Test Smell Detection Tools: A Systematic Mapping Study

Wajdi Aljedaani
wajdialjedaani@my.unt.edu
University of North Texas
Denton, Texas, USA

Anthony Peruma
axp6201@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Ahmed Aljohani
ama1177@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Mazen Alotaibi
mfa2886@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Mohamed Wiem Mkaouer
mwmvse@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Ali Ouni
ali.ouni@etsmtl.ca
ETS Montreal, University of Quebec
Montreal, Quebec, Canada

Christian D. Newman
cnewman@se.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Abdullatif Ghallab
Abdullatif.Ghallab@unt.edu
University of North Texas
Denton, Texas, USA

Stephanie Ludi
Stephanie.Ludi@unt.edu
University of North Texas
Denton, Texas, USA

ABSTRACT

Test smells are defined as sub-optimal design choices developers make when implementing test cases. Hence, similar to code smells, the research community has produced numerous test smell detection tools to investigate the impact of test smells on the quality and maintenance of test suites. However, little is known about the characteristics, type of smells, target language, and availability of these published tools. In this paper, we provide a detailed catalog of all known, peer-reviewed, test smell detection tools.

We start with performing a comprehensive search of peer-reviewed scientific publications to construct a catalog of 22 tools. Then, we perform a comparative analysis to identify the smell types detected by each tool and other salient features that include programming language, testing framework support, detection strategy, and adoption, among others. From our findings, we discover tools that detect test smells in Java, Scala, Smalltalk, and C++ test suites, with Java support favored by most tools. These tools are available as command-line and IDE plugins, among others. Our analysis also shows that most tools overlap in detecting specific smell types, such as General Fixture. Further, we encounter four types of techniques these tools utilize to detect smells. We envision our study as a one-stop source for researchers and practitioners in determining the tool appropriate for their needs. Our findings also empower the community with information to guide future tool development.

ACM Reference Format:
Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3463274.3463335

EASE 2021, June 21–23, 2021, Trondheim, Norway

ACM ISBN 978-1-4503-9053-8/20/06…$15.00
https://doi.org/10.1145/3463274.3463335

1 INTRODUCTION

Software testing is an essential part of the software development life cycle. As part of the software development process, developers create and update their system's test suite to ensure that the system under test adheres to the requirements and provides the expected output. However, test code, similar to production code, is subject to bad programming practices (i.e., smells), which hamper the quality and maintainability of the test suite [48]. Formally defined in 2001 [67], the catalog of test smells has been steadily growing throughout the years. While most test smells focus on traditional Java systems, researchers have also studied the impact of these smells on other programming languages, and platforms [27, 55, 59]. With the growth of the test smell catalog, the research community, in turn, has utilized these smells to study the impact test smells have on the maintainability of test suites. These studies show that test smells negatively impact the comprehension of a test suite and increase change- and defect-proneness of the test suite, thereby increasing its flakiness [63]. In addition to defining test smells, researchers have also provided the community with various tools to detect such test smells. Furthermore, research has shown that early detection of bad smells reduces maintenance costs, highlighting the importance of such detection tools.

With the growth of test smells studies, recent literature reviews [35, 36] have been proposed to study various dimensions related to these anti-patterns. These literature reviews have explored the various definitions of test smells, empirical analysis of their survival, spread, refactoring, and their relationship with change and bug proneness of source code. However, little is known about the toolsets used to detect test smells. The availability of tools is vital for software engineering researchers and practitioners. In research, tools facilitate the reproducibility of studies while developers benefit from improved productivity through tool adoption. Without a thorough understanding of available tools and how these tools compare to one another, it will be difficult to conduct future research that uses the right toolset for a given research problem. Therefore, our work complements these reviews by not only extracting all the

test smell detection tools published in peer-reviewed venues, but also providing more in-depth details about them. To facilitate their adoption, we compare and contrast multiple attributes of these tools, such as supported smell types, target environment, detection mechanisms, etc. Hence, our work provides a catalog for developers and researchers to support the adoption of these tools.

1.1 Goal & Research Questions

The goal of this study is to provide developers and researchers with a one-stop source that offers a comprehensive insight into test smell detection tools. The information in this study will *allow researchers to select the right tool for their research task and provide data-driven advice on how test smell tools can be advanced through future work.* Through this study, the community will be better equipped to determine the correct tool they need to utilize to satisfy their requirement, along with the shortcomings of these tools. This work also provides the research community with insight into areas that require improved automation. Hence, we aim at addressing the following research questions (RQs):

RQ_1: What test smell detection tools are available to the community, and what are the common smell types they support? This RQ investigates the volume of test smell detection tools released by the research community. We answer this RQ by performing an extensive and comprehensive search on six popular digital libraries among the software engineering community. We investigate the frequency of tool release, and the spectrum of test smells detected by the tools.

RQ_2: What are the main characteristics of test smell detection tools? In this RQ, we examine the design-level features that are common to test smell detection tools, such as platform support and smell detection mechanisms. This RQ provides us with details into how the research community constructs such tools and provides insight into the development of future tools.

1.2 Contributions

Through this study, we provide the community of researchers and practitioners with a view and insights on the history of the availability of test smell detection tools. More specifically, our contributions are outlined below:

- A catalog of 22, peer-reviewed, test smell detection tool publications, and publications that utilize these tools. These publications provide an initial platform for future research in this area.
- A series of experiments highlighting the growth of such tools, along with a comparison of key tool attributes.
- A discussion of how our findings provide insight into future research areas in this field along with details that need to be considered when selecting a test smell detection tool.
- A replication package of our survey for extension purposes [1].

2 RESEARCH METHODOLOGY

Being a Systematic Mapping Study (SMS), our research explores published scientific literature to gather information about a specific topic in software engineering, and to provide a high-level understanding and/or answering exploratory research questions [21]. To this extent, our SMS aims at proving a high-level understanding of the existence of test smell detection tools, their characteristics, and their adoption in academic studies. Prior studies in cataloging detection-based tools, were associated with technical debt, bad smells, bug localization, and architectural smells. Further, while Garousi and Küçük [36] provide a list of test smell detection tools as part of their SMS on test smells, our study aims to expand on this listing. Hence, in this section, we describe the procedure we adopt to search and select the relevant publications for analysis. In brief, our methodology consists of three phases– (1) planning, (2) execution, and (3) synthesis. In the following subsections, we elaborate on these phases.

Table 1: The digital libraries queried in our study.

Digital Library	URL
ACM Digital Library	https://dl.acm.org/
IEEE Xplore	https://ieeexplore.ieee.org/
Science Direct	https://www.sciencedirect.com/
Scopus	https://www.scopus.com/
Springer Link	https://link.springer.com/
Web of Science	https://webofknowledge.com/

2.1 Planning

In this phase, we detail our publications search strategy. In conformance with systematic mapping studies, we utilize a specific set of domain-specific (i.e., test smell related) keywords to search, in popular digital libraries, for publications that meet our requirements.

Digital Libraries. To locate publications for our study, we search six digital libraries. These libraries, listed in Table 1, either contain or index publications from computer science and software engineering venues and are utilized by similar studies (e.g., [32]).

Inclusion/Exclusion Criteria. Inclusion and exclusion criteria are crucial in pruning our search space, reducing bias, and retrieving relevant peer-reviewed scientific publications. Selected publications of these criteria become our starting point for manual filtering, to see whether they fit in our study, i.e., propose or adopt a test smells tool. The initial pool of publications also serves for: (1) backward snowballing, i.e., analyzing publications cited by the selected pool; and (2) forward snowballing, i.e., analyzing publications citing our pool publications. Table 2 lists the inclusion and exclusion criteria considered in this study. With regards to the time range, we did not set a starting date. However, our end date was set to the end of December 2020. Hence, the date range criterion allows the selection of any tool as long as it appeared before December 31, 2020.

Table 2: Our inclusion and exclusion search criteria.

Inclusion	Exclusion
Published in Computer Science	Websites, leaflets, and grey literature
Written in English	Published in 2021
Available in digital format	Full-text not available online
Propose or use test smell detect tool	Tools not associated with peer-reviewed papers

Search Keywords. To determine the optimal set of search keywords, we conducted a pilot search on two well-known digital libraries, i.e., IEEE and ACM. This process intends to identify relevant words or synonyms utilized in test smell publications. We performed multiple instances of the pilot search, where each instance involved refinement of the keyword terms in the search query. We conduct our query only on the title and abstract of the publication. We decided to apply the search on a publication's metadata instead of on the full-text to avoid false positives. The finalized search string is presented below.

```
Title:("tool*" OR "detect*" OR "test smell" OR
"test smells") AND Abstract:("test smell" OR "test
smells" OR "test code" OR "unit test smell")
```

2.2 Execution

In this phase, we detail how we process and filter the publications we obtain from our digital library search. Our initial search of the six digital libraries results in 436 publications, with ScienceDirect resulting in the highest number of publications (126). Next, we employ a four-stage quality control process to filter out publications that were not part of our inclusion criteria. Figure 1 depicts the volume of publications filtered at each stage. This quality control process involves three authors manually reviewing the publications to determine if a publication can pass from one stage to another. The first stage starts with removing duplicate and retracted publications; 74 publications were removed, and 362 candidate publications made it to the next stage. In the second stage, we apply our inclusion and exclusion criteria to the title and abstract. For instance, we discard all publications that were not peer-reviewed or did not propose or adopt a tool. This thorough procedure resulted in only including 54 publications. The third stage involves a full-text analysis of each selected publication. Using our inclusion and exclusion criteria, we retain only 30 publications. In the last stage, we perform the forward and backward snowball sampling, resulting in 17 publications. Therefore, we ended up with a final set of 47 publications.

2.3 Synthesis

This phase synthesizes the extracted data to answer our RQs. First, we classify the primary set of publications into one of two types, i.e., tool development or tool adoption. Tool development publications are studies that propose a test smell detection tool, either as part of proposing a new catalog of smells or detecting existing smell types. Tool adoption publications are studies that utilize an existing test smell detection tool as part of their study design. Additionally, we also classify studies by publication year and venue; this helps with partly answering RQ_1. For the remainder of RQ_1 and RQ_2, we manually review the full-text of each tool development publication. As part of this review, we evaluate each study based on concrete evidence present in the publication (and its supporting artifacts, if any) without any vague assertions. For each tool, we extract the types of test smells the tool detects and other tool features, such as supported programming language, testing framework, correctness, etc. We elaborate further on the tool features in Section 3.2 when presenting our findings for RQ_2.

Finally, all RQ-related data collected during the publication review task was peer-reviewed to ensure bias mitigation, with conflicts resolved through discussions. We utilized a spreadsheet to hold the manually extracted data to facilitate collaboration during the author-review process. The authors, participating in the filtering stages and manual review, are experienced with this research. They have published work in this area, including defining test smell types, tool development, and adoption [55, 56].

3 RESEARCH FINDINGS

In this section, we present the findings for our proposed RQs based on the synthesis of the finalized set of 47 test smell tool detection-related publications, which are composed of **22 publications that propose new tools and 25 publications that adopt these tools.**

3.1 RQ_1: What test smell detection tools are available to the community, and what are the common smell types they support?

This RQ comprises of four parts. In the first part, we provide a breakdown of the publications by publication date and venue. In the second part, we present the tools identified in our systematic search, while in the third part, we provide insight into the types of test smells detected by the identified tools. Finally, the fourth part looks at the programming languages supported by the test smells.

3.1.1 ***Publication Years & Venues.***

Figure 2 depicts the yearly breakdown of tool publications. The first test smell tool, TRex [20], appeared in 2006. Since then, there was a steady trend of one or two tools appearing every one or two years, until 2018. The years 2019 and 2020 witnessed a notable increase in tool-based publications compared to the prior years, with approximately 51% of tool development and adoption publications occurring in these two years. There can be many factors influencing this recent hype. We have observed the following: The dynamic nature of detection mechanisms of traditional state-of-the-art tools made them require compilable projects with constraints over how test files should be written and located. Therefore, traditional tools are implemented to run as standalone applications or plugins in Integrated Development Environments (IDEs). Besides being constrained to their environments, they are not intended to run on large-scale software systems. However, the tools that appeared in recent years were developed as APIs, facilitating their deployment to mine software repositories. While their detection strategies carry the false-positiveness of static analysis, they allowed the analysis of a wide variety of software systems. Therefore, the number of empirical publications, adopting these tools, has significantly increased, reaching up to 16 in two years, higher than the number of all previous tool adoption publications combined. These studies have explored various characteristics of test smells, including co-occurrence, survivability, severity, refactoring, impact on flaky tests, proneness to changes and bugs, etc. Next, in Figure 3, we provide a pictorial representation, in the form of a timeline, depicting the release of the 22 test smell detection tools. Analyzing the smell types detected by each tool, we specify, in green, the total number of smell types detected by each tool. Additionally, we also indicate, in red, the number of net new test smell types and the number of existing smell types in blue. Reading Figure 3 from left to right (i.e., the oldest tool to newest), a smell type first introduced by a tool is in blue text, while its subsequent appearance in another tool is in red. For example, the General Fixture smell first appears in TestQ (hence it is shown in blue text), this smell next appears in the unnamed tool (hence it is shown in red text), and so on.

In terms of venues, looking at the complete set of primary publications, 40 publications are associated with a conference/workshop/symposium, while seven publications appear in journals. Looking at just tool development publications, 20 of these publications

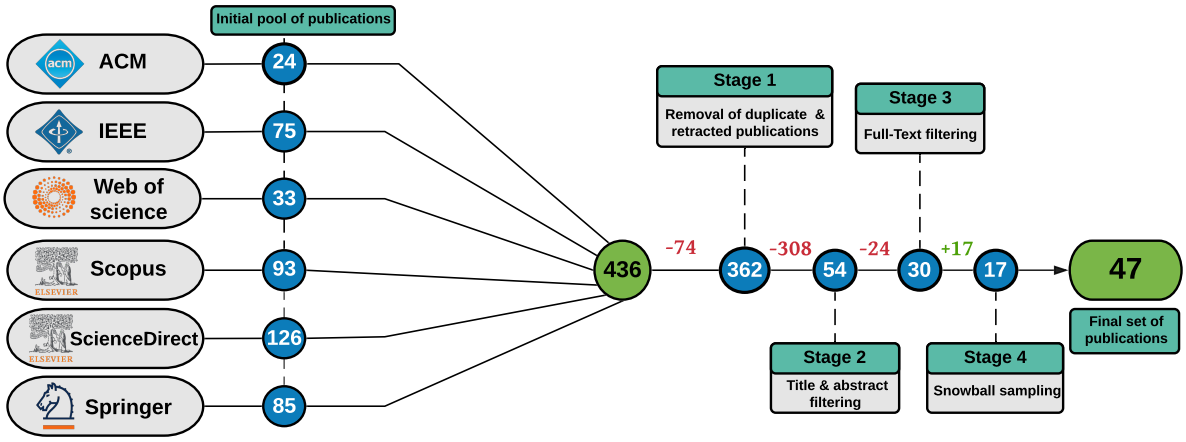

Figure 1: Overview of the volume of publications resulting from our filtering process.

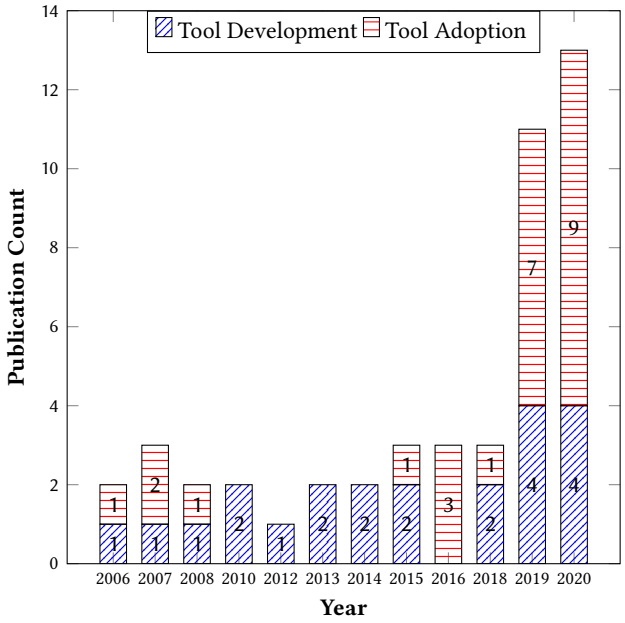

Figure 2: Yearly breakdown of tool publications.

are associated with a conference/workshop/symposium. Finally, the most popular venue for a tool development publication is the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, with four publications. The complete breakdown is available in our replication package.

3.1.2 ***Test Smell Detection Tools.*** In this part of the RQ, based on available documentation (i.e., full-text of the publication and any of its supporting artifacts), we provide an overview of each tool, from the oldest to the most recent.

Released in 2006 is **TRex** by Baker et al. [20]. This tool analyzes for TTCN-3 test suites for issues specific to this testing framework. The tool also provides developers with the ability to correct identified issues. **TestLint**, released by Reichhart et al. [59] in 2007, is a rules-based tool that detects 27 quality violations in the unit test code in Smalltalk systems. In 2008, Breugelmans and Van Rompaey released **TestQ** [27]. The tool provides a visual interface for developers to explore test suites and detects 12 test smells in C++ test suites. The tool facilitates customizations such as smell prioritization.

In 2009 Koochakzadeh and Garousi released **TeReDetect** [45] (Test Redundancy Detection), a test redundancy detection tool for JUnit tests that work in conjunction with a code coverage tool. The authors then released **TeCReVis** (Test Coverage and Test Redundancy Visualization) in 2010 [44], an Eclipse plugin that provides developers with a visualization of a project's test coverage and test redundancy. **Bavota et al.** [22] released an unnamed test smell detection tool in 2012. The tool detects nine test smell types in Java test suites. The tool prioritizes recall over precision resulting in a long list of potential issues and thereby require manual reviews.

2013 saw the release of two detection tools. Greiler et al. introduce **TestHound** [39]. This static analysis tool focuses on test smells related to test fixtures in Java test suites and recommends refactorings to address the detected issues. In a user study, the authors show that developers are appreciative of the tool with regards to understanding test fixture code. Greiler et al. improve on their prior tool by releasing **TestEvoHound** [40]. This improved tool analyzes Git or SVN repositories to analyze the evolution of a system's test fixture code. As part of the analysis process, the tool does a checkout and build of each revision of the project and then passes the revision to TestHound to detect test fixture smells.

Zhang et al. released **DTDetector**, a JUnit supported test dependency detection tool in 2014 [75]. Also released in 2014 by Huo et al. [42], is **OraclePolish**, which utilizes a dynamic tainting-based technique for the detection of two test smell types in JUnit test suites. The tool's empirical evaluation demonstrates that it can detect both brittle assertions and unused inputs in real tests at a reasonable cost. In 2015, Bell et al. released **ElectricTest** [24], another dependency detection tool for JUnit test suites. The authors demonstrate that their tool outperforms DTDetector in test parallelization. Also released in 2015 was **PolDet** by Gyori et al. [41], a test pollution smell detection tool. The tool analyses heap-graphs and file-system states during test execution for instances of state pollution (e.g., tests reading/writing shared resources).

Palomba et al. [52] released **Taste** (Textual AnalySis for Test smEll detection) in 2018. This tool utilizes information retrieval

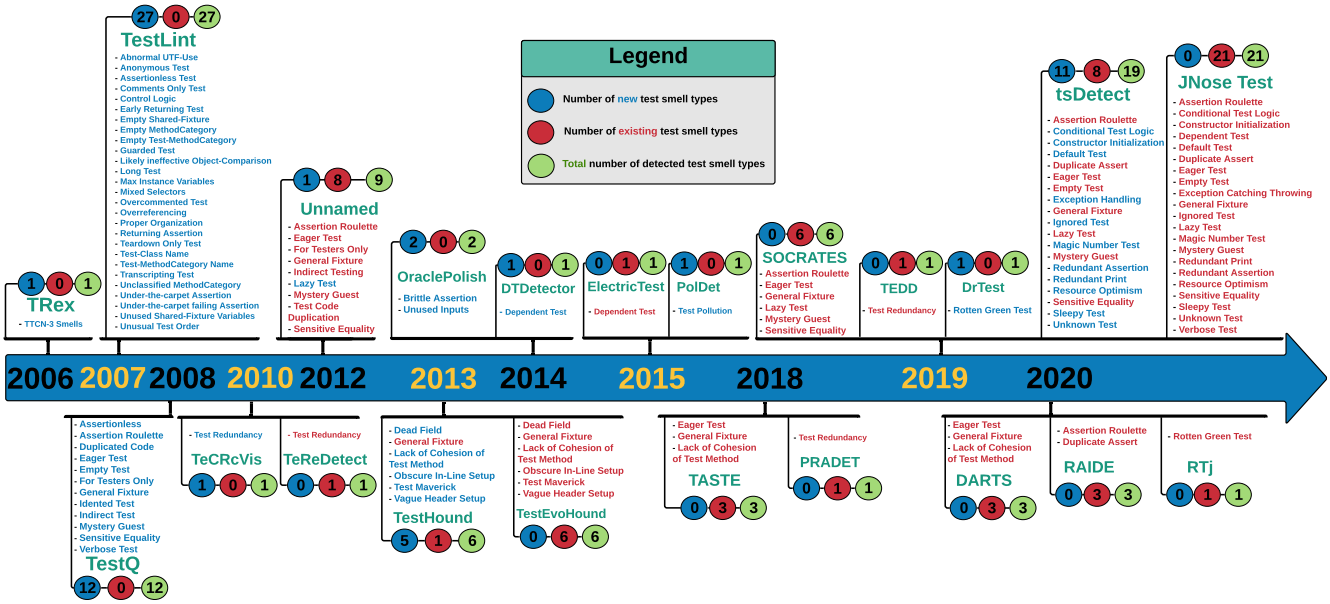

Figure 3: Timeline of the release of test smell detection tools by the research community.

techniques to detect three test smell types in Java test suites. Results from an empirical study show that the tool is 44% more effective in detecting test smells when compared to structural-based detection tools. Also released in 2018 is **PraDeT**, by Gambi et al. [34]. This tool detects manifest test dependencies and can analyze large projects containing a vast quality of tests.

There were four test smell detection tools released in 2019. **tsDetect**, released by Peruma et al. [56], detects a total of 19 test smell types. The smell types comprise of 11 newly introduced types and 8 existing types. The tool utilizes an abstract syntax tree to analyzes JUnit test suites and reports an average F-score of 96.5% for each smell type. Further, one smell type (i.e., *Default Test*) is exclusive to Android applications, while the remaining types apply to all Java systems. **SoCRATES** (SCala RAdar for TEst Smells) by De Bleser et al. [30] detects the presence of six smell types in Scala systems using static analysis. Virginio et al. released **JNose Test**, a tool with the ability to detect 21 test smell types in Java systems. Additionally, the tool also provides ten metrics around code coverage. Biagiola et al. released **TEDD** (Test Dependency Detector), a tool to detect test dependencies in end-to-end web test suites [25]. The tool presents a list of manifest dependencies as output from its execution. Delplanque et al. [31] released **DrTest**, a tool that detects *Rotten Green Test* smell in the Pharo ecosystem.

2020 saw the release of two IDE plugins and one command-line tool. Lambiase et al. [46] released **DARTS** (Detection And Refactoring of Test Smells), a plugin that utilizes information retrieval to detect three smell types. The tool also offers refactoring support. **RAIDE**, an Eclipse plugin was released by Santana et al. [60]. This plugin detects and provides semi-automated refactoring support for two test smell types in JUnit test suites. Martinez et al. [47] released **RTj**, a command-line tool that supports the detection and refactoring of *Rotten Green Test* smells.

Finally, when compared against the catalog of Garousi and Küçük [36], our dataset contains ten of the 12 listed tools. The tools excluded from our study are not proposed in peer-reviewed literature. The common set of tools are indicated in RQ2.

3.1.3 *Detected Test Smell Types.*

Next, we examine the types of test smells detected by the identified tools in our set. For completeness, we provide, in Table 3, brief definitions for each unique smell type detected by the identified tools. We also provide their references for more details. When analyzing the definitions of these smell types, we observe that there are smells that are associated with more than one name, but with a similar description of its symptoms. For example, *Assertionless*, *Assertionless Test*, and *Unkown Test* define the absence of an expected assert in the test method. Similarly, *Duplicated Code* and *Test Code Duplication* define the same issue of the existence of code clones.

Looking at our list of tools, TestLint detects the highest number of smell types (26). JNose Test is the second highest with 21 detected smell types, followed by tsDetect (19 smell types). Furthermore, from Figure 3, it is common to see various tools detecting the same smell types. Per analogy to code smells, while there is an agreement on the meanings of smells, there is no consensus on identifying them. Therefore, it is evident to see various tools containing different detection strategies for similar smell types. Hence, in this analysis, we look at the overlap of detected smell types by the tools in our dataset. The smells, detected by the tools *TestLint*, *OraclePolish*, *TRex*, and *PolDet* are unique to the respective tool. The remaining 16 tools share the detection of some overlapping smells.

In Table 4, we identify the overlapping of smells across 16 tools. For each tool, we indicate if the tool detects a specific smell by the √ symbol. From this table, we observe that the three most common smell types are *General Fixture*, *Eager Test*, and *Assertion Roulette*, which are respectively detected by 9, 7, and 6 tools.

Table 3: Definition of the test smells detected by the tools in our dataset.

No.	Test Smell Name	Abbreviation	Definition	Ref.
01	**Abnormal UTF-Use**	**AUU**	Overriding the default behavior of the testing framework by test-suite.	[59]
02	**Anonymous Test**	**AT**	A test method with a meaningless and unclear method name.	[59]
03	**Assertion Roulette**	**AR**	A test method with multiple assertions without explanation messages.	[22]
04	**Assertionless**	**AL**	A test that is acting to assert data and functionality but does not.	[27]
05	**Assertionless Test**	**ALT**	A test that does not contain at least one valid assertion.	[59]
06	**Brittle Assertion**	**BA**	A test method that has assertions that check data input.	[42]
07	**Comments Only Test**	**COT**	A test that has been put into comments.	[59]
08	**Conditional Test Logic**	**CTL**	A test method that contains a conditional statement as a prerequisite to executing the test statement.	[56]
09	**Constructor Initialization**	**CI**	A test class that contains a constructor.	[56]
10	**Control Logic**	**ConL**	A test method that controls test data flow by methods such as debug or halt.	[59]
11	**Dead Field**	**DF**	When a class has a field that is never used by any test methods.	[39]
12	**Default Test**	**DT**	Default or an example test suite created by Android Studio.	[56]
13	**Dependent Test**	**DepT**	A test that only executes on the successful execution of other tests.	[72]
14	**Duplicate Assert**	**DA**	Occurs when a test method has the exact assertion multiple times within the same test method.	[56]
15	**Duplicated Code**	**DC**	A test method that has redundancy in the code.	[27]
16	**Eager Test**	**ET**	A test method that calls several methods of the object to be tested.	[22]
17	**Early Returning Test**	**ERT**	A test method that returns a value too early which may drop assertions.	[59]
18	**Empty Method Category**	**EMC**	A test method with an empty method category.	[59]
19	**Empty Shared-Fixture**	**ESF**	A test that defines a fixture with an empty body.	[59]
20	**Empty Test**	**EmT**	A test method that is empty or does not have executable statements.	[56]
21	**Empty Test-Method Category**	**ETMC**	A test method with an empty test method category.	[59]
22	**Exception Handling**	**EH**	Occurs when custom exception handling is utilized instead of using JUnit's exception handling feature.	[56]
23	**For Testers Only**	**FTO**	A production class that contains methods that are only used for test methods.	[22]
24	**General Fixture**	**GF**	This smell emerges when `setUp()` fixture creates many objects, and test methods only use a subset.	[22]
25	**Guarded Test**	**GT**	A test that has conditional branches like `ifTrue:aCode` or `ifFalse:aCode`.	[59]
26	**Ignored Test**	**IgT**	A test method that uses an ignore annotation which prevents the test method from running.	[56]
27	**Indented Test**	**InT**	A test method that contains a large number of decision points, loops, and conditional statements.	[27]
28	**Indirect Testing**	**IT**	A test that interacts with a corresponding class by using another class.	[22]
29	**Lack of Cohesion of Methods**	**LCM**	When test methods are grouped in one test class, but they are not cohesive.	[39]
30	**Lazy Test**	**LT**	Occurs when multiple test methods check the same method of production object.	[22]
31	**Likely Ineffective Object-Comparison**	**LIOC**	A test that performs a comparison between objects will never fail.	[59]
32	**Long Test**	**LoT**	A test with many statements.	[59]
33	**Magic Number Test**	**MNT**	A test method that contains undocumented numerical values.	[56]
34	**Max Instance Variables**	**MIV**	A test method that has a large fixture.	[59]
35	**Mixed Selectors**	**MS**	Violates test conventions by mixing up testing and non-testing methods.	[59]
36	**Mystery Guest**	**MG**	A test that uses external resources, such as a database, that contains test data.	[22]
37	**Obscure In-line Setup**	**OISS**	A test that has too much setup functionality in the test method.	[39]
38	**Overcommented Test**	**OCT**	A test with numerous comments.	[59]
39	**Overreferencing**	**OF**	A test that causes duplication by creating unnecessary dependencies.	[59]
40	**Proper Organization**	**PO**	Bad organization of methods	[59]
41	**Redundant Assertion**	**RA**	A test method that has an assertion statement that is permanently true or false.	[56]
42	**Redundant Print**	**RP**	A test method that has print statement.	[56]
43	**Resource Optimism**	**RO**	A test that make an assumption about the existence of external resources.	[22]
44	**Returning Assertion**	**RA**	A test method that has an assertion and returns a value.	[59]
45	**Rotten Green Tests**	**RT**	Occurs when intended assertions in a test are never executed.	[31]
46	**Sensitive Equality**	**SE**	Occurs when an assertion has an equality check by using the `toString` method.	[22]
47	**Sleepy Test**	**ST**	Occurs when a test method has an explicit wait.	[56]
48	**Teardown Only Test**	**TOT**	Exists when a test-suite is only specifying teardown.	[59]
49	**Test Code Duplication**	**TCD**	Occurs when code clones contained inside the test.	[22]
50	**Test Maverick**	**TM**	Exists when a test class has a test method with an implicit setup; however, the test methods are independent.	[39]
51	**Test Pollution**	**TP**	Test that introduces dependencies such as reading/writing a shared resource.	[41]
52	**Test Redundancy**	**TR**	Occurs when the removal of a test does not impact the effectiveness of the test suite.	[44]
53	**Test Run War**	**TRW**	A test that fails when more than one programmer runs them.	[22]
54	**Test-Class Name**	**TCN**	A test that has a class with a meaningless name.	[59]
55	**Test-Method Category Name**	**TMC**	A test method has a meaningless name.	[59]
56	**Transcripting Test**	**TT**	A test that is printing and logging to the console.	[59]
57	**TTCN-3 Smells**	**TTCN**	Collection of smells specific to TTCN-3 test suites.	[20]
58	**Unclassified Method Category**	**UMC**	A test method that is not organized by a method category.	[59]
59	**Under-the-carpet Assertion**	**UCA**	A test that has assertions in the comments.	[59]
60	**Under-the-carpet failing Assertion**	**UCFA**	A test method that has failing assertions in the comments.	[59]
61	**Unknown Test**	**UT**	A test method without an assertion statement and non-descriptive name.	[56]
62	**Unused Inputs**	**UI**	Inputs that are controlled by the test.	[42]
63	**Unused Shared-Fixture Variables**	**USFV**	Occurs when a piece of the fixture is never used.	[59]
64	**Unusual Test Order**	**UTO**	A test that is calling other tests explicitly.	[59]
65	**Vague Header Setup**	**VHS**	A field that is initialized in the class header but not explicitly defined in code.	[39]
66	**Verbose Test**	**VT**	Test code that is complex and not simple or clean.	[27]

Table 4: Distribution of test smells detected by the test smell detection tools.

Tool \ Smell Type	AL	AR	CI	CTL	DA	DC	DepT	DF	DT	EH	EmT	ET	FTO	GF	IgT	InT	IT	LCM	LT	MG	MNT	OISS	RA	RO	RP	RT	SE	ST	TM	TR	TRW	UT	VHS	VT
DARTS [46]												√		√				√																	
DrTest [31]																										√								
DTDetector [75]							√																											
ElectricTest [24]							√																											
JNose Test [72]	√	√	√	√	√		√		√	√	√	√		√					√	√	√		√	√	√		√	√				√		√
PRADET [34]							√																											
RAIDE [60]		√			√																													
RTj [47]																										√								
SoCRATES [30]		√										√		√					√	√							√							
TASTE [52]												√		√				√																
TeCReVis [44]																														√				
TEDD [25]							√																											
TeReDetect [45]																														√				
TestEvoHound [40]								√						√				√				√							√			√		
TestHound [39]								√						√				√				√							√			√		
TestQ [27]	√	√				√				√	√	√	√	√		√	√			√							√						√
TSDETECT [56]		√	√	√	√				√	√	√	√		√	√				√	√	√		√	√	√		√	√				√		
Unnamed [22]		√				√						√	√	√			√		√	√							√				√			
Total	**2**	**6**	**2**	**2**	**3**	**2**	**5**	**2**	**2**	**2**	**3**	**7**	**2**	**9**	**1**	**1**	**2**	**4**	**4**	**5**	**2**	**2**	**2**	**2**	**2**	**2**	**5**	**2**	**2**	**2**	**1**	**2**	**2**	**2**

3.1.4 ***Supported Programming Languages.***

Next, we look at the programming languages supported by the various smell types we identify in this study. From Table 5, we observe that the smell types support four programming languages, specifically Java, Scala, Smalltalk, and C++. From this set, Java is the most popular programming language for test smell support, supporting 39 smell types, followed by Smalltalk (28 smell types). In Table 5, we also list the publications (tool and tool adoption) in our dataset that analyze these smell types. From this, we observe that a subset of the Java-supported test smells also support Scala unit test code. Although the developed XUnit guidelines can be applied to various languages [48], including dynamically typed ones such as JavaScript and Python, we did not locate tools that analyzes test suites written in these languages, which represents a noticeable limitation in terms of supporting the high quality of test suites.

> *Summary.* While there has been a steady release of test smell detection tools over the years, there has been an uptick in both tool development and adoption recently, specifically in 2019 and 2020. While most of the tools detect test smells occurring in Java test suites, there is a lack of support for other popular languages, such as JavaScript and Python.

3.2 RQ_2: What are the main characteristics of test smell detection tools?

This RQ comprises of two parts that examine the common characteristics of test smell detection tool. The first part examines specific high-level features of such tools, while the second part looks at the types of smell detection techniques implemented by the tools.

3.2.1 ***Common Characteristics.*** While the number and types of test smells detected by the tools are essential in selecting an appropriate tool, there are other features that can be considered to make a more informed decision. Similar to prior literature [32, 36], we review our set of test smell detection tools with respect to the following characteristics:

(1) **Programming Language** - This feature comprises of the programming language that the tool is implemented with and the programming language(s) the tool supports.
(2) **Supported Test Framework** - These frameworks provide an environment for developers to write unit tests. As part of the detection strategy the tool may be depended on the presence of specific framework API's in the test code.
(3) **Correctness** - Provides insight into how accurately the tool can detect smells. We look for instances where the tool authors provide values for precision and recall or F-measure.
(4) **Detection Technique** - Strategy the tool utilizes to analyze test code for the presence of smells.
(5) **Interface** - Indicates how developers interact with the tool.
(6) **Usages Guide Availability** - Indicates if documentation on how to use the tool is available (either in the tool's publication or website).
(7) **Adoption in Research Studies** - This provides insight into the popularity of the tool in the research community.
(8) **Tool Website** - Supplementary documentation about the tool, such as (where available) its source code repository, installation/execution instructions, etc.

Table 6 detail our findings for each tool. In case we cannot locate the needed information, we label it as 'UNK' in the table.

It is evident from Table 6 that the majority of test smell detection tools (≈ 86%) focus on detecting test smells exclusively for Java-based systems and are mostly focused on identifying any deviation from the guidelines of JUnit testing framework. This further corroborates our prior RQ finding where we show that most test smell types are geared towards Java systems, and thereby most research around test smells focuses on datasets composed of Java systems. Additionally, there are three tools, namely TestQ, TestHound, and TestEvoHound, which support two or more testing frameworks. In terms of correctness, most tools do not publish details around their accuracy. Additionally, the majority of tools do not report on performance speeds and execution times. Our findings show that only six tools published their detection accuracy, in terms of precision, recall or F-measure. From this set, only DARTS, TASTE, and TSDETECT report values for each smell type it supports; hence the correctness score is reported as a range. From our set of 22 detection tools, only five tools provide refactoring support. TestHound provides textual information on smells correction. While RAIDE provides a semi-automated correction, RTj, DARTS and TRex provide the automated refactoring of their detected smells. However, none of these tools provide details concerning the accuracy of their refactoring capabilities. From detection standpoint, we observe that tools, focusing on test dependency and rotten green test smells, use a dynamic

Table 5: Distribution of Test Smells Per Programming Languages.

Programming Language	Supported Test Smell Types				Literature Usage
Java	(01) Assertion Roulette (AR)	(11) Eager Test (ET)	(21) Magic Number Test (MNT)	(31) Test Maverick (TM)	[20, 49, 50, 73, 74]
	(02) Assertionless (AL)	(12) Empty Test (EmT)	(22) Mystery Guest (MG)	(32) Test Pollution (TP)	[22, 39, 40, 44, 75]
	(03) Brittle Assertion (BA)	(13) Exception Handling (EH)	(23) Obscure In-line Setup Smell (OISS)	(33) Test Redundancy (TR)	[23, 24, 41, 51, 65]
	(04) Conditional Test Logic (CTL)	(14) For Testers Only (FTO)	(24) Redundant Assertion (RA)	(34) Test Run War(TRW)	[34, 55, 58, 63, 72]
	(05) Constructor Initialization (CI)	(15) General Fixture (GF)	(25) Redundant Print (RP)	(35) TTCN-3 Smells (TTCN)	[25, 37, 38, 52, 61]
	(06) Dead Field (DF)	(16) Ignored Test (IgT)	(26) Resource Optimism (RO)	(36) Unknown Test (UT)	[46, 56, 64, 70, 71]
	(07) Default Test (DT)	(17) Indented Test (InT)	(27) Rotten Green Tests (RT)	(37) Unused Input (UI)	[26, 43, 53, 54, 62]
	(08) Dependent Test (DepT)	(18) Indirect Test (IT)	(28) Sensitive Equality (SE)	(38) Vague Header Setup(VHS)	[33, 42, 47, 57, 66]
	(09) Duplicate Assert (DA)	(19) Lack of Cohesion of Test Method (LCM)	(29) Sleepy Test (ST)	(39) Verbose Test (VT)	[45, 60]
	(10) Duplicated Code (DC)	(20) Lazy Test (LT)	(30) (31) Test Code Duplication (TCD)		
Scala	(01) Assertion Roulette (AR)	(03) Exception Handling (EH)	(05) Mystery Guest (MG)		[29, 30]
	(02) Eager Test (ET)	(04) General Fixture (GF)	(06) Sensitive Equality (SE)		
SmallTalk	(01) Abnormal UTF-Use (AUU)	(08) Empty Shared-Fixture (ESF)	(15) Under-the-carpet failing Assertion (UCFA)	(22) Test-Class Name (TCN)	[31, 59]
	(02) Anonymous Test (AT)	(09) Empty Test-MethodCategory (ETMC)	(16) Overcommented Test (OCT)	(23) Test-MethodCategory Name (TMC)	
	(03) Assertionless Test (AL)	(10) Guarded Test (GT)	(17) Overreferencing (OF)	(24) Transcripting Test (TT)	
	(04) Comments Only Test (COT)	(11) Likely ineffective Object-Comparison (LIOC)	(18) Proper Organization (PO)	(25) Unclassied MethodCategory (UMC)	
	(05) Control Logic (ConL)	(13) Long Test (LoT)	(19) Returning Assertion (RA)	(26) Under-the-carpet Assertion (UCA)	
	(06) Early Returning Test (ERT)	(12) Max Instance Variables (MIV)	(20) Rotten Green Tests falls (RT)	(27) Unused Shared-Fixture Variables (USFV)	
	(07) Empty MethodCategory (EMC)	(13) Mixed Selectors (MS)	(21) Teardown Only Test (TOT)	(28) Unusual Test Order (UTO)	
C++	(01) Assertion Roulette (AR)	(04) Eager Test (ET)	(07) General Fixture (GF)	(10) Mystery Guest (MG)	[27]
	(02) Assertionless Test (ALT)	(05) Empty Test (EmT)	(08) Indented Test (InT)	(11) Sensitive Equality (SE)	
	(03) Duplicated Code (DC)	(06) For Testers Only (FTO)	(09) Indirect Test (IT)	(12) Verbose Test (VT)	

Table 6: Characteristics of test smell detection tools.

Tool	Programming Language Implemented	Language Analyzed	Supported Test Framework	Correctness	Detection Technique	Interface	Usage Guide	Adoption in Studies	Tool Website
DARTS ‡ [46]	Java	Java	JUnit	F-Measure: 62%-76%	Information Retrieval	IntelliJ plugin	Yes	–	[3]
DrTest [31]	Smalltalk	Pharo ∇	SUnit	UNK	Rule Dynamic Tainting	Pharo plugin	Yes	–	[4]
DTDetector ★◇ [75]	Java	Java	JUnit	UNK	Dynamic Tainting	Command-line	Yes	–	[5]
ElectricTest [24]	Java	Java	JUnit	UNK	Dynamic Tainting	Command-line	No	–	UNK
JNose Test [70]	Java	Java	JUnit	UNK	Rule	Local web application	Yes	[71, 72]	[6]
OraclePolish ★ [42]	Java	Java	JUnit	UNK	Dynamic Tainting	Command-line	Yes	–	[7]
PolDet [41]	Java	Java	JUnit	UNK	Dynamic Tainting	UNK	No	–	UNK
PraDeT [34]	Java	Java	JUnit	UNK	Dynamic Tainting	Command-line	Yes	–	[8]
RAIDE ‡ [60]	Java	Java	JUnit	UNK	Rule	Eclipse plugin	Yes	–	[10]
RTj ‡ [47]	Java	Java	JUnit	UNK	Rule Dynamic Tainting	Command-line	Yes	–	[11]
SoCRATES [30]	Scala	Scala	ScalaTest	Precision: 98.94% Recall: 89.59%	Rule	IntelliJ plugin	Yes	[29]	[12]
Taste [52]	UNK	Java	JUnit	Precision: 57%-75% Recall: 60%-80%	Information Retrieval	UNK	No	[54]	UNK
TeCReVis ★ [44]	Java	Java	JUnit	UNK	Metrics Dynamic Tainting	Eclipse plugin †	Yes	–	[14]
TEDD [25]	Java	Java	JUnit	Precision: 80% Recall: 94%	Information Retrieval	Command-line	Yes	[26]	[13]
TeReDetect ★ [45]	Java	Java	JUnit	UNK	Metrics Dynamic Tainting	Eclipse plugin †	Yes	–	[14]
TestEvoHound [40]	Java	Java	JUnit, TestNG	UNK	Metrics	UNK	No	–	UNK
TestHound ‡★ [39]	Java	Java	JUnit, TestNG	UNK	Metrics	Desktop application	No	–	[15]
TestLint ★ [59]	Smalltalk	Smalltalk	Sunit	UNK	Rule Dynamic Tainting	UNK	Yes	–	[16]
TestQ ★ [27]	Python	C++, Java	CppUnit, JUnit, Qtest	UNK	Metrics	Desktop application	Yes	–	[17]
TRex ‡§★ [20]	Java	Java	TTCN-3	UNK	Rule	Eclipse plugin	Yes	[49, 50, 73, 74]	[18]
tsDetect [56]	Java	Java	JUnit	Precision: 85%-100% Recall: 90%-100%	Rule	Command-line	Yes	[43, 55, 61, 64] [33, 53, 57, 62]	[19]
Unnamed [22]	UNK	Java	JUnit	Precision: 88% Recall: 100%	Rule	Command-line	No	[23, 51, 65, 66] [37, 38, 53, 58, 63]	UNK

‡ Provides support for refactoring. | † Embedded inside the CodeCover [2] plugin. | § Enhanced version released in [49]. | ★ Included in the catalog of Garousi and Küçük [36].
◇ Also known as 'TestIsolation' in the catalog of Garousi and Küçük [36]. | ∇ Pharo is a dialect of Smalltalk.

detection strategy, while most static analysis-based smells prefer to utilize a rules-based approach. We discuss in detail the different techniques later on. In terms of adoption in research studies, only seven of the tools have been utilized by the research community to study test smells.

Next, in terms of how developers run/interact with these tools, there are two categories– graphical user interface (GUI) and command-line (i.e., non-GUI) tools. Most of the GUI tools are in the form of IDE plugins or web and desktop applications. In terms of tool availability, we searched for a link to the tool website or binaries. In case the link is absent or no longer functional, we contacted the publication's corresponding author. From these 22 publications, we were able to only locate 17 tools. It should be noted that, except for TestLint, the website for these 17 tools points to the tool's source code repository. Further, when examining the project repositories, we observe that tsDetect was the most forked repository (21 forks). Furthermore, an examination of a tool's publications, website, and the README file in the source code repository (where available) yields only 16 tools presenting guidelines on how to setup and/or execute the tool. Finally, from this set, only tsDetect and the tool by Bavota et al. [22] show a high adoption rate, having been used in at least eight other studies. The majority of the tools are only limited to the studies to which they were first introduced.

3.2.2 *Smell Detection Techniques.*

Each tool represents an implementation of a smell detection strategy. Each detection strategy reflects an interpretation of how the smell type manifests in the source code, i.e., how the smell symptoms can be identified. Our analysis shows that, while most of the tools in our study rely on static analysis of source code, some tools identify smells through dynamic analysis. These detection strategies can be grouped into four categories, namely Metrics, Rules/Heuristic, Information Retrieval, and Dynamic Tainting. With this clustering of strategies, we aim to familiarize future smell detection tool researchers/developers with smell detection techniques.

Metrics. The use of metrics to profile smells, is one of the early techniques, and popular ones, for all code smells in general. In a nutshell, the smell symptoms are measured through their impact on structural and semantic measurements, where their values go beyond pre-defined threshold. These metrics, and their corresponding threshold values, are combined into a rules-based tree to make a binary decision, of whether the code under analysis suffers from a smell or not. In a typical metrics-based smell detection approach, the source code is parsed and converted into an abstract syntax tree (AST). This AST is then subjected to a metrics-based analysis to identify and capture the test smells. For instance, Van Rompaey et al. [69], utilize metrics such as Number of Object Used in setup, Number of Production-Type, Number of Fixture Objects, Number Of Fixture Production Types, and Average Fixture Usage to detect smells such as *General Fixture* and *Eager Test*. The threshold used are chosen by the user or empirically derived from a representative set.

Rules/Heuristic. Rules or heuristics smell detection augments the metrics-based techniques with patterns that can be found in the source code. The smell is detected when the input matches a pre-defined set of metric thresholds with the existence of some code patterns. For example, the *Assertion Roulette* smell is detected by defining a heuristic that checks whether a test method contains several assertion statements without an explanation message as a parameter for each assertion method.

Information Retrieval. In this technique, the main steps include extracting information/content from the test code and normalizing it. As part of the extraction process, the textual content from each JUnit test class (e.g., source code identifier and source comments) are taken as the necessary features for identifying the test smells. These characteristics are normalized via multiple text pre-processing steps. These steps include stemming, decomposing identifier names, stop word, and programming keyword removal. The normalized text is then weighted using Term Frequency and Inverse Document Frequency. The end-result is applying machine learning algorithms to extract textual features that can discriminate between classes, i.e., smell types.

Dynamic Tainting. It monitors the source-code while it executes. Dynamic tainting enables the analysis of the actual data from the code based on run-time information. In particular, it works in two steps: (1) run the source-code along with predefined taint value/mark (i.e., user input), and (2) reason which executions are affected by that value/mark.

Our categorization of the tools, in our study, based on the four smell detection techniques are shown in Table 6. The Rule/Heuristic-based technique is a frequently adopted detection technique as the definition for most smells are based on well-defined rules [48, 67]. The metric-based mechanism is less frequently utilized due to: (1) not all known metrics proposed by [27, 39, 68, 69] have the ability to detect all test smells, since they can go beyond traditional design anomalies, and (2) the reliance on determining an appropriate set of thresholds is considered to be a significant challenge. Looking at tools that utilize a dynamic-based detection technique, we observe that most test dependency and rotten green test detection tools utilize this technique. Finally, DARTS, Taste, and TEDD utilize Information Retrieval techniques. However, as these tools rely on source code feature extraction, the lack of such information could affect the detection accuracy [52].

> *Summary.* JUnit is the most popular testing framework supported by test smell detection tools, with most static analysis based tools opting to utilize a rules-based detection strategy to identify smells. Even though most of the tools publish their source code, information about the tool's accuracy is seldom available. Our analysis only shows that only six tools publish their scores related to correctness.

4 DISCUSSION

As a systematic mapping study, our findings provide a high-level understanding of the current state of test smell detection tools. In brief, as seen from our RQ findings, the research community has produced many test smell detection tools that support the detection of various test smell types. These tools, in turn, have been utilized in studies on test smells to understand how they influence software development. However, our findings also demonstrate areas of concern and expansion in this field. In this section, through a series of takeaways, we discuss how our findings can support the research and developer community in selecting the right tool as well as provide future directions for implementing and maintaining future test smell tools.

Takeaway 1: Standardization of smell names and definitions. From RQ_1, we observe an overlap of the smell types detected by the tools. For instance, *Assertion Roulette* is detected by six JUnit supported tools. However, the implementation of the detection rules may vary between these tools. Additionally, there can be instances where some test smell types with the same/similar definitions are known by different names. This fragmentation of smell definitions is not unique to test smells; Sobrinho et al. [28] experience this in code smells. This phenomenon provides the opportunity for future research in this area to compare and contrast such smell types. The agreement on smells definitions, does not necessarily induce similar interpretations. Since there is no consensus on how to measure smells, each smell type can be identified using different detection strategies, and the choice of the strategy becomes part of the developer's preferences.

Takeaway 2: Improve support for non-Java programming languages and testing frameworks. While our findings from RQ_1 show the existence of multiple test smell detection tools, our RQ_2 findings show that most of these tools are limited to supporting Java systems that utilize the JUnit testing framework, thereby narrowing test smell research to Java systems. Hence, restricting research to a single environment/language will not accurately reflect reality. While it can be argued that as most test smells are based on xUnit guidelines [48] research findings on Java systems can carry

over to other similar languages (e.g., C#), actual practitioners of non-Java systems gain no benefit without a tool to use in their development workflow. Furthermore, recent trends have shown a rise in the popularity of dynamically typed programming languages (e.g., Python and JavaScript) [9] giving more urgency for the research community to construct tools that support non-traditional research languages.

Takeaway 3: Do not reinvent the wheel. Researchers/practitioners need to evaluate if implementing another detection tool is necessary for their specific needs or if modifying an existing tool would suffice. Reusing existing tools will not only save effort, but also develops more mature and robust frameworks. For instance, Spadini et al. [61] integrate tsDetect into a code quality monitoring system, while the tool implemented by Tufano et al. [66], HistoryMiner, utilizes the tool of Bavota et al. [22] to detect test smells in the lifetime of a project. Additionally, some of the tools in our dataset are based on other tools in the dataset. For instance, JNose Test and RAID are built on top of tsDetect and DARTS is based on Taste. It is important that, when introducing new tools, tool maintainers should design their tools to be ready for customization. For instance, Spadini et al. [64] customize tsDetect by introducing thresholds to meet their research objective. This can help reduce the release of near-duplicate tools. It also further strengthens the case for the importance of public availability of a tool's source code. Having access to the code enables the improvement of the tool's quality. It also facilitates extensions in improving current detection strategies or introducing the detection of new smell types, which, in the long run, improves the tool's usefulness.

Takeaway 4: Improve transparency on the quality of tools. As reported in RQ_2, only a few tools report on the correctness of the tool. Furthermore, clarity around bias mitigation is not completely addressed for the tools that do report correctness scores. While our objective here is not to discredit the validity of the current set of test smell detection tools, we only highlight inconsistencies that might lead to research studies obtaining varying results based on the tool in use. As stated by Panichella et al. [53], there is a need for a community-maintained gold-set/standard of smelly test files to validate the current and future smell detection tools. We understand that the process involved in the creation of a community-curated gold-set might be time-consuming. Hence, in the meantime, we recommend that the peer-review process be adjusted, such that providing metrics for precision and recall at the smell type level (instead of an overall tool accuracy score) along with the evaluation dataset is made mandatory.

Takeaway 5: Expand from just detecting test smells to interactive refactoring.

Granted that the primary purpose of these tools is the detection of test smells, developers will also immensely benefit from suggested refactoring templates for each smell type. While there have been initial efforts to include such functionality in detection tools, it does not generalize to the current popular detected smells, and more research is needed to elaborate on their ability to appropriately change the test files without introducing any regression.

5 CONCLUSION AND FUTURE WORK

This study identifies 22 test smell detection tools made available by the research community through a comprehensive search of peer-reviewed scientific publications. As part of our analysis, we identify the smell types detected by these tools and highlight the smell types that overlap. Additional comparisons between these tools show that most of these tools support the JUnit framework, and while the source code is made available, details around the tool's detection correctness are not always made public. We envision our findings act as a one-stop source for test smell detection tools and empower researchers/practitioners in selecting the appropriate tool for their requirements. Future work in this area includes a hands-on evaluation of each tool to determine the extent to which tools detect common smell types and create a benchmark for such smell types.

REFERENCES

[1] [n.d.]. https://doi.org/10.5281/zenodo.4726288.
[2] [n.d.]. CodeCover. http://codecover.org/.
[3] [n.d.]. DARTS. https://github.com/StefanoLambiase/DARTS.
[4] [n.d.]. DrTest. https://github.com/juliendelplanque/DrTests.
[5] [n.d.]. DTDetector. https://github.com/winglam/dtdetector.
[6] [n.d.]. JNose. https://github.com/arieslab/jnose.
[7] [n.d.]. OraclePolish. https://bitbucket.org/udse/.
[8] [n.d.]. PRADET. https://github.com/gmu-swe/pradet-replication.
[9] [n.d.]. PYPL. http://pypl.github.io/PYPL.html.
[10] [n.d.]. RAIDE. https://raideplugin.github.io/RAIDE/.
[11] [n.d.]. RTj. https://github.com/UPHF/RTj.
[12] [n.d.]. SOCRATES. https://github.com/jonas-db/socrates.
[13] [n.d.]. TEDD. https://github.com/matteobiagiola/FSE19-submission-material-TEDD.
[14] [n.d.]. TeReDetect & TeCReVis. https://sourceforge.net/projects/codecover/.
[15] [n.d.]. TestHound. https://github.com/SERG-Delft/TestHound.
[16] [n.d.]. TestLint. http://scg.unibe.ch/wiki/alumni/stefanreichhart/testsmells.
[17] [n.d.]. TestQ. https://code.google.com/archive/p/tsmells/.
[18] [n.d.]. TRex. https://www.trex.informatik.uni-goettingen.de/trac.
[19] [n.d.]. tsDetect. https://testsmells.github.io/.
[20] Paul Baker, Dominic Evans, Jens Grabowski, Helmut Neukirchen, and Benjamin Zeiss. 2006. TRex-the refactoring and metrics tool for TTCN-3 test specifications. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*. IEEE, 90–94.
[21] Balbir Barn, Souvik Barat, and Tony Clark. 2017. Conducting Systematic Literature Reviews and Systematic Mapping Studies. In *Proceedings of the 10th Innovations in Software Engineering Conference* (Jaipur, India) *(ISEC '17)*. Association for Computing Machinery, New York, NY, USA, 212–213.
[22] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. 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)*. IEEE, 56–65.
[23] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.
[24] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 770–781.
[25] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 154–164.
[26] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella. 2020. Dependency-Aware Web Test Generation. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 175–185.
[27] Manuel Breugelmans and Bart Van Rompaey. 2008. TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*.
[28] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia. 2021. A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering* 47, 1 (2021), 17–66.
[29] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Assessing diffusion and perception of test smells in scala projects. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 457–467.
[30] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. SoCRATES: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. 22–26.
[31] J. Delplanque, S. Ducasse, G. Polito, A. P. Black, and A. Etien. 2019. Rotten Green Tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 500–511. https://doi.org/10.1109/ICSE.2019.00062
[32] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment*

in Software Engineering. 1–12.

[33] G. Fraser, A. Gambi, and J. M. Rojas. 2020. Teaching Software Testing with the Code Defenders Testing Game: Experiences and Improvements. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 461–464. https://doi.org/10.1109/ICSTW50294.2020.00082

[34] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11.

[35] V. Garousi, B. Küçük, and M. Felderer. 2019. What We Know About Smells in Software Test Code. *IEEE Software* 36, 3 (2019), 61–73.

[36] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software* (2018).

[37] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software* 156 (2019).

[38] G. Grano, F. Palomba, and H. C. Gall. 2019. Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2903057

[39] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 322–331.

[40] Michaela Greiler, Andy Zaidman, Arie Van Deursen, and Margaret-Anne Storey. 2013. Strategies for avoiding text fixture smells during software evolution. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 387–396.

[41] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 223–233.

[42] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 621–631.

[43] D. J. Kim. 2020. An Empirical Study on the Evolution of Test Smell. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 149–151.

[44] Negar Koochakzadeh and Vahid Garousi. 2010. Tecrevis: a tool for test coverage and test redundancy visualization. In *International Academic and Industrial Conference on Practice and Research Techniques*. Springer, 129–136.

[45] Negar Koochakzadeh and Vahid Garousi. 2010. A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering* 2010 (2010).

[46] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In *Proceedings of the 28th International Conference on Program Comprehension*. 441–445.

[47] Matias Martinez, Anne Etien, Stéphane Ducasse, and Christopher Fuhrman. 2020. RTj: A Java Framework for Detecting and Refactoring Rotten Green Test Cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 69–72.

[48] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[49] Helmut Neukirchen and Martin Bisanz. 2007. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Testing of Software and Communicating Systems*, Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–243.

[50] Helmut Neukirchen, Benjamin Zeiss, and Jens Grabowski. 2008. An approach to quality engineering of TTCN-3 test specifications. *International Journal on Software Tools for Technology Transfer* 4 (2008).

[51] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code: An empirical study. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 5–14.

[52] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 311–322.

[53] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn. 2020. Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 523–533. https://doi.org/10.1109/ICSME46990.2020.00056

[54] Fabiano Pecorelli, Gianluca Di Lillo, Fabio Palomba, and Andrea De Lucia. 2020. VITRuM: A Plug-In for the Visualization of Test-Related Metrics. In *Proceedings of the International Conference on Advanced Visual Interfaces* (Salerno, Italy) *(AVI '20)*. Association for Computing Machinery, New York, NY, USA, Article 101.

[55] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[56] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 5. https://doi.org/10.1145/3368089.3417921

[57] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. An Exploratory Study on the Refactoring of Unit Test Files in Android Applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 350–357. https://doi.org/10.1145/3387940.3392189

[58] Abdallah Qusef, Mahmoud O Elish, and David Binkley. 2019. An Exploratory Study of the Relationship Between Software Test Smells and Fault-Proneness. *IEEE Access* 7 (2019), 139526–139536.

[59] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. 2007. Rule-based Assessment of Test Quality. *Journal of Object Technology* 6, 9 (2007), 231–251.

[60] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 374–379.

[61] Martin Schvarcbacher, Davide Spadini, Magiel Bruntink, and Ana Oprescu. 2019. Investigating developer perception on test smells using better code hub-Work in progress. In *2019 Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE*.

[62] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André Santos. 2020. Refactoring Test Smells: A Perspective from Open-Source Developers. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing* (Natal, Brazil) *(SAST 20)*. Association for Computing Machinery, New York, NY, USA, 50–59.

[63] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–12.

[64] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. 2020. Investigating Severity Thresholds for Test Smells. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 311–321. https://doi.org/10.1145/3379597.3387453

[65] Amjed Tahir, Steve Counsell, and Stephen G MacDonell. 2016. An empirical study into the relationship between class features and test smells. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 137–144.

[66] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 4–15.

[67] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. 92–95.

[68] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. 2006. Characterizing the relative significance of a test smell. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 391–400.

[69] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 2007. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering* 33, 12 (2007), 800–817.

[70] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java Test Smell Detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*. 564–569.

[71] Tássio Virgínio, Luana Almeida Martins, Larissa Rocha Soares, Railana Santana, Heitor Costa, and Ivan Machado. 2020. An empirical study of automatically-generated tests from the perspective of test smells. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*. 92–96.

[72] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. 2019. On the influence of Test Smells on Test Coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. 467–471.

[73] Edith Werner, Jens Grabowski, Helmut Neukirchen, Nils Röttger, Stephan Waack, and Benjamin Zeiss. 2007. TTCN-3 Quality Engineering: Using Learning Techniques to Evaluate Metric Sets. In *SDL 2007: Design for Dependable Systems*, Emmanuel Gaudin, Elie Najm, and Rick Reed (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–68.

[74] Benjamin Zeiss, Helmut Neukirchen, Jens Grabowski, Dominic Evans, and Paul Baker. 2006. Refactoring and Metrics for TTCN-3 Test Suites. In *System Analysis and Modeling: Language Profiles*, Reinhard Gotzhein and Rick Reed (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 148–165.

[75] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 385–396.