Investigating Developers' Perception on Software Testability and its Effects

Tushar Sharma[1] ✉, **Stefanos Georgiou**[2], **Maria Kechagia**[3], **Taher A. Ghaleb**[4], **Federica Sarro**[3]

[1]Dalhousie University, Canada; [2]simpleTechs, Germany; [3]University College London, United Kingdom; [4]University of Ottawa, Canada

✉ **For correspondence:** tushar@dal.ca

Data availability: Replication package can be found on GitHub - https://github.com/SMART-Dal/testability

Abstract

The opinions and perspectives of software developers are highly regarded in software engineering research. The experience and knowledge of software practitioners are frequently sought to validate assumptions and evaluate software engineering tools, techniques, and methods. However, experimental evidence may unveil further or different insights, and in some cases even contradict developers' perspectives. In this work, we investigate the correlation between software developers' perspectives and experimental evidence about testability smells (*i.e.,* programming practices that may reduce the testability of a software system). Specifically, we first elicit opinions and perspectives of software developers through a questionnaire survey on a catalog of four testability smells, we curated for this work. We also extend our tool DesigniteJava to automatically detect these smells in order to gather empirical evidence on testability smells. To this end we conduct a large-scale empirical study on 1,115 Java repositories containing approximately 46 million lines of code to investigate the relationship of testability smells with test quality, number of tests, and reported bugs. Our results show that testability smells do not correlate with test smells at the class granularity or with test suit size. Furthermore, we do not find a causal relationship between testability smells and bugs. Moreover, our results highlight that the empirical evidence does not match developers' perspective on testability smells. Thus, suggesting that despite developers' invaluable experience, their opinions and perspectives might need to be complemented with empirical evidence before bringing it into practice. This further confirms the importance of data-driven software engineering, which advocates the need and value of ensuring that all design and development decisions are supported by data.

Keywords: Software testability; software test quality; testability smells; developers' opinions and perspectives; software quality.

1. Introduction

The opinions and perspectives of software developers matter significantly in software engineering research. The research in the domain relies on the experience and knowledge of software practitioners to validate assumptions and evaluate the tools, techniques, and methods addressing software engineering problems. Numerous studies reveal the importance of the practitioners' perspectives [14, 55, 53, 1]. However, empirical evidence may not always agree with developers' perspective, opinions, or beliefs. For example, Devanbu *et al.* [13] show that software developers

have very strong beliefs on certain topics, but are often based on their personal experience; such beliefs and corresponding empirical evidence may be inconsistent. Similarly, despite developers' fairly common negative perspectives about code clones, Rahman *et al.* [54] did not find sufficient empirical evidence to prove a strong correlation between code clones and bug proneness; however, there could be effects of code clones other than bugs. Along the similar lines, a study by Murphy *et al.* [43] casts doubts on practitioners' and researchers' assumptions related to refactoring. Finally, Janes *et al.* [28] challenge the practitioners' perspective on agile process by bringing out the dark side of agility. In this paper, we present a case study of *testability smells*, *i.e., programming practices that may negatively affect testability of a software system*, to investigate whether the software developers' perspectives about testability smells is backed up by empirical evidence .

Researchers and practitioners have proposed various, yet inconsistent, definitions of software testability [21]. The most common definition refers to *the degree to which the development of test cases can be facilitated by the software design choices* [9, 73, 7]. Specifically, several researchers [9, 73, 50] define testability as the *ease of testing*. In addition, some researchers [9, 7, 75] emphasized that testability is not a binary concept but must be expressed in *degree* or *extent*. Additionally, other researchers [9, 75, 52] explicitly connect software *design choices* with the definition of software testability. Furthermore, some studies [9, 50, 75] identify the degree of *effectiveness* by which test development is facilitated as another characteristic of testability definition.

There has been a significant amount of work on test smells and their effects [30, 60, 4]. Test smells are bad programming practices in test code that negatively affect the quality of test suites and production code [23]. Though looks similar, practices that affect testability are completely different than test smells. Test smells occur in test code while issues affecting testability arise in production code. Also, test smells indicate the poor quality of test cases whereas testability impacts the ability to write tests.

Several researchers have proposed frameworks for measuring and empirically evaluating software testability by considering (1) software design choices, including programming language features [71, 59], (2) software quality metrics, including cohesion, depth of inheritance tree, coupling, and propagation rate of methods [39], metrics including the number of method calls, dependencies, and attributes of a class with testability [41], and other software metrics [32, 59, 8], as well as (3) testing effort [64]. However, there are still various aspects related to software testability that remain unexplored, including the extent to which specific programming practices (*e.g.,* handling of dependencies, coding style, and access of modifiers) impact software testability.

The *goal* of this study is to find experimental evidence to validate current practitioners' perspectives about testability smells. To achieve this goal, we first curate a catalog of four programming practices which can affect software testability, referred to as *testability smells*. We then gather software developers' perspectives through a questionnaire survey on testability in general, and our proposed testability smells in particular. Finally, we conduct a large-scale empirical study guided by three research questions to explore the effect of testability smells on test cases, their quality, and on reported bugs at different granularity levels. To support the detection of testability and test smells, we develop a tool named DesigniteJava. To answer the research questions, we curated a dataset of 1,115 Java software projects which are publicly available in GitHub, and analyzed them using our DesigniteJava tool.

Our survey shows that software developers consider testability as a factor that impedes software testing and overwhelmingly acknowledged the proposed testability smells. Our results suggest that testability smells show a low positive correlation with test smells at the repository granularity; however, at the class-level, testability smells and test smells do not correlate. Our exploration of the relationship between testability smells with test density reveals no correlation at repository and class granularity. Finally, our observations from our experiment indicate that testability smells do not contribute to bugs. Therefore, developers' opinions and perspectives might need to be complemented with empirical evidence before bringing it into practice.

This study makes the following contributions to the field.

1. We investigate the extent to which developers' perspectives is in line with the empirical evidence we found in the context of *i.e.,* testability smells.
2. We consolidate a set of programming practices that affect testability of a software system in the form of a catalog of testability smells. This catalog provides a vocabulary for researchers and practitioners to discuss specific programming practices potentially impacting the testability of software systems.
3. We extend DesigniteJava to detect the proposed testability smells and eight test smells. The tool facilitates further research on the topic of testability smells. Also, interested software developers may use this tool to detect testability smells in their source code to better understand the impact of design choices on testing.
4. We explore the relationships between testability smells and several aspects relevant to test development and bugs. Such an exploration improves our understanding, both as software developers and researchers, of testability and lays the groundwork for devising new tools and techniques to improve test development.

We have made publicly available our DesigniteJava to identify testability smells as well as a replication package at https://github.com/SMART-Dal/testability. We hope this facilitate other researchers to replicate, reproduce and extend the presented study.

The rest of this paper is organized as follows. First, we present related work in Section 2. Section 3 provides overview of the methods. Section 4 presents the initial catalog of testability smells, our questionnaire survey targeting to software practitioners and obtained results, and tool implementation to detect testability and test smells. We present the mechanism followed to select and download repositories from GitHub in Section 5. We discuss results in Section 6 and their implications in Section 7. Threats to validity are discussed in Section 8. Finally, we conclude in Section 9.

2. Related work

Software testability. From existing studies, we found that testability was initially considered for hardware design [36, 72, 37]. The concepts of hardware testability were then used for software testability [44, 34]. Afterwards, a great deal of studies has been conducted exploring various aspects of software testability. To measure testability for data-flow software, Nguyen *et al.* [45] suggested an approach that uses the SATAN method, which transforms the source code into a static single assessment form. The form is then fed into a testability model to detect source code parts with testability weaknesses. Bruntink *et al.* [8] collected a large number of source code metrics (*e.g.,* depth of inheritance tree, fan out, and lack of cohesion of methods) and test code metrics to explore the relationship with testability. The analysis focused on open-source Java applications. The results suggest that there is a significant correlation between class-level metrics (most notably fan out, LOC per class, and response for class) and test-level metrics (LOC per class and the number of test cases). Vincent *et al.* [69] investigated software components testability written in C++ and Java in workstations and embedded systems. Moreover, the authors have suggested an approach named *built-in-test* for run-time-testability which can provide more testable, reliable, and maintainable software components. Filho *et al.* [18] used ten testability attributes, proposed in previous studies, to examine their correlation with source code metrics and test specification metrics (*e.g.,* number of test cases, test coverage) on two Android applications. They found that testability attributes are correlated with several source code metrics and test specification metrics. Chowdhary [9] presented experiences while applying testability concepts and introduced guidelines to ensure that testability is taken under consideration during software planning and testing. Based on these findings, the authors introduced a testability framework called SHOCK. Furthermore, various resources [27, 75] discussed their interpretation of testability and impact of smells affecting testability.

Assessing testability. Voas [29] surveyed the factors that affect software testability, arguing that a piece of software that is likely to reveal faults within itself during testing is said to have high

testability. According to this work, *information loss* is a phenomenon that occurs during program execution and increases the likelihood that a fault will remain undetected. Finally, Voas [71] compared the testability of both object-oriented and procedural systems, as well as whether testability is affected by programming language characteristics.

Surveys on testability. Various literature surveys on testability have been carried out. Freedman [19] investigated the testability of software components. Freedman argued that the concept of domain testability of software is defined by applying the concepts of observability and controllability to software. Garousi *et al.* [21] examined 208 papers on testability (published between 1982 and 2017) and also found that the two most commonly referred factors affecting testability are observability and controllability. Furthermore, their survey argues that common ways to improve testability are testability transformation, improving observability, adding assertions, and improving controllability. Similarly, Hassan *et al.* [25] conducted a systematic literature review on software testability to investigate to what extent it affects software robustness. Results show that a variety of testability issues are indeed relevant, with observability and controllability issues being the most researched. They also found that fault tolerance, exception handling, and handling external influence are prominent robustness issues.

Test smells. A wide variety of studies explored test smells and their effect and relationship on various aspects of software development, including change and bug proneness [60], maintainability [5, 33, 67], and test flakiness [16]. Specifically, Spadini *et al.* [60] investigated the relationship between the presence of test smells and the change-and defect-proneness of test code, as well as the defect-proneness of the tested production code. Among their findings, they observed that tests with smells are indeed more change- and defect-prone. Regarding maintainability, Bavota *et al.* [5] presented empirical studies on test smells, and showed that test smells have a strong negative impact on program comprehension and maintenance. They, also, found that comprehension is 30% better in the absence of test smells. Furthermore, Kim *et al.* [33] conducted an empirical study to study the evolution and maintenance of test smells. They found that the number of test smells increases as a system evolves, and through a qualitative analysis they revealed that most test smell removal is a maintenance activities. Additionally, Tufano *et al.* [67] showed that test smells are usually introduced when the corresponding test code is committed in the repository for the first time. Then, those test smells tend to remain in a system for a long time, hindering software maintenance. Fatima *et al.* [16] developed an approach called *Flakify*, which is a black-box, language model-based predictor for flaky test cases.

Despite extensive work on testability, the existing literature does not translate high-level principles such as observability and controllability into actionable programming practices. Due to that though the high-level testability principles have been known to the community for a long time, there has been no tool support to detect them. We provide a tool that supports the detection of testability smells. Furthermore, we explore the relationship between testability practices and test quality and size, which is our other contribution.

3. Overview of the Methods

In the pursuit of weighing developers' perceptions with empirical evidence in the context of testability smells, we formulate the following research questions.

RQ1. *To what extent do testability smells and test smells correlate?*
Testability smells refer to bad programming practices that are believed to make test case design and development difficult. Developers may choose to follow non-optimal practices when it is not easy to write tests, leading to poor-quality test cases. Test smells refer to bad programming practices in unit test code, compromising test code quality by violating the best practices recommended for writing test code [12]. This research question explores whether and to which extent testability smells and test smells correlate.

RQ2. *Do testability smells correlate with test suite size?*
By definition, testability smells make the design and development of test cases difficult. With this research question, we aim to empirically evaluate whether the presence of testability smells can hinder test development and consequently lead to a fewer number of test cases. By answering this research question, we can inform developers about testability smells that might impede a smooth test development.

RQ3. *Do testability smells cause more bugs?*
Testability smells make it harder for developers to test a software system. This implies that the software under test lacks appropriate testing, leaving more bugs uncovered during software development. This research question aims to investigate whether and to which extent testability smells can lead to a higher number of reported bugs.

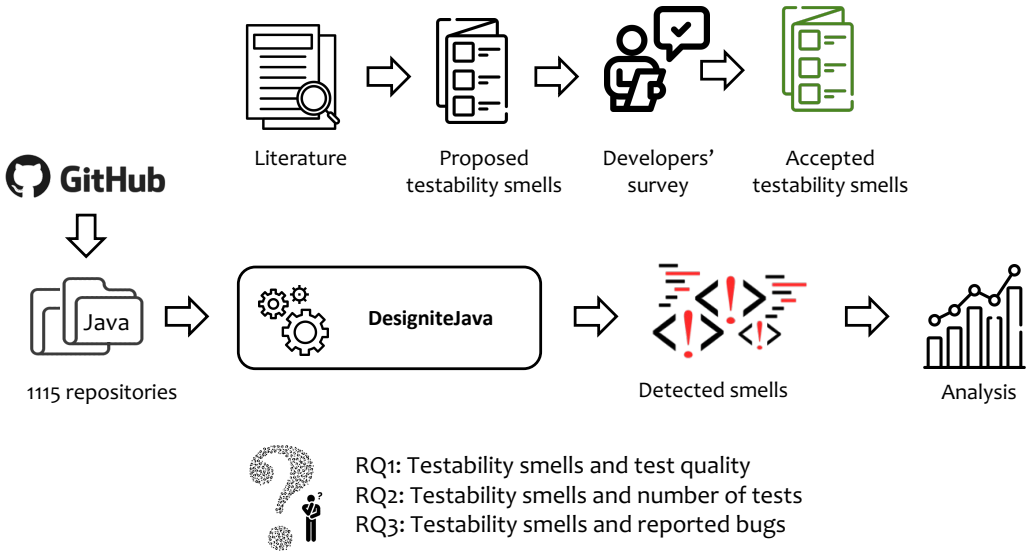

Figure 1. Overview of the study

Towards the goal of the study, as outlined in Figure 1, we first prepared a set of potential testability smells based on the available literature and recommended practices. We then carried out an online survey to understand developers' perspectives on software testability and to gauge the extent to which they agree that those smells really impact testability negatively. We extended our tool (DesigniteJava) to detect testability and test smells. We analyzed 1,115 Java repositories downloaded from GitHub. After identifying the smells, we reported our observations and findings with respect to each research question.

4. Testability smells

We define testability smells as the programming practices that reduce the testability of a software system. This section presents an initial catalog of testability smells, validates them by carrying out an online developers' survey, and discusses the implementation details of our tool.

4.1 Initial Catalog of Testability Smells

In order to identify specific programming practices that negatively affect testability, we carried out a light-weight multi-vocal literature (MLR) review, which surveys writings, views, and opinions in diverse form and format [22]. The review process has three main stages: *search*, *selection*, and *information extraction*.

In the search stage, two of the co-authors searched for a set of search terms (including testability, ease of testing, and software design+test) on Google Scholar and Google Search. For each

search term, we manually searched minimum seven pages of search results. After the minimum threshold of seven pages, we continued the search until we get two continuous search pages without any new and relevant articles. Adopting this mechanism avoided missing any relevant articles in the context of our study.

We applied inclusion and exclusion criteria to filter out irrelevant sources. The main inclusion criterion was that the source's content must relate to testability. Examples of exclusion criteria include dropping gray literature that is too short, written in language other than English, or presented without objectivity in presentation. These examples map to the *Objectivity* requirement of MLR process guidelines [22].

In the last stage, we read or observed the selected resources and extracted information relevant to our study. Specifically, we strived for concrete recommendations in terms of programming practices that influence testability from the selected sources. We grouped the practices based on similarity and assigned an appropriate name reflecting the rationale. We identified four potential testability smells discussed by more than one selected source. We present the consolidated set of smells below. It is the first attempt, to the best of our knowledge, to document specific programming practices as testability smells. It is by no means a comprehensive list of testability smells; we encourage the research community to further extend this initial catalog of testability smells.

4.1.1 Hard-wired dependencies

This smell occurs when a concrete class is instantiated and used in a class resulting in a *hard-wired dependency* [9, 75, 26]. A *hard-wired dependency* creates tight-coupling between concrete classes and reduces the ease of writing tests for the class [9]. Such a *hard-wired dependency* makes the class difficult to test because the newly instantiated objects are not replaceable with test doubles (such as stubs and mocks). Hence, the test will check not only the CUT (class under test) but also its dependencies, which is undesirable.

In Listing 1, the `parse`[1] method creates an object of the `BindingOperation` class (line 4) and calls a few methods (lines 6 and 7). The object cannot be replaced at testing execution due to the concrete object creation and its use within this method. Hence, the hard-coded dependency is reducing the ease of writing tests for the class.

```
1  private void parse(String name, String namespace, WsdlParser parser) throws WsdlParseException
       {
2      if (WSDL_NS.equals(namespace)) {
3          if (OPERATION.equals(name)) {
4              BindingOperation operation = new BindingOperation(definitions);
5              operation.read(parser);
6              operations.put(operation.getQName(), operation);
7          }
8      }
9  //rest of the method
10 }
```

Listing 1. Example of hard-coded dependency

4.1.2 Global state

Global variables are, in general, widely discouraged [40, 63]. This smell arises when a global variable or a Singleton object is used [26, 62, 17, 65]. Global variables create hidden channels of communication among abstractions in the system even when they do not depend on each other explicitly. Global variables introduce unpredictability and hence make tests difficult to write by developers.

The `Builder`[2] class in Listing 2 is accessible, and hence can be read/written, within the entire project. Such practice makes it difficult to predict the state of the object in tests.

[1] https://github.com/forcedotcom/wsc/blob/master/src/main/java/com/sforce/ws/wsdl/Binding.java
[2] https://github.com/forcedotcom/wsc/blob/master/src/main/java/com/sforce/async/JobInfo.java

```
public static class Builder {
    //class definition
}
```

Listing 2. Example of global state

4.1.3 Excessive dependency

This smell occurs when the class under test has excessive outgoing dependencies. Dependencies make testing harder; a large number of dependencies makes it difficult to write tests for the class under test in isolation [62, 39, 74]. For example, the `Error`[3] class in the open-source project wsc refers to nine other classes within the project --- `Bind`, `BulkConnection`, `TypeInfo`, `StatusCode`, `XmlOutputStream`, `XmlInputStream`, `ConnectionException`, `TypeMapper`, and `Verbose`. Such a high number of dependencies on other classes increases the effort to write tests for this class to be tested in isolation.

4.1.4 Law of Demeter violation

This smell arises when the class under test violates the law of Demeter *i.e.,* the class is interacting with objects that are neither class members nor method parameters [38, 31, 65]. In other words, the class has a chain of method calls such as `x.getY().doThat()`. Violations of the law of Demeter create additional dependencies that a test has to take care of. For example, lines 4 and 5 of the snippet[1] given in Listing 3 call a method to obtain an object that in turn calls another method on the obtained object. Such method chains introduce indirect dependencies that reduce the ease of writing tests for the class.

```
public Iterator<Part> getAllHeaders() throws ConnectionException {
    HashSet<Part> headers = new HashSet<Part>();
    for (BindingOperation operation : operations.values()) {
        addHeaders(operation.getInput().getHeaders(), headers);
        addHeaders(operation.getOutput().getHeaders(), headers);
    }
    return headers.iterator();
}
```

Listing 3. Example of the law of Demeter violation

4.2 Developer Survey

We carried out an anonymous online questionnaire survey targeting software developers to understand their perspectives on software testability. Specifically, we aimed to consolidate developers' perspectives *w.r.t.* the definition of testability as well as the relevance of our identified testability smells. We divided our survey into three sections. In the first section, we collected information about developers' experience. In the second section, we asked developers how they define testability. The final section presented our initial catalog of testability smells and asked the respondents whether and to what extent they agree that the presented practices negatively affect testability. All the questions in this section were Likert-scale questions. The questionnaire that we used is available online [57].

Before rolling out the survey to a larger audience, we ran a pilot for the survey, collected feedback, and improved the survey. We shared the survey on all online professional social media channels (such as Twitter, LinkedIn, and Reddit) and sought participation from the software development community. We kept the survey open for six weeks. We received 45 complete responses.

4.2.1 Findings from the survey

Figure 2 presents the demographic distribution of participants in terms of years of experience classified by their roles. We asked them to check all applicable roles and hence the total number

[3] https://github.com/forcedotcom/wsc/blob/master/src/main/java/com/sforce/async/Error.java

of responses shown in the figure is more than the number of participants. It is evident that most of the participants belong to the "software developer" role; a significant number of the participants belong to the highly experienced group (11-20 years).

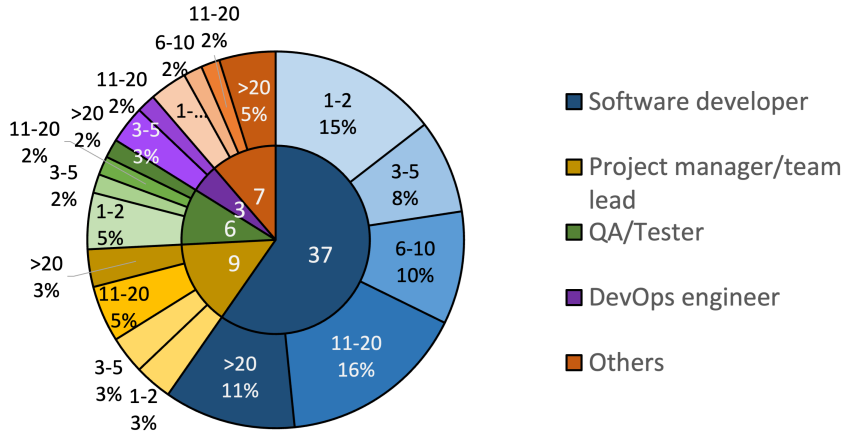

Figure 2. Demographics (role and experience in number of years) of participants

Definition of software testability: We asked the participants a question to elicit the definition of software testability. Most of the responses point to **the degree of ease with which automated tests can be written**. Some of the actual responses are: *"The extent to which a software component can be tested"*, *"software testability is the degree that software artifacts support testing"*, *"easy testing"*, and *"a measure of how easy it is for the code to be tested through automated tests"*. An interested reader may look at the raw anonymized responses in our replication package [57].

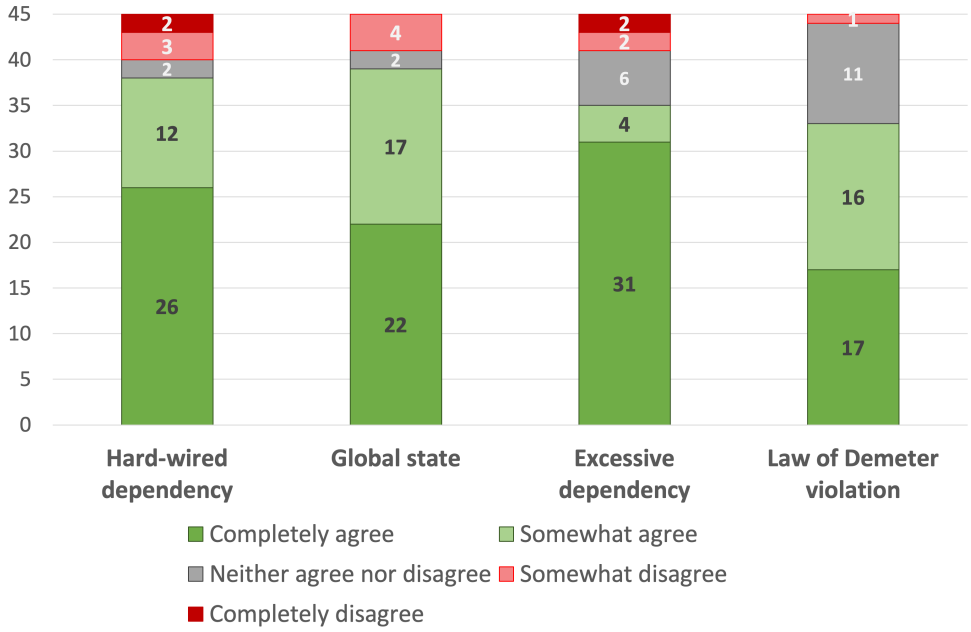

Figure 3. Respondents' perspective of considered testability smells

Programming practices affecting testability: The next set of questions presented four programming practices corresponding to each potential testability smell and asked the respondents whether and to what extent these practices negatively affect software testability. We also asked

about the rationale for their choice. Figure 3 presents the consolidated responses for all four considered smells. A very large percentage (84%, 87%, 78%, and 73% respectively for the four considered smells) of the responses agreed (either completely or somewhat agree) to mark the presented practices as testability smells.

We looked into the rationale provided by respondents for other options (*i.e.,* neither agree nor disagree, somewhat and completely disagree). Specifically, for *hard-wired dependency*, one of the respondents who marked *completely disagree* did not offer any justification; another respondent suggested to use mocking. One respondent with a *somewhat disagree* option for the same smell basing his/her answer on the assumption that dependencies are trivial (*i.e.,* internal class or trivial class from a library) most of the time. Respondents who opted for the option "*neither agree nor disagree*" either expressed their ignorance about the specific question or left the rationale question unanswered.

The respondents of "*somewhat disagree*" option for *global state* smell justify their selection by providing a workaround to test a unit with global variables; for example, one of the respondents provided the following rationale: "Logging where and when the global state is altered is usually good enough for testing the code I work with".

Whereas, those who chose the "*completely disagree*" option for the *excessive dependency* smell, seem, however, to agree that it is difficult to test source code containing this smell based on their open answers (for example, one such an answer states "If designed properly then testing won't be difficult but yes more dependencies need extra setups").

For the *law of Demeter violation* smell, a considerable number of respondents chose the "*neither agree nor disagree*" option; however, they did not provide any fruitful rationale towards this testability smell.

Additional programming practices affecting testability: We also enquired about other programming practices that may negatively influence the testability of a software system. The responses provided us with additional practices such as poor separation of concern (mixing UI and non-UI aspects), interaction with external resources, such as sockets, files, and databases, time dependencies, asynchronous operations, reflection, invoking command line from code, methods that do not return anything but changes internal state, and requirements for authentication credentials that hinder testability. In addition, the respondents mentioned spaghetti code, highly tangled code, large methods, and non-standard environments as practices that reduce testability. Some of the indicated practices are covered by the proposed smells. For example, interaction with external resources has been captured by the *hard-wired dependency* smell since external resources such as a network connection need to be instantiated.

The results from the survey not only suggest that the investigated smells are indeed considered practices that affect testability negatively but also provide indicators for the community to extend the proposed catalog.

4.3 The DesigniteJava Tool

We extended our tool DesigniteJava [56], to support for testability and test smells detection.[4] We select DesigniteJava to extend because the tool detects a variety of code smells and it has been used in various studies [46, 58, 15, 68, 2]. Architecturally, DesigniteJava is structured in three layers as shown in Figure 4. Eclipse Java Development Toolkit (JDT) forms the bottom layer. DesigniteJava utilizes JDT to parse the source code, prepare ASTs, and resolve symbols. The source model is the middle layer. The model invokes JDT and maintains a source code model from the information extracted from an AST with the help of JDT. The top layer of the tool contains the business logic *i.e.,* the smell detection and code quality metrics computation logic. The layer accesses the source model, identifies smells and computes metrics, and outputs the generated information in either .CSV or .XML files. Due to the existing support to detect smells and compute metrics, various fea-

[4] https://www.designite-tools.com/blog/understanding-testability-test-smells

tures (such as the source model) can be reused in our context. To support testability and test smell detection, we added code in the code smell detection layer. We also modified the source model layer to extract additional information required for our purpose. The extended version of the tool can be downloaded from its website.[5]

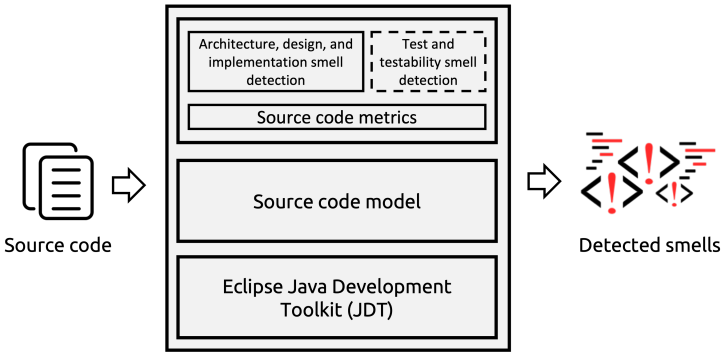

Figure 4. Architecture of DesigniteJava tool

Existing explorations have proposed a few tools to detect test smells. We first tried to utilize existing tools, specifically JNose [70] and TsDetect [51]. We were able to use JNose after taking help from its authors and developing a wrapper to use the tool as a console application. However, a quick analysis of the produced results showed a considerable number of false positive and false negative smell instances. Similarly, we were unable to use TsDetect because it is not suitable to analyze a large number of repositories due to a manual step requiring a mapping of test files and corresponding production files. Finally, we decided to develop our own test smell detector to identify the following eight test smells---*Assertion roulette, Conditional test logic, Constructor initialization, Eager test, Empty test, Exception handling, Ignored test,* and *Unknown test*. We selected these smells because these were commonly known test smells and both the tools, *i.e.,* TsDetect and JNose, support them. We implemented the support to detect test smells in DesigniteJava along with testability smells.

4.3.1 Detection rules for testability smells

We summarize below the detection strategies used for the testability smells.

***Hard-wired dependency*:** We first detect all the objects created using the `new` operator in a class. Then, if the functionality of the newly created object is used (*i.e.,* at least one method is called) in the same class, we detect this smell.

***Global state*:** If a class or a field in a class is declared with `public static` modifiers, we detect this smell.

***Excessive dependency*:** We compute *fan-out* (*i.e.,* total number of outgoing dependencies) of a class. If the fan-out of the class is more than a pre-defined threshold, we detect the smell. The literature [47, 48, 76] suggests a threshold value for fan-out between 5 and 15 with a varying compliance rate. We adopted 7 as the threshold value as suggested by Arar et al. [76]. We ensure that the threshold value is configurable; hence, future studies may change any of the thresholds used.

***Law of Demeter violation*:** We detect all the method invocation chains of the form `aField.getObject().aMethod()`. We detect this smell when method calls are made on objects that are not directly associated with the current class.

4.3.2 Detection rules for test smells

The tool uses the definition of test smells and their detection strategies from existing studies [70, 51, 3]. We present a summary of the detection strategies for the considered test smells below.

[5] https://www.designite-tools.com/designitejava/

Assertion roulette: We detect this smell when a test method contains more than one assertion statement without giving an explanation as a parameter in the assertion method.
Conditional test logic: We detect this smell when there is an assertion statement within a control statement block (e.g., `if` condition).
Constructor initialization: We detect this smell when a constructor of a test class initializes at least one instance variable.
Eager test: We detect this smell when a test method calls multiple production methods.
Empty test: We detect this smell when a test method does not contain any executable statement within its body.
Exception handling: We detect this smell when a test method asserts within a catch block or throws an exception, instead of using `Assert.Throws()`.
Ignored test: We detect this smell when a test method is ignored using the `Ignore` annotation.
Unknown test: We detect this smell when a test method does not contain any assert call or expected exception.

4.3.3 Validation

We curated a *ground truth* of smells in a Java project to manually validate the tool, as explained below.
Subject system selection: We used the RepoReapers dataset [42] and applied the following criteria to select a subject system.

1. The repository must be implemented mainly in the Java programming language
2. The repository must be of moderate size (between $10K$ and $15K$) to avoid toy projects on one side and excessive manual effort on the other
3. The repository must have a unit-test ratio > 0.0 (number of SLOC in the test files to the number of SLOC in all source files)
4. The repository must have a documentation ratio > 0.0 (number of comment lines of code to the number of non-blank lines of source code)
5. The repository must have a community size > 1 (more than one developer).

We applied the criteria and sorted the list by the number of stars. We obtained *j256/ormlite-jdbc*, *paul-hammant/paranamer*, and *forcedotcom/wsc* as the top three projects satisfying our criteria. The majority of the source code belonging to *j256/ormlite-jdbc* and *paul-hammant/paranamer* was in test cases. Hence, we selected *j256/ormlite-jdbc*,[6] as our subject system for test smells validation. However, such repositories were not suitable for validating testability smells, since we detect testability smells in non-test code. Hence, we selected *forcedotcom/wsc*,[7] a project that offers a high performance web service stack for clients, as our subject system for the manual validation of testability smells.
Validation protocol: Two evaluators manually examined the source code of the selected subject systems and documented the testability and test smells that they found. Both the evaluators hold a PhD degree in computer science and have more than 5 years of software development experience. Before carrying out the evaluation, they were introduced to testability and test smells. They were allowed to use IDE features (such as "find", "find usage" (of a variable) and "find definition" (of a class) and external tools to collect code quality metrics to help them narrow their search space. Both evaluators carried out their analyses independently. It took approximately three full work days to complete the manual analysis. After their manual analysis was complete, they matched their findings to spot any differences. We used *Cohen's Kappa* [6] to measure the inter-rater agreement between the evaluators. The obtained result, 89% and 93% respectively for testability and test smells, shows a strong agreement between the evaluators. The evaluators discussed the rest of their findings and resolved the conflicts.

[6] https://github.com/j256/ormlite-jdbc
[7] https://github.com/forcedotcom/wsc

Table 1. Results of manual validation for testability smells

Testability Smells	Manually Verified Instances	TP	FP	FN
Hard-wired dependencies	64	63	2	1
Global state	22	22	0	0
Excessive dependencies	20	19	0	1
Law of Demeter violation	66	57	0	9
Total	**172**	**161**	**2**	**11**

Table 2. Results of manual validation for test smells

Testability Smells	Manually Verified Instances	TP	FP	FN
Assertion roulette	214	212	0	2
Conditional test logic	11	11	0	0
Constructor initialization	0	0	0	0
Eager test	13	13	0	0
Empty test	0	0	0	0
Exception handling	3	2	0	1
Ignored tests	2	2	0	0
Unknown test	58	58	0	0
Total	**301**	**298**	**0**	**3**

Validation results: We used our tool, DesigniteJava, on the subject systems and identified testability and test smells. We manually matched the ground truth prepared by the evaluators and the results produced by the tool. We classified each smell instance as true positive (TP), false positive (FP), and false negative (FN). We computed precision and recall metrics using the collected data.

Table 1 presents the results of the manual evaluation for testability smells. The tool identified 161 instances of testability smells out of a total of 172 manually verified smell instances. The tool produced two false positive instances and eleven false negative instances. The false positive instances were detected mainly because the tool identified the *hard-wired dependency* even when an object was instantiated in a method call statement. Similarly, the tool reported false negatives due to an improper resolution of enumeration types; we traced back the inconsistent behavior to the JDT parser library. The precision and recall of the tool for testability smells based on the analysis is $161/(161+2) = \mathbf{0.99}$ and $161/(161+11) = \mathbf{0.94}$, respectively. Similarly, Table 2 shows the results of the manual evaluation carried out for test smells. Out of 301 test smells in 428 test methods, the tool correctly detected 298 smell instances. The cause of three instances of false negative is traced back to inconsistent behavior of the parser library. The precision and recall of the tool for test smells based on the analysis is $298/(298+0) = \mathbf{1.0}$ and $298/(298+3) = \mathbf{0.99}$, respectively. An interested reader may find the detailed manual analysis in our replication package [57].

Generalizability of conclusions: The above validation shows that the tool produces reliable results in almost all cases. Given that the tool has been used by many researchers and practitioners, occasional issues reported by them were promptly fixed, thus further improving the reliability of the tool. A few known issues and limitations of the tool remain. First, due to a symbol resolution issue in JDT, in some very peculiar cases, the tool cannot resolve the symbol that leads to issues such as inability to identify the type of a variable. Also, the tool can identify test smells only when the tests are written in the JUnit framework.

Table 3. Characteristics of the analyzed repositories

Characteristics	Count
Total number of repositories	1,115
Total lines of code	46,176,914
Total number of classes	691,481
Total number of methods	4,031,216
Total number of test cases	415,527
Total number of testability smells	637,118

5. Mining GitHub repositories

We use the following mechanism to select and download repositories from GitHub.

1. We use RepoReapers [42] to filter out low-quality and too small repositories on GitHub. We use quality characteristics provided by the RepoReapers to define a suitable criteria for repository selection. RepoReapers assesses repositories on eight characteristics and assigns a score typically between 0 and 1. We select all Java repositories in the RepoReapers dataset where architecture (as evidence of code organization), community and documentation (as evidence of collaboration), unit tests (as evidence of quality), history and issues (as evidence of accountability) scores are greater than zero. Further, we filter out repositories containing less than 1,000 lines of code (LOC) and having less than 10 stars.
2. We obtain 1,500 repositories after applying the above selection criteria.
3. We analyze all the selected repositories using the DesigniteJava tool that we developed to identify testability and test smells.

Table 3 presents the characteristics of the analyzed repositories. We attempted downloading and analyzing all the selected repositories; however, we could not download (either due to deleted or made private) and analyze (due to missing tests developed using JUnit framework) some of the repositories. Specifically, we did not find JUnit tests in 300 repositories. We successfully analyzed 1,115 repositories containing approximately 46 million LOC. Our replication package [57] includes the initial set of repositories, the names of all the successfully analyzed repositories along with the raw data generated by the employed tool, DesigniteJava.

6. Results

6.1 RQ1. To what extent do testability smells and test smells correlate?

6.1.1 Approach

The goal of this RQ is to explore the degree of correlation between test smells and testability smells in a repository. To achieve the above-stated goal, we first detect all the considered testability and test smells using DesigniteJava in the selected repositories. We calculate the sum of all testability smells and test smells per repository. Then, we compute smell density [58] to normalize the total number of smells to eliminate the potential confounding factor of project size. Testability smell density is defined as the total number of testability smells per one thousand lines of code (*i.e.,* (number of testability smells $\times$ 1,000)/total lines of code). Test smell density is defined as the total number of test smells in each test method (*i.e.,* number of test smells/total number of tests). We use the Spearman's correlation coefficient [61] to measure the degree of association between these two smell types.

Furthermore, we explore the relationship at the class-level. By the fine-grained analysis, we aim to see whether a class C that suffers from testability smells shows a high number of test smells in the test cases that primarily test the class C, and vice-versa. Testability smells occur in production (non-test) code and test smells arise in test code. Hence, we require a mechanism to map a production class with corresponding test classes that test the production class. We implemented the logic

of identifying the production class under test for each test case in DesigniteJava. For the analysis, we first find out all the method calls in each test case. Then, we identify the classes of the methods that are called from the test case. It is possible that a test case calls methods belonging to multiple classes; in that case, we attempt to identify the primary class that is being tested by the test case. To do so, we match the name of the test class and the names of candidate primary classes; typically, a test class is named by appending `Test` in the class name that the test class is testing. If the test class name does not follow the specified pattern and there are multiple candidate classes to be designated as the primary production class, then we pick the first candidate class whose method is called from the test case. Using the above information, we prepare a reverse index mapping to obtain a list of test cases corresponding to each production class. We use the mapping to retrieve the number of test smells corresponding to each production class. As explained above, we compute the testability smell density and test smell density at the class level. Finally, we compute the Spearman's correlation coefficient between testability smell density and test smell density.

6.1.2 Results

Figure 5 shows the scatter plot between testability smell density and test smell density in the software systems under examination. We obtain the Spearman's correlation coefficient $\rho = 0.246$ (p-value $< 2.2e-16$); the coefficient indicates that testability and test smells share a low positive correlation.

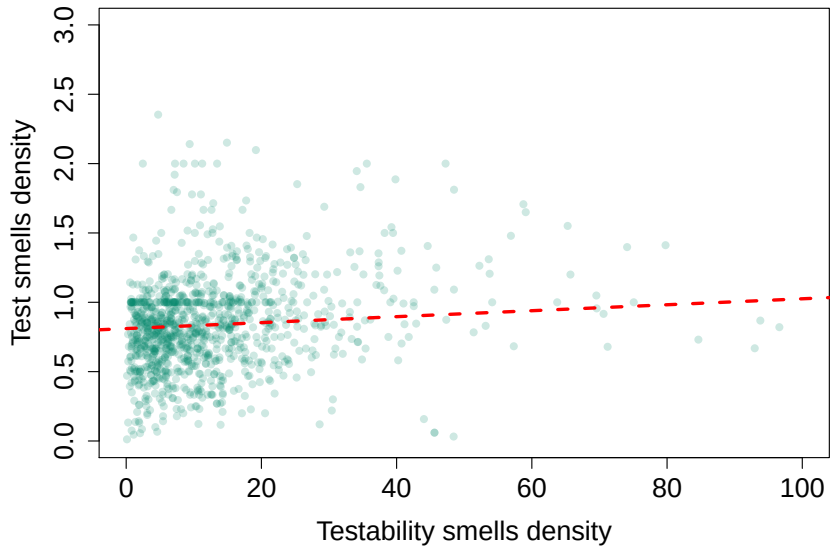

Figure 5. RQ1. Correlation between testability and test smell density

We extend our analysis by computing the correlation between the density of individual testability smells and the test smell density per repository. We observe that *law of Demeter violation* shows the highest correlation $\rho = 0.358$ (p-value $< 2.2e-16$) with test smells. On the other hand, the *global state* exhibits the lowest correlation $\rho = 0.076$ (p-value $< 2.2e-16$). *Hard-wired dependency* and *excessive dependency* show correlation $\rho = 0.328$ (p-value $< 2.2e-16$) and $\rho = 0.248$ (p-value $< 2.2e-16$), respectively.

We also investigate the relationship at the class-level. We identify the test cases and corresponding test smells for each production class and compute the Spearman's correlation between the normalized values of testability smells and test smells. We obtain $\rho = 0.050$ (p-value $= 2.903e-08$). The results indicate that testability smells and test smells do not share any correlation at the class-level

granularity.

> **Answer to RQ1.** Testability smells show a low positive correlation with test smells. A fine-grained analysis at the class-level reveals that testability smells and test smells do not correlate with each other.

6.2 RQ2. Do testability smells correlate with test suite size?

6.2.1 Approach

RQ2 investigates whether and to what extent the presence of testability smells leads to fewer test cases. To study this relationship, we first compute the testability smells in all the considered repositories using DesigniteJava. In addition to smells, we use the tool to figure out the total number of test cases in a repository; the tool marks each method as a test method or a normal non-test method. For simplicity, we treat each test method (*i.e.,* a method with a *@Test* annotation) as a test case. Next, we compute the *testability smell density* as described in RQ1 and the *test density* of each repository. Test density is a normalized metric that represents the total number of test cases per one thousand lines of code. We compute the Spearman's correlation coefficient between the testability smell density and the test density for each repository.

Similar to RQ1, we explore the correlation at the class-level. For the analysis, as we explain in RQ1, we first find out the production classes that a test case is testing. With this information, we prepare a mapping between production classes and their corresponding set of test cases. We use the mapping to retrieve the number of test smells corresponding to each production class. We compute testability smell density and test case density at the class level. We measure the correlation between testability smells and the number of test cases using Spearman's correlation coefficient.

6.2.2 Results

Figure 6 presents a scatter plot between the testability smell density and the test density of the selected repositories. We obtain $\rho = -0.033$ (p-value $= 0.308$), which is not statistically significant. Therefore, testability and test smells do not correlate with each other.

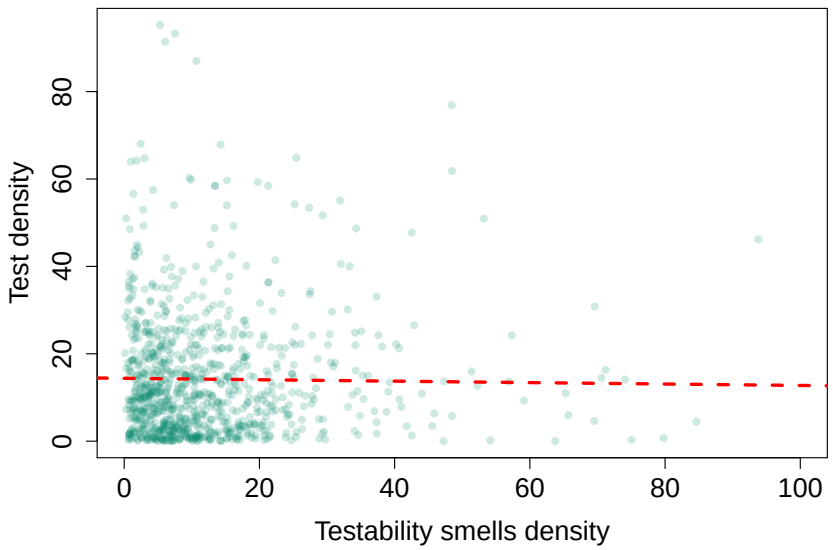

Figure 6. RQ2. Correlation of testability smells with test density

We extend our analysis by segregating the repositories into two categories by size. In the first set, we put all the repositories that have less than $50,000$ lines of code and, then, we put the rest of the repositories in the second set. We carry out the same analysis on both of these sets. We obtain $\rho = -0.009$ (p-value = 0.789) for the first set and $\rho = -0.050$ (p-value = 0.492) for the second set between testability smells and test density. The obtained results are not statistically significant.

Furthermore, we observe the relationship between testability smells and the number of tests at the class-level granularity. We compute the total number of testability smells for each non-test class and figure out the total number of tests written for the class. In the computation, we did not include the classes where the number of tests for the entire project is zero indicating that either the test cases are not written for the project or the test cases are written using a framework other than JUnit. We perform the above step to reduce the noise in the prepared data. We obtain $\rho = -0.179$ (p-value $< 2.2e-16$) as the correlation coefficient. The results clearly show that testability smells show a very low correlation with the size of the test suite.

> **Answer to RQ2.** Testability smells do not exhibit any correlation with the test density of a software system.

6.3 RQ3. Do testability smells cause more bugs?

6.3.1 Approach

RQ3 aims to investigate whether and to what extent testability smells relate to, and even cause, bugs in a given software system. To answer this question, we choose five subject systems manually and perform a trend analysis by extracting information from multiple commits for each of these subject systems.

We use the following protocol to identify the subject systems for this research question. First, we obtain a sorted list of repositories by their number of commits in descending order from our selected initial set of repositories (see Section 5) using the GitHub API. The intent here is to choose repositories with a rich commit history to facilitate detailed trend analysis. Then, we manually check these repositories one by one to assess whether a repository uses GitHub issues and whether these issues are labelled as "bugs". In addition, we execute DesigniteJava on the latest commit of each of these repositories to ensure that it does not take too long to run, as, otherwise, it might be prohibitive for us to run it to analyze the entire repository containing multiple (hundreds of) commits. Finally, we select the first five repositories that satisfy the above criteria, which are: `Magarena`,[8] `XP`,[9] `Rundeck`,[10] `MyRobotlab`,[11] and `Ontrack`.[12]

In order to perform a trend analysis, it is crucial to select a suitable set of commits from each of these repositories. One common way is to select commits at a fixed interval either by commit number (for example, every 100^{th} commit) or by commit date (for example, one commit per month). However, such a mechanism may result in a skewed set of commits where either significant changes in the commits are missed or commits with hardly any change are analyzed. To overcome this limitation, Sharma *et al.* [58] proposed a commit selection algorithm where commits are selected based on the amount of changes introduced in a commit *w.r.t.* the previous selected commit. In this work, we follow this strategy to select commits for each of the five identified repositories. Specifically, we first obtain all the commits in a repository in the main branch, and then we choose the first and the last commit to get started. Then, we compute five code quality metrics (*i.e.,* weighted methods per class (WMC), number of children (NC), lack of cohesion among methods (LCOM), fan-in, and fan-out) and identify changed classes based on the changes in any of these metrics. If the changed number of classes between two analyzed commits differ by a threshold (set

[8] https://github.com/magarena/magarena
[9] https://github.com/enonic/xp
[10] https://github.com/rundeck/rundeck
[11] https://github.com/MyRobotLab/myrobotlab
[12] https://github.com/nemerosa/ontrack

to 5%), we consider the commit having significant changes [58]. We then pick the middle commit (*i.e.,* the commit between the currently selected two commits) and repeat the process until we find commits with non-significant changes [58].

Once we identify the set of commits for the trend analysis, we detect testability smells in each of the selected commits for each repository by using our DesigniteJava tool. Also, we identify the total number of open and closed issues that has the tag "bugs" when the commit was made. The GitHub API does not provide a direct way to figure out issues at the time of a specific commit. To identify issues at the time of a specific commit, we first fetch the issues that are open (or closed) since the time of a commit and subtract them from the total open (or closed) issues at present. It gives us the total number of open (or closed) issues at the time of a specific commit. We record this information along with the total detected testability smells for each selected commit. Using this information, we compute the Spearman's correlation coefficient between the total detected testability smells and the total number of issues (*i.e.,* the sum of open and closed for each considered commit).

We also carry out a causal analysis to figure out whether testability smells *cause* bugs. We use Granger's causality [24] analysis for this purpose. The method has been used in similar studies [10, 49, 58] to explore the causal relationship within the software engineering domain. Equation 1 presents Granger's method mathematically.

$$a(t) = \sum_{j=1}^{k} f(s_{t-j}) + \sum_{j=1}^{k} f(b_{t-j}) \tag{1}$$

In our context, time series S and B represent the testability smell density *i.e.,* total number of testability smells per one thousand lines of code, and reported bugs computed over a period of time. Variables s_t and b_t represent testability smell instances and total reported bugs at time t. If the predictions of variable b with the past values of both s and b are better than the predictions using only the past values of b, then testability smells *cause* the bugs.

In such analysis, we must ensure the *stationary* property of a time-series before analyzing it and drawing conclusions based on that. A time-series is stationary if its statistical properties, such as mean, variance, and autocorrelation, are constant over time [11]. A non-stationary time-series shows seasonal effects, trends, and fluctuating statistical properties changing over time. Such effects are undesired for the causality analysis and thus a time-series must be made stationary before we perform the causality analysis. We carried out the augmented Dickey-Fuller unit root test [20] to check the stationary property of our time-series. Initially, our time-series was non-stationary. There are a few techniques to make a non-stationary time-series a stationary one [35]. We addressed this issue by applying a *difference* transformation, *i.e.,* subtracting the previous observation from the present observation for all columns. Techniques such as *differencing*, that we applied, help stabilize the mean of a time series by removing changes in the time series, and therefore eliminate or reduce the non-stationary nature of the series [35]. After the transformation, we obtain a stationary time-series that we confirmed by performing the augmented Dickey-Fuller unit root test again. Finally, we carry out the causality analysis using Equation 1.

6.3.2 Results

Table 4 presents the results of the experiment. The number of analyzed commits ranges between 38 (for `XP`) and 180 (for `Rundeck`). The size of the selected repositories varies between ≈ 71 KLOC (for `Magarena`) to ≈ 181 KLOC (for `XP`) as measured for the most recent analyzed commit. We compute the total number of testability smells in each selected commit as well as the total reported (open and closed) issues marked as bugs at the time of the corresponding commits for each of the selected repositories individually. The table shows the total number of testability smells detected in the most recent analyzed commit. We compute the Spearman correlation coefficient between the reported issues and testability smell density. We observe mixed results for the correlation analysis; two repositories show strong, one repository shows moderate, and one repository shows low

Table 4. RQ3. Correlation and causation relationships between testability smell density with the number of reported bugs

Repository	#Commits	LOC	Testability Smells	Correlation Coefficient (p-value)	Causality p-value
Magarena	66	71,567	1,425	$0.482(4.4e-4)$	0.119
MyRobotlab	76	118,532	2,643	$0.761(< 1.4e-15)$	0.661
Ontrack	107	17,009	72	$0.105(< 0.280)$	0.708
Rundeck	180	81,198	1,570	$0.193(< 0.009)$	0.825
XP	38	181,278	2,143	$0.937(< 2.2e-16)$	0.249

correlation. We observe that the correlation coefficient is not statistically significant for the *Ontrack* repository.

The last column of Table 4 presents the results of the causality test. Each cell in the column shows the p-value computed for the causal relationship of testability smells with the reported bugs. The results for all the analyzed repositories show that testability smells *do not* cause bugs as all the obtained p-values are greater than 0.05.

Answer to RQ3: The causality analysis reveals that **testability smells do not cause bugs**.

7. Implications and Discussion

The results of our first research question reveal that there is no correlation between testability smells and test smells. This suggests that writing good-quality test code is possible even with poor testability, at least for all testability smells considered herein. The results also indicate that either the difficulty in writing tests due to the considered testability smells is orthogonal to test smells, or existing testing frameworks, *e.g.,* mocking frameworks, make it easier to overcome the challenges posed by poor testability. Researchers may investigate further the influence of tools' features, such as mocking, to facilitate testing despite poor testability.

We explore the effect of testability smells on test suite size, represented by the number of test cases, in RQ2. Figure 7 shows box-plots of the categories of testability smell density with test density. We divide the repositories into four categories C1 to C4 based on the value of the testability smell density. For example, repositories with a testability smell density of less than five are put into category C1. We observe that the median test density for the first category is the highest among all the categories and the test density dips for the category C2. However, against the common belief of developers, test density rises again in categories C3 and C4. **The analysis further reaffirms that testability smells do not share a linear monotonic relationship with test density.**

Our experiment to investigate the correlation of testability smell density with the number of reported bugs does not show a consistent strong relationship. The strong correlation in two repositories and the moderate correlation in a repository show that the density of testability smell increases as the size of the software grows since the total number of reported bugs always increases with time. Hence, a strong correlation implies that the rate of testability smells increases as the software systems grow.

In the context of our study, one might wonder about the relationship of testability smells with traditional code smells. Given the definition and scope, it is likely that some code smells are also considered testability smells if they impact testability. However, this interpretation is not uniquely

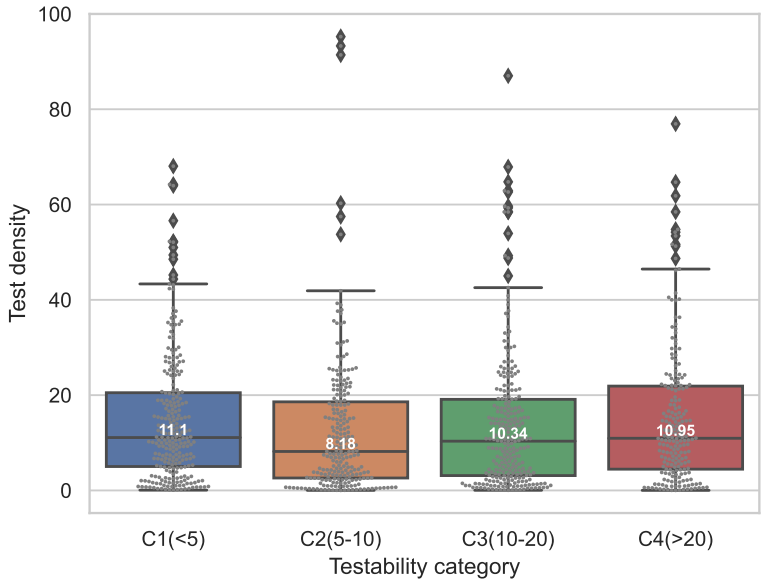

Figure 7. Box-plots of the categories of testability smell density with test density

applicable only in this context. For example, a violation of the 'single responsibility principle' may introduce incohesive class (or multifaceted abstraction) at design and 'feature concentration' smell at architecture granularity. Nevertheless, we perform an analysis of testability smell density with code smell density not only at the repository-level but also at a fine-grained granularity of class-level (where we compute the total number of smells for each class of the considered repositories). We use the DesigniteJava tool to detect code smells and testability smells. We compute the Spearman's correlation coefficient between the normalized total number of smells. At the repository-level, we obtain $\rho = 0.851$ (p-value $< 2.2e-16$) as the Spearman's correlation coefficient. Similarly, we get $\rho = 0.857$ (p-value $< 2.2e-16$) when we compute the correlation at the class-level. The strong correlation indicates that the presence of a large population of code smells is associated with the presence of a large number of testability smells and vice versa.

The elicited developers' perspective clearly emphasizes the importance of testability smells and the potential negative impact on testing aspects. However, the empirical evidence observed in the study does not agree with the perspective. We observed that testability smells, at the class-level fine-grained granularity, do not correlate with test smells. Also, the smells do not show any influence on test density. Furthermore, the results show that testability smells do not contribute to a higher number of bugs. The results suggest that despite developers' invaluable experience, their opinions and perspectives might need to be complemented with empirical evidence before bringing it into practice.

8. Threats to Validity

This section discusses the potential threats to the validity (construct, internal, and external) of our reported results.

Construct validity. Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our study, we used our DesigniteJava tool to identify the four testability smells. However, the strategies used to identify testability smells may not capture all testability cases. To mitigate this threat, we thoroughly tested the tool using different cases for each smell, and also fine-tuned the tool based on testing. Then, we performed a manual analysis of the four testability smells on a complete project, namely `wsc`. The results of the man-

ual validation show a very high recall and precision. Similarly, we also implemented support to detect test smells by following detection strategies proposed in the existing literature to identify test smells.

Internal validity. Internal threats to validity are concerned with the ability to draw conclusions from our experimental results. We carried out an online anonymous survey targeting developers by posting our survey on social media professional channels (Twitter, LinkedIn, and Reddit). Given the anonymity of the survey, we do not have any mechanism to verify the level of experience claimed by the participants. However, based on the quality of the responses provided by the participants, we believe that such a threat is mild. In addition, software developers participated in our online survey were not selected based on the repositories we analyzed. As a result, opinions of developers could be influenced by the repositories they usually contribute to and might not agree with our empirical results. To mitigate this, we did not target developers from specific repositories but rather expanded our participation range by posting invitations on online professional social media channels.

Agreement bias (or *acquiescence bias*) refers to the participants' tendency to agree with a statement rather than disagreeing with it [66]. We design our questionnaire in a neutral tone and provide options by using a Likert-scale to mitigate this threat. A similar threats to validity is participants' acquaintance to the authors. To avoid this threat, we did not circulate the survey in our internal organization groups. Also, we restricted the sharing to *professional* social media channels and hence did not sharing the survey on our, for example, Facebook profiles or groups.

RQ3 investigates causality between testability smells and the number of reported bugs; the analysis reveals that the testability smells do not cause bugs. There are two possible threats to the conclusion. First, it is possible that the testability smells other than those considered in the study have a larger impact on bugs. However, though there could be many other testability smells, the considered smells are representative as shown by our developers' survey. Second, the study only considered reported known bugs. It is possible that there are many more unknown bugs that may influence the results and conclusion of the experiment.

External validity External threats are concerned with the ability to generalize our results. The $1,115$ GitHub repositories analyzed in this paper were selected using well-defined criteria from the RepoReapers dataset. However, some repositories might have switched from *public* to *private* or no longer exist on GitHub, which might affect the criteria used to select repositories in this paper. In addition, all the selected repositories contain software written in Java, which might affect the generalizability of our findings. The major reason for focusing on Java is that the majority of research on software quality analysis has been done on Java code, and hence we can leverage existing tools to achieve the goals of our study. Along the same lines, the implemented test smell detection works only if the tests are written using JUnit. The rationale behind this decision is that JUnit is the most commonly used testing framework for Java. We encourage future research to expand the analyses conducted in this paper to software written in different programming languages.

9. Conclusions

This study explores practitioners' perspectives about testability smells as well as experimental evidence gathered via a large-scale empirical study on $1,115$ Java repositories containing approximately 46 million lines of code in order to better understand the relationship of testability smells with test quality, number of tests, and reported bugs.

Specifically, the study surveyed software developers to elicit their opinions and perspectives about testability smells. The survey showed that software developers consider testability a factor that impedes software testing; the survey also revealed their strong agreement with the proposed testability smells. Then, we conducted an extensive empirical evaluation to observe the relationship between testability smells and test-related aspects such as test smells and test suit size. Our results show that testability smells do not correlate with test smells at the class granularity and

with test suit size. Furthermore, we did not find evidence that testability smells cause bugs.

Our study has implications for both the research and industrial communities. Software developers often have strong opinions about software engineering concepts; however, experimental evidence may not support them in general. Specifically, this study shows that developers' opinions about testability do not concur with the experimental evidence. Hence, opinions and perspectives must be complemented with empirical evidence before bringing into practice. This also highlight the importance of data-driven software engineering, which advocates the need and value of adopting design and development decisions supported by data. Researchers can use our tool to detect testability smells to further evaluate and confirm our observations. Also, researchers may propose additional testability smells and investigate their collective impact on other relevant testing aspects, such as testing efforts.

Conflict of interest statement: The authors declare that they have no conflict of interest.

References

[1] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, and M. Harman. App store effects on software engineering practices. *IEEE Transactions on Software Engineering*, 47(2):300--319, 2021.

[2] M. Alenezi and M. Zarour. An empirical study of bad smells during software evolution using designite tool. *i-Manager's Journal on Software Engineering*, 12(4):12, 2018.

[3] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi. Test smell detection tools: A systematic mapping study. In *Evaluation and Assessment in Software Engineering*, EASE 2021, page 170–180, New York, NY, USA, 2021. Association for Computing Machinery.

[4] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56--65, 2012.

[5] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Softw. Engg.*, 20(4):1052–1094, aug 2015.

[6] K. J. Berry and J. Paul W. Mielke. A Generalization of Cohen's Kappa Agreement Measure to Interval Measurement and Multiple Raters. *Educational and Psychological Measurement*, 48(4):921--933, 1988. _eprint: https://doi.org/10.1177/0013164488484007.

[7] R. V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, sep 1994.

[8] M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219--1232, Sept. 2006.

[9] V. Chowdhary. Practicing testability in the real world. In *2009 International Conference on Software Testing Verification and Validation*, pages 260--268, 2009.

[10] C. Couto, P. Pires, M. T. Valente, R. da Silva Bigonha, A. C. Hora, and N. Anquetil. Bugmaps-granger: A tool for causality analysis between source code metrics and bugs. 2013.

[11] D. Cox and H. Miller. *The Theory of Stochastic Process*. Chapman and Hall, London, 1 edition, 1965.

[12] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001*, pages 92--95, 2001.

[13] P. Devanbu, T. Zimmermann, and C. Bird. Belief & evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 108–119, New York, NY, USA, 2016. Association for Computing Machinery.

[14] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 830–840, New York, NY, USA, 2019. Association for Computing Machinery.

[15] A. Eposhi, W. Oizumi, A. Garcia, L. Sousa, R. Oliveira, and A. Oliveira. Removal of design problems through refactorings: Are we looking at the right symptoms? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 148--153, 2019.

[16] S. Fatima, T. A. Ghaleb, and L. Briand. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*, 2022.

[17] M. Feathers. *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.

[18] F. G. S. Filho, V. Lelli, I. d. S. Santos, and R. M. C. Andrade. Correlations among software testability metrics. In *19th Brazilian Symposium on Software Quality*, SBQS'20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] R. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553--564, 1991.

[20] W. A. Fuller. *Introduction to Statistical Time Series*. John Wiley and Sons New York, 1 edition, 1976.

[21] V. Garousi, M. Felderer, and F. N. Kılıçaslan. A survey on software testability. *Information and Software Technology*, 108:35--64, 2019.

[22] V. Garousi, M. Felderer, and M. V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, 2018.

[23] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52--81, 2018.

[24] C. W. J. Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424--438, 1969.

[25] M. M. Hassan, W. Afzal, M. Blom, B. Lindström, S. F. Andler, and S. Eldh. Testability and software robustness: A systematic literature review. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 341--348, 2015.

[26] M. Hevery. Writing Testable Code, Aug. 2008. https://testing.googleblog.com/2008/08/by-miko-hevery-so-you-decided-to.html.

[27] M. Human. Why You Should Be Replacing Full Stack Tests with Ember Tests, Dec. 2022. https://www.mutuallyhuman.com/blog/why-you-should-be-replacing-full-stack-tests-with-ember-tests/.

[28] A. A. Janes and G. Succi. The dark side of agile software development. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, page 215–228, New York, NY, USA, 2012. Association for Computing Machinery.

[29] V. Jeffrey M. Factors that affect software testability. Technical report, 1991.

[30] N. S. Junior, L. Rocha, L. A. Martins, and I. Machado. A survey on test practitioners' awareness of test smells, 2020.

[31] T. Kaczanowski. *Practical Unit Testing with JUnit and Mockito*. Tomasz Kaczanowski, 2013.

[32] R. A. Khan and K. Mustafa. Metric based testability model for object oriented design (mtmood). *SIGSOFT Softw. Eng. Notes*, 34(2):1–6, feb 2009.

[33] D. J. Kim, T.-H. P. Chen, and J. Yang. The secret life of test smells - an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5):100, July 2021.

[34] R. Kolb and D. Muthig. Making testing product lines more efficient by improving the testability of product line architectures. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, pages 22–--27, New York, NY, USA, 2006. Association for Computing Machinery.

[35] D. Kwiatkowski, P. C. Phillips, P. Schmidt, and Y. Shin. Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root? *Journal of Econometrics*, 54(1):159 -- 178, 1992.

[36] Y. Le Traon and C. Robach. From hardware to software testability. In *Proceedings of 1995 IEEE International Test Conference (ITC)*, pages 710--719, 1995.

[37] Y. Le Traon and C. Robach. Testability measurements for data flow designs. In *Proceedings Fourth International Software Metrics Symposium*, pages 91--98, 1997.

[38] K. J. Lienberherr. Formulations and benefits of the law of demeter. *SIGPLAN Not.*, 24(3):67–78, mar 1989.

[39] B. Lo and H. Shi. A preliminary testability model for object-oriented software. In *Proceedings. 1998 International Conference Software Engineering: Education and Practice (Cat. No.98EX220)*, pages 330--337, 1998.

[40] L. Marshall and J. Webber. Gotos considered harmful and other programmers' taboos. *Department of Computing Science Technical Report Series*, 2000.

[41] S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Information and Software Technology*, 47(15):979--997, 2005. Most Cited Journal Articles in Software Engineering - 1999.

[42] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, 22(6):3219--3253, Dec. 2017.

[43] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5--18, 2012.

[44] T. Nguyen, M. Delaunay, and C. Robach. Testability analysis for software components. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 422--429, 2002.

[45] T. B. Nguyen, M. Delaunay, and C. Robach. Testability Analysis of Data-Flow Software. *Electronic Notes in Theoretical Computer Science*, 116:213--225, Jan. 2005.

[46] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 346--357. IEEE, 2019.

[47] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik. Rttool: A tool for extracting relative thresholds for source code metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 629--632, 2014.

[48] P. Oliveira, M. T. Valente, and F. P. Lima. Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 254--263, 2014.

[49] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99:1 -- 10, 2018.

[50] J. E. Payne, R. T. Alexander, and C. D. Hutchinson. Design-for-testability for object-oriented software. *Object Magazine*, 7(5):34--43, 1997.

[51] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1650–1654, New York, NY, USA, 2020. Association for Computing Machinery.

[52] B. Pettichord. Design for testability. In *Pacific Northwest Software Quality Conference*, pages 1--28, 2002.

[53] D. Pina, C. Seaman, and A. Goldman. Technical debt prioritization: A developer's perspective. In *Proceedings of the International Conference on Technical Debt*, TechDebt '22, page 46–55, New York, NY, USA, 2022. Association for Computing Machinery.

[54] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 72--81, 2010.

[55] D. M. Ribeiro, F. Q. B. da Silva, D. Valença, E. L. S. X. Freitas, and C. França. Advantages and disadvantages of using shared code from the developers perspective: A qualitative study. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery.

[56] T. Sharma. DesigniteJava, Dec. 2018. https://github.com/tushartushar/DesigniteJava.

[57] T. Sharma, S. Georgiou, M. Kechagia, T. A. Ghaleb, and F. Sarro. Replication Package for Testability Study, Nov. 2022. https://github.com/SMART-Dal/testability.

[58] T. Sharma, P. Singh, and D. Spinellis. An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 25(5):4020--4068, 2020.

[59] P. K. Singh, O. P. Sangwan, A. P. Singh, and A. Pratap. An assessment of software testability using fuzzy logic technique for aspect-oriented software. *International Journal of Information Technology and Computer Science (IJITCS)*, 7(3):18, 2015.

[60] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1--12, 2018.

[61] C. Spearman. The proof and measurement of association between two things. 1961.

[62] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 1 edition, 2014.

[63] R. E. Sward and A. Chamillard. Re-engineering global variables in ada. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*, pages 29--34, 2004.

[64] V. Terragni, P. Salza, and M. Pezzè. Measuring software testability modulo test quality. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 241–251, 2020.

[65] D. Thomas and A. Hunt. *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional, 2019.

[66] B. Toner. The impact of agreement bias on the ranking of questionnaire response. *The Journal of Social Psychology*, 127(2):221--222, 1987.

[67] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 4–--15, New York, NY, USA, 2016. Association for Computing Machinery.

[68] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenilio, R. Lima, A. Garcia, and C. Bezerra. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511--522, 2020.

[69] J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of Built-In-Test for Run-Time-Testability in Component-Based Software Systems. *Software Quality Journal*, 10(2):115--133, Sept. 2002.

[70] T. Virgínio, L. Martins, L. Rocha, R. Santana, A. Cruz, H. Costa, and I. Machado. Jnose: Java test smell detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 564–569, New York, NY, USA, 2020. Association for Computing Machinery.

[71] J. M. Voas. *Object-Oriented Software Testability*, pages 279--290. Springer US, Boston, MA, 1996.

[72] H. Vranken, M. Witteman, and R. Van Wuijtswinkel. Design for testability in hardware software systems. *IEEE Design Test of Computers*, 13(3):79--86, 1996.

[73] L. Zhao. A new approach for software testability analysis. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 985–988, 2006.

[74] Y. Zhou, H. Leung, Q. Song, J. Zhao, H. Lu, L. Chen, and B. Xu. An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. *Science china information sciences*, 55(12):2800--2815, 2012.

[75] G. Zilberfeld. Design for Testability – The True Story , Jan. 2012. https://www.infoq.com/articles/Testability/.

[76] Ömer Faruk Arar and K. Ayan. Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies. *Expert Systems with Applications*, 61:106--121, 2016.