# Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies

Zi Peng,Tse-Hsun (Peter) Chen, *Member, IEEE,* and Jinqiu Yang, *Member, IEEE*

**Abstract**—In continuous testing, developers execute automated test cases once or even several times per day to ensure the quality of the integrated code. Although continuous testing helps ensure the quality of the code and reduces maintenance effort, it also significantly increases test execution overhead. In this paper, we empirically evaluate the effectiveness of test impact analysis from the perspective of code dependencies in the continuous testing setting. We first applied test impact analysis to one year of software development history in 11 large-scale open-source systems. We found that even though the number of changed files is small in daily commits (median ranges from 3 to 28 files), around 50% or more of the test cases are still impacted and need to be executed. Motivated by our finding, we further studied the code dependencies between source code files and test cases, and among test cases. We found that 1) test cases often focus on testing the integrated behaviour of the systems and 15% of the test cases have dependencies with more than 20 source code files; 2) 18% of the test cases have dependencies with other test cases, and test case inheritance is the most common cause of test case dependencies; and 3) we documented four dependency-related test smells that we uncovered in our manual study. Our study provides the first step towards studying and understanding the effectiveness of test impact analysis in the continuous testing setting and provides insights on improving test design and execution.

**Index Terms**—empirical study, test smells, continuous testing, test impact analysis

## 1 Introduction

Continuous integration (CI) is widely used in modern software development. The CI practice integrates developers' code changes to a central code repository once or even several times per day. Such frequent code integration reduces software maintenance overheads and allows developers to provide the latest working software to customers.

To ensure the quality of the integrated code, developers need to run a set of test cases for each code integration (i.e., CI build) in a continuous fashion – called continuous testing. However, running test cases is time-consuming and requires a significant amount of computing resources. To reduce testing overhead, prior studies have proposed techniques to reduce the test cases that need to be executed [1, 2, 3, 4, 5, 6]. In particular, Orso *et al*. [7] and Legunsen *et al*. [8] found that, by analyzing the static class dependencies between test cases and source code files, we can effectively identify and only execute the test cases that are *"impacted"* by the code changes (i.e., have dependencies with the changed code) to reduce testing overhead. In this paper, we call such techniques test impact analysis [9].

Although prior studies have shed light on the potential of test impact analysis, its effectiveness is closely related to the design of test cases; and in particular, code dependencies. Due to the frequent code changes and increased system complexity, the maintenance and quality of test cases may degrade. There may be a certain degree of dependencies between test cases and source code files, or among test cases. We define the degree of dependencies as the number of dependencies between a test case and other source code files, among test cases, or between a source code file and other test cases. Unlike traditional software development where test cases are executed once in a while, having a high degree of code dependency may have a larger accumulated testing overhead in CI due to frequent code integration and testing. A high degree of dependencies (i.e., a test case has dependencies with multiple source code files or other test cases, or a source code file is tested by multiple test cases) may reduce the effectiveness of test impact analysis and increase the difficulty of test maintenance.

To better understand the effectiveness of test impact analysis in CI settings and provide insights on improving the modularity of test case design, we study CI test cases (i.e., test cases that are executed as part of the CI process) by analyzing the code dependencies in 11 large-scale open-source systems. We study code dependencies from two different perspectives: CI execution in relation to test impact analysis, and CI test case design. To study the effect of code dependencies on test impact analysis, we follow prior studies on static test impact analysis [7, 8] to uncover the class dependency graph and identify the percentage of test cases that are impacted (i.e., need to be executed) by the changed code. If there is a high degree of dependencies between test cases and source code files, the number of impacted test cases would be high; and thus, the effectiveness of test impact analysis will be impacted. We analyze the daily code changes (i.e., commits) in the studied systems for a period of 12 months. We found that daily changes on a relatively small number of files (median values range from 3 to 28 files

- *Z. P, T. Chen and J. Yang are with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada.*
  *E-mail: zi_peng,peterc,jinqiuy@encs.concordia.ca*

per day) can impact (i.e., have dependencies with) around 50% or more of the test cases.

To study code dependencies in relation to test case design, we analyze the dependencies between test cases and source code files (and vice versa), and the dependencies among test cases. We found that most test cases cover the integrated behaviour of the software (i.e., 75% to 97% of the test cases cover multiple source code files), and there exists a certain degree of dependencies among test cases (i.e., 18% of the test cases have dependencies with other test cases). Finally, we conducted a qualitative study on the reasons that cause the dependencies among test cases. We uncovered four dependency-related test smells that may negatively affect test maintenance and execution.

The paper makes the following contributions:

- We found that the number of daily changed files is often small (median range values range from 3 to 28 classes across the studied systems). However, most of the test cases (around 50% or more) have dependencies with the modified files and may need to be re-executed in every build.
- The studied test cases often focus on testing the integrated behaviour of the system. On average, 15% of the test cases have dependencies with 20 or more source code files. We also found that the source code files in a test case often belong to different packages (i.e., cover various business logics).
- We found that, on average, 18% of the test cases have dependencies with other test cases. Our manual study found that most dependencies are caused by four reasons: inheritance between test cases, test cases containing public test utility methods, shared variables among test cases, and test cases creating instances of other test cases or using the instances as parameters.
- We documented four dependency-related test smells that we manually uncovered. We reported some instances of each test smell to developers, and the instances are either confirmed or fixed.

Our paper provides an important first step to study and understand the design of CI test cases in terms of code dependencies and their impact on the effectiveness of test impact analysis. Our findings can help developers improve test case design by refactoring unneeded code dependencies (e.g., the test smells that we identified), and may inspire future software testing research to further improve test design and test execution efficiency in continuous testing. We release the replication package of this study, including the studied data, code, and the results of the manual studies (https://sites.google.com/view/codedependencies).

**Paper Organization.** Section 2 introduces the background and related work of CI testing. Section 3 describes our studied systems and the methodology to study code dependency and test impact analysis. Section 4 presents the result of test impact analysis and further shows the dependency between test cases and source code classes, and among test cases. Section 5 summarizes the key findings and their implications. Section 6 discusses the threats to validity. Finally, Section 7 concludes the paper.

## 2 Background and Related Work

In this section, we discuss the background and related work of continuous integration and testing, test dependency, and test case prioritization and selection.

**Continuous Integration and Testing**. Continuous integration (CI) is the practice of frequently integrating and merging developers' code changes to a central code repository. To reduce the manual effort in the CI process, developers often use automation tools such as Jenkins [10] for automated code integration, compilation, and testing. The CI process can be triggered by the automation tools based on a customized schedule to integrate the most recent code changes. As part of the CI process, continuous testing is to verify the quality of the integrated code by automatically executing test cases. A common practice is to run the test cases either after a fixed period (e.g., once per day in Hadoop) or after a consecutive number of commits (e.g., eight commits in Ericsson [11]). A prior study [12] found that CI improves software development productivity and helps reveal more bugs. Developers often use different build systems, such as Maven, to manage the test execution. In each build, Maven would automatically execute all the test cases specified in its build script and generate a final test report. The concept of continuous testing was introduced by Saff *et al.* [13] as a way to help developers rapidly identify regression errors at an early stage and to reduce development waste. Muslu *et al.* [14] discussed the benefit of leveraging the continuous testing process to detect system errors caused by incorrect data. Their research illustrates that continuous data testing can be used to address important data debugging problems. Chen *et al.* [15] documented their industrial experience on how they integrate non-functional (i.e., performance) tests into CI automation pipeline.

Despite the success, practitioners are faced with challenges when deploying continuous testing in practice due to the high overhead. A recent study by Memon *et al.* [16] describes techniques adopted by Google to scale up continuous testing (i.e., 2 billion LOC and 150 millions of test runs per day). On the core of the scalability problem is the code dependency, i.e., between source code and test, and among test cases. In this work, we study continuous testing from the perspective of code dependencies (between source code classes and test cases, and among test cases) in 11 large-scale open-source systems. We study code dependencies from two different perspectives: CI execution in relation to test impact analysis, and CI test case design. Our study provides a different perspective and presents an important step towards understanding and improving test case execution and design.

**Test Case Prioritization and Selection.** There exist many prior studies on test case prioritization (TCP) and selection (TCS) techniques. TCP techniques prioritize the executions of test cases that are likely to fail (i.e., developers can work on resolving the failing test cases as soon as possible) [17, 18, 19, 20, 21, 22, 23, 24]. On the other hand, TCS techniques select a subset of test cases to significantly reduce the execution time [1, 2, 3, 4, 5, 6]. Recently, researchers proposed to perform test case prioritization and selection in continuous testing for better efficiency. Elbaum *et al.* [25] adopt a combination of test case prioritization and selection

techniques in practice to make continuous testing more cost-effective. Memon *et al.* [16] aimed to provide more prompt feedback from continuous testing (i.e., reducing waiting time) for developers after a commit is submitted. They proposed an approach that leverages factors on test breakages or fixes to improve test case prioritization. Marijan *et al.* [26] present a case study of using test case prioritization in industry. Zhu *et al.* [27] propose to re-prioritize the test cases that are more likely to fail based on historical test executions in continuous testing. Luo *et al.* [28] compared four static TCP techniques with the state-of-the-arts dynamic-based approach. Their evaluation of 58 Java systems shows that static techniques can be very effective in terms of fault detection and its cost reduction.

Companies such as Microsoft [9] use test case selection and test impact analysis to select only the impacted test cases (i.e., test cases that have dependencies with the changed code) to reduce test execution overhead and make continuous testing scalable to large codebases. Engstrm *et al.* [29] studied 28 test case selection techniques and classified them according to properties such as software language, test selection approach, and test selection granularity. The study highlights that there are no strong differences between these techniques and no technique was found to be definitely superior to others. Legunsen *et al.* [8] compared class-level and method-level static TCS techniques with Ekastazi (a dynamic TCS technique) in 985 revisions of 22 Java systems. Their result showed that static TCS at the class-level shows promising results and have comparable performance with dynamic-based techniques. They also found that method-level analysis contains too many false positives. They suggested that researchers should continue to improve static TCS techniques at a coarser granularity. One major difference between our work and the study by Legunsen *et al.* [8] is that they did not consider the accumulative testing overhead if the test cases need to be executed frequently. Our study analyzes the amount of time that can be saved if applying test impact analysis over a one-year period.

Prior studies on TCP and TCS are evaluated in a non-CI setting. Unlike traditional software development where test cases are executed once in a while, having a high degree of code dependency may have a larger accumulated testing overhead in CI due to frequent code integration and testing. Hence, better modularity will not only help improve test maintenance but will also benefit the effectiveness of test case prioritization/selection. To better understand the effectiveness of test impact analysis in CI settings and provide insights on improving the modularity of test case design, we study CI test cases by analyzing the code dependencies in 11 large-scale open source systems. We first study the effect of code dependencies on test impact analysis in CI settings, then we zoom in and further study test case design from the perspective of the dependencies between test cases and source code files, and among test cases. We also discuss some dependency-related test smells that we manually uncovered.

**Studies on Test Case Dependencies and Quality.** There exist a few studies related to dependencies among test cases from different perspectives. Zhang *et al.* [30] empirically studied the assumption of test independence, which is required by many test case prioritization (TCP) and test case selection (TCS) techniques. They found that the dependencies between test cases may cause TCP and TCS techniques to fail. Gambi *et al.* [31] presented an approach named PRADET to detect test dependency through a systematic and data-driven process. Mocking practices may also reduce test case dependencies. Spadini *et al.* [32] conducted an empirical study on the mocking practices in unit tests. Their study showed that developers often mock external dependencies (e.g., web services), and maintaining mocking codes introduces additional overhead. Pinto *et al.* [33] studied the evolution of test cases. They found that developers often refactor and modify/delete test cases in addition to repairing test cases. There are also many studies that focus on studying the test quality concerns, e.g., flaky tests. Luo *et al.* [34] empirically identified the root causes of flaky tests, i.e., a test sometimes fail and sometimes pass. They found that flaky tests are commonly caused by test orders, concurrency, and asynchronous waits. Vahabzadeh *et al.* [35] empirically studied bugs in test code and found that flaky, semantic, and environment-related bugs are common problems. Palomba *et al.* [36] found that more than 50% of the flaky tests contain test smells, and removing the smells can help improve software design and test flakiness.

Different from prior studies, we first study the effectiveness of test impact analysis. We find that, even though the number of changed files is often small, the number of impacted test cases is high. Our study on code dependency shows that most test cases are related to system integration, which has a higher degree of dependencies. For the dependencies among test cases, we find that most dependencies are caused by test case inheritance. Our qualitative study further reveals several test smells in which unnecessary dependencies could be removed. Such test smells may cost additional maintenance effort, increase maintenance difficulty, cause unstable test environments and test results (i.e., flaky tests), and reduce the effectiveness of test impact analysis.

## 3 Experimental Settings and Methodology

**Experimental Settings.** In this paper, we conduct our case study on 11 open-source Java systems. Table 1 shows an overview of the studied systems. The domain of the studied systems ranges from databases, distributed computing, and cloud computing to communication and web services. We analyze all the Java files in the 11 studied systems. We choose these systems because they are large in scale, follow the continuous testing practice, actively maintained, and commonly used in industry. The studied systems strictly follow the continuous integration (CI) practices, and use Jenkins or TravisCI for test automation and code integration [10]. They all schedule daily builds on Jenkins or TravisCI that compile the system and run the test cases.

**Uncovering Code Dependency Graph** We use JavaParser [37] to statically uncover the dependencies in the studied systems. JavaParser is an open-source Java static analysis framework that supports the latest version of Java. We first construct a class-level dependency graph that includes both the test cases and source code files. The dependency graph stores the information about whether there

TABLE 1
An overview of the studied systems.

| System | Version | Release date | LOC in source | LOC in test | Num. files in source | Num. files in test |
|---|---|---|---|---|---|---|
| CXF | 3.3.0 | Jan. 2019 | 694K | 413K | 3.9K | 3.2K |
| Flink | 1.7.1 | Dec. 2018 | 483K | 492K | 3.9K | 2.6K |
| Hadoop | 3.2.0 | Jan. 2019 | 1,097K | 896K | 6.4K | 3.5K |
| HBase | 2.1.2 | Jan. 2019 | 554K | 327K | 2.2K | 1.5K |
| jclouds | 2.1.2 | Feb. 2019 | 332K | 237K | 3.6K | 2.2K |
| Kafka | 2.1.0 | Nov. 2018 | 181K | 136K | 1.3K | 0.6K |
| BookKeeper | 4.9.0 | Feb. 2019 | 193K | 114K | 1.5K | 0.5K |
| Hive | 3.1.0 | Jul. 2018 | 1,221K | 327K | 4.6K | 1.3K |
| Jetty | 10.0.0.beta0 | May. 2020 | 307K | 237K | 1.6K | 1.3K |
| Cucumber-JVM | 6.2.2 | Jul. 2020 | 29K | 31k | 0.4k | 0.4K |
| Californium | 2.3.0 | Jun. 2020 | 88K | 46K | 0.6K | 0.2K |

exists a dependency between two files (i.e., either a test case or source code file). We identify a Java file as a test case if it uses APIs from testing frameworks such as JUnit or TestNG (i.e., using the *@Test* annotation). Similar to the work by Orso *et al*. [7], we also consider the inheritance relationships when constructing the dependency graph. For each class, we consider its dependency with the related classes that are on the same inheritance hierarchy. In addition, we exclude binaries and only analyze the dependency if we can find a corresponding source code file (i.e., only analyze the system source code and exclude external libraries).

**Applying Test Impact Analysis.** Similar to prior studies [7, 8, 1, 2], we consider a test case is impacted by a given commit if the test case may need to be executed due to having dependencies with the changed files (i.e., either source code or test files). Our test impact analysis follows the technique proposed by Orso *et al*. [7], which can be understood as a conservative static coverage analysis at the class-level. Prior studies [8, 28] found that static test impact analysis techniques achieve a similar level of performance compared to dynamic-based techniques, and class-level dependency gives better results compared to method-level dependency. As a result, we perform the impact analysis by analyzing the static dependencies at the class level between test cases (i.e., test classes) and the changed source code files. In particular, we identify the impacted test cases based on the two following criteria: 1) the test case directly or indirectly calls the changed file; or 2) the changed file directly or indirectly calls the test case (e.g., the changed file is a test case that calls another test case). To formalize our approach using the dependency graph, we call that node_A (i.e., represents either a test or a source code class) is an ancestor of node_B if there exists a path from node_A to node_B (i.e., node_A directly or indirectly calls node_B). From the dependency graph, we first collect all the ancestor nodes of the nodes that represent the changed files in one commit, named *all_ancestors*. Then, the set of *all_descendants* is inferred by taking the union of all the descendant nodes of every node in *all_ancestors*. Finally, we only consider the nodes that represent test cases in *all_descendants* as impacted test cases that have dependencies with the changed files.

## 4 Studying Code Dependencies of Test Cases

In this section, we study code dependencies in test cases from two different perspectives, CI execution in relation to test impact analysis, and CI test case design, by answering four research questions (RQs). In RQ1, we study the effectiveness of test impact analysis in the CI setting and study the accumulated test execution overhead. In RQ2 and RQ3, we zoom in to study test design from the perspective of code dependencies: between source code files and test cases (RQ2) and among test cases (RQ3). Finally, in RQ4, we manually explore potential dependency-related test smells that may help reduce code dependencies in test cases and inspire future testing research. For each RQ, we provide the motivation, approach, and results.

### RQ1: What is the impact of dependencies from the perspective of test impact analysis in continuous testing?

**Motivation.** Due to the high frequency of test executions in CI settings (e.g., at least once or even several times on a daily basis), there may be a larger accumulated testing overhead if there is a high degree of dependencies between source code and tests, and among test cases. In this RQ, we seek to study, for each run of continuous testing triggered by code integration, how many test cases are impacted (i.e., have dependencies with the changed code and need to be executed). We also investigate the test execution time that can be potentially saved if developers apply test impact analysis.

**Approach.** Our studied systems run continuous testing on a daily basis. We use the same frequency (i.e., every day) to analyze the impacted test cases in each run of continuous testing. We apply the test impact analysis approach described in Section 3. We consider the code changes in the past year (i.e., 365 days) prior to the release of the studied version of the systems (see Table 1). For each day, we construct the dependency graph, collect all the commits on that day, and identify which files (i.e., either test cases or source code files) are changed in the commits. Finally, based on the dependency graph (Section 3) and the list of changed files, we calculate the percentage of test cases that was impacted in each day.

Furthermore, we analyze the potential reduced time that continuous testing can benefit from test impact analysis. We crawl the readily-available execution logs from the CI platforms of the studied systems (i.e., Jenkins and Travis CI). Although the format of the log may be different in each CI platform or system, the logs generally contain information such as the name of the test case, test execution result (i.e., pass or fail), and test execution time. We analyze the execution time of the test cases that are not impacted by the code change (i.e., potentially saved time) and compare that with the actual execution time (i.e., the time to run all the test cases). To obtain enough data for analysis, we monitored the CI platforms for 30 days (i.e., from 2020-06-07 to 2020-07-05) and collected the generated execution logs. Note that some systems configure the CI platform to keep almost one month of test execution logs, while some systems only store the logs for a few days. In addition, some systems did not execute the test cases due to having no code changes. Hence, the number of collected execution logs varies across the studied systems.

**Results.** *Although the median number of changed files is less than 10 in most studied systems, the median percentage of impacted test cases may go up to 72%.* Table 2 shows the median number of daily commits and changed files. Note that we

TABLE 2
Median number of daily commits and changed files, excluding the days when there are no code changes.

| System | Med. num. of commits | Med. num. of changed files |
|---|---|---|
| CXF | 3 | 5 |
| Flink | 7.5 | 17 |
| Hadoop | 8 | 28 |
| HBase | 3 | 8 |
| jclouds | 1 | 3 |
| Kafka | 3 | 8 |
| Hive | 6 | 22 |
| BookKeeper | 2 | 7 |
| Jetty | 6 | 12 |
| Cucumber-JVM | 2 | 3 |
| Californium | 3 | 6 |

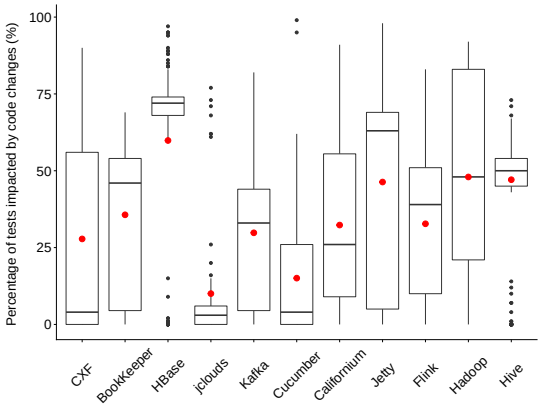



Fig. 1. The distributions of the percentage of test cases that are impacted by the daily code changes. The red dot illustrates the mean value.

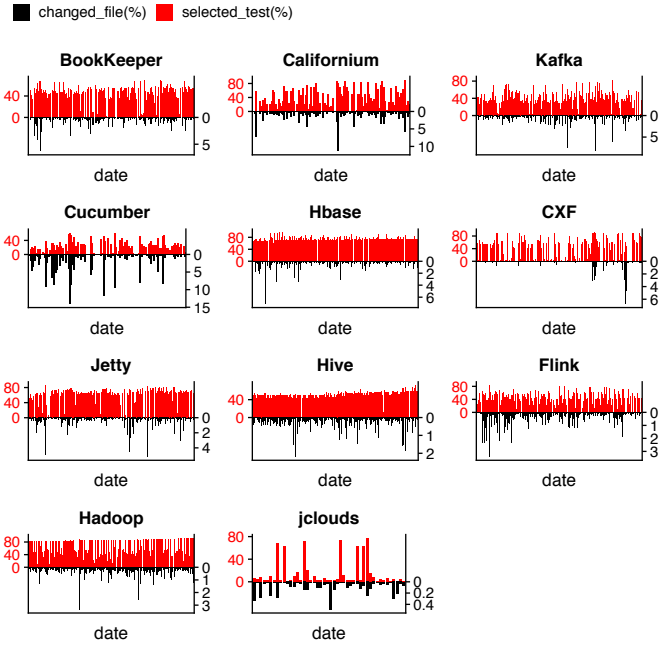



Fig. 2. The comparison of the percentage of changed files and the percentage of impacted tests over the one-year period.

TABLE 3
The details of the collected CI builds for analyzing the execution time of the impacted tests.

| System | Num. of collected CI builds | Ave. test execution time (in seconds) |
|---|---|---|
| CXF | 19 | 39,301.00 |
| Flink | 31 | 4,828.30 |
| Hadoop | 59 | 13,595.00 |
| HBase | 116 | 34,510.00 |
| jclouds | 40 | 571.49 |
| Kafka | 77 | 10,886.90 |
| Hive | 46 | 7,598.84 |
| BookKeeper | 45 | 64,406.00 |
| Jetty | 41 | 2,188.00 |
| Cucumber-JVM | 100 | 60.31 |
| Californium | 30 | 401.00 |
| **Total** | **604** | **178,346.84** |

exclude the days when there are no code changes. We find that developers usually only change a small number of files in a day. As shown in Table 2, the median number of changed files is less than 10 in 7 out of 11 studied systems. Considering the total number of both source code and test files in the studied systems (Table 1), developers, on average, make changes to less than 0.5% of the files per day. We also find that the median number of daily commits is less than 10 in all studied systems.

Figure 1 shows the distributions of the percentage of test cases that are impacted by the daily commits. In general, the median percentage of impacted test cases goes up to 72%. For Flink, Hadoop, Kafka, Hive, Jetty, and BookKeeper, the median percentage of impacted test cases is around 40% to 63%. Californium and Kafka have a median percentage of around 26% and 33%. Among all the studied systems, HBase's test cases are impacted the most (the median percentage of impacted test cases is 72%). In other words, developers still need to execute most test cases even if they apply test impact analysis to reduce test execution overhead. We find that the number of impacted test cases is smaller for CXF, Cucumber, and jclouds: the median percentage of the impacted test cases is less than 10%. After some investigation, we find that the median number of daily changed files is also small for CXF, Cucumber, and jclouds (i.e., 5, 3, and 3 files, respectively), so the number of impacted test cases is lower than that of the other studied systems. Nevertheless, the average number of impacted test cases is still high for CXF and Cucumber (e.g., over 20%).

Figure 2 visualizes the relationship between the percentage of changed files (i.e., the upper part of each plot) and the percentage of the impacted test cases (i.e., the lower part of each plot) of each daily build in the one-year time period. We can see that the studied systems had experienced different levels of development activities over the studied period. In general, we see that if there are more changed files, more test cases are impacted. We further compute the Spearman correlation between the percentage of changed files and impacted test cases, and we find that there is only a moderate positive correlation (i.e., correlation ranges from 0.43 to 0.68) except for Cucumber (i.e., the correlation is 0.88). Our finding shows that, even though there is a certain degree of correlation, the correlation is not very strong. *Namely, changes to some source code files may have a larger effect on the number of impacted test cases.*

*The effectiveness of test impact analysis may vary across systems, depending on the test execution overhead. The median percentage of saved test execution time is around 50% for most of the studied systems.* Table 3 lists the details of the collected CI builds for the analysis on the test execution. We find that

the average test execution time varies significantly across the studied systems. For larger systems such as BookKeeper, one run of continuous testing (i.e., executing all test cases) takes over 17 hours to complete. For smaller systems such as Cucumber, the test cases only take one minute to run. The results indicate that the benefit of test impact analysis is significantly higher for larger or more complex systems compared to smaller systems. Reducing the accumulated test execution overhead may further reduce the needed resources for testing and increase test execution frequency.

Figure 3 shows the distribution of the percentage of test execution time that can be potentially saved after applying test impact analysis. Each data point represents one CI build that we collected (Table 3). For CXF, jclouds, Kafka, Cucumber, and Jetty, over 80% (median) of the test execution time may be saved after applying test impact analysis. For the remaining systems, namely BookKeeper, HBase, Californium, Flink, Hadoop, and Hive, a median of 50% or less of test execution time may be saved after applying test impact analysis. Among the studied systems, we find that the median percentage of reduced test execution time is very small for HBase. After some investigation, we find that for the analyzed CI builds, developers modified some files that are highly dependent upon, which causes many test cases to be selected. For example, developers modified HBaseTestingUtility 13 times in the analyzed CI builds, and this class has dependencies with more than 760 test cases. Therefore, such files that have many dependencies with other files may greatly reduce the effectiveness of test impact analysis, even worse if such files are changed frequently.

Our finding shows that, although the number of daily changed files is often small in the studied systems, the percentage of the impacted test cases can be large (median around 50% for 6 out of 11 studied systems). We also concluded a similar finding in terms of execution time of the impacted tests, and the corresponding percentage of saved time. For 6 out of the 11 studied systems, the impacted test cases may consume over 50% of the total test execution. The result indicates that there is a high degree of dependencies among test cases or between test cases and source code files, which affect the effectiveness of test impact analysis. To further understand the reasons for such dependencies and provide insights on reducing test execution overheads, we conduct detailed analysis on code dependencies in RQ2 and RQ3.

*Although the number of daily changed files is small, most test cases (around 50% or more) are impacted. Such a high degree of dependencies affects the effectiveness of using test impact analysis to reduce testing overheads. Future research should consider more specialized techniques (e.g., through refactoring) that can reduce certain code dependencies, to improve the effectiveness of test impact analysis.*

### RQ2: What is the degree of dependencies between test cases and source code files?

**Motivation.** To reduce test execution overheads, developers may need to find a subset of the test cases that are impacted by the given code changes. However, as we found in RQ1, even though the number of changed files is small in each test execution, many test cases are still impacted. In this RQ, we further study test design from the perspective of the dependency between the test cases and source code files. The findings will provide an overview of how the systems are tested and may provide insights on how to help improve the current testing practices and inspire future research.

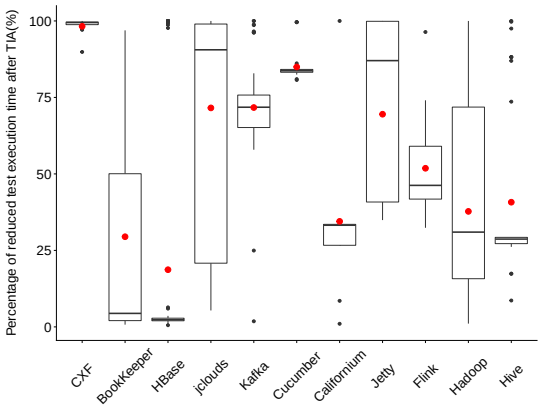



Fig. 3. The distributions of the reduced test execution time after applying test impact analysis. The red dot illustrates the mean value.

**Approach.** To answer this RQ, we statically identify test cases and uncover their dependencies with source code files. In continuous testing, developers leverage build systems, such as Maven, to execute all the test cases under the *test* directory or the directory specified in the build script. However, there may be other non-test files, such as utility or helper files, in *test* directories. Therefore, the first step is to identify true test cases. As discussed in Section 3, we identify a file as a test case if it contains the *@Test* annotation (i.e., regardless of the directory where the file is located). We identify a file as a source code file (i.e., code related to the actual business logic) if the file is located outside of the *test* directory (i.e., the directory path does not contain the word "test" to avoid including test utility files) and does not contain any testing related APIs. Note that we only consider the direct dependency between a test case and its corresponding source code files in this RQ. Direct dependency better reflects test design, so focusing on direct dependencies allows us to analyze the composition of CI tests and CI test design: When developers design a test case for Class A, developers may not care much about the dependencies of Class A (i.e., the indirect dependencies), but only Class A itself (i.e., the direct dependency of the test case).

The second step is to analyze how many source code files a test case tests. If a test case depends on multiple source code files, we further investigate the average package distance among the source code files [38]. The average package distance gives us insights on the semantic similarity of the source code files covered by the same test case based on the structural closeness (e.g., whether a test case is testing the interaction of various components in the system) [39, 40, 38]. If two source code files are used to implement the same or similar functionality, they are more likely to be located in the same or nearby package; thus, have a low package distance. Because the package structure is closely associated with the directory structure, we implement the following three steps

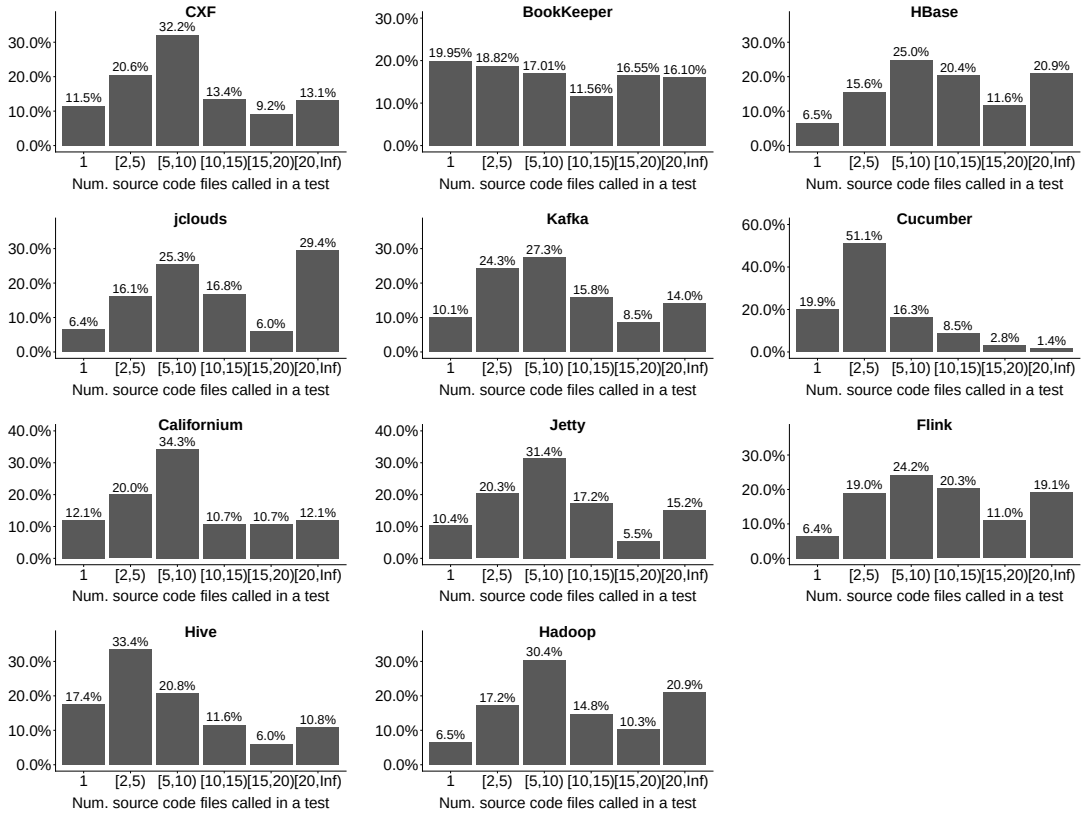



Fig. 4. The distribution of the number of source code files that are called in a test case.

to calculate the average package distance among source code files. The calculation is an iterative process based on pairwise package distance calculations.

1) We build a directory tree (i.e., each node in the tree represents a directory name in the path) for all the source code files that are invoked by a test case and calculate the depth of the tree (*N*).
2) For every pair of source code files, from the top to bottom in the built directory tree, we find their last common parent node at depth *M*. The package distance of the two source code files equals to *N-M-1*.
3) We iteratively compare every pair of the source code files and calculate the average package distance among the source code files.

As an example, test case `TestNamenodeResolver.java` in *Hadoop* calls three source code files below. The paths are simplified for illustration.

ClassA: `./hadoop-common/src/java/hadoop/conf/Configuration.java`

ClassB: **`./hadoop-hdfs-rbf/src/java/hadoop/hdfs/server/federation/`** `store/MembershipState.java`

ClassC: **`./hadoop-hdfs-rbf/src/java/hadoop/hdfs/server/federation/`**`router/RBFConfigKeys.java`.

The paths of the three source code files form a tree with a depth of *N=9*. ClassB and ClassC have a common parent at depth *M=7*. The package distance of ClassB and ClassC is calculated as *N-M-1=1*. Similarly, the other pair-wise package distances are calculated as 8 for ClassA and ClassB, and 8 for ClassA and ClassC. Finally, the average package distance is calculated as $(8 + 8 + 1)/3 = 5.67$.

Mocking, as a common testing practice, may introduce false dependencies. To avoid counting mocked objects as dependencies, we automatically check whether a dependent class is a mocked object or not. *Mockito*, *EasyMock*, *PowerMock*, and *MockWebServer* of *OkHttp* are the four mocking frameworks that are used in our studied systems. These mocking frameworks follow a similar way of creating mock objects: 1) using annotations (e.g., `@Mock`) and 2) invoking mocking methods (e.g., `Mockito.mock()`). To exclude the dependencies of mocked objects, we follow a similar approach that is proposed in a prior study [41]. We first identify the mocking frameworks used in each studied system. Then, if a test case mocks the implementation of an object, we mark the corresponding class as *mocked* and exclude the dependency of the mocked object.

**Results.** *Most test cases focus on testing the integrated behavior of the system.* Figure 4 shows the number of source code files that are tested in a test case (after removing source code files that are called due to mocking). Even though many prior studies [42, 43, 44] focus on studying unit tests and help reduce unit test execution overheads [42, 43, 44, 45, 46, 47, 48, 49], we find that most test cases focus on testing multiple source code files. In all the studied systems, only 6%–24% of the test cases focus on testing a single source code file, and all other test cases test multiple source code files. In general, most test cases (16% to 34%) in the studied systems test 5 to 10 source code files. We also find that a large number of test cases test more than 20 source code files in HBase

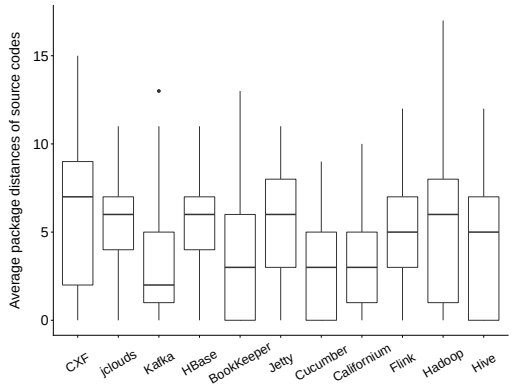



Fig. 5. The distributions of the average package distance of the source code files in a test.

and jclouds (20.9%, 29.4%, respectively). Figure 5 further illustrates the average package distance of the source code files in each test case. The majority of the studied systems have a median package distance that is larger than five. Such a large package distance shows that many test cases cover a wide variety of functionality. In short, our findings highlight that developers spend a significant amount of efforts on testing the integrated behaviour of the system. Based on our finding, a potential way to improve the effectiveness of test impact analysis in continuous testing is to prioritize the test cases based on their degree of dependencies. Inspired by the principle of *fail-fast* [50], if the basic functionality of a source code file cannot even pass the test cases, developers may then consider skipping executing other more complicated test cases that test the integrated behaviour of the system (i.e., test cases that cover more than one source code file).

*Most test cases executed in continuous testing cover multiple source code files in different packages, and many test cases even cover more than 20 files. Future studies should consider the peculiarity of the focus in continuous testing when designing approaches to help developers reduce test execution overheads.*

### RQ3: What is the degree of dependencies among test cases?

**Motivation.** Test cases should be independent of each other to ensure the quality of individual source code file or module. As systems evolve, there may be a certain degree of dependencies among test cases [30], which can increase the test maintenance difficulty and reduce the effectiveness of test impact analysis. Therefore, in this RQ, we empirically study the degree of dependencies among test cases.

**Approach.** We conduct both quantitative and qualitative studies on the dependencies among test cases. For the quantitative study, we study the degree of dependencies among test cases (i.e., how many test cases call other test cases) using the dependency that we uncovered (Section 3). Similar to RQ2, we analyze the direct dependency among these test cases. For the qualitative study, we take a statistical sample to understand the reasons for the dependency. In total, we studied 326 sampled test cases out of 2,160 in the studied systems (based on a 95% confidence level and 5% confidence interval [51]) that have direct dependencies with other test cases (identified in the quantitative study step).

TABLE 4
Number of test cases, test utility files, and the number of test cases that have dependencies with more than one test cases. *Med. dep.* and *Max. dep.* shows the median and the maximum number of dependencies between one test case and other test cases. We exclude the test cases that do not have any dependencies with other test cases when calculating the median.

| System | No. test cases | No. test utils. | No. test cases with dep. | Med. dep. | Max. dep. |
|---|---|---|---|---|---|
| BookKeeper | 449 | 76 | 16 (3.6%) | 1 | 1 |
| CXF | 1,226 | 2,039 | 127 (10.4%) | 1 | 5 |
| Flink | 1,946 | 641 | 463 (23.8%) | 1 | 4 |
| Hadoop | 2,823 | 619 | 509 (18.0%) | 1 | 6 |
| Hive | 819 | 437 | 104 (12.7%) | 1 | 3 |
| HBase | 1,177 | 287 | 164 (13.9%) | 1 | 2 |
| jclouds | 2,063 | 119 | 867 (42.0%) | 2 | 13 |
| Kafka | 437 | 124 | 17 (3.9%) | 1 | 1 |
| Jetty | 711 | 580 | 51(7.2%) | 1 | 2 |
| Cucumber-JVM | 149 | 230 | 6(4.0%) | 1 | 1 |
| Californium | 143 | 65 | 0(0%) | 0 | 0 |

The process of our manual study contains three phases:

- Phase I: A1 derives a list of 326 randomly sampled test cases that have code dependencies with other test cases. The sample is based on a 95% confidence level and 5% confidence interval.
- Phase II: A1 studies 100 randomly sampled test cases and studies the reason for the dependency. A1 derives a draft list of categories based on the observation. The three authors then collaboratively label the 100 test cases using the draft list of categories. During the process, the categories are revised and refined.
- Phase III: A1 , A2, and A3 independently apply the derived categories to the remaining sampled test cases. Any disagreement is discussed until a consensus is reached. In this phase, no new categories were derived.

**Results.** *On average, 18% of the test cases have dependencies with other test cases.* Table 4 shows the number of test cases, test utility files (i.e., non-test files in the *test* directory), and the number of test cases that have dependencies with other test cases. We find that there are many test utility files in the studied systems. However, even though developers may be using test utility files to refactor test code and assist in writing test cases, we still find that 3.6% to 42% of the test cases (an average of 15%) have dependencies with other test cases. Table 4 also shows the median and the maximum number of dependencies among test cases (we exclude the test cases that do not have dependencies with other test cases when calculating the median). We find that most of the test cases only have a direct dependency with one other test case (median is 1 in most systems), but some test cases may have dependencies with up to 13 other test cases. In short, our finding shows that even though dependencies between test cases may increase test maintenance difficulties [52, 30], such dependencies are still common in the studied systems.

*Most test case dependencies are caused by test case inheritance and having public test utility methods in test cases.* Table 5 shows the five manually-uncovered reasons for dependencies among test cases and the corresponding distribution. Inheritance is the most common reason and accounts for 79.4% of the dependencies among the studied test cases. We also find that 13.2% of the dependencies is caused by a test case calling other test utility methods that are declared in

TABLE 5
An overview of the manually derived reasons for dependencies among test cases.

| Reason | Definition | Percentage |
|---|---|---|
| Inheritance | Test cases extend another test case. | 79.4% |
| Test cases contain *test utilities* | Test cases contain test utility methods which are also used by other test cases. | 13.2% |
| Shared variables | Variables are accessed by more than one test cases. | 2.1% |
| Shared test case | Test cases use instances of other test cases as parameters or create objects of other test cases. | 4.6% |
| Others | Other reasons such as returning an instance of a logger that refers to another test case class. | 1.2% |

TABLE 6
Number of test cases extending other classes. *Med. DIT* shows the median DIT number of all test cases with non-zero inheritance. *Max. inherit layer* shows the maximum DIT number among all test cases in one system.

| System | No. tests inherit other classes | Med. DIT layer | Max. DIT layer |
|---|---|---|---|
| BookKeeper | 224 (49.9%) | 1 | 10 |
| Hadoop | 1,071 (37.9%) | 2 | 13 |
| HBase | 383 (32.5%) | 1 | 8 |
| CXF | 629 (51.6%) | 2 | 9 |
| Flink | 1,390 (71.4%) | 2 | 11 |
| jclouds | 1,375 (66.7%) | 3 | 11 |
| Kafka | 56 (12.8%) | 1 | 6 |
| Hive | 189 (23.1%) | 1 | 6 |
| Jetty | 257(36.1%) | 1 | 9 |
| Californium | 21(14.7%) | 1 | 2 |
| Cucumber-JVM | 7(4.7%) | 1 | 1 |

another test case. Namely, some test cases have public test utility methods that are used by other test cases. However, such a design creates unnecessary dependencies and should be refactored. A better design that may help improve test code comprehension is to create a separate test utility file instead of having the utility method implemented in a test case [52]. 2.1% of the dependencies among test cases are caused by having more than one test cases that access a static class field (e.g., static instance variable). As shown in a prior study [30], such dependency may cause flaky tests and reduce the reliability of the test result. 4.6% of the dependencies among test cases are caused by an instance of a test case being created in another test case or being passed as a parameter to another test case. In such cases, the assumption of test case independence may be violated and the test cases should be refactored [30]. Finally, we find that 1.2% of the dependencies belong to the "Other" category. For example, developers may be mistakenly using the class of TestCaseB to create a logger in TestCaseA, which may cause difficulties when using logs for debugging [53].

Since inheritance is the most dominating reason for dependencies among test cases, we further study the inheritance relationship in the test cases (Table 6). We find that inheritance is widely used in the test cases: 4.7% to 71.4% of the test cases extend other classes (an average of 46.91%). We use *DIT* (i.e., depth of inheritance) that is proposed by Chidamber and Kemerer [54] to study class inheritance. *DIT* is a classic metric in object-oriented design to quantify the level of inheritance, and is calculated as the length of the maximum path from a class to the root of the inheritance tree. For most of the studied systems, the average *DIT* of all test cases is either two or three. The maximum *DIT* ranges from 1 to 13 in the studied systems. Even though developers may use inheritance to share common test setup and tear down code across test cases [52], such complex inheritances (i.e., a large *DIT*) may increase the dependencies among test cases and increase maintenance difficulty [55, 56, 57]. Our findings reveal common types of test case coupling in continuous testing. Future studies may build upon our findings and help developers refactor the test cases, and reduce test maintenance and execution overhead.

*On average, 18% of the test cases have dependencies with other test cases. Most dependencies are caused by test case inheritance and accessing utility methods in test cases.*

### RQ4: What are the dependency-related test smells that may negatively impact test case design?

**Motivation.** Dependencies may introduce spaghetti code and increase the difficulty of test maintenance. For example, the independence assumption of test cases may be violated due to unnecessary code dependencies [30]. RQ3 shows that there exists non-trivial code dependencies among test cases. In this RQ, we further conduct a qualitative analysis on the sampled test cases to identify dependency-related test smells and recommend how to fix them. Identifying common patterns of excessive and even problematic dependencies will shed light to reduce code dependencies among test cases; and hence, improve the effectiveness of test impact analysis and test quality [58, 59, 60].

**Approach.** Similar to RQ3, we manually examine the design of each test case in a randomly sampled set of 326 test cases based on 95% confidence level and 5% confidence interval (the same set that we used in RQ3). Three of the authors collaboratively examine the design of each test case, the dependencies with other test cases, and potential negative effects caused by the identified dependencies. Any disagreement is discussed until a consensus is reached. At the end of this phase, we obtain four types of test smells that might have a negative effect on test dependencies and design. Finally, A1 reported at least one instance for each type of the test smell to the issue tracking system (i.e., Jira) to get confirmation from developers.

**Results.** In total, we uncovered four dependency-related test smells (26 instances in the 326 manually studied test cases) [61]. We reported at least one instance for each type of the test smell, and all of the reported instances are either confirmed[1][2] or fixed by developers[3][4][5]. Below, we discuss the test smells that we uncovered during our manual analysis. For each test smell, we provide a description and an example, and discuss the negative effect and possible solutions.

**Test Smell 1: Duplicate test runs caused by inheritance.**
***Description.*** We find that in some situations, the inherited test methods may be executed twice: once in the parent test case (i.e., a non-abstract class) and once in the child test case.

1. https://issues.apache.org/jira/browse/HBASE-22814
2. https://issues.apache.org/jira/browse/JCLOUDS-1508
3. https://issues.apache.org/jira/browse/CXF-8092
4. https://issues.apache.org/jira/browse/CXF-8086
5. https://issues.apache.org/jira/browse/CXF-8087

The test methods from the base test case will be executed when running the base test case. Then, the inherited test methods will be executed again when running the child test case. Note that the child test case may inherit the test fixture from the parent test case; thus, the same test methods are executed multiple times in the same test environment.

***Example.*** In CXF, there are 14 test methods inherited by test case `AssociatedManagedConnectionFactoryImplTest` from the non-abstract base test case `ManagedConnectionFactoryImplTest`. These 14 test methods will be executed twice, once by each of the two test cases. In total, we found 8 instances of this test smell in the manually studied test cases.

***Negative Effect.*** Duplicate test runs will waste testing resources and increase test overhead. The frequent executions of continuous testing (i.e., due to frequent code changes), exaggerate the negative effect of this test smell. Moreover, the base test case and its child test cases may fail together due to the same issue, which increases the challenge of failure diagnostic, especially in continuous testing. The severity of the test smell may also be increased when more test methods are inherited by child test cases, or more test cases inherit the same base test case.

***Possible Solutions.*** Developers should try to avoid inheritance from non-abstract test cases. If two test cases share many test methods, a better solution would be to either create a test utility class or refactor the common test methods to a separate test case.

**Test Smell 2: Scattered test fixtures caused by inheritance.**

***Description.*** Test fixtures are defined in test cases to set up the environment for test execution. A base test case may define a general test fixture (i.e., a method annotated by *@BeforeClass* or *@Before*). When a child test case inherits a base test case, the child test case may define its own test fixture, but may also call the test fixture of the base test case. In multi-level inheritance, when each child test case has its own test fixture, the test fixture code becomes scattered and difficult to maintain.

***Example.*** In HBase, the test case `TestWALReplay` extends `AbstractTestWALReplay` and implements the test fixture using the *@BeforeClass* annotation (Listing 1). The test fixture method also calls the test fixture of the base test case (i.e., *setUpBeforeClass*() on line 6). `TestWALReplay` is further extended by `TestWALReplayBoundedLogWriterCreation` and it adds specific test fixtures and invokes the test fixture of the parent test case on lines 12–16. In total, we found 12 instances of this test smell in the studied test cases.

***Negative Effect.*** Test fixture methods are either executed before each test method (e.g., *@Before*), or before each test case (e.g., *@BeforeClass*). Scattered test fixtures in multi-level inheritance may introduce extra test overhead, such as reducing the understandability of test design and makes test case evolution more error-prone in continuous testing. In our manual analysis, we even found several inconsistencies when developers implement test fixtures through multi-level inheritance. For instance, in HBase, the test case `TestSecureWALReplay` extends `TestWALReplay` (defined on line 1 in the above-mentioned code snippet). However, in contrast to the sibling test case shown in the above code snippet (`TestWALReplayBoundedLogWriterCreation`, line 10), the overridden test fixture in `TestSecureWALReplay` does not call the test fixture method from the base class (`TestWALReplay`) while the test fixture in the all other sibling test cases do (e.g., `TestWALReplayBoundedLogWriterCreation`, line 14). Such inconsistency may introduce insufficient setup in the subclass. We also find a few cases where the test fixture method is overridden, but the implementation in the test fixture methods are the same. Such unnecessary code clones may increase test maintenance difficulties. Finally, the inheritance hierarchy of test fixtures may violate the assumption of test case independence, i.e., there exists an *anticipated* order of calling different test fixture methods in one inheritance tree. Conflicting configurations in test fixtures of different sibling test cases may result in unanticipated test behaviours and unstable test results. Such negative effects introduce additional diagnosis challenges when relevant test cases are deployed in continuous testing. One example of such consequence is flaky tests, which are often ignored by practitioners and lead to negligence of true test failures.

Listing 1. The child class extends the parent `TestWALReplay` class, but both classes have *@BeforeClass* annotation for test case setup. The child class will execute both *@BeforeClass* methods and may cause maintenance issue or even unexpected test results.

```
1  public class TestWALReplay extends
       AbstractTestWALReplay {
2    @BeforeClass
3    public static void setUpBeforeClass() throws
         Exception {
4      ...
5      conf.set(WALFactory.WAL_PROVIDER, "filesystem");
6      AbstractTestWALReplay.setUpBeforeClass();
7    }
8  }
9  ------------------------------------------------
10 public class TestWALReplayBoundedLogWriterCreation
     extends TestWALReplay {
11   @BeforeClass
12   public static void setUpBeforeClass() throws
         Exception {
13     TEST_UTIL.getConfiguration().setBoolean( );
14     // invoke parent's fixture
15     TestWALReplay.setUpBeforeClass();
16   }
17   ...
18 }
```

***Possible Solutions.*** Developers should maintain the independence of test fixtures in one inheritance tree. Developers may use test utility classes to manage test fixtures instead of using inheritance. Another possible solution is to manage test fixtures individually and independently in each test case (i.e., including set-up and tear-down test environment), so the dependencies of test fixture between base test case and child test cases can be removed. Developers may also use JUnit's *@Rule* annotation to refactor the code that needs to be executed before and after a test.

**Test Smell 3: Using test case inheritance to test source code polymorphism.**

***Description.*** Developers may implement inheritance in test cases to test source code polymorphism so that code duplication of common test methods can be reduced (i.e., following the DRY principle – *"Don't Repeat Yourself"*). However, sometimes the unnecessary inheritance relationship in test cases may increase the difficulty of test maintenance and make test evolution more error-prone.

***Example.*** In jclouds, there are 49 test cases that extend from one parent test case (i.e., `BaseProviderMetadataTest`).

All the 49 test cases inherit all the test methods from the parent test case. On line 3 and line 10 in the code snippet below, the two child test cases call the constructor of the parent test case with arguments of different types. However, the arguments (e.g., Class `SkaliCloudMalaysiaProviderMetadata` on line 3, and class `GleSYSProviderMetadata` on line 10) inherit the same parent class. In this example, developers test the source code polymorphism through the constructors in the inheritance tree: the constructor of each child test case will pass arguments of different types to the constructor of the parent test case. The arguments of different types either extend or implement the argument types of the parent test case's constructor. Such test design may introduce unnecessary inheritances that may result in more scattered code (e.g., having 49 separate child test cases that do not implement any other test methods). In total, we found four instances of this test smell in the studied test cases.

Listing 2. 49 child test classes inherit the same base class `BaseProviderMetadataTest`), but all the child test classes calls the base class directly and only differ in one parameter.

```
1  public class SkaliCloudMalaysiaProviderTest extends
       BaseProviderMetadataTest {
2      public SkaliCloudMalaysiaProviderTest() {
3        super(new SkaliCloudMalaysiaProviderMetadata(),\
4        new ElasticStackApiMetadata());
5      }
6  }
7  ------------------------------------------------
8  public class GleSYSProviderTest extends
       BaseProviderMetadataTest {
9      public GleSYSProviderTest() {
10       super(new GleSYSProviderMetadata(), \
11       new GleSYSApiMetadata());
12     }
13 }
```

***Negative Effect.*** Such test design requires extra maintenance effort and might introduce errors due to the low maintainability. First, if the parent test case is modified (e.g., add additional test methods), developers would need to review all the child test cases to make sure the modification is valid to all of the child test cases (49 in total in the above-mentioned example). Second, if the parent test case is problematic (i.e., some test methods are buggy or flaky), many child test cases may also be affected and developers would need to spend extra time to isolate the issue. Third, insufficient testing might be overlooked in such a test design. If only using the common test methods from the parent test case, developers might neglect to test the unique aspects of each child test case. Due to the rapid development cycle and more frequent code changes, modern software development heavily relies on automated test execution (e.g., continuous testing) for quality assurance purposes. Failing to test certain aspects may lead to insufficient quality assurance in continuous testing.

***Possible Solutions.*** Such coupling can be reduced by utilizing features commonly provided by testing frameworks. For example, parameterized tests from JUnit [62] can be used to run one test case multiple times with different parameters in one code location (i.e., no need for multiple copies of the code or using inheritance). Compared to the scattered locations using inheritance, parameterized tests provide better maintainability by centralizing the code; and are thus, easier to maintain. More importantly, to improve the test efficiency and coverage, developers would also need tooling support to visualize the coupling among test cases and to highlight the coupling among the tested classes. Future research may study automated techniques that help developers verify whether source code polymorphism is well tested by existing test cases.

**Test Smell 4: Flaky tests caused by accessing shared resources or variables.**

***Description.*** Test cases may be flaky in different test runs because of accessing shared resources or variables.

***Example.*** The test case `JAXRSclientServerWebSocketTest` in CXF defines some test methods and is extended by several child test cases. One of the child test cases (the code is shown below) will modify a system property (i.e., *System.setProperty*) before running the test methods inherited from the base test case (line 4 and 5). However, the child test case does not reset the modified system property after the completion of a test run (lines 9–11). Hence, the system property may be changed when running other test cases, resulting in unstable test environments. In total, we found three instances of this test smell.

Listing 3. The child class shown below modifies the system property but does not change it back. Therefore, unexpected test results may happen when running other test cases.

```
1  public class JAXRSClientServerWebSocketNoAtmosphereTest
       extends JAXRSClientServerWebSocketTest {
2      @BeforeClass
3      public static void startServers() throws Exception {
4        System.setProperty("org.apache.cxf.transport." + \
5        "websocket.atmosphere.disabled", "true");
6        ...
7      }
8      @AfterClass
9      public static void cleanup() {
10       //this method is empty.
11     }
12 }
```

***Negative Effect.*** Accessing shared variables violates the assumption of test case independence [30]. In the above-mentioned code snippet, the system property is shared among test cases (e.g., the base and the child test cases). Modifying the system property and failing to reset it in the child test case may result in having different test environments when the order of test case execution changes. Thus, the test results may become flaky and unstable, which introduce additional challenges for understanding and performing diagnosis in continuous testing.

***Possible Solutions.*** Tooling support is needed to improve developers' awareness of the usage and consequence of shared variables. Once developers know the existence of shared variable access, they can take actions to eliminate the side-effects of such accesses. In the above-mentioned code example, developers can clean up the test environment and reset the modified system property in the *cleanup* method (lines 9–11, annotated by *@AfterClass*).

We also analyze the time when the dependency was introduced for the 26 test smell instances that we uncovered. We find that three test smell instances were introduced during the development of new features, one was introduced in a bug fixing commit, and the remaining were introduced at the beginning when source code files were created. Based on our preliminary analysis, developers may need to pay more attention to dependency-related test smells when they initially design and implement the test cases. To further show the generalizability of the uncovered test smells, we

implemented a static checker and applied it to the studied systems. In total, the preliminary static checker detects 924 test smell instances. Our static checker is publicly available and can be found in the replicate package. Future studies should further investigate the impact of these test smell instances.

*We uncover and document four dependency-related test smells through a manual analysis. We reported instances of the test smells and they are either confirmed or fixed by developers. Fixing such test smells may reduce excessive dependencies and improve test case design.*

## 5 Implication and Future Directions

In this section, we discuss our key findings and their implications. We also highlight future research opportunities and provide recommendations on the adoption of test impact analysis in a CI setting.

**Adopting test impact analysis in CI settings can reduce accumulated test execution overhead**. Our test impact analysis results in CI settings demonstrate potential in reducing the accumulated testing overhead in CI settings. Even for the least-effective case in our studied systems, an average of over 20% of the test execution time can be saved for each run of continuous testing triggered by code integration. Across all the studied systems, the median percentage of saved time is around 50%. With the increasing test execution frequency, test impact analysis, with its current effectiveness, already shows some benefits if integrated into CI practices.

**Needs for further improving the effectiveness of test impact analysis in a CI setting**. Despite the non-trivial test execution time saved by test impact analysis, we believe the effectiveness of test impact analysis can be further improved. Our study reveals that there exists a high degree of dependencies between source code and test cases, and among test cases: on average, 15% of the test case have dependencies with over 20 source code files. As code dependencies play a pivot role in the effectiveness of test impact analysis, reducing code dependencies can unleash the full potential of test impact analysis in reducing test overhead in the CI setting. Practitioners can adopt common approaches that can reduce code dependencies concerning test cases, such as refactoring and mocking. Moreover, in this work, we took a closer look at the dependencies among test cases and concluded four dependency-related test smells. Such test smells negatively impact test quality and introduce unneeded dependencies. We implemented and released a prototype tool that detects these test smells. Future studies and practitioners may leverage the tool to help maintain test quality on a regular basis.

**Needs and future research opportunities for proposing specialized techniques that reduce code dependencies to improve the effectiveness of test impact analysis**. Our findings highlight the need for more specialized techniques that can reduce code dependencies in a more targeted way. We find that the number of changed files only has a moderate positive correlation with the percentage of the impacted test cases. This indicates that some changed files may have a larger impact than the others. For example, we uncovered that the file HBaseTestingUtility in HBase has direct and indirect dependencies with 760 test cases and was frequently modified. Such files reduce the effectiveness of test impact analysis and cause significant accumulated test execution overhead. Future research on test impact analysis should consider various properties of a changed file, such as its change-proneness and the importance in the dependency graph (e.g., the number of direct dependencies may be small, but the propagation scope may be large). In addition, such specialized techniques can be integrated in a just-in-time fashion, which provides more prompt feedback to developers. For example, techniques that detect and reduce unneeded dependencies (such as our implemented test smell detection) can be deployed to check every code integration, and seeks for early resolution of unneeded dependencies, before which may degrade the effectiveness of test impact analysis. Moreover, future techniques may examine the possibility of providing an interactive development environment that allows developers to exclude files that have high dependencies but the changes are less error-prone.

**Needs for revisiting and improving test case design**. Our manual analysis reveals that inheritance and utility methods are the two major causes of dependencies among test cases (inheritance: 79.4%, utilities: 13.2%). While inheritance and utilities are standard in improving code maintenance, modularization, and usability, the necessity and negative impacts of such code reuse in test cases should be revisited more thoroughly by future work. Intuitively, inheritance and sharing utility code may violate the test independence assumption, and inheritance may even reduce test code readability and increase test execution overhead [63]. Our manually-uncovered dependency-related test smells confirm such intuition to some extent: Code reuse through inheritance and utility may cause hard-to-maintain test fixtures (Test Smell #3), unstable test environments (Test Smell #2), and flaky tests (Test Smell #4). In short, our work calls for future research efforts to revisit the current test design and its impact on test quality, and the effectiveness of test impact analysis. Moreover, automated approaches are needed to refactor test cases for better design and improved effectiveness of test impact analysis.

## 6 Threats to Validity

**External Validity.** We conduct our study on eleven large-scale open source systems in different domains. We find that the overall findings hold in all the studied systems. Our studied systems are all implemented in Java, so the results may not be generalizable to systems in other programming languages. Future studies should validate the generalizability of our findings in systems that are implemented in other programming languages. Although the studied systems have different levels of development activities, our test impact analysis in CI is limited to a one-year period and does not cover the very beginning of the system development. Future studies can more thoroughly perform test impact analysis and expand it to the entire development lifecycle of software systems.

**Construct Validity.** In this paper, we use static analysis to uncover the dependencies between test cases and other source code files. During our manual study, we did not find

any false positives that are caused by our static analysis approach. We choose static analysis over dynamic analysis for recovering dependencies at the class-level because of the three following reasons: 1) Prior studies [8, 28] found that static test impact analysis have similar performance compared to dynamic-based techniques, and class-level dependency gives better results compared to method-level dependency. 2) A large number of test cases suffer from flakiness [64]. Flaky tests expose different behaviors among multiple runs and may result in differences in code coverage. 3) In each CI execution, there may exist many test cases not being executed due to various reasons (e.g., failures of preceding test cases) [65, 66]. 4) Using static analysis is shown to have high accuracy in identifying class dependencies [67]. Future studies may use dynamic analysis to re-evaluate our findings. We identify a file as a test case if it contains calls to testing frameworks such as JUnit or TestNG (i.e., contain *@Test* annotation). Although we did not find any false positives during our manual study, some of the identified test cases may be skipped by developers during the build process. Future studies are needed to verify the accuracy of our test identification approach. Note that, older versions of the testing frameworks (e.g., JUnit 3) use inheritance to define a test case (i.e., by calling `extends TestCase`) and not *@Test* annotation. In such cases, our test identification approach may not work properly. However, the studied systems are using newer versions of the testing frameworks, which use *@Test* annotation to define test cases. Our study focuses on studying Java source code and test cases. Although the majority of the studied systems are written in Java, some of them may contain code that is written in different programming languages. For example, Flink contains 25% non-Java tests, Kafka contains 27% non-Java tests, Hadoop contains 0.8% C++ tests, and Hbase contains 4.2% JavaScript and 1.5% Ruby tests. After some manual investigation of the build script and the executed tests in the CI process, we find that these non-Java tests are either excluded in the daily CI build, or are executed in a separate CI job (i.e., does not affect the CI process of the Java components). Future studies are needed to evaluate the effect of the polyglot nature of a system on its test design and execution.

**Internal Validity.** In this paper, we use static analysis to uncover the class dependencies. However, there may be some limitations in the static analysis (e.g., difficult to analyze reflection) that cause inaccurate results. Even though we did not find such cases in our manual study, future studies should validate our findings on other systems.

## 7 Conclusion

To reduce test execution overhead, prior studies have proposed and evaluated techniques such as test impact analysis in the traditional software development settings. However, the effectiveness of test impact analysis on reducing the accumulative testing overhead remains unknown in the continuous integration setting. In this paper, we first study the effectiveness of static test impact analysis on 11 open-source systems in the continuous testing setting. We analyzed one year of software development history. We found that most test cases (around 50% or more) are impacted in the daily test execution due to high code dependencies, although the number of changed files is small. Motivated by our finding, we further studied the code dependencies between the source code files and test cases, and among test cases. We found that most test cases cover the integrated behavior of the system, and many test cases cover more than 20 source code files. We also found that 18% of the test cases have dependencies with other test cases. Finally, we documented four dependency-related test smells that we manually uncovered. In short, our study highlights the needs and provides insights on reducing test execution overheads and improving test design.

## References


[1] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 211–222.

[2] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 237–247.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[4] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering*, ser. ISSRE '11, 2011, pp. 170–179.

[5] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 210–221.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[7] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. FSE'12, 2004, pp. 241–251.

[8] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 583–594.

[9] Microsoft, "Test impact analysis in visual studio test," https://docs.microsoft.com/en-us/azure/devops/pipelines/test/test-impact-analysis?view=azure-devops, 2019, last accessed May 2019.

[10] A. Jenkins, "Apache Jenkins CI test results," https://builds.apache.org/, 2019, last accessed Nov 2019.

[11] A. Najafi, W. Shang, and P. C. Rigby, "Improving test effectiveness using test executions history: An industrial experience report," in *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[12] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 805–816.

[13] D. S. andMichael D. Ernst, "Reducing wasted development time via continuous testing," in *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA*, 2003, pp. 281–292. [Online]. Available: https://doi.org/10.1109/ISSRE.2003.1251050

[14] K. Muslu, Y. Brun, and A. Meliou, "Data debugging with continuous testing," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 631–634. [Online]. Available: https://doi.org/10.1145/2491411.2494580

[15] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *Proceedings of the 39th International Conference on Software Engineer-*

*ing: Software Engineering in Practice Track*, ser. ICSE-SEIP '17, 2017, pp. 243–252.

[16] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

[17] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 700–711.

[18] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[19] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, Nov. 2012.

[20] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[21] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 268–279.

[22] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, Feb. 2014.

[23] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering Methodology*, vol. 22, no. 3, pp. 19:1–19:29, Jul. 2013.

[24] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 192–201.

[25] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 235–245.

[26] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 540–543.

[27] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME18)*. IEEE, 2018.

[28] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, "How do static and dynamic test case prioritization techniques perform on modern software systems? an extensive study on github projects," *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1054–1080, 2019.

[29] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[30] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610404

[31] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, 2018, pp. 1–11. [Online]. Available: https://doi.org/10.1109/ICST.2018.00011

[32] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock?: An empirical study on mocking practices," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17, 2017, pp. 402–412.

[33] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 33:1–33:11.

[34] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 643–653.

[35] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15, 2015, pp. 101–110.

[36] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME 2017, 2017, pp. 1–12.

[37] JavaParser, https://javaparser.org/, 2019, last accessed Feb 1 2019.

[38] E. Hautus, "Improving java software through package structure analysis," in *The 6th IASTED International Conference Software Engineering and Applications*, 2002.

[39] S. Grant, J. R. Cordy, and D. B. Skillicorn, "Using heuristics to estimate an appropriate number of latent topics in source code analysis," *Science of Computer Programming*, vol. 78, no. 9, pp. 1663 – 1678, 2013.

[40] S. Grant, J. R. Cordy, and D. B. Skillicorn, "Using topic models to support software maintenance," in *16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 403–408.

[41] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems," *Empirical Software Engineering*, Nov 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9663-0

[42] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 492–501.

[43] D. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, Sep. 2005.

[44] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Proceedings of the 25th International Symposium on Software Reliability Engineering*, ser. ISSRE '14, 2014, pp. 201–211.

[45] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007, pp. 417–420.

[46] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2015, pp. 61–70.

[47] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 267–276. [Online]. Available: https://doi.org/10.1109/ISSRE.2005.28

[48] A. Qusef, R. Oliveto, and A. De Lucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10.

[49] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "Ezunit: A framework for associating failed unit tests with potential programming errors," in *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming*, ser. XP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 101–104. [Online]. Available: http://dl.acm.org/citation.cfm?id=1768961.1768979

[50] J. Shore, "Fail fast [software debugging]," *IEEE Software*, vol. 21, no. 5, pp. 21–25, Sep. 2004.

[51] S. Boslaugh and P. Watters, *Statistics in a Nutshell: A Desktop Quick Reference*, ser. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[52] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[53] Z. Li, T.-H. P. Chen, J. Yang, and W. Shang, "DLFinder: Characterizing and detecting duplicate logging code smells," in *Proceedings of the 41th International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 1–1.

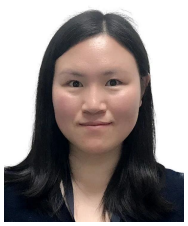
[54] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.

[55] L. Moonen and A. Yamashita, "Do code smells reflect important maintainability aspects?" in *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, ser. ICSM '12, 2012, pp.

306–315.
[56] "Three reasons why we should not use inheritance in our tests," https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/, 2019, last accessed May 2019.
[57] L. Prechelt, B. Unger, M. Philippsen, and W. F. Tichy, "A controlled experiment on inheritance depth as a cost factor for code maintenance," *Journal of Systems and Software*, vol. 65, no. 2, pp. 115 – 126, 2003. [Online]. Available: https://doi.org/10.1016/S0164-1212(02)00053-5
[58] F. Palomba and A. Zaidman, "The smell of fear: on the relation between test smells and flaky tests," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2907–2946, 2019.
[59] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proceedings of the International Conference on Software Maintenance and Evolution*, ser. ICSM '18, 2018, pp. 1–12.
[60] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, 2001, pp. 92–95.
[61] "Test smell found in manual study," https://docs.google.com/spreadsheets/d/1umM_zQeyHMPTyFTKnjqtq4apfYH7JA38d8z44YR5V8E/edit?usp=sharing.
[62] "Parameterized tests in JUnit," https://github.com/junit-team/junit4/wiki/parameterized-tests, 2019, last accessed May 2019.
[63] L. Koskela, *Effective Unit Testing: A guide for Java developers*. Manning Publications, 2013.
[64] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 433–444.
[65] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 356–367. [Online]. Available: https://doi.org/10.1109/MSR.2017.62
[66] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17, 2017, pp. 345–355.
[67] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovi, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018.

**Tse-Hsun (Peter) Chen** Tse-Hsun (Peter) Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work has been published in flagship conferences and journals such as ICSE, FSE, TSE, EMSE, and MSR. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSME, SANER, and ICPC, and he is a regular reviewer for software engineering journals such as JSS, EMSE, and TSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.

**Jinqiu Yang** Jinqiu Yang is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Her research interests include automated program repair, software testing, software text analytics, and mining software repositories. Her work has been published in flagship conferences and journals such as ICSE, FSE, EMSE. She serves regularly as a program committee member of international conferences in Software Engineering, such as ASE, ICSE, ICSME and SANER. She is a regular reviewer for Software Engineering journals such as EMSE and JSS. Dr. Yang obtained her BEng from Nanjing University, and MSc and PhD from University of Waterloo. More information at: https://jinqiuyang.github.io/.



**Zi Peng** Zi Peng is a Masters student at the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. She obtained her B.Eng. from Zhejiang University, China. Her research interests include software testing, program analysis, and mining software repositories.