to the parent *Maven* build script (i.e., the main pom file).

3.3.2 Acquiring Test Oracle. In software testing, *Test Assertion* plays a critical role in assessing whether the actual behavior of the program aligns with the expected behavior specified by the developers (i.e., the test oracle) [8, 14, 15, 53]. If the observed values of the program state differ from the oracle, the test assertion fails, indicating that the program is incorrect. Test failures indicate software regression caused by buggy code introduced through developer modification. Hence, in addition to coverage, the quality of assertions becomes crucial in assessing the effectiveness of test cases in capturing faults within the source code [16, 21, 37]. Hence, to detect redundancies in test cases, we use both code coverage and test assertion, ensuring effective identification of redundant test executions. We instrument the test assertions to collect the state

Table 2: Systems Studied and Their Inheritance Statistics.

Project	#Inheritance tree	#Test classes constituted by inheritance tree	#Test classes within entire codebase
Commons-collections	10	206 (85%)	243
Zookeeper	6	301 (80%)	378
Avro	46	233 (49%)	478
Maven	13	83 (37%)	227
Shiro	10	45 (36%)	126
Commons-math	27	124 (31%)	400
Feign	6	30 (28%)	108
Iotdb	21	110 (25%)	437
Shenyu	7	133 (16%)	838
Dubbo	32	122 (14%)	856
Graphhopper	11	36 (14%)	252
Rocketmq	5	23 (11%)	214
Commons-lang	6	16 (10%)	161
Pdfbox	3	17 (8%)	213
Biojava	3	11 (4%)	259
Total	202	1,691 (31%)	5,322

of the program for both expected and actual behavior during test execution. Our test instrumentation is lightweight. We examine the static import of the studied systems to uncover all potential testing frameworks (e.g., JUnit/Hemcrest) that developers may use to write test cases. From these frameworks, we extract all the APIs used for assertion. During the test instrumentation, we identify such APIs and replace these APIs with `print` statements, which collect the program states during test execution. However, during our instrumentation, we uncovered that assertions are written in a variety of contexts in the test code. In particular, (1) The test cases may rely on reusable method which performs the oracle analysis. In this case, the test case itself may not have assertions. Hence, we leverage *Spoon* to instrument the entire codebase to collect more accurate test oracles.

3.3.3 Execution Time of our Technique. As discussed previously, our technique consists of three important steps: detecting 1) *Redundancy-Inducing Inheritance*, 2) *Inheritance-Induced Redundant Test Candidates*, and 3) *Inheritance-Induced Redundant Test Executions*. The execution time for steps 1) and 2) takes a few seconds to less than 3 minutes, which is relatively trivial. We do not consider the execution time for dynamic analysis since *JaCoCo* is third-party software. However, based on *JaCoCo* developers, the performance overhead is approximately a 10% increase from normal test execution time [18]. However, our static analysis help us locate inheritance-related issues and saves time for our dynamic analysis.

4 Studying Inheritance-Induced Test Case Redundancy

In this section, we first introduce the studied systems. Then, we study inheritance-induced test redundancy by answering four RQs.

Studied Systems. We conduct our analysis on 15 open-sourced systems. The selection of 15 systems was influenced by time constraints, as analyzing all 503 systems from Section 2 would have been time-consuming. To identify the 15 systems, we applied additional criteria to identify highly maintained systems: the presence of inheritance, usage of *JUnit* in the *Maven* configuration files, sufficient test files, containing commit activity between 2022-2023, high popularity (stargazer count > 600), and non-forked repositories. The criteria for systems to have inheritance is to ensure that redundancies are common issues for systems that involve inheritance. From the pool of systems, we randomly selected the following 15 systems: Commons-math, Commons-lang, Iotdb, Maven, Pdfbox, Shiro, Shenyu, Biojava, Rocketmq, dubbo, Avro, Zookeeper, Commons-collections, Feign, and Graphhopper. As shown in Table 2, these studied systems cover different domains, from distributed databases to stream processing frameworks, message brokers, and group chat servers. Table 2 also displays the number of test inheritance extracted from the studied systems using the technique from Section 3.1.2. The results reveal that despite the relatively small number of test inheritances tree in some systems, like *commons-collections* (i.e., 10 inheritances), their impact on the codebase was substantial, constituting 85% of the total number of test classes. In contrast, in systems like *Avro*, which had a higher number of test inheritances tree (i.e., 46), the percentage of impacted test classes was lower, at 49%. This finding suggests that even a few instances of test inheritance can substantially influence overall testing structure. It underscores the need to analyze the intricacies inheritance in each system to grasp the true impacts of test inheritance. In conclusion, the results highlight the diverse nature of test inheritance in different systems.

RQ1: How Prevalent are Inheritance-Induced Redundant Test Candidates?

Motivation. As discussed in Section 2, inheritance is widely adopted in practice, which raises an intriguing question about the number of test cases inherited from `superclasses` and whether test cases may become redundant. Particularly, when extensive test inheritance occurs, test cases may be inherited by numerous test `subclasses`, leading to challenges in understanding the test logic inherited from the `superclass` and the possibility of redundant test execution. This research question aims to investigate the occurrence of *Inheritance-Induced Redundant Test Candidates* to shed light on the implications of test inheritance in real-world projects. By exploring these test case relationships, we aim to gain valuable insights into the impact of inheritance in software testing.

Approach. In Section 3.1, we presented our static analysis approach to identify *Inheritance-Induced Redundant Test Candidates*. These candidates arise when two or more `subclasses` inherit the same test cases from the `superclass`, leading to the execution of inherited test cases multiple times (may or may not be redundant). To assess the prevalence of these candidates, we compare them with all executable test cases in the studied systems, using the *mvn-surefire* test execution strategy by following the approach that is described in Section 3.3.

Result. ***On average, inheritance-induced redundant test candidates account for approximately 13% of the total executable test cases.*** This finding is based on a comparison of the number of *Inheritance-Induced Redundant Test Candidates* against the total number of executable test cases, which is reported in the last column, i.e., # Discovered Candidates, as shown in Table 3. Specifically, out of the 40,420 executable test cases in the examined systems, 5,080 (13%) are attributed to potential redundancies, specifically through test inheritance. This indicates a potentially significant impact of test inheritance on the overall testing efforts. For instance, Table 3 highlights that 50% of the test cases in *Commons-C.* are contributed through inheritance, followed by 21% in both *Commons-M.*

Table 3: Revealing the landscape of redundant test candidates: A summary of the static analysis result.

	Inheritance-Induced Redundant Test Candidates						
	#Unique Test Cases	#Occurrence in various subclasses				#Discovered	#Total Executable
Project	Defined in `Superclass`	Mean	Max	Min	Multiplier	Candidates	Test Cases
commons-collections	230	14	93	2	14x	3167 (50%)	6367
commons-math	251	3	17	2	3x	866 (21%)	4036
feign	69	4	7	2	3x	299 (21%)	1441
shiro	14	5	11	2	5x	65 (8%)	822
dubbo	66	4	6	2	4x	236 (7%)	3626
biojava	30	3	3	2	3x	86 (6%)	1436
avro	31	4	5	2	4x	122 (3%)	3984
graphhopper	44	2	3	2	2x	98 (3%)	3347
maven	18	2	2	2	2x	36 (3%)	1058
shenyu	1	26	26	26	26x	26 (3%)	874
zookeeper	23	2	2	2	2x	46 (1%)	3156
pdfbox	4	3	4	2	3x	12 (1%)	1905
iotdb	5	2	2	2	2x	10 (1%)	1618
commons-lang	3	3	3	3	3x	9 (<1%)	6137
rocketmq	1	2	2	2	2x	2 (<1%)	613
Total	790	N/A	N/A	N/A	6x	5080 (13%)	40420

and *Feign*. The results highlight the high prevalence of *Inheritance-Induced Redundant Test Candidates* among the test cases. While the average percentage of *Inheritance-Induced Redundant Test Candidates* may seem modest at 13%, the presence of such test cases is not negligible, given the large number of test cases in the examined systems. Furthermore, certain systems, such as *Commons-C.*, show a remarkably high percentage of inherited test cases, highlighting the importance of examining inheritance relationships and their impact on testing.

Initially, redundant test candidates come from 790 unique test cases. However, through inheritance the number of redundant candidates can increase sixfold. We also analyze the number of different `subclasses` from which the redundant candidates can be inherited. For instance, if a *Inheritance-Induced Redundant Test Candidates* test case is inherited from two `subclasses`, then we consider this test case to be multiplied two times through inheritance. We depict this multiplier in column 8 (e.g., Multiplier) in Table 3. Furthermore, we provide three summary statistics (mean, max, min) to show the diversity of inheritance. Hence, Table 3 shows that *Commons-C.* has 230 unique test cases that are defined in `superclasses`. These test cases undergo inheritance through an average of 14 `subclasses`, with a maximum of 93 `subclasses` and a minimum of 2 `subclasses`. Through various inheritance practices, the number of redundant candidates then multiplies by 14x, increasing to 3167 test cases. Notably, all systems initially have a smaller subset of test cases (790). However, through inheritance, the total number of potential redundant candidates can increase sixfold (to 5,080). These initial findings underscore the potential of identifying and addressing redundant test candidates to optimize testing resources.

Answers to RQ1. We discovered that 13% of the total executable test cases are *Inheritance-Induced Redundant Test Candidates*. These instances of redundancies are primarily caused by a small subset of test cases, but their occurrence increases six-fold through the inheritance process.

RQ2: Are the Inheritance-Induced Redundant Test Candidates Truly Redundant?

Motivation. In RQ1, based on our static analysis results, we observed that systems that utilize inheritance consistently exhibit potential redundant candidates. However, assessing the redundancy of a test case solely through static analysis presents challenges. For example, consider a scenario where two subclasses inherit the same test case. Due to program dependencies, each subclass may have different execution contexts (e.g., through test fixtures [24]) of the test case differently; thus, these test cases cannot be considered redundant. Hence, in this RQ, our objective is to delve deeper into the true redundancy of these uncovered redundant candidates by conducting a comparative analysis of their code coverage and test oracles. By examining these quality attributes, we aim to gain comprehensive insights into the true redundancy of these test cases. We believe that this investigation will contribute significantly to enhancing the understanding of the effectiveness of these redundant test candidates and their overall impact on testing quality.

Approach. We define test case redundancy as the condition where the coverage and test oracles are identical. We collect coverage and oracle following the approach from Section 3.3.1. For complete coverage comparison, we compare branch, line and instruction. We use Algorithm 1, to identify truly redundant tests from the initial ***Inheritance-Induced Redundant Test Candidates***. Note that, employing dynamic analysis to compare all test executions can be resource-intensive, as not all redundant test cases are induced by inheritance, which makes our static analysis an important intermediate step to focus on *Inheritance-Induced Redundant Test Executions*. The algorithm proceeds through three key steps. In step ①, a set of redundant test candidates, along with their corresponding coverage and oracle information, is provided as input. In step ②, as shown in *line 4*, the algorithm generates all possible pairwise combinations of the redundant candidates. For each pair, it compares both their coverage and oracle information. A pair is deemed redundant only when both coverage and oracle are found to be equivalent. This comparison is represented by a triplet, denoted as $< Test1, Test2, Boolean >$, where the boolean value is `True` if and only if both coverage and oracle are equal, and `False`

otherwise. Given that pairwise comparisons are employed among the redundant candidates, the number of comparisons performed follows the formula $\frac{n!(n-r)!}{r!}$. In step ③, as shown in *line 5*, we use *union-find* [51] algorithm to establish the connected component relationships within the pairwise comparisons. The output of the relationship is denoted as *Group*. In step ④, as shown in *line 7-12*, we examine the *Group* and flag component that has at least two candidates to be redundant, i.e., have the same code coverage and oracle. For example, given $< A, B, False >$, $< A, C, False >$, and $< B, C, True >$, the algorithm will flag that the presence of A always leads to *False*, indicating that B and C are redundant, while A is non-redundant.

Algorithm 1 Redundancy Analysis

Input: Array `Coverage`, Array `Oracle`
Output:

```
1:  Global Var1: RedundantTest
2:  Global Var2: noRedundantTest
3:  procedure findRedundantTest(Coverage,Oracle)
4:      Pairs ← doPairWiseCombination(Coverage, Oracle)
5:      Group ← unionFind(Pair)
6:      for group in Groups do
7:          if len(group) > 1 then
8:              RedundantTest ← group
9:          else
10:             noRedundantTest ← group
11:         end if
12:     end for
13: end procedure
```

Result. ***45% of identified redundant test candidates are truly redundant.*** Table 4 provides an overview of the identified redundant test executions among the studied redundant test candidates. We uncover that the majority of systems exhibit redundant tests, with around 50% or more of the candidates falling into redundant tests in most systems. Notably, *Commons-C.* initially contained the highest number of redundant test candidates. However, through redundancy analysis, a significant portion of these candidates (86%) were identified as non-redundant, leaving only 14% as truly redundant. This may be related to developers' design and usage of inheritance in test cases. For example, in *Commons-C.* developers commonly use test inheritance to help test diverse implementations of different algorithms, e.g., sorting algorithms. Namely, these sorting algorithms share the same test setup, i.e., create new lists, prior to testing the sorting algorithm. Developers use test inheritance to reuse code and avoid duplication.

We uncover that on average 45% of the redundant test candidates are truly redundant when considering both code coverage and test oracles, while the remaining 55% demonstrate differences in either code coverage, test oracles, or both, making them non-redundant. These results yield two important insights: Firstly, the significant presence of redundancy (45%) among the identified candidates of redundant test cases suggests opportunities for eliminating tests that may not contribute to fault localization. Secondly, from another perspective, the presence of a significant number (55%) of non-redundant test cases highlights the potential benefits of test case inheritance in not only facilitating code reuse but also diversifying code coverage and assertions, enhancing testing practice.

Table 4: The prevalence of redundant tests and their impact on test execution time. *Candidates* **refers to** ***Inheritance-Induced Redundant Test Candidates*** **and** *Redundant* **refers to true redundant test cases.**

Project	# Test Execution		Test Execution Time (seconds)	
	Candidates	Redundant	Total Tests	Redundant
Feign	291	291 (100%)	183.996	146.428 (79.6%)
Commons-C.	2677	385 (14%)	16.076	4.373 (27.2%)
Avro	106	87 (82%)	215.396	36.988 (17.2%)
Shiro	65	30 (46%)	86.845	13.696 (15.8%)
Commons-M.	817	415 (51%)	75.371	5.377 (7.1%)
Rocketmq	2	2 (100%)	267.950	1.974 (0.7%)
Pdfbox	12	8 (67%)	66.070	0.257 (0.4%)
Biojava	86	63 (73%)	774.207	2.615 (0.3%)
Iotdb	10	4 (40%)	1362.134	3.540 (0.3%)
Maven	36	18 (50%)	48.927	0.103 (0.2%)
Dubbo	201	99 (49%)	1322.386	0.493 (0.1%)
Commons-L.	9	0	N/A	N/A
Graphhopper	92	0	N/A	N/A
Shenyu	26	0	N/A	N/A
Zookeeper	42	0	N/A	N/A
Average	% Redundancy in Candidates		% Contribution to Execution Time	
	45%		14%	

Answers to RQ2. We uncover that 45% of the redundant test candidates are truly redundant tests, whereas the remaining test cases facilitate diversification of coverage and assertion oracle.

RQ3: How Much Time does Inheritance Induced Redundant Test Contribute to the Overall Test Execution Time?

Motivation. As seen in RQ2, there is a large occurrence of redundant test cases exhibiting identical coverage and oracle results. Considering the prevalence of these redundant tests, it is important to investigate their impact on the overall test execution time and to what extent to which these redundant test cases prolong the testing process.

Approach. Following Section 3.3, we employed *Maven-Surefire* to execute the redundant test cases and collect the test execution time. Specifically, we use `mvn clean test` to run all the test cases and `mvn clean test -DTest=RedundantCandidates` to sequentially run redundant test cases for each studied project within a single JVM. Note that the test execution time excludes the time for code compilation. To enhance the reliability of our findings, we conducted data collection five times and calculated the average results.

Result. ***On average, the detected inheritance induced redundant tests contributes to 14% of the total test execution time.*** Table 4 provides detailed insights into the execution time of redundant tests and the total number of tests for different studied systems. Among the studied systems, 11 out of 15 systems contained redundant test cases. The presence of these redundant tests had a large impact on the overall test execution time, constituting approximately 14% of the total time. As anticipated, the extent to which redundant test cases contributed to the execution time was

closely related to their proportion within the total number of test cases. For example, systems such as *Feign* and *Commons-C.* exhibited a higher overall execution time due to a substantial number of redundant test cases. Interestingly, for *Commons-C.*, although it has many redundant candidates (e.g., 2677), it has fewer truly redundant tests (e.g., 385), which is the opposite for other systems like *Feign*, where 100% of candidates are truly redundant. More importantly, we find that while *Commons-C.* has a lower percentage of truly redundant tests (e.g., 17.2%) compared to *Commons-M.* and *Avro*, its redundant test execution contributes much more to the overall execution time. Therefore, the impact of redundant tests depends on the test design, and some systems may be more affected by test execution time. This finding also underscores the potential benefits of removing redundancy, as it has the potential to significantly improve testing resources.

Answers to RQ3. We uncover that 14% of total test execution time is spent on redundant test cases that do not provide any additional benefits.

Discusson. Expanding upon the findings of our RQ3, it becomes evident that using inheritance in the test code leads to an increased occurrence of redundant test execution. Such redundancies are not immediately noticeable in the test code, as they can be inherited from its superclass. Our methodology allows for the detection of such redundancies caused by inheritance, providing developers with awareness of potential bottlenecks in test code. Developers can bypass these redundancies by using the build system (Maven) to exclude test cases from execution. Another bypassing strategy is to override the inherited test cases with another test case in the subclass annotated with @Ignore (Apache Ignite - 63b9e1653d), as often shown in prior work [25]. However, these strategies do not completely remove redundancies in the test code; they only skip their execution, overlooking the complexity of redundancy removal. In RQ4, we elaborate on the complexity of removing redundancies related to inheritance test cases.

RQ4: Assessing the Feasibility of Reducing Inheritance-Induced Redundant Test Execution

Motivation. In prior RQs, we uncovered many test redundancies. Consequently, the next natural step is to eliminate these redundancies that do not contribute effectively to fault localization capabilities in order to improve test execution time. While developers could temporarily bypass such tests, the removal of redundant test cases within an inheritance context presents a more significant challenge. As observed in RQ1, in extreme cases, a test case can be inherited and executed as many as 96 times, demonstrating complex coupling and making the task of redundancy removal challenging. Hence, in this RQ, we conduct an empirical analysis to understand the feasibility of removing redundant test cases in inheritance. Our aim is to provide insights to aid future research in the development of test case minimization tools.

Approach. We conduct a feasibility analysis because, unlike previous works on test case minimization that typically involve straightforward removal of redundant test cases [10, 28, 36], the scenario of test case redundancies related to inheritance contains complex coupling and necessitates careful refactoring decisions. Hence, we conduct two analysis to study the challenges of removing inheritance induced redundant test cases. We list them below:

A. Can inherited test cases become both redundant and non-redundant test executions?

B. How far are the redundant test cases from their definition of the superclasses in the inheritance trees?

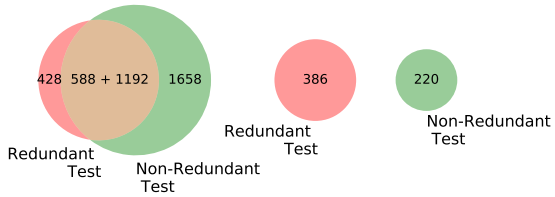

Figure 3: Overlap between Redundant and Non-redundant Tests in Test Execution. The overlapping region (orange) indicates that an inherited test case is redundant in one `subclass` but non-redundant in another `subclass`. The one figures on the left correspond to five systems, while the two figures on the right correspond to the rest.

RQ4(A): Can inherited test cases contain both redundant and non-redundant groups of tests?

Motivation. In RQ3, We uncovered that 45% of identified redundant candidates are truly redundant, whereas the remaining is non-redundant, i.e., through code coverage or test oracle. In this RQ, we hypothesize that it is possible for inherited test cases to be redundant in one context, i.e., redundant in one `subclass` but not redundant in another `subclass`. The existence of such a complex scenario will give us an initial glimpse of the challenges for efficient test case minimization.

Approach. We modify Algorithm 1 to check if the equivalent group contains both redundant test cases and non-redundant test cases resulting from the same *Inheritance-Induced Redundant Test Candidates*, and denote this as *Co-existence* group.

Result. ***Out of 1,402 detected redundant tests, 588 (41%) test cases co-occurs with non-redundant tests.*** As illustrated in Figure 3, inherited tests can result in both redundant and non-redundant test executions. In other words, inheriting a test case results in redundancies in one `subclass`, but not in another `subclass`, due to the different execution contexts specified by developers, e.g., through test fixtures. Specifically, our analysis reveals that among the 1,402 detected truly redundant test cases, 588 test cases actually co-exist with their non-redundant test case counterparts. Notably, 5 out of 15 studied systems (i.e., *Avro*, *Commons-C.*, *Commons-M.*, *Dubbo*, and *Shiro*) encompass this co-existence of redundancy and non-redundancy in the inherited test cases. This suggests that even for the same test case defined in a superclass, inheritance of this test case does always cause redundancy, and some may be utilized in different `subclass` contexts (e.g., to ease maintenance and improve coverage). However, whether test inheritance is beneficial to test design remains a future research problem. While it may improve code coverage, it can also increase code complexity, which may become difficult to maintain in the long run. Nonetheless, the variability in the nature of redundancy may be related to the design of

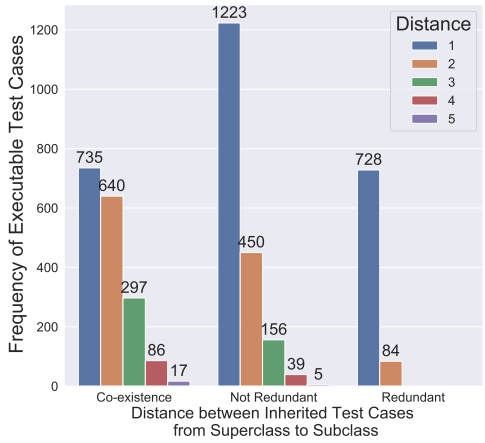

Figure 4: Analysis of distance in the inheritance tree between the parent test case and the child test.

test cases in different systems. These findings highlight a complex scenario for effective test case minimization.

RQ4(B): How far are the redundant test cases from their definition of the superclasses in the inheritance trees?

Motivation. As seen in RQ4(A), *Inheritance-Induced Redundant Test Candidates* can be found in many different `subclass` contexts, co-existing with non-redundant test cases. In this RQ, we delve into the distance of these executable test cases in the `subclass`, i.e., where the test case is executed, from their `superclass`, i.e., where the test case is declared. Analyzing class distance will reveal the potential existence of complex hierarchical relationships, i.e., how many `subclasses` do these inherited test cases impact through inheritance? Analyzing such distance is beneficial to understand the challenges of removing redundant tests that impact multiple classes.

Approach. We investigate class distance in all of the *Redundancy-Inducing Inheritance* (e.g., 4,472 test candidates), including redundant test, non-redundant test, and co-existence of both. To analyze class distance, we leverage our inheritance tree from Section 3.1.2. We use the *shortest-path* algorithm [50] to find the path it takes to reach the impacted `subclass` from a `superclass`.

Result. ***Inherited test cases that lead to both redundant and non-redundant test executions in*** `subclasses` ***exhibit a highly variable number of inheritance distance from*** `superclass`**.** As shown in Figure 4, majority of redundant tests (728/812 - 90%) are executed by the direct `subclass`, whereas remaining 10% have executions that executed by two `subclasses` downstream. The finding shows that these redundant test cases may be easier to resolve. Interestingly, we find that for tests that co-exist with non-redundant tests, there are more diverse sets of class distances. In particular, some non-redundant tests may be executed up to five class distances in the downstream `subclass`. Namely, for systems *Avro*, *Commons-C.*, *Commons-M.*, *Dubbo*, and *Shiro*, which contains co-existence of redundant and non-redundant tests, there is a more complex inheritance distance. This reveals that inheritance relationships within *Inheritance-Induced Redundant Test Executions* may have significant variability in hierarchical structures. Specifically, it is possible that test cases designed with such complexity in class distance are less likely to be redundant, as they impact a higher number of `subclasses`, whereas much simpler class hierarchies have a higher tendency to be redundant. Nonetheless, the presence of a complex hierarchy constitutes additional complexity that makes test case minimization challenging, as it may impact many downstream `subclasses`.

Answers to RQ4. Removing redundant tests need careful preservation of code coverage. This is particularly important when dealing complex inheritance relationships, where co-existence of both redundant and non-redundant can contribute to code coverage and may impact multiple classes.

Discusson. Expanding upon the findings of our study, which demonstrate a significant overlap between redundant and non-redundant test cases, it becomes evident that the removal of redundant test cases, as often seen in traditional test case reduction strategies, may not represent a valid strategy for inherited test cases. Our results underscore the complexity of the issue. In other words, while redundancies must be addressed, it is also apparent that certain test cases leverage inheritance to enhance coverage and assertions, indicating their value in the testing process. Consequently, this raises an interesting question: How can we reorganize the test interfaces to reduce redundancies and improve test maintainability? Namely, our results show the possibility of exploring higher-level architectural refactoring to enhance test quality. Nevertheless, our approach provides initial insights to eliminate the impact on test execution overhead.

5 Implications & Future Works

Based on our empirical findings, we present actionable implications and future research directions for researchers and practitioners.

5.1 Implication for Researchers.

Future research should explore test removal while preserving code coverage, as inheritance-induced redundant test cases may overlap with non-redundant tests. While our analysis revealed many *Inheritance-Induced Redundant Test Executions*, in RQ4 we also found a 41% overlap with tests that contribute to increased/different code coverage. This presents a challenge in determining how to remove redundant test executions while preserving non-redundant test cases that aim to increase code coverage. The co-existence of redundant and non-redundant tests complicates test case reduction, as both types of tests serve different purposes. Redundant test cases may increase execution time and hinder fault localization capability, while non-redundant test cases play a role in increasing code coverage. Hence, future research is necessary to comprehend the trade-offs associated with using inheritance to achieve code coverage and its potential increase in redundancy.

Further research may investigate trade-offs between using inheritance to make tests reduce maintenance cost and not using inheritance to reduce test case redundancies. While inheritance in test [17, 43, 45] is a controversial practice, we find that 40% amongst 503 sampled systems utilize inheritance in test code, which is widely adopted in practice. In particular, projects like Commons-collections and Commons-math, despite their heavy reliance on inheritance, exhibit fewer redundancies, hinting at the compactness and superior quality of their tests. This opens the

door to future research avenues, exploring the trade-offs between employing inheritance and abstaining from it. Future research may also delve into quality attributes, such as the time required for activities like bug fixing, coverage enhancement, and feature addition, comparing tests that employ inheritance to those that do not. Moreover, for test cases that result in redundancies, future studies may also investigate how they manifest in the code and provide preventative measures.

In general, future research is needed to understand how to remove complex inheritance relationships in the test code. In RQ4, we revealed that redundant tests may exhibit complex inheritance hierarchy relationships. Removing redundant tests in such scenarios poses a challenge, as redundancy impacts multiple class relationships. Further research is needed to explore effective strategies and tools to refactor these complex inheritance relationships in general, which may also help remove inheritance-induced redundant test cases. Namely, our paper show the possibility of exploring higher-level architectural refactoring to enhance test quality.

We uncovered the widespread existence of inheritance-induced redundant test cases. How these test cases impact fault localization can be further explored in future research. As redundant test cases are inherited from other test classes, test failures may be difficult to localize using fault localization. For instance, in *Apache-Avro*, the *TestProtocolSpecific* class contains 15 test cases that are inherited and executed by five different *subclasses*. Interestingly, all 15 test cases fail in one *subclass* while passing in the remaining four, which might be due to specific bugs associated with the test setup in that particular *subclass*. As these failures are not indicative of source code defects, they could potentially mislead developers and fault localization algorithms, which attempt to localize source code defects [52], causing them to identify faults incorrectly. We encountered a similar scenario in *AbstractOrderedBidiMapDecoratorTest* from *Commons-collections*. Considering that *Commons-collections* is part of the *defects4j* benchmark and contains many inheritance-induced dependencies, future studies could also investigate how eliminating such redundancies can improve fault localization techniques focus on distinct failure.

5.2 Implication for Practitioners.

Practitioners need better support for detecting repetitive test candidates. Inheritance is a double-edged sword, while it may improve test compactness and maintainability, it can also introduce test case redundancies. For example, as seen in RQ2, while many redundant test cases are caused by inheritance, they are related to a small subset of parent test cases. Furthermore, some test cases may repeat up to 93 times due to inheritance. Therefore, it would be beneficial to raise awareness among developers about these issues. Future work should provide tools to assists developers to be aware of the redundant test cases.

6 Related Work

Inheritance Evolution and Maintenance. Many works investigated the evolution of inheritance in source code. For example, Shaheen and du Bousquet [42] 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. [32] 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. [33] 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. [12] 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.

Inheritance Maintenance in Test Code. Limited works investigated the evolution and maintenance of inheritance in the test code. The work by Wang et al. [49] 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. In contrast, our analysis shows the existence of inheritance in the test case design and its implication on test execution overhead. There is another body of work relevant to our work. Peng et al. [40] 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. While their work is the most relevant to our work, they emphasize test dependencies and little on the impact of test code redundancies.

Test Case Minimization/Reduction. Our work is related to research in test case reduction/minimization, which focuses on eliminating redundant test cases while preserving fault detection capability. Nadeem et al. [31] developed `TestFilter` that uses the statement-coverage criterion for the reduction of test cases. Fang and Lam [11] used assertion fingerprints to detect similar test cases that can be refactored into one single test case. Alipour et al. [3] presented an approach that reduces a test suite by compromising a certain amount of coverage while preserving the overall fault-finding ability. Our work, on the contrary, is more related to the design of the test code, which makes test case removal non-trivial. Complementary to test case reduction, other works focus entirely on reducing time execution of test executions [13, 23, 26]. Our work, on the contrary, focuses on identifying redundant test cases, which also help reduce test execution time. Moreover, the issues of redundancy still exist in these works. Vahabzadeh et al. [48] performed fine-grained test case minimization by merging all test cases that have the same code coverage. However, due to the complexity of inheritance relationships, they do not fully explore inheritance in their study. Our work, on the contrary, not only detect redundancy but can also point out the underlying causes (i.e., inheritance).

7 Threats to Validity

Internal Validity. Firstly, our findings depend on the accuracy of the third-party tool (e.g., *spoon*) to mine *Redundancy-Inducing Inheritance* and *Inheritance-Induced Redundant Test Candidates* in the source code and also the accuracy of the dynamic analysis tool (e.g., *Jacoco*) to execute *Inheritance-Induced Redundant Test Executions*. It is important to note that validating the precision of these third-party tools is not within the scope of our responsibility, However, both *spoon* and *Jacoco* are widely used in prior research and in practice and we did not find any false positives during our manual examination of the results.

External Validity. Our studied systems are all open source systems implemented in Java, so the result may not be generalized to all systems. However, to minimize the threat, we follow a set of criteria to popular systems from various domains, large in scale, and actively maintained. Within this criteria we randomly sample 15 studied systems to obtain diverse studied systems. However, we acknowledge that the three projects containing over 85% of the redundancy candidates might indicate a concentration of the issue in certain projects. Nevertheless, the representatives of the entire open-source Java project ecosystem is a complex matter. Our intent was not to claim that this issue is uniformly distributed across all projects but to highlight that our findings are dependent on how different systems use inheritance in their test code design. Hence, our work may not be applicable to all systems, and the impact may be more significant in larger projects. Although our tool is designed for analyzing Java systems, we have made our source code available, where our implementations may inspire writing similar analyses for other programming languages. We encourage future studies to replicate our experiments on other systems and projects implemented in different programming languages.

Construct Validity. Our dynamic analysis encountered some test failures and environmental errors, resulting in un-executed test cases and potentially under-representing our analysis for *Inheritance-Induced Redundant Test Executions*. However, the number of un-executed tests is small. There may be bugs in the tools that we use. For example, prior to *JaCoCo* version 0.8.10 (i.e., most updated version), the *report-aggregate* plugin contained a bug where it only collects coverage of dependent module except for its current module [19]. We noticed the issue and migrated to the fixed version of Jacoco. However, there may still be undiscovered bugs in the tools that can affect the results. Our technique leverage functionality from third-party software, such as *Spoon* and *Jacoco*. We leverage *Spoon* to extract *Redundancy-Inducing Inheritance* and *Inheritance-Induced Redundant Test Candidates*, whereas we leverage *Jacoco* to identify *Inheritance-Induced Redundant Test Executions*. Moreover, for extracting assertion of the test cases we also rely on the *Spoon* API. It is important to note that validating the precision of these third-party tools is not within the scope of our work. However, our manual investigation of the results from *Redundancy-Inducing Inheritance* achieved 100% precision.

8 Conclusion

This paper presents the first empirical study on test case redundancy caused by inheritance. We propose a hybrid approach that combines static and dynamic analysis to detect and verify inheritance-induced redundant test cases. We apply our approach to 15 open-source Java systems. We find that (1) Despite controversies surrounding test inheritance, non-negligible tests (14%) of test case executions are redundant. (2) The redundant test cases take, on average, 13% of the total execution, which adds additional test execution overhead. (3) Many inherited test cases (40%) are redundant in some *subclass* but non-redundant in others, making it difficult to eliminate redundancy while preserving code coverage. This complexity calls for careful refactoring decisions to address the issue effectively. Finally, we also discuss challenges and future research directions on resolving inheritance-related issues.

References

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439* (2017).

[2] Haldun Akoglu. 2018. User's guide to correlation coefficients. *Turkish journal of emergency medicine* 18, 3 (2018), 91–93.

[3] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. 2016. Evaluating non-adequate test-case reduction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 16–26.

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

[5] Robert Biddle and Ewan Tempero. 1996. Explaining inheritance: A code reusability perspective. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*. 217–221.

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

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

[8] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A snowballing literature study on test amplification. *Journal of Systems and Software* 157 (2019), 110398.

[9] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2013. Coverage-based test case prioritisation: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 302–311.

[10] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 (2015), 371–396.

[11] Zheng Felix Fang and Patrick Lam. 2015. Identifying test refactoring candidates with assertion fingerprints. In *Proceedings of the Principles and Practices of Programming on The Java Platform*. 125–137.

[12] Giammaria Giordano, Antonio Fasulo, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. 2022. 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)*. IEEE, 947–958.

[13] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 243–252.

[14] Neha Gupta, Arun Sharma, and Manoj Kumar Pachariya. 2019. An insight into test case optimization: ideas and trends with future perspectives. *IEEE Access* 7 (2019), 22310–22327.

[15] Soneya Binta Hossain, Matthew B Dwyer, Sebastian Elbaum, and Anh Nguyen-Tuong. 2023. Measuring and Mitigating Gaps in Structural Testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1712–1723.

[16] Soneya Binta Hossain, Matthew B Dwyer, Sebastian Elbaum, and Anh Nguyen-Tuong. 2023. Measuring and Mitigating Gaps in Structural Testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1712–1723.

[17] https://lists.apache.org/list.html?dev@maven.apache.org. 2023. *https://lists.apache.org/list.html?dev@maven.apache.org*. https://lists.apache.org/thread/cpm046p745j7nj0dvw9mtxfmthkgobp6

[18] Jacoco. 2003. *[java code coverage] Reasons for huge performance impact*. Retrieved March 2, 2005 from https://jacoco.narkive.com/adoFZzfD/java-code-coverage-reasons-for-huge-performance-impact#:~:text=Usually%20the%20performance%20impact%20of,so%20a%20factor%20of%2010.

[19] Jacoco. 2013. *Add parameter to include the current project in the aggregated report*. https://github.com/jacoco/jacoco/pull/1007

[20] Jacoco. 2023. *JaCoCo Java Code Coverage Library*. Retrieved March 26, 2023 from https://www.jacoco.org/jacoco/

[21] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.

[22] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[23] Shadi Abdul Khalek and Sarfraz Khurshid. 2011. Efficiently running test suites using abstract undo operations. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 110–119.

[24] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Peter Chen, and Jinqiu Yang. 2021. Studying Test Annotation Maintenance in the Wild. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. 62–73.

[25] Dong Jae Kim, Bo Yang, Jinqiu Yang, and Tse-Hsun Chen. 2021. How disabled tests manifest in test maintainability challenges?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1045–1055.

[26] Jinhan Kim, Junhwi Kim, and Shin Yoo. 2017. GPGPGPU: Evaluation of parallelisation of genetic programming using GPGPU. In *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*. Springer, 137–142.

[27] Patipat Konsaard and Lachana Ramingwong. 2015. Total coverage based regression test case prioritization using genetic algorithm. In *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE, 1–6.

[28] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. 417–420.

[29] Cristina Marinescu and Mihai Codoban. 2014. 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)*. IEEE, 246–253.

[30] Scott McMaster and Atif Memon. 2007. Fault detection probability analysis for coverage-based test suite reduction. In *2007 IEEE International Conference on Software Maintenance*. IEEE, 335–344.

[31] Aamer Nadeem, Ali Awais, et al. 2006. TestFilter: a statement-coverage based test case reduction technique. In *2006 IEEE International Multitopic Conference*. IEEE, 275–280.

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

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

[34] Oracle. 2022. *Inheritance*. https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

[35] Oracle. 2022. *Polymorphism*. https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

[36] Rongqi Pan, Taher A Ghaleb, and Lionel Briand. 2023. ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1700–1711.

[37] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[38] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[39] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[40] Zi Peng, Tse-Hsun Chen, and Jinqiu Yang. 2020. Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies. *IEEE Transactions on Software Engineering* (2020).

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

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

[43] stackoverflow. 2013. *Prefer composition over inheritance?* https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance

[44] stackoverflow. 2013. *why inheritance is strongly coupled where as composition is loosely coupled in Java?* https://stackoverflow.com/questions/19146979/why-inheritance-is-strongly-coupled-where-as-composition-is-loosely-coupled-in-j

[45] stackoverflow. 2020. *How can I resolve this redundancy caused by inheritance and nested class?* https://stackoverflow.com/questions/56063575/how-can-i-resolve-this-redundancy-caused-by-inheritance-and-nested-class

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

[47] Amjed Tahir, Steve Counsell, and Stephen G MacDonell. 2016. An empirical study into the relationship between class features and test smells. In 2016 23rd Asia-Pacific Software Engineering Conference (APSEC).

[48] Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. 2018. Fine-grained test minimization. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 210–221.

[49] Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. 2021. 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*. 540–552.

[50] Wikipedia. 2023. *Dijkstra's algorithm*. Retrieved 17 July 2023 from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[51] Wikipedia. 2023. *Disjoint-set data structure*. Retrieved 17 July 2023 from https://en.wikipedia.org/wiki/Disjoint-set_data_structure

[52] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[53] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

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