Empirical Software Engineering (2023) 28:20
https://doi.org/10.1007/s10664-022-10251-1

Smells in system user interactive tests

Renaud Rwemalika[1] · Sarra Habchi[1] · Mike Papadakis[1] · Yves Le Traon[1] · Marie-Claude Brasseur[2]

Accepted: 26 September 2022 / Published online: 17 December 2022

Abstract

Test smells are known as bad development practices that reflect poor design and implementation choices in software tests. Over the last decade, there are few attempts to study test smells in the context of system tests that interact with the System Under Test through a Graphical User Interface. To fill the gap, we conduct an exploratory analysis of test smells occurring in System User Interactive Tests (SUIT). We thus, compose a catalog of 35 SUIT-specific smells, identified through a multi-vocal literature review, and show how they differ from smells encountered in unit tests. We also conduct an empirical analysis to assess the diffuseness and removal of these smells in 48 industrial repositories and 12 open-source projects. Our results show that the same type of smells tends to appear in both industrial and open-source projects, but they are not addressed in the same way. We also find that smells originating from a combination of multiple code locations appear more often than those that are localized on a single line. This happens because of the difficulty to observe non-local smells without tool support. Furthermore, we find that smell-removing actions are not frequent with less than 50% of the affected tests ever undergoing a smell removal. Interestingly, while smell-removing actions are rare, some smells disappear while discarding tests, i.e., these smells do not appear in follow-up tests that replace the discarded ones.

Communicated by: Jinqiu Yang

✉ Renaud Rwemalika
renaud.rwemalika@hotmail.com

Sarra Habchi
sarra.habchi@uni.lu

Mike Papadakis
michail.papadakis@uni.lu

Yves Le Traon
yves.letraon@uni.lu

Marie-Claude Brasseur
marie-claude.brasseur@bgl.lu

[1] University of Luxembourg, Luxembourg, Luxembourg

[2] BGL BNP Paribas, Luxembourg, Luxembourg

Keywords Test smells · System user interface tests · GUI tests · Refactoring · Multivocal literature review

1 Introduction

User interfaces are designed to provide a means for the user to interact with an application. Unfortunately, as interfaces become more user-friendly, the underlying technology becomes more complex (Myers 1994). In addition to the increase in complexity, nowadays, Graphical User Interfaces (GUI) are becoming more and more common (Myers and Rosson 1992; Myers 1995; Brooks et al. 2009; Memon and Nguyen 2010). Thus, the question of testing GUI-based application becomes of upmost importance.

To increase the release pace while maintaining a high quality of their products, software companies adopt continuous integration. To this end, much effort has been devoted towards GUI test automation, notably focusing on desktop applications (Nguyen et al. 2014; Advolodkin 2018; Pezzè et al. 2018), web applications (Mesbah and Van 2009; Biagiola et al. 2019), mobile applications (Machiry et al. 2013; Gomez et al. 2013; Mao et al. 2016; Salihu et al. 2019; Yu et al. 2019), and cross-platform applications (Canny et al. 2020).

All these techniques rely on the same principle: A test framework executes events belonging to GUI components and monitors the resulting changes in the state of the System Under Test (SUT) (Nguyen et al. 2014). In other words, each technique generates a sequence of test steps, that are executed against the SUT (input) and captures some indication as to what is the current state of the system (output). We refer to this category of tests as System User Interactive Tests (SUIT) that can be defined as an automatic test exercising a software application with a graphical front-end to check whether it meets its specifications by performing a sequence of events against the GUI elements (Cunha et al. 2010; Banerjee et al. 2013; Issa et al. 2012). As such, SUITs present unique characteristics in their (1) structure, (2) interaction with the SUT, and (3) the actors involved in their maintenance.

Thanks to the maturity of test automation frameworks, both commercial and open-source projects are increasingly adopting automated test suites to ensure software quality at the system level (Mabl 2021). Keyword-Driven Testing (KDT) forms such an automated approach that is widely adopted by industry (Mabl 2021; Katalon 2018). It allows writing test scripts that are readable (by all the stakeholders) and easily understandable by humans.

Unfortunately, KDT is associated with high maintenance effort (Gao et al. 2015; Coppola et al. 2019; Rwemalika et al. 2019a) due to their fragility, structure and usage that is significantly impacted by the evolution of the projects. To aid testers in reducing the involved maintenance cost, the research community proposed solutions by automating test maintenance (Hurdugaci and Zaidman 2012) or by setting best practices, in some sense exposing sub-optimal processes (Labuschagne et al. 2017) that make test maintenance hard. This work aims at defining and monitoring frequent sub-optimal patterns of SUITs during the projects development process.

Previous research has that many issues emerge in test code that results in increase of the code entropy (Hanssen et al. 2010). For example, prior work has shown the existence of sub-optimal design choices that lower the quality of the test code (van Deursen et al. 2001; Meszaros 2007; Reichhart et al. 2007; Van et al. 2007; Bavota et al. 2015; Tufano et al. 2016; Bowes et al. 2017; Kim 2020; Peruma et al. 2020) in the form of test smells. This concept of smells was first introduced by (Fowler et al. 1999), who defined code smells as poor design and implementation choices that hinder the system maintainability. It was later extended to test code, with the introduction of test smells (van Deursen et al. 2001).

Interacting through the user interface, SUITs present unique characteristics that originate from the way they interact and synchronize with the SUT. Indeed, unlike unit tests, SUITs are executed in separated processes than the SUT. Thus, they require mechanisms to identify and interact with GUI elements thereby allowing to reach new states in the SUT. To avoid any race condition between the tests and the system they exercise, synchronization mechanism must be set into place to allow SUT to reach their new state before the SUIT continue with their execution. Generally, SUITs treats the SUT as a black box thereby focusing on end-to-end behavior by ignoring implementation details.

The above means that SUITs are fundamentally different from unit tests and as a result, the conclusions drawn from previous work on the detection of smells (van Deursen et al. 2001; Tufano et al. 2016; Bowes et al. 2017), the analysis of their impact and diffusion (Bavota et al. 2015; Tufano et al. 2016; Kim 2020), and their automatic removal (Van et al. 2007; Reichhart et al. 2007; Peruma et al. 2020) does not generally hold in the context of SUITs. For example, the smell *Mystery Guest*, which is well studied in test smell literature (van Deursen et al. 2001; Bavota et al. 2015; Tufano et al. 2016; De Bleser et al. 2019; Peruma et al. 2020; Virginio et al. 2020) appears when a test class relies on an external resource file. However, in the case of SUITs, extracting test data outside the test code is considered a good practice (thus, the opposite of a smell) and is highly used in data-driven testing (Baker et al. 2008) to decouple the data from the behavior specifications. Indeed, one of the functions of a SUIT being communication between different roles, namely business analysts, testers, and developers, technical details are hidden away to ease communication between the stakeholders. Therefore, while there might be an overlap between smells in unit tests and SUITs, some smells are specific to each category.

Thus, this study aims to consolidate our understanding on the understudied SUITs smells. Specifically, we aim to answer the following research questions:

– **RQ1**: Which are the SUIT smells that are mentioned in academic and grey literature?
 Goal: This question aims to explore and identify test smells that are specific to SUITs. Identifying and categorizing test smells that are mentioned and discussed in the grey literature bridges the gap between research community and practitioners since the majority of grey literature is based on the practitioners views. In view of this, our intent is to produce a catalog of known SUIT smells along with their definition and impact.
– **RQ2**: How prevalent the SUIT smells are?
 Goal: This question aims at assessing the prevalence of SUIT smells in industrial and open-source projects and how different smells appear in the test codebase. An additional aim in analysing this question is the analysis of how smells are viewed and managed by practitioners. Answering this question provides evidence on the importance of the SUIT smells, i.e., very rare smells are probably not interesting as they do not appear in practice.
– **RQ3**: How often do we observe smell-removing changes in SUIT smell symptoms?
 Goal: This question aims to analyze the smell-removing actions performed by maintainers to remove SUIT smells. While there may be a large amount of smells present in the test codebase, practitioners may be unaware of them. To explore this, we identify and detect fine-grained smell-removing actions that reflect an interest by the maintainers of the test suite. Answering this question provides evidence in the importance of the smells, i.e., they are associated with development time and effort.

To answer these questions, we combine a multi-vocal literature review and an empirical study on a large industrial project and 12 open-source repositories. Relying on the multivocal literature review, we answer RQ1 and build a catalog of 35 SUIT smells. For 16 out of the 35 SUIT smells of this catalog we propose an automated approach for detecting their diffusion and removal. Based on this approach, we measure the prevalence of SUIT smells and their removal in more than two million tests from our dataset composed of industrial and open-source projects. In summary, the main contributions of this study are:

- We introduce the first catalog of SUIT-specific smells building on the knowledge of both researchers and practitioners.
- We develop an open-source automated tool[1] to detect smells and their removals for 16 SUIT smells. Alongside with it, we implemented a SonarQube Plugin[2] able to detect smell instances in Robot Framework code.
- We show that smells caused by combinations of multiple code entities (functions, files, suites) are harder to detect and thus, tend to appear more often than smells with symptoms localized on a single line.
- Our results suggest that less than half of SUITs ever experience smell removal actions. *Missing Assertion* is a unique exception with up to 90% of the symptomatic tests refactored. However, assertions are rapidly added which suggesting it is an artifact of the development process. Tests are created and once they work, concrete assertions are added. Interestingly, while smell-removing actions are rare, SUIT smells like *Narcissistic* and *Middle Man* still disappear from the codebase as a side effect of unintended maintenance and refactoring operations.

2 Background

In this section, we discuss current techniques used to generate SUITs. The techniques are grouped based on their theoretical potential for automation during the generation process.

2.1 Random GUI Testing

In Random GUI Testing, tests are automatically created by generating steps in the hope to explore the possible state of the SUT to reveal failures. Because no requirements are passed as input, the generated tests rely on specific constraints that are application or domain invariant (Mesbah and Van 2009). This property makes random GUI testing cheap to run on a large number of version/configuration of an application. Thus, random GUI testing offers a cheap alternative to classical test scripting which can be useful to detect crashes and other invariants in an exploratory fashion. However, because of the absence of functional requirements and the lack of knowledge of the lifecycle of the application, exploring deep states of the application and exposing complex behavior remains challenging. This is why, while companies do use random GUI testing tools such as Sapienz (Mao et al. 2016) in companies like Facebook, they use it in addition to the other techniques that we present below.

2.2 Model-Based Testing

Model-Based Testing (MBT) can be defined as an approach encompassing the process and techniques for the automatic derivation of abstract test cases from abstract models (Utting et al. 2012). Functional specifications of a system, with the assumption that they are precise

[1] Available at https://github.com/serval-uni-lu/ikora-evolution
[2] Available at https://github.com/serval-uni-lu/sonar-ikora-plugin

enough, are used as input to design process models which in turn are used to generated automated test cases (Gupta and Surve 2011). Unfortunately, today, GUI-based application are not typically derived from a model. Thus, QA engineers cannot rely on an *a priori* model encompassing all the expected behaviors of the application. Besides, even though advances have been made in Model-Based Software Engineering, complex GUI applications exhibit behaviors that cannot be expressed by models found in GUI testing (Lelli et al. 2015). To tackle this shortcoming, authors propose to rely on reverse-engineering to automatically derive a model of the SUT by automatically exploring the application (in a similar fashion as random GUI testing). Unfortunately, models built that way do not guarantee completeness, because of limitations such as assessing if the coverage offered by the model is sufficient (Yuan et al. 2007) or choosing inputs during exploration (Biagiola et al. 2019). These limitations cause MBT to offer limited applicability which explains why practitioners (with the exception of safety critical domains) tend to shy away from the model-based solutions proposed in the scientific literature.

2.3 Record & Replay

When adopting Record & Replay, test cases are generated from a sequence of user interactions provided by the tester. As its name suggest Record & Replay works in a two phase process: a record phase and a replay phase. During the record phase, the tester manually interacts with the application, thereby generating events on the SUT. The tooling records these interactions and a part of the resulting SUT state as specified by the tester. Then, during the replay phase, the recorded test cases can be replayed on subsequent versions of the SUT and the captured states of the SUT are used as test oracles. The generated test cases require human intervention during the record phase, but are executed automatically during the replay phase. Hence, Record & Replay reduces the overall effort of regression testing when compared to manual testing. Additionally, because the test cases are generated by the framework, no particular skills are needed from the tester. (Di Martino et al. 2021) show that even when human testers have limited information about the system, they achieve better coverage metrics than automatic test generation techniques when using Record & Replay.

Despite its advantages when generating the test cases, the generated tests tend to be fragile (Hammoudi et al. 2016b). To address this drawback, the research community proposes different ways to tackle the problem of fragility by automatically repairing broken tests (Hammoudi et al. 2016a), by improving the recording (Ronsse and De Bosschere 1999), or by generating a model of the SUT instead of directly building test cases (Saddler and Cohen 2017). Unfortunately, even with these advances from the scientific community, the approach suffers from high cost in terms of the maintainability of the generated test suites. This issue is all the more relevant as the generated tests are often obscure, lacking any type of internal hierarchy, thus, compromising any attempt to manually fix the tests. The solution offered by the industry where most of the major tool vendor offer Record & Replay capabilities (*e.g.* IBM Rational Functional Tester, TestComplete, HP UTF, SmartBear, Katalon) is to represent the generated test as script and allow test automation engineer to manually improve them as they are being built (Section 2.4).

2.4 Test Scripting

Like Record & Replay, SUIT scripts are typically used by teams performing acceptance testing. Acceptance tests ensure that a specific acceptance criterion, which can be functional or non-functional, is met (Pandit and Tahiliani 2015). Conforming to the acceptance criteria both verifies that the SUT delivers the business value expected by the customer and guards against regressions or defects that break preexisting functions of the SUT (Humble and Farley 2010). Therefore, acceptance tests are not concerned with the internal implementation of the SUT, but with the overall behavior of the system. Additionally, because such tests are business-facing, they are involved in the discussion between testers, developers, and business analysts, and, as such, should be readable by all stakeholders.

To ease this communication process, practitioners propose design patterns to separate different concerns into different abstraction layers. For example, (Humble and Farley 2010) propose the three following layers adopted by open-source and commercial tools alike (*e.g.* Robot Framework, Cucumber, JBehave, Finess, or TestComplete): (1) the acceptance criteria which describes the functional behavior, usually written in a form close to natural language; (2) the test implementation layer that contains the underlying implementation of the test using the vocabulary from the application domain; and (3) the application driver layer that understands how to interact with the SUT and is expressed in the domain language of the driver that is used to communicate with the SUT.

One approach commonly used in industry that promotes this architecture is Keyword-Driven Testing (KDT). KDT aims at separating test design from the technical implementation of tests, thus, limiting exposure to unnecessary details. To do so, KDT relies on *Keywords*, *i.e.* named procedures, that are composed of steps expressing a behavior, where each step represent a subsequent call to a *Keywords*. KDT advocates that this separation of concerns makes tests easier to write and more maintainable. On top of that, the separation enables experts from different fields and backgrounds to work together, at different levels of abstraction (Tang et al. 2008). This enables the unobstructed collaboration in the creation and analysis of the tests between different experts. This paradigm is used by the major tool offering across the market such as TestComplete, HP UTF, SmartBear, Katalon, Ranorex, and Robot Framework, which are jointly used by about 20-30% of the teams having adopted automated acceptance testing (Katalon 2018; Mabl 2021).

Notably, another paradigm is largely used, namely, Behavior Driven Development (BDD) with the extensive use of the tool Cucumber representing another 20-25% of the market share. Similarly to KDT, BDD implements the three-layer architecture prescribed by (Humble and Farley 2010), but instead of relying on keyword for the test implementation layer, general-purpose languages such as Python or Java are used. Finally, many teams still rely on general purpose languages and their associated testing tools (*e.g.* Junit, Pytest) to write their tests and call specific drivers like Appium[3] or Selenium[4]. However, with the consolidation of the offering and the improvement of the tooling, this practice tend to reduce (Mabl 2021).

Unfortunately, test scripting comes with its challenges as well. Indeed, because the GUI layer exhibits a lot of variation across versions and configurations (Gao et al. 2015), SUITs tend to be fragile, *i.e.* breaking following non-functional changes resulting from the natural evolution of the SUT. Hence, test cases fail or require maintenance even though the specific functionalities they test remain unchanged (Coppola et al. 2019; Di Martino et al. 2021).

In this work, we focus on the KDT paradigm and more specifically on its supporting open-source framework, Robot Framework, as it is the technology deployed at our partner BGL BNP Paribas. Furthermore, the open-source nature of the tooling allows us to extend our study to other open-source projects mined from Github. In the remainder of this section, we describe in more details KDT and Robot Framework.

[3]https://appium.io/

[4]https://www.selenium.dev/

2.4.1 Robot Framework

Robot Framework is an open-source KDT framework that was originally developed by Nokia Networks and is adopted by 7% to 10% of the companies practicing acceptance testing (Katalon 2018; Mabl 2021). As a KDT solution, it relies on human-readable keywords that make the code easy to understand by non-technical people. One of the key selling point of Robot Framework is its high modularity. This modularity allows the core of the framework to be platform-agnostic by relying on a driver plugin architecture. Indeed, the application driver layer is created either by the Robot Framework core team and integrated in the language, developed by third parties and distributed as library plugins, or developed directly by the test team in which case the library code can be maintained alongside the test code. Thus, the concrete implementation of an action in a Robot Framework test script is hidden from the test, imposing a clear separation between the test implementation layer and the application driver layer.

Figure 1 shows an example of a Robot Framework test adapted from the official Robot Framework documentation. The *Test Case* `A user logs in with his username and password`, at line 6, is responsible for validating the correct behavior of the login form in an imaginary SUT by interacting with its user interface. As can be seen from the listing, most parts of this fully automated test are written in plain English. As defined by KDT, the test implementation layer is composed of *Keywords*. To further extend the reuse of these *Keywords*, they can accept *Arguments*. For instance, the *Keyword* `Open browser` (line 21) is called with two arguments, `${LOGIN URL}` and `${BROWSER}`. At lines 6–8 we see three `test steps` that represent the test acceptance criteria. As described in Section 2.4, in KDT, each step is implemented as a *Keyword*. In turn, these *Keywords* are defined in their respective definition blocks between lines 12 and 41. The steps composing the body of a *Keyword* are subsequent *Keyword* calls. At the end of the call tree, we find *Library Keywords* performing concrete actions such as interacting with the SUT (application driver layer) or marshalling the control flow of the test (*e.g.* branching). For example, at line 2, the script is using the web driver automation library *Selenium* to interact with the SUT. This library allows the test script to perform concrete actions such as navigating through the website using the `Go To` *Library Keywords* at line 26 and interact with elements present on a web page with library keywords such as `Input Text` (lines 31 and 35) or `Click Button` (line 38). Furthermore, *Library Keywords* can be used to perform assertions such as `Title Should Be` at line 41 which ensures that the title of a webpage is of a specific value.

Thus, Robot Framework makes it easy to follow the three-layer architecture described by (Humble and Farley 2010). In this model, the *Test Case* and its steps are the acceptance criteria, the intermediate *User Keywords* are the implementation layer and, finally, the *Library Keywords* interacting with the system under test represent the application driver layer.

3 Experimental Design

3.1 RQ1: Identification of SUIT Smells

To perform a study on the impact of smells in SUITs, we first need to build a catalog of test smells. To this end, we start our investigation by collecting SUIT smells presented in both academic and grey literature. We consider as academic literature (*a.k.a* white literature) papers that are published in peer reviewed conferences or journals. On the other hand,

```
*** Settings ***
Test Teardown     Close All Browsers
Library           Selenium2Library

*** Test Cases ***
A user logs in with his username and password
    Given browser is opened to login page
    When user "demo" logs in with password "mode"
    Then welcome page should be open

*** Keywords ***
Browser is opened to login page
    Open browser to login page

User "${username}" logs in with password "${password}"
    Input username    ${username}
    Input password    ${password}
    Submit credentials

Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Title Should Be    Login Page

Go To Login Page
    Go To    ${LOGIN URL}
    Login Page Should Be Open

Input Username
    [Arguments]    ${username}
    Input Text    username_id    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_id    ${password}

Submit Credentials
    Click Button    validate_id

Welcome Page Should Be Open
    Title Should Be    Welcome Page

*** Variables ***
    ${SERVER}         localhost:7272
    ${BROWSER}        Chrome
    ${LOGIN URL}      http://${SERVER}
```

Fig. 1 Example of robot framework test

we consider as grey literature white-papers, magazines, online blog-post, question-answers sites, survey results, and technical reports following the methodology presented in (Ricca and Stocco 2021). Indeed, the grey literature constitutes a rich source of documents where practitioners share their experiences, propose guidelines or even ask and answer questions. Thus, we conduct a multivocal literature review following the steps depicted by (Garousi and Küçük 2018). Figure 2 summarizes our adoption of these steps in the process of building a catalog of SUIT smells. It is worth noting that our literature review focuses on building a first catalog of SUIT smells and does not aim for completeness.

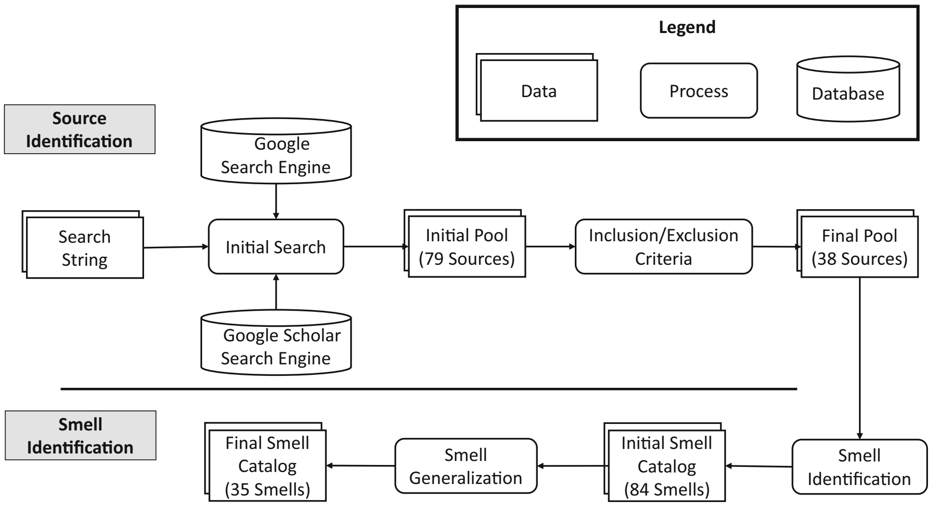


Fig. 2 Overview of the process to establish the SUIT smell catalog

Initial search To mine the academic sources we rely on the Google Scholar Search engine, whereas for the grey literature we rely on the Google Search engine. These two search engines are known to subsume other databases and repositories (Garousi and Küçük 2018) hence we limit our investigation to them. We compile a list of search terms such as "acceptance test", "test automation", "end-to-end test", "system test" and "behavior test" to define the type of tests that we are targeting and combine those strings with each of the following: "smell", "symptom", "anti pattern" and "bad practice". This allows us to generate the following search string for the engines: *software and ("acceptance test" OR "test automation" OR "end-to-end test" OR "system test" OR "functional test") AND ("smell" OR "symptom" OR "anti pattern" OR "bad practice").* To validate the search string we run it and make sure that relevant documents known *a priori* appear in the results (Kitchenham and Charters 2007; Ricca and Stocco 2021).

Note that when gathering sources, a source is considered as academic if it appears in the proceedings of a peer reviewed conference, it is an article from a peer reviewed journal, or it is a Ph.D. thesis. During this process, we do not take any decision about the rank or impact factor of the venues nor the type of paper (*e.g.* tool paper, main track). Finally, papers present in non-peer-reviewed repositories, (*e.g.* ArXiv) are excluded from the results.

Then, we proceed to a review of all the sources in order to identify test smells. However, because the search string returns about 1,370,000 results in the Google Search Engine and about 12,100 results in the Google Scholar Search Engine, we restrict our analysis to an initial pool composed of the first 200 results in each engine, which is the point at which no new results were found in both search engines. From those 400 entries, the two first authors read the title and abstract available to decide whether or not to further analyze the articles. When a disagreement was encountered, the source was added for further analysis. During the selection process, the decision was based on the following criteria:

- **Topic** We exclude all articles that do not present SUIT smells. Indeed, many articles were related to the topic of SUITs but did not introduce any smells for these tests.
- **Language** We exclude articles that are not written in English.

Table 1 The form leveraged to extract smell information from the analysed articles

Data	Explanation
Smell name(s)	We extract the names attached to the smell in the article.
Smell definition	We extract and summarize the definition given to the
Smell impact(s)	We infer the smell impact from the definition.
Example	When possible, we extract a concrete example of the presented smell.

- **Redundancy** Some articles are published on multiple platforms and hence, we had to ignore the duplicates.

This leads to an initial list composed of 40 sources in the grey literature and 31 sources in the academic literature for a total of 71 sources.

Inclusion/Exclusion criteria Given our study objectives, we decided to exclude articles that:

- **Do not focus on black-box tests** To be eligible for our study, a smell must address the code of a SUIT. However, the terms adopted in both industry and academia might not refer to how the test interacts with the application, but rather to what is the intent of the test (*e.g.* acceptance testing) or its scope (*e.g.* system test). This leads to a series of sources discussing tests that are not SUITs and are not compatible with them, typically, white-box tests. This criterion led to the exclusion of 22 articles.
- **Do not discuss codebase-related smells** Some sources address testing issues that are not related to the codebase such as organizational smells, which are outside the scope of this study. Indeed, some smells do not target the test code but testers' behavior *e.g.*, *Making Intermittent Bugs Low Priority* (StackExchange 2017) for which bugs making the functional test suite fail intermittently at low intervals are ignored. This criterion led to the exclusion of 11 articles.

Since our study considers both white and grey literature, we did not exclude articles based on their publisher or format. We admitted academic articles with different formats (*e.g.* long or short) as well as non-peer-reviewed articles. Note that whenever disagreement was encountered between the authors as whether or not to include a source, a discussion was triggered until a consensus was reached. Using the aforementioned exclusion criteria, we reduce our list to 32 sources from the grey literature and 6 sources in the academic literature for a total of 38 sources. The drastic drop in the academic literature is due to the large number of studies targeting unit tests instead of SUITs. The selection process is depicted in the summary sheet[5].

Smell Identification Relying on this final pool of sources, we proceed to the smell identification by extracting from each source the smells that it formally defines. We considered a source to define formally a smell if it specifically identifies a bad practice and defines its impact on the codebase. Table 1 summarizes the information extracted from each article.

We filter out all the smells that are not suitable for our analysis. Indeed, some smells are not applicable to the framework under study *e.g.*, *Dependence on Record and Playback*

[5] Available as https://docs.google.com/spreadsheets/d/e/2PACX-1vQ78jmOjU3qTOlGzwCSkidJOliPKNDQhmuOxSsfTaRqFVjmFP41JUbYQeupqU_lGCK6L4EpQ3FHNGhU/pubhtml a spreadsheet

(StackExchange 2017) in which tests are created using Record and Playback, a feature that is not supported by Robot Framework. Other smells are too technology specific *e.g.*, not using the page object pattern in Selenium tests (Advolodkin 2018). We select the smells for which the two authors who performed the literature review reach a consensus, classifying them as generic SUIT smell.

Finally, we manually label whether the impact is affecting the readability, execution, or maintenance. To do so, each author first relies on general understanding to label the smell, then, in a second stage, all authors discuss each of the categories where disagreement is observed until a consensus is reached for each smell and its labelling. The outcome of this process is an initial smell catalog of 84 unique SUIT smells for which we can derive a metric by analyzing the code representing the symptom observed.

Smell generalization Some SUIT smells gathered in the previous step exhibit large overlaps; thus, we proceed to a smell generalization step. For instance, the smell *Enter Enter Click Click* (Buwalda 2015) is grouped with *Comments and documentation instead of abstraction* (Klarck 2014), since both smells target the presence of low-level actions in the acceptance criteria but in the latter a specific emphasis is put on the documentation aspect. Hence, we consider those two smells to present similar symptoms and effects and we group them in one smell named *Lack of encapsulation* (Chen and Wang 2012; Klarck 2014; Buwalda 2015; England 2016; Renaudin 2016). Moreover, we observe that some test smells are subsuming others. In this case we only keep the subsuming smell. From this list, we filter out any test smell that would require the test to be executed in order to be observed. As a result, the outcome of this step is a list of 35 SUIT smells[6] that can be detected statically (Table 6).

Finally, conducting our study using Robot Framework, we exclude four test smells that cannot be observed in this specific language and its associated framework. Furthermore, we omit 15 test smells because, despite our best efforts, no automated metric avoiding false positive could be constructed by analyzing the test code. Hence, in this work, we present a list of 16 SUIT smells alongside with a metric to automatically detect them. A comprehensive description of each smell is presented in Section 4.2.

3.2 Dataset

To answer RQ2 and RQ3, we conduct a case study on 13 test suites written in Robot Framework. We establish two sets of projects: the first set is composed of 48 repositories from our partner site, BGL BNP Paribas and the second set of projects is composed of 12 open-source projects mined from Github. Table 2 summarizes the overall properties of these projects. In the following we describe the projects and present the collection process.

3.2.1 Industrial Project:

We leverage the codebase of our industrial partner, BGL BNP Paribas. The test suite is maintained by a dedicated quality assurance team which role is to ensure the compliance to the requirements of the products deployed to production. As such, the team tests desktop,

[6]The complete catalog is available at https://github.com/serval-uni-lu/suit-smell-catalog

web, and mobile applications that are depending on services developed following a service-oriented architecture (SOA). The goal of the team is to assess compliance to functional requirements and applications are tested in a black box fashion through their user interface.

Historically, the team relied on manual testing to perform its tasks. However, with the release cycles becoming shorter and the number of tests increasing, the execution burden on the team became unmaintainable. Thus, in the end of 2016, the team started migrating from manual testing to automated acceptance testing relying on Robot Framework. Robot Framework was chosen because of the ease it provides to develop tests targeting application written in different technologies requiring different mode of interactions.

Today, the test suite consists of 559 Robot Framework tests stored across 48 repositories on an on premise GitLab instance. While some repositories are defining *Test Cases*, others are used as resources of common *Keywords* where a series of generic *Keywords* specific to the BGL BNP Paribas architecture have been created to help with the development effort as well as to avoid code duplication. A typical example is the login to the ecosystem which is common to a large amount of services, and consequently can be mutualized. Hence, in this study, we merge all the repositories to count them as one project, which explains the larger number of *User Keywords* observed in Table 2 for the project *bgl*.

3.2.2 Open-Source Projects:

To collect the open-source test suites, we use the Github Search API to mine repositories containing suitable test suites, *i.e* Robot Framework test suites. Thus, using the API, we select all project that contains at least one Robot Framework Test Cases and that have at least 10 stars to avoid collecting toy projects. Additionally, we ignore any forked repository to avoid duplication. From the results of this first step we obtain 51 candidate projects. Then, we manually analyze each project by reading their readme section (and if needed looking at the test and the test harness) and filter out training projects (3 projects) and projects using

Table 2 Metrics for the 13 projects under study

Name	LoC	#Commits	#TestCases	#Keywords
bgl	57030	309	559	5000
apinf	1068	345	74	138
bikalims	4446	1279	76	172
collective-cover	1317	943	24	31
cumulus-ci	1599	1429	132	34
harbor	7270	3131	246	472
ifs	8466	5606	494	605
mms-alfresco	1797	504	156	8
mystamps	2361	353	212	111
ozone	2636	271	230	90
plone	1046	211	59	163
plone-intranet	2225	1759	213	134
rspamd	3397	685	429	151

LoC is the number of lines of Robot Framework Code, *#Commits* is the number of commits implicating Robot Framework code, and *#TestCases* and *#Keywords* are the number of test cases and user keywords respectively in the last commit of the projects

Robot Framework as a tool or extending it, and not using it as a test runner (12 projects). Following this approach, we gather 36 repositories from Github. We then gather metrics about the size of the projects and discard any projects with less than 10 test cases or less than 500 lines of code. Finally, to ensure that maintenance was performed on the test suite itself, we analyze the number of commit involving the Robot Framework test suite for each project, *i.e.* SUIT modifying commit, and discard any project with less than 100 SUIT modifying commits. This process yields a total of 12 repositories presented in Table 2.

The data collection process leads to a total of 2,884,383 SUITs analyzed across all the versions of all the projects where 2,742,271 originate from the open-source projects and 142,112 from the industrial projects gathered at BGL BNP Paribas.

These 13 projects (12 open source and 1 industrial) cover various technology stacks. For instance, BGL BNP Paribas developed internal services using Java Swing which rely on either the API exposed by Swing or rely on computer vision, through the use of the Sikuli[7] library. BGL BNP Paribas also uses Robot Framework to test its mobile client application relying on the Appium library. Finally, the majority of the test suites in both industrial and open-source projects from this dataset target web applications through the use of the Selenium library. The SUITs interact with such applications (1) through their webpages by interacting with the DOM, (2) through their APIs or (3) directly accessing the state of the database after an operation. Note that some operations generate emails or reports, thus, some tests rely on operating system interaction to check that an email was properly sent through a mail client or assess the existence and parses the content of a generated PDF documents.

To conclude this section, we observe that the dataset is composed of projects covering various domains, technology stacks, sizes, and modes of development. We believe that this diversity allows to avoid biases observed in one type of projects and improves the generalization of the observations.

3.3 RQ2: SUIT Smell Symptoms Distribution

For each of the SUIT smells that we gathered, we compute a metric to automatically measure a smelliness score for the affected test by relying on heuristics to construct these metrics. To this end, we apply the high-level investigation mechanism framed by (Marinescu 2004) called "detection strategy". As defined by the authors, a "detection strategy" is *a generic mechanism for analyzing a source code model using metrics*. The detection strategies are formulated in a series of steps:

1. Break-down informal rules into symptoms that can be captured by a single metric;
2. Select a proper set of metrics to quantify each of the symptoms;
3. Define thresholds that classify a test as smelly or not;
4. Use operators to correlate the symptoms leading to the final rule for detecting the smells.

Note that the step 3 of this approach describes the definition of a threshold. Even though different approaches have been proposed, *e.g.* using semantical properties and statistical distributions (Marinescu 2004) or using Bayesian belief networks (Khomh et al. 2009), defining a good threshold remains a hard and error prone task. Thus, more recently, researchers have been trying to avoid this limitation by using machine learning to directly learn what a smell looks like avoiding altogether the use a threshold (Arcelli et al. 2016). In

[7] http://sikulix.com/

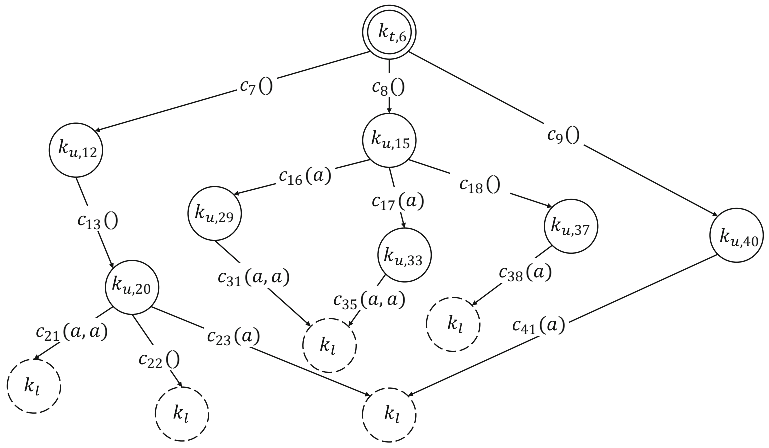


Fig. 3 Call graph of the script from Fig. 1 where k_t represent a test case node, k_u, a user keyword node, k_l, a library keyword node, and $c()$, a keyword call site with its optional arguments, a. The number suffix x in $c_x()$, $k_{t,x}$ and, $k_{u,x}$ indicates the corresponding line in the script

this work, in order to avoid any bias generated by a binary classification, we do not apply any empirical threshold (with the exception of one smell, *Long Test Steps*) but focus our analysis on the observation of the symptoms associated with SUIT smells. Hence, for each test we attribute a metric that represents the number of symptoms that are observed in a test. Furthermore, we propose a density metric, which normalizes the number of symptoms observed over the worst case scenario for a given test. Indeed, our goal is to analyze how the symptoms are treated by the maintainers of the test codebase and not to classify tests as smelly or not.

To extract code metrics, we rely on a graph representation of the test code. Indeed, a Robot Framework script can be represented using its call graph. A call graph $G(N, E)$ is a directed graph where every node $n \in N$, represents a function (or procedure) and an edge $e_{n_1 \rightarrow n_2} \in E$ exists if function n_1 invokes function n_2 (Hall and Kennedy 1992). Figure 3 shows the graph representation of the test presented in Fig. 1. The nodes of the tree represent functions from the Robot Framework language: (*Test Case*, *User Keyword*, and *Library Keyword*). The edges represent *Keyword* call sites. At the root of the graph, we find the entry point of test, the *Test Case* (double line). Each *test sep* composing the acceptance criteria is represented by an edge from the *Test Case* to a *Keyword*. The intermediate nodes (plain

Table 3 Categories assigned to *Library Keywords*

Name	Description
Interaction with the SUT	Performs an action on the SUT capable of modifying its state.
Assertion	Verifies that a predicate is true at a specific point.
Control Flow	Modifies the control flow of the test execution.
Getter	Retrieves a value available from the SUT or the environment.
Setter	Modifies a value available from the environment.
Logging	Dumps log during execution.
Synchronization	Synchronizes the execution of the test with the one of the application.

Table 4 Types assigned to *Library Keywords* argmuents

Name	Description
Object	Generic type that can be converted to all the other types. The arguments for *User Keywords* always assume that type.
String	Type accepting a chain of characters.
Duration	Type accepting a time interval representation (*e.g.* `1second`).
Boolean	Type accepting a boolean value.
Condition	Type accepting the definition of a condition that returns a boolean value (*e.g.* `$value == 1`).
Number	Type accepting a numerical value.
Keyword	Type accepting a keyword call and its arguments.
Locator	Type accepting the description of a GUI element.

line) represent *User Keywords*. *User Keywords* are calling subsequent *Keywords* which can be either other *User Keywords* or *Library Keywords*. Finally, the leaf nodes (dashed line) are the *Library Keywords*. Because they perform concrete actions that are hidden from the test script, they appear in the graph as leaf nodes.

When building the call graph of the tests, we annotate each *Library Keyword* with a category describing its intent (Table 3), *e.g.* `Input Text` modifies the state of the SUT, therefore, is from the category *Interaction with the SUT*. Furthermore, even though Robot Framework is a dynamic language, the type of *Library Keywords* is defined. Hence, in addition to their category, *Library Keyword* nodes are annotated with the type of the arguments they expect (Table 4), *e.g.* `Input Text` takes two arguments of types *Locator* and *String*.

Finally, when traversing the arguments during the call graph construction, arguments are annotated as *constant* if the value is a literal or *variable* if the argument is a variable. In the latter case, a link is created to the definition of the variable, allowing partial resolution (the necessary information might not be available statically) of the variable value set. The algorithm is implemented in a Maven package[8].

The generated call graphs are used to identify unique patterns that can be associated to smells. A pattern here is nothing else than a subtree that matches some predifined rule set extracted from the definition of the smells in Section 4.1. Note that the heuristics chosen do not capture all the possible symptoms for a given smell, but are straightforward and focus on manifestations that can be captured in the anoted call graph.

When computing the metrics, we introduce two types of metrics : a count metric S and a density metric D. The count metric S (# Symptoms in Table 5) counts the number of instances of a symptom observed in a test. On the other hand, the density metric D (% Symptoms in Table 5) provides an indication of the number of instances over a maximum value that could have appeared in the test. Thus, the density metrics vary between 0 and 1, 0 indicating the absence of any of the symptoms and 1 indicates that for each of the possible location for the symptom to be present, it appeared in the test. Note that we focus on metrics with a high precision at the detriment of the recall. Indeed, some of the metrics only cover one form of symptoms leading to a smell but lead to low rate of false positive. All the patterns are implemented in a Maven package [9].

[8]https://github.com/serval-uni-lu/ikora-core

[9]https://github.com/serval-uni-lu/ikora-smells

Table 5 Definitions of the metrics used in the study

Name	Notation	Description
# Symptoms	S_s	The number of symptoms for a smell s counted in a single test.
% Symptoms	D_s	The number of symptoms for a smell s counted in a single test divided by the number of location they could have appeared in that test.
# Smell-Removing Action	$\|SRA_s\|$	The total number of removal/changes in nodes exhibiting a symptom SRA_s for a smell s counted across all tests in all versions.
% Smell-Removing Action	–	The proportion of symptomatic tests undergoing at least one smell-removing action through their lifespan.

3.4 RQ3: SUIT Smell-Removing Actions

Refactoring can be defined as a technique to improve the design of a system and enable its evolution (Fowler et al. 1999) and is typically associated to smell removal. However, because our tooling cannot capture the intent of the change but merely its consequence, in this work we focus on a related phenomenon, smell-removing actions which are changes targetting specifically the removal of a smell but without guaranteeing a similar behavior. Relying on this definition, to answer our third research question, we conduct an analysis of the smell-removing actions during the maintenance of the test suites. To do so, we collect every pair of subsequent versions and identify smell-removing changes occurring in the test codebase. A key component in this type of analysis is to properly identify what constitutes a valid smell-removing action. Indeed, just observing a decrease in smell metrics does not imply that a smell has been specifically addressed. Other actions such as changing the scope of a test can, as a side effect, remove symptoms of a SUIT smells without specifically targeting it. Thus, following the line of work present in the literature to identify refactoring activity (Tsantalis et al. 2013; Silva and Valente 2017), we address this limitation by using heuristics based on static test code analysis. More specifically, we use the fine-grained change algorithm presented in (Rwemalika et al. 2019a) to extract instances of smell-removing patterns. These patterns are derived from the definition of the symptoms present in the literature.

More specifically, let's consider a graph G_1 and its subsequent version G_2. The set of actions $A_{G_1 \rightarrow G_2}$ is composed of all the edit required to go from G_1 to G_2. To extract $A_{G_1 \rightarrow G_2}$, the fine-grained change algorithm works in two phases:

1. Finding a match between the nodes from G_1, N_{G_1}, and the nodes from G_2, N_{G_2} to come up with a mapping for each node $n_{G_1} \rightarrow n_{G_2}$.
2. Finging a minimum edit script that transforms G_1 into G_2 given the computed mapping.

As a result, the algorithm produces an edit script composed of the actions to transforms G_1 into G_2, $A_{G_1 \rightarrow G_2}$. Each action, $a_{G_1 \rightarrow G_2} \in A_{G_1 \rightarrow G_2}$ is one of: *Insert* ($\emptyset \rightarrow n_{G_2}$), *Delete* ($n_{G_1} \rightarrow \emptyset$), or *Update* ($n_{G_1} \rightarrow n_{G_2} | n_{G_1} \neq n_{G_2}$).

Concurrently, for each SUIT smell s we store the set of nodes from G_1, N_{G_1}, that are exhibiting a symptom of the SUIT smell and the smell-removing actions $A_{s,n_{G_1}}$ that when performed on one or more elements of $n_{G_1} \in N_{G_1}$ removes the symptom. We obtain a set of tuples $< s, n_{G_1}, a_{s,n_{G_1}} >$ for each pair of subsequent versions which describes the fine-grained smell-removing actions that were performed on one or more nodes to fix a specific SUIT smell symptom, *i.e.* the smell-removing actions.

To generate this list, first we compile the set of potential actions $A_{s,n_{G_1}}$ that can fix a symptom of a SUIT smell s given a set of nodes N_{G_1} (the complete list is presented in Section 4.1). Then, for each action $a_{G_1 \to G_2}$ provided by the fine-grained change extraction algorithm, we check if $a_{G_1 \to G_2}$ matches any pattern from $A_{s,n_{G_2}}$. If it is the case, then the fine-grained change $a_{G_1 \to G_2}$ is considered to be a smell-removing action and added to the list of tuples $< s, N, a_{G_1 \to G_2} >$.

To assess to which extent practitioners refactor SUIT smell symptoms, we count the number of smell-removing actions across all the versions, $|a_{s,N}|$ (# Smell-Removing Action ($|SRA_s|$) in Table 5). Thus, the higher this number, the more often developers remove the symptoms. However, because some symptoms might appear much more often than others, this number might be skewed towards symptoms that are frequent. In the result section, we avoid this limitation by proposing a generalization metric computing the proportion of symptomatic tests undergoing at least one smell-removing action through their lifespan (% Smell-Removing Action in Table 5). The results of this analysis are presented in Section 4.5.

3.5 Interviews

Finally, to validate the results obtained by the literature review as well as our tooling, we conduct interviews with Robot Framework practitioners. The pool of participants is composed of a practitioner from the open-source projects we collected, an automation engineer from BGL BNP Paribas, as well as two other test automation engineers contacted through the official *Slack* channel of the Robot Framework community. All participants actively work on Robot Framework Test Suites and have at least five years of experience with the technology. Thus, this leads to a pool of four experienced test automation engineers.

The interviews take the form of a semi-structured questionary where for each of the 16 smells for which we have a metric, we present their definition along with their impacts, and an example displaying a symptomatic instance captured by our tooling. We ask for each smell four questions:

1. According to your experience, do you consider this pattern a smell? (Yes/No)
2. How would you prioritize the refactoring of this smell? (1 (lowest) to 5 (highest))
3. How often have you seen this smell? (1 (never) to 5 (all the time))
4. Do you have any additional comments? (free text)

With question 1, we validate the fact that the test is indeed consider as a smell by practitioners and that the instance captured by the tooling reflects their perception. Questions 2 and 3 characterized the perceived harmfulness and frequency of each smell. And finally, question 4 offers an opportunity for the practitioners to share some experiences or thoughts about the smells in an exploratory fashion.

In addition to the questions about the smells themselves, we show the practitioners the results of our study regarding the diffusion of smells and the smell-removing actions. Here, for each of the two experiments, we present the results in the form of a graph with a legend and a short description and ask the single open-ended question: Do you have any comments about your experience with smells? The goal of this question is to offer perspective from our measures from practitioners in an exploratory fashion.

The results as well as the quotes obtained from the interviews are presented alongside the results of the other research questions.

Table 6 Catalog of SUIT Smells and their description

Name	Description
Army of Clones	Different tests perform and implement similar actions, leading to duplicated pieces of test code.
Complicated Setup Scenarios	The test performs actions to set the system under test in a valid state for the focus of the test. These actions should be performed in the setup.
Conditional Assertions	The test verifies different properties depending on the environment when the environment state may change from one execution to the next.
Conspiracy Of Silence	An assertion in a test is failing with no indication as to why.
Data Creep	Test data is stored in ways that are hard to access.
Dependencies between tests	Tests have dependencies between them, be it within the test themselves or their fixtures.
Directly Executing UI Scripts	Since technologies such as web technologies allow to directly exercise the system under test using scripts such as Javascript, it can be tempting to completely bypass the user interface and just run such scripts.
Duplicate Check	Different test in the test suite perform the same exact check against the system under test.
Eager Test	The test exercises several non-related features of the system under test.
Hardcoded Environment	The test contains hardcoded references to the environment when the same requirement must be run against different test environments.
Hiding Test Data	The data are not directly visible and understandable in the test but are hidden in the fixture code.
Implementation Dependent	The test is dependent on the implementation details and data structures present in the system under test.
Inconsistent Hierarchy	The test component hierarchy (class hierarchy, keyword hierarchy, macro hierarchy, etc.) is inconsistent with the structure of the GUI.
Inconsistent Wording	The domain concept are not used in a consistent way across tests.
Lack of Encapsulation	The implementation details of a test are not properly hidden in the implementation layer and start appearing in its acceptance criteria.
Lack Of Early Feedback	Regression suites grow too big to be accommodated in the automation pipeline where execution would last for hours.
Lifeless	The test is missing the lifecycle steps of the business objects (e.g. CRUD) and such basic operations are scattered throughout the test suite.
Long Test Steps	One or many test steps are very long, performing a lot of actions on the system under test.
Middle Man	A test component (keyword, macro, function) delegates all its tasks to another test component.
Missing Assertion	The test lacks any explicit assertions.
Narcissistic	The test uses the first person pronoun I to refer to its actors and does not uniquely qualify those actors.
Noisy Logging	The test logs the state of the fixtures.
Obscure Test	The test behavior is difficult to understand because the test does not clearly state what it is verifying. Typical symptoms are hardcoded values, high cyclomatic complexity and/or function or procedure calls with high number of parameters.

Table 6 (continued)

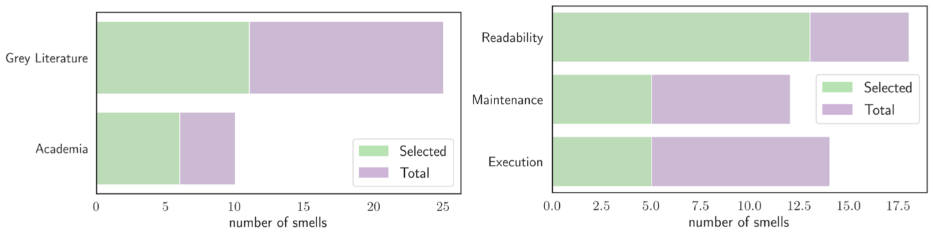
Name	Description
On the Fly	The test calculates an expected results during its execution instead of relying on pre-computed values.
Over-Checking	The test performs some assertions that are not relevant for its scope.
Pointless Scenario Descriptions	The description of the step does not give any extra information about its intent while remaining long.
Sensitive Locators	The test uses element identification selectors that have long chains to identify an element in the user interface. e.g. complex x-pass or CSS selector for web application.
Sneaky Checking	The test hides its assertions in actions that are at the wrong level of details.
Stinky Synchronization	The test fails to use proper synchronization points with the system under test.
Test Data Loss	Fixtures used by the test are always the same and require non trivial computation. When not saved, these values need to be computed at each test execution.
Testing Data Not Code	The data the test rely on are dependent on the environment they lie in and need to be generated for each environment at every release.
Unnecessary Navigation	The test performs actions that are not directly connected to the focus of the test.
Unrealistic Data	The test is written using trivial data that are not representative of the production data.
Unsecured Test Data	The test is using real user data and reads private information which might be violating certain laws.
Unsuitable Naming	The name of the test does not capture the essence of what the test is doing.

Smell in bold are associated with a metric which is discussed in Section 4.2

4 Results

4.1 RQ1: SUIT Smells Catalog

Table 6 lists all the 35 smells identified in both grey and academic literature along with a short description for each smell. The entries in bold refer to smells for which we defined a metric. The origin of the smell is depicted in Fig. 4a which shows the number of test smells that were found in academic literature and in grey literature. We show both the 35 initial SUIT smells extracted from the literature and the subset of 16 SUIT smells for which we could extract a metric in the test code. Note that a SUIT smell is considered as covered by academia if at least one of the smells grouped in the generalization step (Section 3.1) is covered by the academic literature. The figure shows that while there exists an interest from practitioners with 35 unique smells discussed, the number of smells discussed in peer reviewed literature remains rather limited with only 10 smells identified. We also see that smells identified from the literature could generate a metric in six out of ten cases. The ones for which no metric could be derived are *Inconsistent Wording* (Hauptmann et al. 2013), *Unsuitable Naming* (Chen and Wang 2012), *Inconsistent Hierarchy* (Hauptmann et al. 2013) and *Data Creep* (Alegroth et al. 2016). While (Hauptmann et al. 2013) propose metrics for



(a) Source (b) Issues

Fig. 4 Properties of the identified SUIT smells. Figure 4a displays the number of smells found in the academic and in the grey literatures. Figure 4b displays the number of smells exhibiting a specific issue. Note that a smell can lead to several issues, hence the total is greater than the total number of SUIT smells

the smells they introduce, during our evaluation we observed a high rate of false positive for *Inconsistent Wording* and *Inconsistent Hierarchy* thus we excluded them for the final list. Chen and Wang (Chen and Wang 2012) and (Alegroth et al. 2016) on the other hand do not propose a metric to automatically measure the smells they present. To conclude this analysis of Fig. 4a, we see that there exists a gap between the grey literature and the academic literature. We believe that more work need to be conducted to better understand and automatically detect and refactor SUIT smells in the academia. The present work is an attempt to fill this gap with the introduction of 25 new smells not previously studied in academia.

Figure 4b presents the effects associated to each SUIT smell. Note that a smell can lead to issues from different categories. Thus, the total number of issues is greater than the number of smells. From the figure, we observe that although readability issues appear the most often with 18 instances, maintenance issues with 12 instances, and execution issues with 14 instances do not fall far behind. Furthermore, to our surprise, SUIT smells affecting readability where the ones for which the most metrics could be computed. One reason explaining this phenomenon is that in the case of execution issues, information about the SUT and its execution are required. The same observation can be made for maintenance where the danger is coming from a divergence of the test from the SUT.

There is a gap between the academic and grey literatures in terms of SUIT smells, with 10 and 35 covered smells respectively. SUIT smells are reported to affect readability, maintenance, and test execution.

Before moving on to the detailed description of each smell, note that for each SUIT smell, we define the sources, and if there exists any, the sources describing the associated unit test smell (Tables 7 and 8). The sources populating the *SUIT* columns are the ones extracted from our MLR, while the ones in the *Unit Test* columns are extracted fom the study by (Garousi and Kůčůk 2018), where each source was reviewed to ensure that it is addressing unit tests specifically. This analysis offers an overview on the differences between the smell addressed in the two types of tests. Note that because the scope of this work targets SUIT smells, only those are reported in this table. One class of smells not appearing in SUITs are the smells appearing in association with production code (Garousi and Kůčůk 2018). For instance, the smell *For Test Only* occurs when a production class has methods only used by test methods (Breugelmans and Rompaey 2008; Garousi and Kůčůk 2018). However, SUIT

Table 7 Sources of SUIT smells and their potential unit test counterpart for which an automatic metric could not be derived

Name	SUIT	Unit Test
Complicated Setup Scenarios	(Scott 2015)	(Greiler et al. 2013)
Conspiracy Of Silence	(Gawinecki 2016; Sheth 2020)	(van Deursen et al. 2001)
Data Creep	(Alegroth et al. 2016; Siminiuc 2019; Shay 2019)	
Dependencies between tests	(Klarck 2014; Advolodkin 2018; Cripsin 2018; Bushnev 2019; Goldberg 2019)	(Zhang et al. 2014)
Directly Executing UI Scripts	(Scott 2015)	
Duplicate Check	(Buwalda 2019)	
Eager Test	(England 2016; Renaudin 2016; Cripsin 2018; Sciamanna 2019; Temov 2020)	(van Deursen et al. 2001; Van et al. 2006; Bavota et al. 2012; Basit et al. 2013; Greiler et al. 2013; Bavota et al. 2015; Delin and Foegen 2016; Tufano et al. 2016; Spadini et al. 2018; Grano et al. 2019)
Implementation Dependent	(Jain 2007; Kapelonis 2018; Goldberg 2019)	
Inconsistent Hierarchy	(Clayton 2014; Gawinecki 2016; Buwalda 2019)	(Reichhart et al. 2007)
Inconsistent Wording	(Hauptmann et al. 2013)	
Lack Of Early Feedback	(Dharmender 2017)	
Lifeless	(Buwalda 2015; Renaudin 2016; Buwalda 2019)	
Pointless Scenario Descriptions	(England 2016)	(Reichhart et al. 2007)
Test Data Loss	(Siminiuc 2019)	
Testing Data Not Code	(Dharmender 2017)	
Unnecessary Navigation	(Archer 2010)	
Unrealistic Data	(Goldberg 2019)	
Unsecured Test Data	(Morlion 2019)	
Unsuitable Naming	(Chen and Wang 2012; Goldberg 2019; Shay 2019; Sheth 2020)	(Reichhart et al. 2007; Basit et al. 2013)

smells functioning in a black box fashion, by design do not suffer from this category of smells.

Looking at Tables 7 and 8, we observe an important overlap in the type of smells with 14 out of 35 smells appearing both in unit test and SUITs. To classify a unit test smell and a SUIT smell as overlapping, we used the methodology described in Section 3.1 to extract and identify the smell from the unit smell source and to map them to the corresponding SUIT category. Thus, for the 21 remaining smells, we did not find a mapping from unit tests. Looking at the smells only appearing in SUITs, we see that they are concerned with issues in the test data management or they tackle the workflow of the SUIT. Since unit test typically do not deal with any external data and are meant to be short and focused, these smell do not impact them. Finally, smell specifically targeting interaction with the user interface, such as *Sensitive Locator* or *Directly Executing UI Scripts* fall out of the scope of unit test smells.

Table 8 Sources of SUIT smells and their potential unit test counterpart for which an automatic metric could be derived (Section 4.2)

Name	SUIT	Unit Test
Army of Clones	(Chen and Wang 2012; Hauptmann et al. 2013; Hauptmann et al. 2015; Knight 2019)	(van Deursen et al. 2001; Lanubile and Mallardo 2007; Breugelmans and Rompaey 2008; Bavota et al. 2012; Basit et al. 2013; Athanasiou et al. 2014; Bavota et al. 2015; Grano et al. 2019)
Conditional Assertions	(Gawinecki 2016)	
Hardcoded Environment	(Gawinecki 2016; Sheth 2020)	
Hiding Test Data	(Jain 2007)	
Lack of Encapsulation	(Chen and Wang 2012; Evangelisti 2012; Klarck 2014; Buwalda 2015; England 2016; Renaudin 2016; Knight 2017a; Goldberg 2019; Knight 2019; Shay 2019)	
Long Test Steps	(Chen and Wang 2012; Hauptmann et al. 2013; Buwalda 2019)	(Reichhart et al. 2007; Breugelmans and Rompaey 2008; Spadini et al. 2020)
Middle Man	(Chen and Wang 2012)	
Missing Assertion	(Klarck 2014)	(Reichhart et al. 2007; Breugelmans and Rompaey 2008; Athanasiou et al. 2014)
Narcissistic	(England 2016; Knight 2017b)	
Noisy Logging	(Jain 2007)	(Reichhart et al. 2007)
Obscure Test	(Hauptmann et al. 2013; Gawinecki 2016; Siminiuc 2019)	(Lanubile and Mallardo 2007; Breugelmans and Rompaey 2008; Athanasiou et al. 2014; Delin and Foegen 2016)
On the Fly	(Archer 2010)	
Over-Checking	(Buwalda 2015; Renaudin 2016)	(van Deursen et al. 2001; Lanubile and Mallardo 2007; Breugelmans and Rompaey 2008; Bavota et al. 2012; 2015; Tufano et al. 2016; Spadini et al. 2018; Grano et al. 2019)
Sensitive Locators	(Scott 2015; Knight 2019; Battat 2020; Sheth 2020)	
Sneaky Checking	(Kirkbride 2014; Buwalda 2015; Renaudin 2016)	
Stinky Synchronization	(Gawinecki 2016; Renaudin 2016; Bushnev 2019; Knight 2019; Sheth 2020)	(Spadini et al. 2020)

There exists an overlap between Unit Test Smells and SUIT smells with 14 smells affecting both unit tests and SUITs. However, 21 SUIT smells are unique to SUITs, and are typically concerned with problems regarding data management, test workflow, and interaction with the UI.

4.2 SUIT Smell Symptom Metrics

Following the notation introduced in Section 3.3, we formally describe each smell for which a metric can be extracted with respect to its symptoms, its impact, and the metrics chosen to measure the prevalence of the symptoms following the protocol presented in Section 3.3. For each smell, the metric is define as the pattern that best represent the definition of the symptom and the smell removal action is the minimal set of action that would result in the removal of the symptomatic pattern. Finally, associated with each metric, we provide the smell dispersion which is the level of localization at which a symptom appears in the test code. It can be:

- *Line*: It affects a single line of test code;
- *Keyword*: It is present in a single Keyword;
- *Test*: It can spread across different keyword from within a single test;
- *Suite*: It is spread across the entire test suite.

4.2.1 Army of Clones (AoC)

Description Different tests perform and implement similar actions, leading to duplicated pieces of test code.

Impact on readability Test sequences which are similar but not identical are not easy to distinguish. It is not easy to grasp the intention of the test in comparison with its clone.

Impact on maintenance The effort to maintain duplicated parts of tests increases. Furthermore, it is difficult to determine where maintenance has to be performed.

Dispersion Suite.

Detection method Code duplication can be observed at different levels. Here, for the body of a *User Keyword*, we detect if there exists a clone of type 1 (code duplication at the token level) or type 2 (code duplication at syntax level allowing for minor syntactic changes such as variables name) in the test suite. Thus, we express the count metric, $S_{AoC}(t)$, as the number of calls to *User Keywords* that have a clone and the density metric, $D_{AoC}(t)$, as the number of calls to *User Keywords* that have a clone over the total number of unique *User Keywords* called by the test. More formally:

$$S_{AoC}(t) = |K_t \cap K_{clone}|$$

$$D_{AoC}(t) = \frac{|K_t \cap K_{clone}|}{|K_t|}$$

where K_t is the set of unique *User Keywords* called by test t and K_{clone} is the set of unique *User Keywords* that have at least one clone in the test suite.

Smell-removing actions The symptom is considered refactored if the *User Keyword* that is called by a test and have at least one clone in the test suite, $k_{t,clone}$, is removed. Thus, we propose the smell-removing pattern, $SRA_{AoC}(k)$, as follow:

$$SRA_{AoC}(k) = k_{t,clone} \xrightarrow{action} \emptyset$$

4.2.2 Conditional Assertions (CA)

Description The test verifies different properties depending on the environment when the environment state may change from one execution to the next.

Impact on readability With more complex logic in the assertions, it becomes harder to capture their meaning.

Impact on execution More complex code might introduce bugs in the test code.

Dispersion Line or Keyword.

Detection method We consider assertions nodes, $C_{assertion}$, to be symptomatic if they have a parent node which is a conditional node and have no sibling nodes in the call graph, $C_{condition}$. Thus, we express the count metric, $S_{CA}(t)$, as the number of conditional assertion calls and the density metric, $D_{CA}(t)$, as the number of conditional assertion calls over the total number of assertion calls.

$$S_{CA}(t) = |C_t \cap C_{assertion} \cap C_{condition}|$$

$$D_{CA}(t) = \frac{|C_t \cap C_{assertion} \cap C_{condition}|}{|C_t \cap C_{assertion}|}$$

where $|C_t \cap C_{assertion}|$ is the size of the set of calls to *Library Keyword* annotated as "assertion" for a test t and $|C_t \cap C_{assertion} \cap C_{condition}|$ is the size of the set of calls to *Library Keyword* annotated as "assertion" for which the caller is a conditional node that has only one child (logging nodes excluded).

Smell-removing action The symptom is considered refactored if the conditional assertion node is removed from the call graph. Thus, we accept the following smell-removing pattern, $SRA_{CA}(c)$, as removing a symptom in a node c:

$$SRA_{CA}(c) = c_{t,assertion,condition} \xrightarrow{action} c_{t,assertion}$$

The assertion $c_{t,assertion}$ replaces its former parent node $c_{t,assertion,condition}$. Note that removing a parent of $c_{t,assertion,condition}$ or adding a sibling to the child assertion node are not considered as fixing the symptom.

4.2.3 Hardcoded Environment (HE)

Description The test contains hardcoded references to the environment when the same requirement must be run against different test environments instead of having an environment-agnostic test.

Impact on maintenance Updating the configuration requires modifying multiple locations in different tests.

Dispersion Line.

Detection method The metric we propose covers the case of multi-browser configuration. Here, when a browser is loaded, the metric ensure that the web-driver is not instantiated with a hardcoded configuration for the browser. Thus, we express the count metric, $S_{HE}(t$, as the number of configuration arguments that are hardcoded and the density metric, $N_{HE}(t)$, as he number of configuration arguments that are hardcoded over the total number of configuration arguments. More formally:

$$S_{HE}(t) = |A_t \cap A_{config} \cap A_{hardcoded}|$$

$$D_{HE}(t) = \frac{|A_t \cap A_{config} \cap A_{hardcoded}|}{|A_t \cap A_{config}|}$$

where $|A_t \cap A_{config}|$ is the size of the set of arguments in calls to *Library Keywords* annotated as "configuration" in a test t and $|A_t \cap A_{config} \cap A_{hardcoded}|$ is the size of the set of arguments in calls to *Library Keywords* annotated as "configuration" for which the value is hardcoded.

Smell-removing action The symptom is considered refactored if the hardcoded argument in a call to a *Library Keyword* annotated as "configuration", $a_{t,config,hardcoded}$, is replaced with a variable, $a_{t,config,variable}$. Thus, we propose the following smell-removing pattern, $SRA_{HE}(a)$, for an argument a:

$$SRA_{HE}(a) = a_{t,config,hardcoded} \xrightarrow{action} a_{t,config,variable}$$

4.2.4 Hidden Test Data (HTD)

Description The data are not directly visible and understandable in the test but are hidden in the fixture code.

Impact on readability The data is completely obscure to the future reader making the intent of the test difficult to understand.

Dispersion Line.

Detection method In this work, we associate the fixture code to the setup of a test. We consider data access as reading input from external resources through a *Library Keyword* annotated as "getter". Thus, we express the count metric, $S_{HTD}(t)$, as the number of calls to getter in the setup of a test and $D_{HTD}(t)$ as the number of calls to getter in the setup of a test over the total number of calls in the setup of that test. More formally:

$$S_{HTD}(t) = |C_t \cap C_{setup} \cap C_{getter}|$$

$$D_{HTD}(t) = \frac{|C_t \cap C_{setup} \cap C_{getter}|}{|C_t \cap C_{setup}|}$$

where $|C_t \cap C_{setup}|$ is the size of the set of calls to *Library Keywords* in the setup of test t and $|C_t \cap C_{setup} \cap C_{getter}|$ is the size of the set of calls to *Library Keywords* annotated as "getter" in the setup of test t.

Smell-removing action The symptom is considered refactored if the call to the *Library Keywords* annotated as "getter" in the setup of test t, $c_{t,setup,getter}$, is removed. Thus, we propose the following smell-removing pattern, $SRA_{HTD}(c)$ performed on a *Library Keyword* call c as follow:

$$SRA_{HTD}(c) = c_{t,setup,getter} \xrightarrow{action} \emptyset$$

Note that removing a parent node of $c_{t,setup,getter}$ is not considered as a fix.

4.2.5 Lack of Encapsulation (LoE)

Description The implementation details of a test are not properly hidden in the implementation layer and start appearing in its acceptance criteria.

Impact on readability The acceptance criteria is meant to convey intention over implementation. Focusing on implementation in the acceptance criteria makes the intent harder to grasp.

Dispersion Line.

Detection method Typically the acceptance criteria makes call to the implementation layer which subsequently relies on the application driver layer. The metric detects the direct calls from the acceptance criteria (test steps) to *Library Keywords*. Thus, we express the count metric, $S_{LoE}(t)$, as the number of direct calls to a driver from the acceptance criteria of a test and the density metric, $D_{LoE}(t)$, as the number the number of direct calls to a driver from the acceptance criteria of a test over the total number of steps of the acceptance criteria of a test. More formally:

$$S_{LoE}(t) = |C_t \cap C_{step} \cap C_{driver}|$$

$$D_{LoE}(t) = \frac{|C_t \cap C_{step} \cap C_{driver}|}{|C_t \cap C_{step}|}$$

where $|C_t \cap C_{step}|$ is the size of the set of steps in the acceptance criteria of the test t and $|C_t \cap C_{step} \cap C_{driver}|$ is the size of the set of steps in the acceptance criteria of the test t directly calling the application driver, *i.e.* a (*Library Keyword*).

Smell-removing action The symptom is considered refactored if the direct call to a *Library Keyword* is removed from the acceptance criteria. Thus, we propose the smell-removing patterns, $SRA_{LoE,1}(c)$ and $SRA_{LoE,2}(c)$, performed on a *Library Keyword* call c as follow:

$$SRA_{LoE,1}(c) = c_{t,step,driver} \xrightarrow{action_1} c_{t,step,\neg driver}$$

$$SRA_{LoE,2}(c) = c_{t,step,driver} \xrightarrow{action_2} \emptyset$$

In the first equation, $SRA_{LoE,1}(c)$, the direct call to a *Library Keyword* in the acceptance criteria is replaced by a call to a *User Keyword* where as in the second equation, $SRA_{LoE,2}(c)$, the call is removed.

4.2.6 Long Test Steps (LTS)

Description One or many test steps are very long, performing a lot of actions on the system under test.

Impact on readability The intention of the step is difficult to grasp.

Impact on execution With each action on the system under test, there is a chance of something going wrong. The higher this number, the more fragile the test becomes.

Dispersion Test.

Detection method For a test step, $K_{t,step}$, of a test t, the metric counts the number of *Library Keyword* annotated as "action" (triggering an event on the SUT) called directly or indirectly by $K_{t,step}$. If the value is greater than a threshold L, then $K_{t,step}$ is considered symptomatic. Thus, we express the count metric, $S_{LTS}(t)$, as the number of steps that are performing a number of actions greater than a threshold and the density metric, $D_{LTS}(t)$, s the number of steps that are performing a number of actions greater than a threshold over the total number of steps. More formally:

$$S_{LTS}(t) = |C_t \cap C_{step \geq L}|$$

$$D_{LTS}(t) = \frac{|C_t \cap C_{step \geq L}|}{|C_t \cap C_{step}|}$$

where $|C_t \cap C_{step}|$ is the number of steps in a test t and $|C_t \cap C_{step \geq L}|$ is the number of steps calling more than L *Library Keyword* annotated as "action".

Because the parameter L needs to be set empirically, we compute a deviation threshold based on the distribution of our dataset. Using the analysis of the evolution of the quantiles, we compute at which point the values start to rapidly deviates by computing the knee curve of the quantiles distribution function proposed by (Satopaa et al. 2011). Following this approach we find the knee point at 13 actions on the SUT for a step (quantile = 0.986, see Fig. 5). Therefore, we set $L = 13$ and consider any step performing a sequence of actions on the SUT greater than 13 to be too long.

Smell-removing action The symptom is considered refactored if the number of actions performed on the SUT by a long step sees its value pass under the threshold L. We do not specify how the calls on the SUT have to be transformed, as long as the test step call is left unchanged. Thus, we propose the smell-removing pattern, $SRA_{LTS}(c)$, where c is a long test step:

$$SRA_{LTS}(c) = c_{t,step \geq L} \xrightarrow{action} c_{t,step < L}$$

where $c_{t,step \geq L}$ is a step yielding at least L actions on the SUT while $c_{t,step<L}$ is a step yielding less than L actions on the SUT.

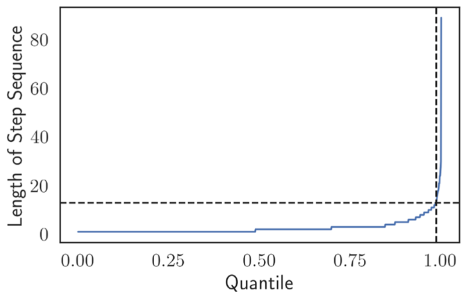


Fig. 5 The blue curve represent the evolution of the length of the sequences of the step as the quantiles increase. The intersection of the black doted lines displays the knee point (0.986, 13) at which the values sequence lengths values start to increase dramatically

4.2.7 Middle Man (MM)

Description A test component (keyword, macro, function) delegates all its tasks to another test component.

Impact on readability The levels of indirection make the test harder to follow by future readers.

Dispersion Keyword.

Detection method The principle of delegating is doing nothing and simply calling another function. Thus, we express the count metric, $S_{MM}(t)$, as the number of *User Keywords* called from a test that are composed of a single call to another *Keyword* and the density metric, $D_{MM}(t)$, as the he number of *User Keywords* called from a test that are composed of a single call to another *User Keyword* and the density metric over the total number of *Keyword* called by the test. More formally:

$$S_{MM}(t) = |K_t \cap K_{delegate}|$$

$$D_{MM}(t) = \frac{|K_t \cap K_{delegate}|}{|K_t|}$$

where $|K_t|$ is the number of *User Keywords* in a test t and $|K_t \cap K_{delegate}|$ is the number of *User Keywords* performing a single call to one other *User Keyword* (we ignore logging action) without performing any subsequent action on their own, *i.e. Delegate Keyword.*

Smell-removing actions The symptom is considered refactored if the *Delegate Keyword* call is replaced with another call which is not simply delegating its actions. Thus, we propose the smell-removing pattern, $SRA_{MM}(c)$ where c is a call to a *Delegate Keyword* as:

$$SRA_{MM}(c) = c_{K_{t,delegate}} \xrightarrow{action} c_{K_{t,\neg delegate}}$$

where $c_{K_{t,delegate}}$ is a call to a *Delegate Keyword* and $c_{K_{t,\neg delegate}}$ is a call to any *Keyword* not performing delegation.

4.2.8 Missing Assertion (MA)

Description The test lacks any explicit assertions.

Impact on readability Future readers are left in the potentially frustrating position of puzzling over the intention of the test.

Dispersion Test.

Detection method The metric detects the absence of call to *Library Keyword* annotated as "assertion" within the test. Because the symptom can at most appear once in the test, both the count metric, $S_{MA}(t)$ and the density metric $D_{MA}(t)$ are express as the absence of an assertion call in the test. More formally:

$$S_{MA}(t) = D_{MA}(t) = \begin{cases} 1, & \text{if } C_t \cap C_{assert} = \emptyset \\ 0, & \text{otherwise} \end{cases}$$

Smell-removing actions The symptom is considered refactored if *Library Keyword* annotated as "assertion" is introduced in test that is missing any assertion. Thus, we propose the smell-removing pattern, $SRA_{MA}(t)$, where t is the test lacking any assertion as:

$$SRA_{MA}(t) = \emptyset \xrightarrow{action} c_{t,assertion}$$

where $c_{t,assertion}$ is a call to a *Library Keyword* annotated as "assertion" in a test t.

4.2.9 Narcissistic (N)

Description The test uses the first person pronoun "I" to refer to its actors and does not uniquely qualify those actors.

Impact on readability The test is harder to read because it is not clear who "I" is and what are the roles that "I" has.

Dispersion Line.

Detection method Using text tagging, we identify calls from the acceptance criteria, *i.e.* "test steps", using a personal pronouns as the subject in there name as symptomatic. Furthermore, the implementation used in our experiments supports the different languages present in our dataset (namely, French and English). Thus, we express the count metric, $S_N(t)$, as the number of steps in the acceptance criteria of a test that are using a personal pronoun and the density metric, $D_N(t)$, as the number of steps in the acceptance criteria of a test that are using a personal pronoun over the total number of steps in the acceptance criteria of the test:

$$S_N(t) = |C_t \cap C_{step} \cap C_I|$$
$$D_N(t) = \frac{|C_t \cap C_{step} \cap C_I|}{|C_t \cap C_{step}|}$$

where $|C_t \cap C_{step}|$ is the number of "test steps" for a test t and $|C_t \cap C_{step} \cap C_I|$ is the number of "test steps" using a personal pronouns as the subject in there name.

Smell-removing actions The symptom is considered refactored if the name of a symptomatic "test steps" is changed so that it does not contain a personal pronoun anymore. Thus, we propose the smell-removing pattern, $SRA_N(c)$, where c is a step of the acceptance criteria of a test t:

$$SRA_N(c) = c_{t,step,I} \xrightarrow{action} c_{t,step,\neg I}$$

where $c_{t,step,I}$ is a *User Keyword* called by a "test steps" using the personal pronoun "I" as the subject and $c_{t,step,\neg I}$ is the *User Keyword* with its new name not using the personal pronoun "I" as the subject. Therefore, a fix is detected only when the name of a *User Keyword* called by a "test steps" is changed.

4.2.10 Noisy Logging (NL)

Description The test logs the state of the fixtures.

Impact on execution There is too much noise in the output from the tests, making its analysis more cumbersome.

Dispersion Test.

Detection method In this work, we associate the fixture code to the setup of a test. Thus, we express the count metric, $S_{NL}(t)$, as the number of calls to *Library Keywords* annotated as "logging" from the setup of a test and the density metric, $D_{NL}(t)$, as the number of calls to *Library Keywords* annotated as "logging" from the setup of a test over the total number of calls from the setup of the test. More formally:

$$S_{NL}(t) = |C_t \cap C_{setup} \cap C_{logging}|$$

$$D_{NL}(t) = \frac{|C_t \cap C_{setup} \cap C_{logging}|}{C_t \cap C_{setup}}$$

where $|C_t \cap C_{setup}|$ is the size of the set of *Library Keyword* called from the setup of a test t and $|C_t \cap C_{setup} \cap C_{logging}|$ is the size of the set of *Library Keyword* annotated as "logging" called from the setup of test t.

Smell-removing action The symptom is considered refactored if the call to the *Library Keyword* annotated as "logging" called from the setup of test t, $c_{t,setup,log}$, is removed. Thus we propose the smell-removing pattern, $SRA_{NL}(c)$, performed on a *Keyword* call c as follow:

$$SRA_{NL}(c) = c_{t,setup,log} \xrightarrow{action} \emptyset$$

Note that removing a parent of $c_{t,setup,log}$ is not considered as fixing the smell.

4.2.11 Obscure Test (OT)

Description The test behavior is difficult to understand because the test does not clearly state what it is verifying. Typical symptoms are hardcoded values, high cyclomatic complexity and/or function or procedure calls with high number of parameters.

Impact on readability Future reader might not understand the meaning of a hardcoded value, hence, missing the intention of the test. As with a high cyclomatic complexity it becomes hard for the future reader to follow the execution flow of the test and grasp what it is doing.

Impact on maintenance It is difficult to determine where to perform changes if hardcoded values are scattered all over the test code. Furthermore, test with high cyclomatic complexity might have side effect overseen during maintenance which might lead to future problems.

Dispersion Test

Detection method In this work, we focus on one of the expression of an obscure test: the overuse of hardcoded values. The code starts to smell when hardcoded values are used directly in calls to both *User Keyword* and *Library Keywords*, instead of relying on variables. Thus, we express the count metric, $S_{OT}(t)$, as the number of hardcoded arguments present in a test and the density metric $D_{OT}(t)$ as the number of hardcoded arguments present in the test over the total number of arguments from that test. More formally:

$$S_{OT}(t) = |A_t \cap A_{hardcoded}|$$

$$D_{OT}(t) = \frac{|A_t \cap A_{hardcoded}|}{|A_t|}$$

where $|A_t|$ is the size of the set of arguments passed to *Keyword* calls in a test t and $|A_t \cap A_{hardcoded}|$ is the size of the set of arguments passed to *Keyword* calls which are hardcoded.

Smell-removing actions Focusing on hardcoded values, the symptom is considered refactored if an argument passed to *Keyword* call as a hardcoded value is replaced by a variable. Thus, we propose the smell-removing pattern, $SRA_{OT}(a)$, where a is a *Keyword* call arguments as:

$$SRA_{OT}(v) = a_{t,hardcoded} \xrightarrow{action} a_{t,variable}$$

where $a_{t,hardcoded}$ is a hardcoded *Keyword* call argument and $a_{t,variable}$ is a variable *Keyword* call argument.

4.2.12 On the Fly (OtF)

Description The test calculates an expected results during its execution instead of relying on pre-computed values.

Impact on readability By embedding the business rule in the assertion, the code for the automated test can become as complicated as the system under test.

Dispersion Line.

Detection method The expected value should be a constant or a reference to a constant and not computed during the test. Thus, we express the count metric, $S_{OtF}(t)$, as the number of expected values that are computed in the test and the density metric, $D_{OtF}(t)$, as the number of expected arguments from assertion calls that are computed in the test over the total number of expected arguments from assertion calls:

$$S_{OtF}(t) = |A_t \cap A_{expected} \cap A_{computed}|$$

$$D_{OtF}(t) = \frac{|A_t \cap A_{expected} \cap A_{computed}|}{|A_t \cap A_{expected}|}$$

where $|A_t \cap A_{expected}|$ is the number of "expected" arguments in calls to *Library Keywords* annotated as "assertion" for a test t and $|A_t \cap A_{expected} \cap A_{computed}|$ is the number of "expected" arguments in calls to *Library Keywords* annotated as "assertion" resolved during the execution of the test t. Note that the identification of the "expected" argument is based on the definition of the *Library Keyword*. When a *Library Keyword* annotated as "assertion" contains a field called "expected", it is considered by the engine as the placeholder for the expected value. For instance, the library keyword *Should be equal* from the Builtin library takes six arguments: *value*, *expected*, *message*, *values*, *ignore_case* and *formatter*. Hence, in this case the expected argument is the second one.

Smell-removing action The symptom is considered refactored when the assertion is preserved but the expected value is not computed on the fly. This leads to the following equation for a smell-removing action, SRA_{OtF}, addressing a symptom in an argument a:

$$SRA_{OtF}(a) = a_{t,expected,computed} \xrightarrow{action} a_{t,expected,\neg computed}$$

where $a_{t,expected,computed}$ is an expected value that is computed on the fly and $a_{t,expected,\neg computed}$ is an expected value that is not computed on the fly, being either hardcoded or provided through a variable pointing to a static value. Thus removing the assertion would not be considered as removing the symptom since $a_{t,expected,\neg computed}$ would not be present.

4.2.13 Over-Checking (OC)

Description The test performs some assertions that are not relevant for its scope.

Impact on readability It becomes harder to understand what is the main intent of the test. Many assertions suggest the test is checking many properties.

Impact on maintenance The test may be too sensitive to the evolution of the SUT, verifying implementation properties instead of behavioral ones.

Dispersion Test.

Detection method As the ratio assertions to actions on SUT increases, the chances that all the assertions are relevant decreases. Thus we express the count metric, $S_{OC}(t)$, as the number of assertions in a test and the density metric, $N_{OC}(t)$, as the number of assertions in a test over the total number of calls in the test. More formally:

$$S_{OC}(t) = |C_t \cap C_{assertion}|$$
$$N_{OC}(t) = \frac{|C_t \cap C_{assertion}|}{|C_t|}$$

where $|C_t|$ is the size of the set *Library Keywords* calls and $|C_t \cap C_{assertion}|$ is the size of the set of calls to *Library Keywords* annotated as "assertion".

Smell-removing actions The symptom is considered refactored if the call to a *Library Keyword* annotated with "assertion", $c_{t,assertion}$, is removed from a test t. Thus, we propose the smell-removing pattern, $SRA_{OC}(c)$ where c is a call to an *Library Keyword* annotated with "assertion" as:

$$SRA_{OC}(c) = c_{t,assertion} \xrightarrow{action} \emptyset$$

4.2.14 Sensitive Locators (SL)

Description The test uses element identification selectors that have long chains to identify an element in the user interface. e.g. complex x-pass or CSS selector for web application.

Impact on maintenance This leads to fragile tests, as any change in that chain from the user interface representation will break the tests.

Dispersion Line.

Detection method The complexity of a locator can be expressed by how deep the locator needs to go in the hierarchy of the UI, be it an x-pass, a CSS selector or any UI representation based on a hierarchy. A locator, $A_{locator}$, can be expressed as the number of GUI element, $|E|$, that have to be traversed to uniquely locate the target GUI element. Thus, we express the count metric, $S_{SL}(t)$, as the number of locator arguments that visit more than one GUI elements and the density metric, $D_{SL}(t)$, the number of of locator arguments that visit more than one GUI elements in a test over the total number of locator arguments present in the test. More formally:

$$S_{SL}(t) = |A_t \cap A_{locator} \cap A_{|E>1}|$$

$$D_{SL}(t) = \frac{|A_t \cap A_{locator} \cap A_{|E>1}|}{|A_t \cap A_{locator}|}$$

where $|A_t \cap A_{locator}| \cap A_{|E>1}|$ is the number of locators that require to visit more than one element E of the GUI to be uniquely identified (*e.g.* the XPath "/html/body/div[4]/button" visits 4 elements to quality the button where the XPath "//button[@id ="unique-id"]" only needs to visit one element). Note that Robot Framework using dynamic types, only *Library Keyword* calls explicitly specify a type for their parameters. Therefore, for each *Library Keyword* call requiring a locator as an argument, the engine resolve all the values possible for the argument within the test to populate the set $A_{locator}$.

Smell-Removing Actions: The symptom is considered refactored if the value of a node l flagged as complex locator sees its length go down to one. Thus, we propose the smell-removing pattern, $SRA_{SL}(l)$, as follow:

$$SRA_{SL}(c) = l_{|E>1|} \xrightarrow{action} l_{|E=1|}$$

where $l_{|E>1|}$ is a node defining the value of a sensitive locator and $l_{|E=1|}$ is the same node but with a simple locator expression. Note that a change is only accounted for when the value of the locator is modified.

4.2.15 Sneaky Checking (SC)

Description The test hides its assertions in actions that are at the wrong level of details.

Impact on readability The future reader is not able to understand what is being tested by just looking at the main steps of the acceptance criteria without a need to inspect how low level details are implemented.

Dispersion Keyword.

Detection method A *User Keyword* only calling an *Library Keyword* annotated as "assertion" can be seen as hiding the assertion to the callers of that *User Keyword*. Thus, we express the count metric, $S_{SC}(t)$, as the number of unique *User Keywords* called by a test and only calling an assertion and the density metric, $D_{SC}(t)$, as the number of unique *User Keywords* called by the test and only calling an assertion over the number of unique *User Keywords* called by that test. More formally:

$$S_{SC}(t) = |K_t \cap K_{assert}|$$

$$D_{SC}(t) = \frac{|K_t \cap K_{assert}|}{|K_t|}$$

where $|K_t|$ is the total number of unique *User Keywords* and $|K_t \cap K_{assert}|$ is the number of *User Keywords* only calling an *Library Keyword* annotated as "assertion" (logging actions are ignored).

Smell-Removing Actions: The symptom is considered refactored if a *User Keywords* only calling a *Library Keyword* annotated as "assertion", $k_{t,assert}$, is removed from the test t. Thus, we propose the smell-removing pattern, $SRA_{SC}(k)$, where k is a *User Keywords* as:

$$SRA_{SC}(k) = k_{t,assert} \xrightarrow{action} \emptyset$$

4.2.16 Stinky Synchronization (SS)

Description The test fails to use proper synchronization points with the system under test.

Impact on execution The test becomes oversensitive to the response time, leading to flaky tests, or very slow tests when choosing very conservative wait points.

Dispersion Line.

Detection method This symptom is associated with the use of explicit and fixed synchronization, independent from the SUT such as a pausing the test for a specific amount of time. Thus, we express the count metric, $S_{SS}(t)$, as the number of calls to explicit pause from a test and the density metric, $D_{SS}(t)$, as the number of calls to explicit pause from a test over the total number of synchronization calls. More formally:

$$S_{SS}(t) = |C_t \cap C_{sync} \cap C_{sleep}|$$

$$D_{SS}(t) = \frac{|C_t \cap C_{sync} \cap C_{sleep}|}{|C_t \cap C_{sync}|}$$

where $|C_t \cap C_{sync}|$ is the size of the set of calls to *Library Keyword* annotated as "synchronization" in a test t and $|C_t \cap C_{sync} \cap C_{sleep}|$ is the size of the set of calls to *Library Keyword* annotated as "synchronization" by pausing the test execution for a specified amount of time. In the case of Robot Framework, it is instantiated by calls to the *Library Keyword* "Sleep".

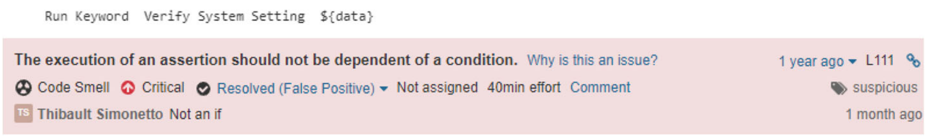
Smell-removing actions The symptom is considered refactored if a *Library Keyword* call annotated as "synchronization" which pausing the test execution of the test for a specified amount of time, $c_{t,sync,sleep}$, is removed or replaced by another *Library Keyword* call annotated as "synchronization", $c_{t,sync,\neg sleep}$. Thus, we propose two smell-removing patterns, $SRA_{SS,1}(c)$ and $SRA_{SS,2}(c)$, as follow:

$$SRA_{SS,1}(c) = c_{t,sync,sleep} \xrightarrow{action_1} \emptyset$$

$$SRA_{SS,2}(c) = c_{t,sync,sleep} \xrightarrow{action_2} c_{t,sync,\neg sleep}$$

4.3 Smell Validation

We proceeded to an external evaluation of the tooling by asking four Ph.D. students from our group to systematically go over the smells and flag the ones that they consider false positive. To assist them during this task we used the SonarQube Plugin (Fig. 6) that we developed which implements the smells reported in this work (with the exception of *Narcissistic* requiring a language model of 400MB that we deem too large for distribution and



```
Run Keyword   Verify System Setting   ${data}
```

The execution of an assertion should not be dependent of a condition. Why is this an issue? 1 year ago ▾ L111
Code Smell Critical Resolved (False Positive) ▾ Not assigned 40min effort Comment suspicious
TS Thibault Simonetto Not an if 1 month ago

Fig. 6 Example of a smell instance deemed as a false positive by one of our volunteer during the evaluation of the smell detection algorithm using our SonarQube Plugin

Army of Clones which conflicts with SonarQube which implements its own token based clone detection mechanism). To do so, we assigned to each of the participants one of the four largest projects (ifs, harbor, bikalims, and ozone) of our dataset and asked them to review systematically all the detected smells instances in the last version of each projects. However, due to the large amount of *Hardcoded Values* (31,161) we only used a random sample and relying on the Central Limit Theorem, we asked the participants to review a sample of 673 smells offering a confidence interval of 99% and a margin of error of 5%. We present the results in Table 9. We see that with the exception of *Conditional Assertion*, all metrics perform well in terms of precision. The low score associated with *Conditional Assertion* originates from the definition of what we define as a condition. Indeed, the tooling label the library keyword `Run Keyword` as conditional since it impacts the control flow, but not doing so conditionally, but rather systematically, it should be excluded when considering a conditional assertion. The false positive associated to *Hardcoded Values* originate from the presence of structures defined by the development teams in python scripts that could not be mapped by our tooling.

Next, because some smells do not appear in the selected versions or are not implemented by the SonarQube Plugin, an additional investigation had to be performed by the authors of the paper to validated the remaining 5 smells. To do so, we randomly select 30 instances of the smell symptoms detected in the industrial project with which we are familiar. We perform a similar analysis as the one performed by the participants but relying on the Ikora Evolution which does not provide a user interface, however, expose both *Narcissistic* and *Army of Clones* instances. Furthermore, the tooling allow to analyze many version of the test suite at once to augment chances to encounter smell symptoms. The results of this analysis are reported in Table 9.

We observe that most result do not expose false positive. Furthermore, after this analysis, we improved the tooling to reach 100% precision for the two smells that performed suboptimally with regard to that metric.

Table 9 Results from the tool evaluation performed using the SonarQube Plugin

Smell	Sample	False Positive	Precision
Army of Clones*	30	0	1.00
Conditional Assertion	42	26	0.42
Hardcoded Environment	31	0	1.00
Hardcoded Values	673	30	0.96
Lack of Encapsulation	949	0	1.00
Long Test Steps	412	0	1.00
Missing Assertion	150	0	1.00
Middle Man*	30	0	1.00
Noisy Logging*	30	0	1.00
On the Fly	6	0	1.00
Over-Checking	489	0	1.00
Sensitive Locator*	30	0	1.00
Sneaky Checking*	30	0	1.00
Stinky Synchronization Syndrome	279	0	1.00

Smells associated with a * indicates an evaluation on the industrial project by the author of the tool

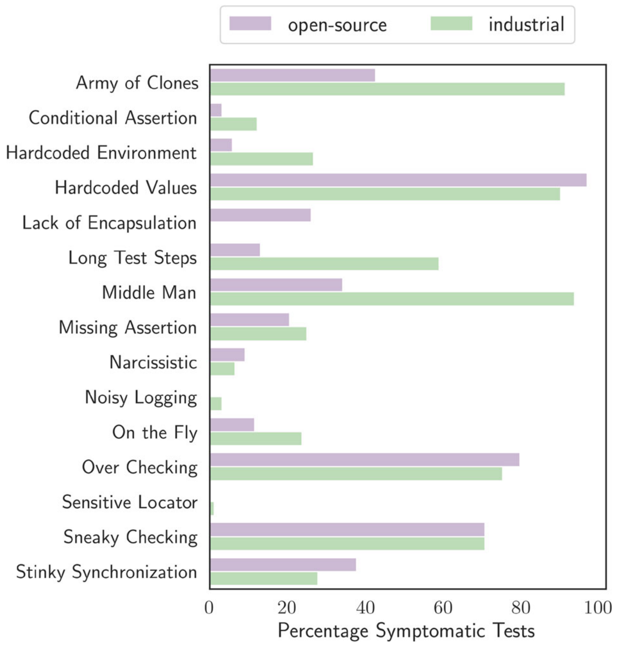


Fig. 7 Percentage of tests exhibiting at least one symptom in open-source and industrial projects

4.4 RQ2: SUIT Smell Symptoms Distribution

Building on the metrics derived in Section 4.1, we show in Fig. 7 the number of tests exhibiting at least one smell symptom. We observe that the smell *Hidden Test Data* does not appear in any of the studied projects. Furthermore, the smells *Noisy Logging*, *Sensitive Locator*, *Narcissistic* and *HardCoded Environment* manifest in less than 10% of the tests in both open-source and industrial test suites. Figure 8 shows answer from the interviews to

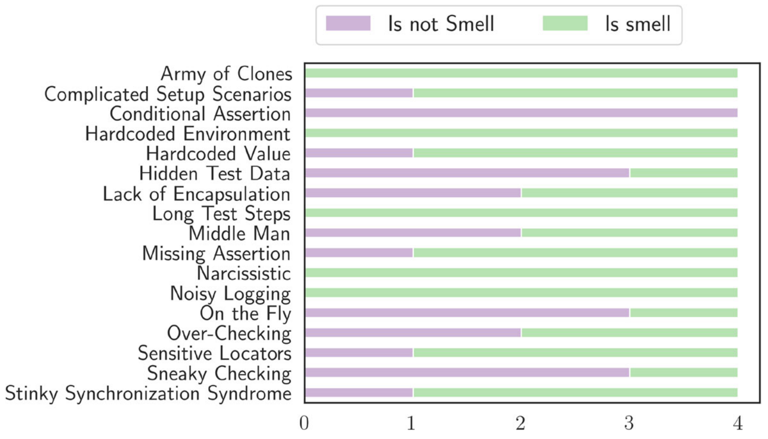


Fig. 8 Answers from the interviews to the question: Do you consider this pattern a smell? The total number of answer (X-axis) corresponds to the number of interviewees

the question : *According to your experience, do you consider it a smell?*. Putting the results from Figs. 7 and 8 into perspective, we observe that the smells *Noisy Logging*, *Narcissistic*, and *HardCoded Environment* are all considered by all interviewees to be a smell and tend to rarely appear in the test suites analyzed suggesting a deliberate intent by practitioners to avoid these patterns. In the case of *Sensitive Locator*, even though one interviewee does not consider it harmful, most do, mentioning strategies and tooling to generate good locators: *I use tools like Playwright to generate stronger locators* or *There also exists website like "test projects" where you can generate good locators*. Furthermore, the literature offers an entire field dedicated to improve locator representation, specifically by offering ways to generate shorter locators (Leotta et al. 2014; 2016; Kirinuki et al. 2019; Nguyen et al. 2021). Finally, *Hidden Test Data* does not appear often despite not being considered by three out of four interviewees as a smell. These results may originate from the limited capacity from the tooling to detect the use of data providing from pythons scripts, thus not being annotated as *GETTER* (Table 3) from our tool.

On the other end of the spectrum, three smells appear in the majority of the tests: the symptom *Hardcoded Values* appears in more than 90% of the tests, *Over Checking* between 75% (industrial) and 79.5% (open-source), and finally *Sneaky Checking* appears in 70% of the tests. Again, contrasting these observations with the results from Fig. 8, we observe that *Sneaky Checking* is considered to not be a smell by three out of four interviewees, which explains its high occurrence. On another hand, the high number for *Hardcoded Values* can be explained by the metric which is very sensitive, since as soon as one hardcoded value is used in a test, regardless of its size, it is considered as being symptomatic. Lastly, *Over Checking* present a test dispersion, which means it can be spread across different *Keywords* located in different files. This characteristic can explain why practitioners are less able to avoid this smell because they might not be aware of its presence.

When comparing industrial and open-source projects, we observe significant differences. Notably, in industrial projects three symptoms appear much more often than in open-source projects, namely, *Middle Man* (93.5%), *Army of Clones* (91.2%) and *Long Test Steps* (58.8%). These results can be explained by the structure of the projects at BGL BNP Paribas. Symptoms for the smells *Long Test Step* and *Army of Clones* manifest because many repositories are interconnected, leading to larger tests and code base where these two smells with a large dispersion, test and suite respectively, are hard to detect. As for the symptoms of *Middle Man*, their presence is explained by the use of *Keywords* that act as translation layers. While two actions can perform the same concrete actions on the SUT, in different contexts they might operate on a different business logic. Thus, developers created translation layers to unify the vocabulary used in the acceptance layer of each test. These translation layers are implemented by the use of a *Keyword* only calling one other *Keyword* with a different name. Along those lines, one interviewee working on another project added: *It depends on the place people come from. For instance, non-english speakers people would not be able to read the english keywords*, when explaining why sometimes they do introduce a *Middle Man*.

We continue our analysis with the evaluation of the number of symptoms appearing per tests (Fig. 9). Note that only tests presenting at least one symptom are considered in our analysis, *i.e.* symptomatic tests. Looking at *Hardcoded Values*, we see that even if both industrial and open-source projects exhibit the same proportion of tests affected by *Hardcoded Values*, the number varies significantly from a median of 27 in the case of industrial projects to 59 for open-source ones. As explained previously, this number can be explained by the presence of larger tests in the industrial projects, which offer mechanically more chances to observe more *Hardcoded Values*. A similar observation can be made for the case

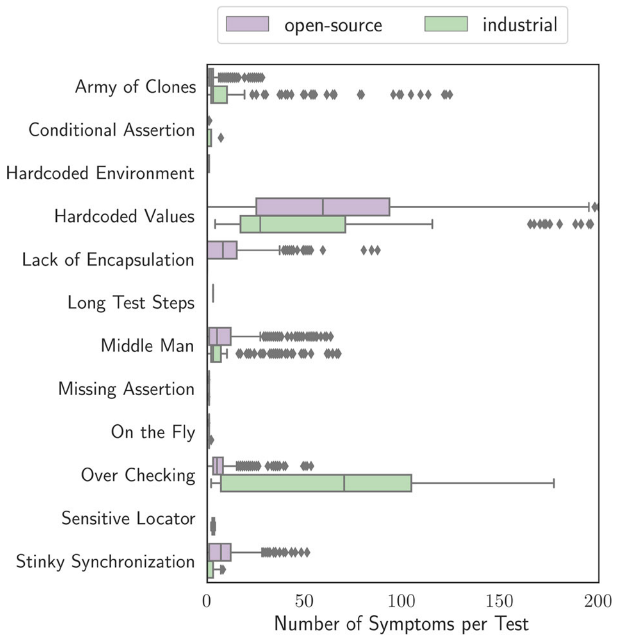


Fig. 9 Distribution of the number of symptoms in symptomatic tests for all projects across all commits

of *Over Checking* where the median varies from 5 (open-source) to 70 (industrial). Finally, the last significant difference regards the symptom *Army of Clones*. Not only the number of tests containing duplicated code (Fig. 7) is higher, but the average number of duplicated *Keywords* in symptomatic tests is higher in industrial projects than in open-source projects. This is explain by the complexity of the project in the industrial dataset, where a duplicated *Keywords* can exist in different sub-project and therefore, it becomes hard for the practitioners to reuse keywords when they are not aware of their existence. Note that this observation is also true in open-source projects. Indeed, during our interviews, one of the interviewee blamed the diffusion of duplicated code on the lack of tooling, saying: *With Robot Framework, there is no proper IDE that allows you to have visibility for the code. [...] The lack of tooling is the issue*.

To put the results of Fig. 9 into perspective, we show the density metrics, $D_S(t)$, in Fig. 10. As a reminder, the density value presents the number of symptoms appearing in a SUIT divided by the number of times it could have appeared. One interesting finding to put in contrast with the previous figure regards the smell *Hardcoded Environment*. While the symptoms do not appear often (only outliers present a value greater than 0 in Fig. 9), whenever an environment variable is used, it will generally be through the use of a hardcoded value which is shown by a median value for its density metric at 1 for both open-source and industrial projects. That is, with regard to its potential occurrences, the SUIT smell *Hardcoded Environment* is actually very frequent. Thus, even though interviewees agree to qualify the use of Hardcoded values to set up environment variable, this smell tends to appear at a high density. On the other hand, the smells *Noisy Logging*, *Hidden Test Data*, *Sensitive Locator*, *Narcissistic* and *Conditional Assertion* present low scores in the density value. This means that even when the conditions for their occurrence are present, the symptoms of these smells do not manifest, which confirms our previous observations.

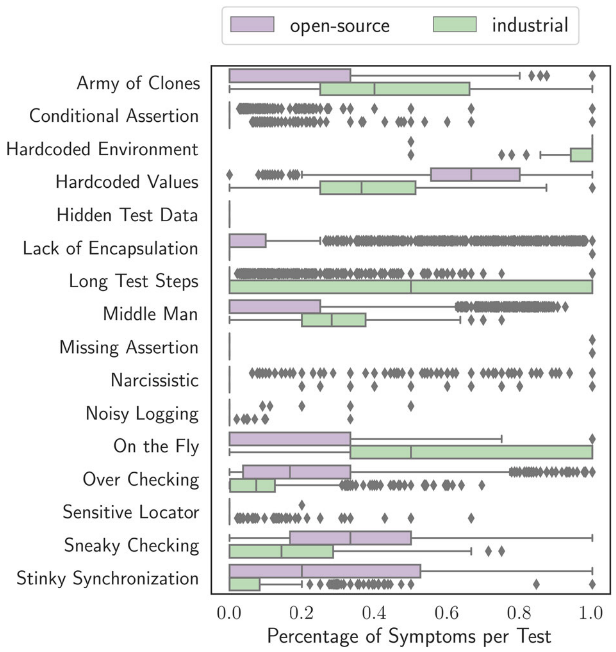


Fig. 10 Distribution of the density of symptoms in each test for all projects across all commits where the density is the percentage of locations that a symptom could appear at which actually exhibit the symptom

We also observe some notable differences between the open-source projects and the industrial projects. In the case of the industrial projects, we observe that *On the Fly* with a median value of 0.67 shows a deviation from the count metric, $S_s(t)$. As for open-source projects, we observe that *Sticky Synchronization* is frequent, with a median value of 0.2. The divergence with the count metric can be explained by the fact that many tests do not use explicit synchronization mechanism, thus, limiting the number of tests where the symptom could appear. Furthermore, the difference from the industrial project can be explained by the hard policy in the QA team at BGL BNP Paribas, where explicit timeouts are considered as a major source of flakiness. As such, explicit timeouts are strongly discouraged by the QA team and are usually removed during code reviews.

To assess its potential to appear in a test suite, we consider a symptom as frequent if the median value of the density metric is greater than 0. Thus, *Hardcoded Values* with median values of 0.67 and 0.33, *Over Checking* with median values of 0.17 and 0.08, and *Sneaky Checking* with median values of 0.33 and 0.14, for open-source and industrial projects respectively, are frequent in both industrial and open-source projects. This aligns with the observations from the count metric and confirms the prevalence of these smells. Similarly, following the same trend from Fig. 9 *Army of Clones* (median value of 0.40), *Long Test Step* (median value of 0.5) and *Middle Man* (median value of 0.29) appear often in industrial projects. As we mentioned earlier, the result for *Hardcoded Values* originates from the sensitivity of the metric. *Sneaky Checking* not being considered as a smell by three out of four interviewees, appears often. On the contrary, even though two out of four interviewees consider *Middle Man* a bad practice, it tends to appear quite often. According to one of the interviewees : *Middle Man creates shorter syntax, making the test easier to read and* *understand*, with the caveat that *if it has the same number of argument, it is quite useless. So reduce the number of arguments instead of just giving it a prettier name*. This comment can explain the apparently contradicting results. It may not be consider a bad practice if the encapsulating *Keyword* consist of less arguments. Finally, *Army of Clones*, *Long Test Step*, and *Over Checking* all follow a characteristic: they are not local, *i.e.* the definition of each symptom covers different places in the test code. As such, especially in the absence of appropriate tooling, it is challenging for testers to spot the presence of these symptoms, thus hindering their effort to avoid them.

Finally, we perform a ranking analysis between industrial projects and open-source projects. The goal of this analysis is to determine if the symptoms appear in the same order in both project. Thus, we use the normalized Levenshtein similarity where the order of the symptoms is determined by the mean number of symptoms and percentage of symptom. This metric provide an indication as the number of permutations that are necessary in order to go from one list to the other, in other words their similarity. A value of 1 indicates a perfect similarity and a value of 0 that the two list are totally dissimilar. We obtain a value of 0.1819 when comparing the number of symptoms and a value of 0.3125 in the case of their density. The higher score when taking into account the density metric can be explain by the difference in term of tests sizes, where the industrial tests tend to be much larger than the open-source ones, thus offering larger raw counts but normalizing for the size effect. More generally, these low scores suggest that the relative importance of the symptoms present in SUITs differs from industrial project and open-source projects. Notably, despite the different in size, one major difference between the two dataset is the scope of the test suites. In the case of open-source project, each test suite target a single application. In the case of the industrial projects, different test suites are reused to create common libraries in an effort to reduce the maintenance and production effort by mutualizing part of the test code. Unfortunately out of the scope of the present study, the impact of the architecture of a test suite on the occurrences of smell would shed more light on the underlying reason behind these differences.

Large projects are prone to multi-location smells such as *Army of Clones*, *Long Test Steps* and *Middle Man* for which the symptom is spread across multiple location in the test code. On the contrary, with the exception of *Hardcoded Environment*, when testers and the literature agree on a smell, if its symptom is localized to a single place in the test code, practitioners are actively avoiding it.

4.5 RQ3: Smell-Removing Actions

While Section 4.4 focuses on the prevalence of smell symptoms across SUITs, this section tackles the question of how often the symptoms are removed by practitioners. We use the methodology described in Section 3.4 to extract the fine-grained changes removing a symptom from the test, *i.e.* smell-removing actions.

Column *Count* in Table 10 shows the number of smell-removing actions across all SUIT-modifying commits. Adding assertions to tests presenting the symptom *Missing Assertion* is the most common type of smell-removing action as it occurs 6,647 times in the industrial project and 137,707 times in open-source projects. This result is explained by the fact that in SUITs, creating a scenario and being able to run it from beginning to end already provide

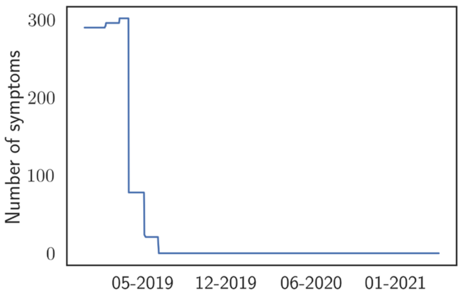
Table 10 Total Number of smell-removing actions (*Count*) and percentage of symptomatic tests where at least one smell-removing action was performed during their lifetime (*Percent*) for industrial and open-source projects

	Industrial		Open-source	
Symptom	Count	Percent	Count	Percent
Army of Clones	738	36.64	10,139	24.42
Conditional Assertions	9	4.95	270	0.32
Hardcoded Environment	0	0.00	882	9.95
Hardcoded Values	226	18.44	28,863	17.09
Hidden Test Data	0	0.00	0	0.00
Lack of Encapsulation	8	0.00	10,944	11.52
Long Test Steps	0	0.00	34	0.19
Middle Man	1,037	39.57	55,509	27.62
Missing Assertion	**6,647**	**90.45**	**137,707**	**72.86**
Narcissistic	0	0.00	0	0.00
Noisy Logging	0	0.00	0	0.00
On the Fly	27	13.85	516	7.93
Over-Checking	35	3.46	21,586	15.50
Sensitive Locators	2	4.55	0	0.00
Sneaky Checking	0	0.00	0	0.00
Stinky Synchronization	38	4.92	23,163	23.28

a signal. However, once the test is ready and behaves as expected, more specific checks, *i.e.* assertions, are added to improve its readability and its fault detection capabilities. Thus, we observe that some SUITs are missing assertion when created but assertions are added in later commits.

In the industrial project, three other types of smell-removing actions see high values compared to the rest, namely, *Army of Clones* with 738 smell-removing actions, *Hardcoded Value* with 226 smell-removing actions, and *Middle Man* with 1,037 smell-removing actions. Indeed, with the introduction of the tool presented in (Rwemalika et al. 2019b), the team at BGL BNP Paribas became aware of the existence of a large amount of code duplication and actively started to work on reducing it, which explains the number of smell-removing actions for *Army of Clones*. As for *Middle Man*, during the year 2019, the team performed a normalization in the naming of *Keywords*, in an effort to improve readability in the codebase. Consequently, names such as “Fill Form Next Page” were changed to more expressive forms such as “Fill Login Form and Validate”. The goal was to increase the expressiveness of the test codebase and consequently, it reduced the need for a translation layer. Note that in this case, the team was targeting another SUIT smell, *Unsuitable Naming*, where the name of the *Keyword* does not provide indication as what it is doing. Thus, the team ended up addressing two smells: *Unsuitable Naming* and *Middle Man*. Finally, observing the number of smell-removing actions addressing *Hardcoded Value* is mainly due to the fact that the symptom appears often. Indeed, when observing the column *Percent* from Table 10, we see that only 18% of the test exhibiting the symptom ever observe the removal of one symptomatic instance through their lifetime.

Fig. 11 Evolution of the number of symptoms for the smell *Narcissistic* over time for the industrial project (BGL BNP Paribas)



Still focusing on the results for the industrial project, for some symptoms, we never observe smell-removing actions. This is the case for: *Hidden Test Data*, *Noisy Logging*, *Narcissistic*, *Sensitive Locator* and *Sneaky Checking*. With the exception of *Sneaky Checking*, these symptoms only rarely occur in the test codebase. Hence, in the absence of symptoms, there is nothing to refactor. *Sneaky Checking*, on the other hand, is considered frequent according to our metric, but is never addressed. This observation confirms our initial assumption that *Sneaky Checking* is not a smell, and therefore, not actively removed from the test code.

However smell-removing actions only offer a partial picture. Indeed, Figure 11 shows the evolution of the symptom *Narcissistic* over time. We observe an abrupt decrease of the number of symptoms until they completely disappear. Where this observation seem to contrast with previous results, this is explained by old tests presenting this pattern are deprecated by the team and replaced by new ones not presenting the smell. Thus, while there is no specific smell-removing action happening, the symptoms are removed from the test suite as new test are introduced and old tests deprecated.

The results for open-source projects depict a similar picture. Symptoms for *Missing Assertion*, *Middle Man*, *Army of Clones*, and *HardCoded Values* show also relatively high values compared to other symptoms. Similarly, we also notice a relatively large number of

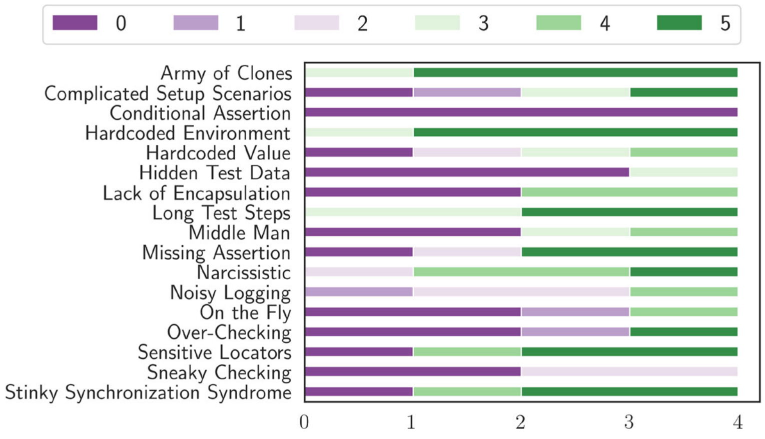


Fig. 12 Answers from the interviews to the question: How would you prioritize the refactoring of this smell? The total number of answer (X-axis) corresponds to the number of interviewees. A value of 0 from the legend indicates that the interviewee does not consider the pattern a smell, a value of 1 that it prioritize its refactoring to low, and a value of 5 corresponds to a high priority for refactoring

smell-removing actions for the symptoms of *Stinky Synchronization*, *Over Checking*, and *Lack of Encapsulation*. The difference between the results from *Stinky Synchronization* can be explained by the active policy in the industrial team preventing such symptoms to appear altogether.

Looking at Fig. 12, we see confirm once again that *Sneaky Checking* is not regarded as a smell and explain the absence of refactoring. Interestingly, where one would have expected high values for the intent of refactoring for *Missing Assertion*, we see that only one developer considers it high where the two others do not. Yet, an hypothesis as the high number despite this result from our interview is the ease and lack of side effect of adding an assertion in a test. Contrarily, *Long Test Steps* almost never see refactoring actions, where all interviewees gave it a high score for refactoring. Indeed, not only the smell is difficult to detect without executing the test, but refactoring the test requires deep changes and a lot of care to avoid breaking its workflow. As such, this smell tends to be overlooked. Surprisingly, *Army of Clones* with similar characteristics was refactored, even in open-source projects. This indicates that testers are ready to pay the cost of maintaining the quality of the test suite, even within a non-trivial setup. But, even more surprising, is the total absence of refactoring for *Hardcoded Environment*. Indeed, looking at Fig. 12, all the interviewees agree that it is a smell and should be removed with high priority. However, not only the smell tend to appear systematically, but also it is never removed, despite being very local and easy to fix (replace a hardcoded value by a variable and put the associated value in the variable block).

Similarly to Section 4.4, we perform a rank analysis using the normalized Levenshtein similarity. The value obtained when comparing the number of fixes performed in open-source and industrial projects is 0.4375 and 0.2667 when accounting for the percentage of smell-removing actions. These low values show that the majority of smell-removing actions are not ranked in the same way in both industrial and open-source projects. Taking into account the results of Section 4.4 this comparison suggests that when generalizing results obtained from open-source projects, researchers should remain careful to their applicability in industrial contexts.

In conclusion, the results presented in this section show that the smell-removing actions performed in industry and open-source projects are different in type and frequency. However, in both cases, the proportion of symptomatic tests that are refactored remain low with the exception of the symptoms for *Missing Assertion*.

For half of the smells less than 50% of the symptomatic tests are ever subjects of smell-removing actions. *Missing assertion* is a unique exception with between 70% (open source) and 90% (industrial) of the symptomatic tests being refactored. Interestingly, while smell-removing actions are rare, test smells like *Narcissistic* and *Middle Man* still disappear from the test codebase as a side effect of the replacement of symptomatic tests by new tests not exhibiting the symptom.

Finally, we conclude this analysis with a cross-comparison across different metrics. Table 11 summarizes the results of the interviews we conducted; the number of commits modifying the test suite between the insertion of the smell and its removal by a smell-removing action; the number of symptoms present in the tests; and finally, the number of actions performed to remove those symptoms. We observe that when practitioners agree to qualify a smell as relevant (*Perception* has a large value) and the smell appears frequently (*Symptoms* has a large value), they tend to refactor the smells (*Fixes* has a large value). This

Table 11 Comparative table that reports results from all the previous research questions where *Perception* is the average refactoring priority (0 to 5) given by the interviewees, *Duration* is the average number of versions between the introduction and the removal of a smell, *Symptoms* is the average number of symptoms per test, and *Fixes* is the total number of fixes across all versions

Smell	Perception	Duration	Symptoms	Fixes
Army of Clones	4.50	39.70	3.94	10,877
Conditional Assertion	0.00	74.25	0.13	279
Hardcoded Environment	4.50	303.10	2.29	882
Hardcoded Values	2.25	303.10	47.21	29,089
Hidden Test Data	0.75	–	0.00	0
Lack of Encapsulation	2.00	11.46	3.51	10,952
Long Test Steps	4.00	13.78	1.07	34
Middle Man	1.75	594.68	4.76	46,546
Missing Assertion	3.00	4.61	0.21	144,354
Narcissistic	3.75	–	0.06	0
Noisy Logging	2.25	–	0.05	0
On the Fly	1.25	229.93	0.07	540
Over Checking	1.50	15.48	3.93	21,621
Sensitive Locator	3.50	411.10	0.01	2
Sneaky Checking	1.00	–	2.18	0
Stinky Synchronization	3.50	87.89	5.36	23,201

is the case for the smells: *Army of Clones*, *Missing Assertion*, and *Stinky Synchronization*. Smells with a high *Perception* and a low *Symptoms* may indicate that developers actively avoid these smells. This is the case for: *Narcissistic*, *Noisy Logging*, and *Sensitive Locator*. Interestingly, we also notice some smells which, despite interviewees agreeing to qualify them as low priority, tend to undergo a lot of refactoring to remove them. This is the case for the smells: *Hardcoded Values*, *Lack of Encapsulation*, and *Middle Man*. There are two cases with different trends. These are the *Long Test Steps* and the *Hardcoded Environment*. In particular, *Long Test Steps* while being considered of high priority by testers and appearing quite often is not addressed. We believe this is due to the difficulty to detect this smell without appropriate tooling. Similarly, *Hardcoded Environment* is an outlier in our dataset, i.e., while being fairly frequent and considered of high priority by the developers, it is not removed. To explain this observation we inspected the code in which this smell appears and found out that the expressions in which the smell appears were readable and easy to maintain.

When observing the time between the introduction of a smell and its removal (by a smell-removing action), we see that *Missing Assertion* have the shortest lifespan. This confirms our previous observation that this type of smell appears during creation time, but is quickly addressed once the test is released. On the other end of the spectrum, some tests take several hundreds of commit before being addressed. One observation that can explain this result is that some smells are removed in bulk following the introduction of a new tool (*e.g.*, our smell detection tool at BGL BNP Paribas). Another explanation regards smells such as *Sensitive Locator* where the locators are only updated when they cause the test to break. As such, where this study focuses on the perceived impact of smells, their dispersion in the test code base, and the way they are addressed, future work should focus on the concrete impact of SUIT smells to offer a better understanding as to why some smells are so long lived or never removed.

5 Lessons Learned

In this section, we describe the lessons learned based on our analysis. These lessons are important for test automation practitioners, tool manufacturers, and researchers.

Lesson 1: Testing community possess untapped knowledge During our multivocal literature review, we identified 25 new smells that were never addressed in the scientific literature when studying SUIT smells. This gap in knowledge between scientists and practitioners highlights the importance for researchers in the field of software engineering practices to read content produced by practitioners.

Lesson 2: SUIT have their unique issues Even though there exists an overlap between unit tests and SUITs, we observe that SUIT contains unique smells and *vice versa*. This suggests that different types of testing practices will exhibit different issues. Thus, it is important when studying tests not to conflate all type of tests but to actually separate them based on their scope, intent, and mode of interaction with the SUT.

Lesson 3: Not everyone agrees When interviewing seasoned practitioners whether or not a pattern constitutes a smell, in half of the cases no consensus was reached. This means that what is viewed as a bad practice within a team or an organization may not be seen the same way by another group. Furthermore, the interviewees consider that four of the 16 smells that we presented to them should not be considered as smell. Thus, they have a diverging position than the author from our grey literature analysis, *i.e.* other practitioners. As such, tool producers should use this result as a caution. When designing tools, sufficient flexibility should be provided to the practitioners so they can adapt the smell detection algorithm to their needs. Indeed, what can be considered a test smell in a team, can be considered not harmful by another one.

Lesson 4: If you cannot see it, you cannot do anything about it Some tests are considered harmful by all the practitioners we interviewed, yet, still appear often in tests and are not subsequently fixed. This behavior can be explained by the amount of code that has to be analyzed to detect the smell. For example, smells like *Narcissistic* which expression is local tend to almost never appear. On the contrary, smells such as *Long Test Step* for which the symptom can be spread across different files are much harder to spot. Indeed, in the absence of proper IDE support multi-location smell symptoms are challenging for testers to detect and address. As such, it is important for researchers and tool producers alike to focus on this large spread tests in order to provide the tooling to support the activities of test automation engineers.

Lesson 5: Avoid smells or no one will remove them The majority of the smells introduced will never be refactored. As such, we suggest to focus on the prevention of smells rather than their removal at a later stage. Providing feedback during creation time could avoid the introduction of the smells altogether. Thus, when designing tooling, integrating them into

IDE is crucial to provide value to test automation engineers. Furthermore, the adoption of strong policies and code review seems to avoid the spread of smells in the test codebase (*e.g.* Strong policies allowed the team BGL BNP Paribas to prevent the proliferation of *Stinky Synchronization* smell in their test codebase.).

Lesson 6: We learn from our mistakes While analyzing the subsequent versions of the test suites, many smells disappear without undergoing smell removing actions. Indeed, in some instances, old smelly tests were disappearing and the newly introduced tests did not express the smells. It shows that developers are aware of the impact of SUIT smells or at the very least tend to avoid them as the project evolves. This shows that when conducting research on smell evolution, not only the action to actively remove smells can lead to conclusion, but also the absence of some patterns in newly created code.

6 Threats to Validity

Threat to construct validity result from the non suitability of the metrics used to evaluate the results. To detect test smells we rely on heuristics based on code metrics. Thus, the validity of our results is bound to the precision with which smells are detected by our tooling. To mitigate the effect of low precision, we first ensured that our tooling offers good performances by writing a comprehensive test suite covering all the targeted smells in their different variation of the patterns. Moreover, we completed our evaluation of the tool with a user evaluation to further mitigate this threat to validity. Unfortunately, we are fully aware that the non-detection of certain smells is a limitation of this work, yet, we are not aware of any mechanism to mitigate this shortcoming.

Threats to the internal validity are due to the design of the study, potentially impacting our conclusions. Such threats typically do not affect exploratory studies like the one in this work. A caveat can be raised on how the changes are extracted. Indeed, changes are recorded when developers check-in their changes to the control version system. Thus, smell-removing actions might be lost if developers do not check-in often their changes or on the contrary, we might flag artifacts of the development process (*e.g.* Assertions added only at the end of the test) as smell-removing actions. To account for this phenomenon, we analyze manually a subset of the results to ensure the soundness of the process.

Finally, the threats to external validity, regarding the generalization of the results, concerns mainly the choice of the projects analyzed. Indeed, conducting our analysis on a limited sample of projects, our results might not generalize to other projects. However, we try to control for this limitation by selecting projects of different sizes, from different domains and in different development cultures. Furthermore, working with Robot Framework, there is no guarantee that the results presented in this work are transferable to other languages or technologies. Finally, it is noted that the proposed catalog reflects the studied sources, which were deemed sufficient, and should not be considered as an exhaustive list of SUIT smells. Additionally, the defined search terms and the sources analyzed were both in English, resources written in other languages were excluded. We argue that our process, be it restricted to English encompasses sufficient knowledge that would have been captured by other sources, notably with English being a language adopted even by non native speakers to increase diffusion. Further replications and analyses would be beneficial to corroborate our findings.

7 Related Work

Considering the uniqueness and impacts of test smells, the research community entailed studies to extend the catalogs of known test smells, empirical studies have been conducted to analyze their introduction, diffusion and suppression, and finally, tools have been proposed to automatically detect them.

(van Deursen et al. 2001) are among the first to introduce the concept of test smells. They describe a catalog of 11 test smells and refactoring operation to address them. Following their work, different studies were conducted to further extend the catalog of test smells. Departing from the analysis of the code to identify test smells, (Bowes et al. 2017) and (Tufano et al. 2016) conducted human studies to isolate test smells and report the perceived impacts on both test code and production code.

With good test smell catalogs established, the community investigated the diffusion and evolution of smells in test code. (Bavota et al. 2015) conducted an empirical studies to analyze the diffusion and the impact of test smells. In their work, (Tufano et al. 2016) and later (Kim 2020) showed that test smells are introduced when the test is written and are long lived.

To help practitioners identify potential sub-optimal patterns in their test codebase, researchers introduced tooling to automatically detect them. Typically, detection tools analyze test code metrics using rule based heuristics (Van et al. 2007; Reichhart et al. 2007; Peruma et al. 2020) to isolate test smells.

Nonetheless, all of the work mentioned above focuses on test smells present in unit testing. (Hauptmann et al. 2013) conducted a study on test smells present in tests expressed in natural language in industrial systems. However, their experiments show that the metrics extracted cannot be used for assessment of the quality of a test suite. (Femmer et al. 2014; Femmer et al. 2017) introduce the concept of requirement smells and conducted an empirical evaluation of their approach using industrial projects. In their work, the author present an automated static analysis technique relying on natural language processing (NLP) to detect smells in the requirements. The main goal at the requirement level is to detect whether or not a requirement contains ambiguities, therefore, the techniques developed rely on concrete instances of ambiguities. Finally (Chen and Wang 2012), propose a catalog of 11 test smell present in KDT along with 16 refactoring methods.

8 Conclusion and Implications

The goal of this paper is to shed light on the smells occurring in SUITs. To do so, we conducted a multivocal literature review and identified SUIT-specific smells from both formal and grey literature. This process lead to a catalog of 35 SUIT smells. Comparing our catalog to previously established catalogs from the literature on Unit Test Smells, we see that while there exist an overlap between the SUIT and Unit Test Smells with 14 common smells, 21 smells are unique to SUITs. These smells, unique to SUITs, are the result of the difference in the scope and workflow between SUITs and unit tests. Smells unique to SUITs typically address issues such as data management, test workflow, and interaction with the GUI.

For 16 of the SUIT smells we derive metrics to characterize the diffusion and removal of their symptoms in the test code. *Large projects* are prone to multi-location smells such as *Army of Clones*, *Long Test Steps* and *Middle Man* for which the symptom is spread across multiple location in the test code. On the contrary, with the exception of *Hardcoded Environment*, when testers and the literature agree on a smell, if its symptom is localized to a single line in the test code, practitioners are actively avoiding it. When a test is affected by a symptom, its suppression is far from systematic, with less than 50% of the symptomatic tests ever undergoing any refactoring actions for 8 of the 16 smells studied. Note that *Missing Assertion* is a unique exception with between 70% (open-source) and 90% (industrial) of the symptomatic tests being addressed. Interestingly, while smell-removing actions are rare, symptoms for smells such as *Narcissistic* and *Middle Man* still disappear from the test codebase as a side effect of the replacement of old symptomatic tests by new tests not exhibiting the symptom.

Though we observe general trends common to both industrial and open-source projects, when performing a statistical comparison between the two sets of projects, we observe significant differences. Indeed, the projects differ both in terms of prevalence of the symptoms and removing operations. These results can be explain by the difference in scope, actors, and lifecycle present in each context. Consequently, these results suggest that when conducting studies researchers should be aware of these fundamental differences which might limit the generalization of their results.

In light of the results from this exploratory analysis, we believe that extending the catalog of known SUIT smells and providing the necessary tooling to highlight them will have an impact on SUITs development process. Indeed, we observed a trend to avoid the introduction of smells by test automation engineers. However, if a smell is not addressed right away, practitioners tend not to remove it later on. Thus, it is of utmost importance to catch the bad practices as soon as they are introduced. Unfortunately, because some smells are harder to detect, they manage to creep in the test code base. As such, we advocate for continuous monitoring of the test suite with tools such as SonarQube that allow to flag and remove smell instances as soon as they are introduced.

For instance, our partner, BGL BNP Paribas, has already started using our tooling to address some of the bad design choices that we observed in their test codebase. The SonarQube Plugin that we developed is now being evaluated at their site with the goal of offering continuous quality evaluation of the test code base at each commit. The report generated are used to allow test automation engineers to keep an eye on the proliferation of anti-pattern in the test code base, and address them as soon as they are introduced. While at BGL BNP Paribas no quality gate is setup, the team is reviewing the quality reports on a regular basis. To do so, a Jenkins job launches the SonarQube runner once a week and keeps a quality report up to date. The report is accessible through the SonarQube web client. Test automation engineers analyze the report at regular intervals and address potential smells revealed by the reporting tool. While this practice has allowed the team to improve the quality of the test codebase through dedicated maintenance session, as suggested in Section 5, running the analysis directly in the development environment would allow the smells to be detected as they are introduced and potentially avoided altogether.

This study alongside with the tooling developed and the available dataset[10] lays the ground for future research on the impact of smells on SUIT suites. Moreover, with our new catalog and these first observations, we open perspective for future research on awareness of bad testing practices and the pitfalls to avoid when evolving SUITs. Our future agenda is focused on the use of Natural Language Processing to address the categories of smells linked to wording and expressiveness that is central in acceptance testing where different stakeholders from different domains need to communicate.

[10]https://github.com/kabinja/suit-smells-replication-package

References

Advolodkin N (2018) Top 17 automated testing best practices (supported by data). https://ultimateqa.com/automation-patterns-antipatterns/

Alegroth E, Steiner M, Martini A (2016) Exploring the presence of technical debt in industrial GUI-Based Testware: A case study. In: Proceedings of the 9th international conference on software testing, verification and validation workshops, IEEE, pp 257?262. https://doi.org/10.1109/ICSTW.2016.47

Arcelli FF, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21(3):1143–1191. https://doi.org/10.1007/s10664-015-9378-4

Archer M (2010) How test automation with selenium can fail. https://mattarcherblog.wordpress.com/2010/11/29/how-test-automation-with-selenium-or-watir-can-fail/

Athanasiou D, Nugroho A, Visser J, Zaidman A (2014) Test code quality and its relation to issue handling performance. IEEE Trans Softw Eng 40(11):1100–1125. https://doi.org/10.1109/TSE.2014.2342227

Baker P, Dai ZR, Grabowski J, Haugen O, Schieferdecker I, Williams C (2008) Data-driven testing. In: Model-driven testing. Springer, pp 87–95

Banerjee I, Nguyen B, Garousi V, Memon A (2013) Graphical user interface (GUI) testing: Systematic mapping and repository. Inf Softw Technol 55(10):1679–1694. https://doi.org/10.1016/j.infsof.2013.03.004

Basit W, Lodhi F, Ahmed F, Bhatti MU (2013) A metric based evaluation of unit tests as specialized clients in Refactoring. Pak J Eng Appl Sci 13:37–53

Battat M (2020) How do you simplify end-to-end test maintenance?. https://dzone.com/articles/how-do-you-simplify-end-to-end-tes t-maintenance-au

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2012) An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: Proceedings of the 28th IEEE international conference on software maintenance. IEEE, pp 56–65. https://doi.org/10.1109/ICSM.2012.6405253

Bavota G, Qusef A, Oliveto R, De Lucia A, Binkley D (2015) Are test smells really harmful? An empirical study. Empir Softw Eng 20(4):1052–1094. https://doi.org/10.1007/s10664-014-9313-0

Biagiola M, Stocco A, Ricca F, Tonella P (2019) Diversity-based web test generation. In: Proceedings of the 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ACM Press, Tallinn, Estonia, vol 1, pp 142-153. https://doi.org/10.1145/3338906.3338970

Bowes D, Hall T, Petric J, Shippey T, Turhan B (2017) How Good Are My Tests? In: Proceedings of the 8th workshop on emerging trends in software metrics.IEEE, pp 9–14. https://doi.org/10.1109/WETSoM.2017.2

Breugelmans M, Rompaey BV (2008) TestQ : Exploring structural and maintenance characteristics of unit test suites. In: Proceedings of the 1st international workshop on advanced software development tools and techniques, i, pp 1–16

Brooks P, Robinson B, Memon AM (2009) An initial characterization of industrial graphical user interface systems. In: Proceedings of the 2nd IEEE international conference on software testing, verification, and validation, pp 11–20. https://doi.org/10.1109/ICST.2009.11

Bushnev Y (2019) Top 15 ui test automation best practices. https://www.blazemeter.com/blog/top-15-ui-test-automation-best-practices-you-should-follow

Buwalda H (2015) Test design for automation: Anti-patterns. https://www.techwell.com/techwell-insights/2015/09/test-design-automation-anti-patterns

Buwalda H (2019) 8 test automation anti-patterns (and how to avoid them). https://dzone.com/articles/8-test-automation-anti-patterns-and-how-to-avoid-t

Canny A, Palanque P, Navarre D (2020) Model-based testing of GUI applications featuring dynamic Instanciation of widgets. In: Proceedings of the international conference on software testing Testing, verification and validation workshops, IEEE, pp 95–104. https://doi.org/10.1109/ICSTW50294.2020.00029

Chen WK, Wang JC (2012) Bad smells and refactoring methods for GUI test scripts. In: Proceedings of the 13th international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing pp 289–294. https://doi.org/10.1109/SNPD.2012.10

Clayton J (2014) Acceptance tests at a single level of abstraction. https://thoughtbot.com/blog/acceptance-tests-at-a-single-level-of-abstraction

Coppola R, Morisio M, Torchiano M (2019) Mobile GUI testing fragility: a study on open-source android applications. IEEE Trans Reliab 68(1):67–90. https://doi.org/10.1109/TR.2018.2869227

Cripsin L (2018) Keep your automated testing simple and avoid anti-patterns. https://www.mabl.com/blog/keep-your-automated-testing-simple

Cunha M, Paiva ACR, Ferreira HS, Abreu R (2010) PETTool: A pattern-based GUI testing tool. In: Proceedings of the 2nd IEEE international conference on software technology and engineering, IEEE, vol 1, pp 202?206. https://doi.org/10.1109/ICSTE.2010.5608882

De Bleser J, Di Nucci D, De Roover C (2019) Assessing diffusion and perception of test smells in scala projects. In: Proceedings of the 16th international conference on mining software repositories, IEEE, pp 457–467 https://doi.org/10.1109/MSR.2019.00072

Delin M, Foegen K (2016) An analysis of information needs to detect test smells. full-scale software engineering/current trends in release engineering, pp 19–24

van Deursen A., Moonen, van Den BA, Kok G (2001) Refactoring test code. In: Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering, pp 92–95

Dharmender K (2017) Automation testing: Anti-patterns. https://alisterbscott.com/2015/01/20/five-automated-acceptance-test-anti-patterns/

Di Martino S, Fasolino AR, Starace LLL, Tramontana P (2021) Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing. Softw Test Verif Reliab 31(3):272–285. https://doi.org/10.1002/stvr.1754

England T (2016) Cucumber anti-patterns (part one). https://cucumber.io/blog/bdd/cucumber-antipatterns-part-one/

Evangelisti A (2012) How to transform bad acceptance tests into awesome ones. https://mysoftwarequality.wordpress.com/2012/12/14/how-to-transform-bad-acceptance-tests-into-awesome-ones/

Femmer H, Fernández DM, Juergens E, Klose M, Zimmer I (2014) Rapid requirements checks with requirements smells: two case studies. In: Proceedings of the 1st international workshop on rapid continuous software engineering, ACM Press, New York, USA, pp 10–19. https://doi.org/10.1145/2593812.2593817

Femmer H, Fernández DM, Wagner S, Eder S (2017) Rapid quality assurance with requirements smells. J Syst Softw 123:190–213. arXiv:1611.08847. https://doi.org/10.1016/j.jss.2016.02.047

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: Improving The Design of Existing Code. Addison-Wesley Longman Publishing Co.,Inc., USA

Gao Z, Liang Y, Cohen MB, Memon AM, Wang Z (2015) Making System User Interactive Tests Repeatable: When and What Should We Control? In: Proceedings of the 37th International Conference on Software Engineering, IEEE, vol 1, pp 55–65, https://doi.org/10.1109/ICSE.2015.28

Garousi V, Kúćùk B (2018) Smells in software test code: A survey of knowledge in industry and academia. J Syst Softw 138:52–81. https://doi.org/10.1016/j.jss.2017.12.013

Gawinecki M (2016) Anti-patterns in test automation. http://nomoretesting.com/blog/2016/05/23/anti-patterns/

Goldberg Y (2019) Node.js and javascript testing best practices 2020. https://yonigoldberg.medium.com/yoni-goldberg-javascript-nodejs-testing-best-practices-2b98924c934

Gomez L, Neamtiu I, Azim T, Millstein T (2013) RERAN: Timing- and touch-sensitive record and replay for android. In: Proceedings of the 35th international conference on software engineering, IEEE, pp 72–81. https://doi.org/10.1109/ICSE.2013.6606553

Grano G, Palomba F, Di Nucci D, De Lucia A, Gall HC (2019) Scented since the beginning: On the diffuseness of test smells in automatically generated test code. J Syst Softw 156:312–327. https://doi.org/10.1016/j.jss.2019.07.016

Greiler M, Van Deursen A, Storey MA (2013) Automated detection of test fixture strategies and smells. In: 2013 IEEE Sixth international conference on software testing,verification and validation, IEEE, pp 322–331. https://doi.org/10.1109/ICST.2013.45

Gupta P, Surve P (2011). In: Proceedings of the 1st ACM international workshop on end-to-end test script engineering, ACM Press, New York, USA, pp 1–7.https://doi.org/10.1145/2002931.2002932

Hall MW, Kennedy K (1992) Efficient call graph analysis. ACM Trans Program Lang Syst 1(3):227–242. https://doi.org/10.1145/151640.151643

Hammoudi M, Rothermel G, Stocco A (2016a) WATERFALL: an incremental approach for repairing record-replay tests of web applications. In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, ACM, New York, NY, USA, vol 13-18-Nove, pp 751–762. https://doi.org/10.1145/2950290.2950294

Hammoudi M, Rothermel G, Tonella P (2016b) Why do Record/Replay Tests of Web Applications Break? In: Proceedings of the international conference on software testing, verification and validation, IEEE, pp 180–190

Hanssen G, Yamashita AF, Conradi R, Moonen L (2010) Software entropy in agile product evolution. In: Proceedings of the 43rd Hawaii international conference on system sciences, IEEE, 2, pp 1–10. https://doi.org/10.1109/HICSS.2010.344. http://ieeexplore.ieee.org/document/5428534/

Hauptmann B, Heinemann L, Vaas R, Braun P (2013) Hunting for smells in natural language tests. In: Proceedings of the 35th international conference on software engineering, IEEE, San Francisco, CA, USA, pp 1217-1220. https://doi.org/10.1109/ICSE.2013.6606682

Hauptmann B, Eder S, Junker M, Juergens E, Woinke V (2015) Generating Refactoring Proposals to remove clones from automated system tests. In: Proceedings of the 23rd international conference on program comprehension, IEEE, vol 2015-August, pp 115–124. https://doi.org/10.1109/ICPC.2015.20

Humble J, Farley DG (2010) Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Upper Saddle River, NJ

Hurdugaci V, Zaidman A (2012) Aiding software developers to maintain developer tests. In: Proceedings of 16th european conference on software maintenance and Reengineering, IEEE, pp 11–20

Issa A, Sillito J, Garousi V (2012) Visual testing of graphical user interfaces: an exploratory study towards systematic definitions and approaches. In: Proceedings of the international symposium on web systems evolution, IEEE, pp 11–15. https://doi.org/10.1109/WSE.2012.6320526

Jain N (2007) Patterns and anti-patterns: Acceptance testing with fitnesse. https://blogs.agilefaqs.com/2007/08/25/patterns-and-anti-patterns-acceptance-testing-with-fitnesse/

Tang J, Cao X, Ma A (2008) Towards adaptive framework of keyword driven automation testing. In: Proceedings of the IEEE international conference on automation and logistics, IEEE, September, pp 1631–1636. https://doi.org/10.1109/ICAL.2008.4636415

Kapelonis K (2018) Software testing anti-patterns. http://blog.codepipes.com/testing/software-testing-anti-patterns.html

Katalon (2018) The most striking problems in test automation : A survey. Tech Rep May, Katalon

Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. In: Proceedings of the 9th international conference on quality software. IEEE Comput Soc, USA, pp 305–314. https://doi.org/10.1109/QSIC.2009.47

Kim DJ (2020) An empirical study on the evolution of test smell. In: Proceedings of the 42nd international conference on software engineering: companion proceedings, ACM, New York, NY, USA, i, pp 149–151.https://doi.org/10.1145/3377812.3382176

Kirinuki H, Tanno H, Natsukawa K (2019) COLOR: correct locator recommender for broken test scripts using various clues in web application. In: Proceedings of the 26th international conference on software analysis, evolution and Reengineering, IEEE, vol 36, pp 310–320

Kirkbride J (2014) Testing anti-patterns. https://medium.com/jameskbride/testing-anti-patterns-b5ffc1612b8b

Kitchenham BA, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Tech. Rep. EBSE 2007-001, Keele University and Durham University Joint Report

Klarck P (2014) Robot framework dos and don'ts. https://slideshare.net/pekkaklarck/robot-framework-dos-and-donts

Knight A (2017a) Bdd 101: Writing good gherkin. https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/

Knight A (2017b) Should gherkin steps use first-person or third-person? https://automationpanda.com/2017/01/18/should-gherkin-steps-use-first-person-or-third-person/

Knight A (2019) Bdd 101: Writing good gherkin. https://techbeacon.com/app-dev-testing/7-ways-tidy-your-test-code

Labuschagne A, Inozemtseva L, Holmes R (2017) Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In: proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp 821–830. https://doi.org/10.1145/3106237.3106288

Lanubile F, Mallardo T (2007) Inspecting automated test code: a preliminary study. In: Agile processes in software engineering and extreme programming, vol 4536, LNCS, Springer Berlin Heidelberg, pp 115–122. https://doi.org/10.1007/978-3-540-73101-6_16

Lelli V, Blouin A, Baudry B, Coulon F (2015) On model-based testing advanced GUIs. In: Proceedings of the 8th international conference on software testing, verification and validation workshops, IEEE, pp 1–10. https://doi.org/10.1109/ICSTW.2015.7107403

Leotta M, Stocco A, Ricca F, Tonella P (2014) Reducing web test cases aging by means of robust XPath locators. In: Proceedings of the IEEE international symposium on software reliability engineering workshops, IEEE, pp 449–454

Leotta M, Stocco A, Ricca F, Tonella P (2016) Robula+: an algorithm for generating robust XPath locators for web testing. J softw: Evol Process 28(3):177–204

Mabl (2021) Benchmark report : the state of testing in DeVops

Machiry A, Tahiliani R, Naik M (2013) Dynodroid: an input generation system for Android apps. In: Proceedings of the 9th joint meeting on foundations of software engineering, ACM Press, New York, USA, pp 224. https://doi.org/10.1145/2491411.2491450

Mao K, Harman M, Jia Y (2016) Sapienz: multi-objective automated testing for android applications. In: Proceedings of the 25th international symposium on software testing and analysis, ACM, New York, NY, USA, pp 94–105. https://doi.org/10.1145/2931037.2931054

Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE international conference on software maintenance, IEEE, pp 350–359. https://doi.org/10.1109/ICSM.2004.1357820

Memon AM, Nguyen BN (2010) Advances in automated model-based system testing of software applications with a GUI Front-End. In: Advances in computers, vol 80, 1st edn, Elsevier Inc., pp 121–162. https://doi.org/10.1016/S0065-2458(10)80003-8

Mesbah A, Van DA (2009) Invariant-based automatic testing of AJAX user interfaces. In: Proceedings of the 31st international conference on software engineering, IEEE, pp 210–220. https://doi.org/10.1109/ICSE.2009.5070522

Meszaros G (2007) xUnit Test Patterns: Refactoring Test Code, 1st edn. Addison-Wesley

Morlion P (2019) Software testing anti patterns. https://www.enov8.com/blog/software-testing-anti-patterns/

Myers B (1994) Challenges of HCI design and implementation. Interactions 1(1):73–83. https://doi.org/10.1145/174800.174808

Myers BA (1995) User Interface Software Tools. ACM Trans Comput Hum Interact 2(1):64–103. https://doi.org/10.1145/200968.200971

Myers BA, Rosson MB (1992) Survey on user interface programming. In: Proceedings of the conference on Human factors in computing systems, ACM Press, New York, USA, pp 195–202. https://doi.org/10.1145/142750.142789

Nguyen BN, Robbins B, Banerjee I, Memon A (2014) GUITAR: an innovative tool for automated testing of GUI-driven software. Autom Softw Eng 21(1):65–105. https://doi.org/10.1007/s10515-013-0128-9

Nguyen V, To T, Diep GH (2021) Generating and selecting resilient and maintainable locators for Web automated testing. Softw Test Verif Reliab 31(3):19. https://doi.org/10.1002/stvr.1760

Pandit P, Tahiliani S (2015) Agile UAT: a framework for user acceptance testing based on user stories and acceptance criteria. Int J Comput Appl 120(10):16–21. https://doi.org/10.5120/21262-3533

Peruma A, Almalki K, Newman CD, Mkaouer MW, Ouni A, Palomba F (2020) tsDetect: an open-source test smells detection tool. In: Proceedings of the 28th joint meeting on European software engineering conference and symposium on the foundations of software engineering, ACM, New York, NY, USA, pp 1650-1654. https://doi.org/10.1145/3368089.3417921

Pezzė M, Rondena P, Zuddas D (2018) Automatic GUI testing of desktop applications. In: Companion proceedings for the ISSTA/ECOOP 2018 Workshops, ACM, New York, NY, USA, i, pp 54–62. https://doi.org/10.1145/3236454.3236489

Reichhart S, Girba T, Ducasse S (2007) Rule-based assessment of test quality. J Object Technol 6(9):231. https://doi.org/10.5381/jot.2007.6.9.a12

Renaudin J (2016) Software testing anti patterns. https://www.slideshare.net/JosiahRenaudin/antipatterns-for-automated-testing

Ricca F, Stocco A (2021) Web test automation: insights from the grey literature. In: Proceedings of the 47th international conference on current trends in theory and practice of computer science, Bolzano,Italy, pp 472-485. https://doi.org/10.1007/978-3-030-67731-2_35

Ronsse M, De Bosschere K (1999) RecPlay: A fully integrated practical record/replay system. ACM Trans Comput Syst 17(2):133–152. https://doi.org/10.1145/312203.312214,

Rwemalika R, Kintis M, Papadakis M, Le Traon Y, Lorrach P (2019a) On the evolution of keyword-driven test suites. In: Proceedings of the 12th international conference on software testing, verification and validation, New York, NY, USA, pp 335-345

Rwemalika R, Kintis M, Papadakis M, Le Traon Y, Lorrach P (2019b) Ukwikora: continuous inspection for keyword-driven testing. In: Proceedings of the 28th international symposium on software testing and analysis, association for computing machinery, New York, NY, USA, pp 402–405. https://doi.org/10.1145/3293882.3339003

Saddler JA, Cohen MB (2017) EventFlowSlicer: A tool for generating realistic goal-driven GUI tests. In: Proceedings of the 32nd international conference on automated software engineering, ASE, IEEE, pp 955–960. https://doi.org/10.1109/ASE.2017.8115711

Salihu IA, Ibrahim R, Ahmed BS, Zamli KZ, Usman A (2019) AMOGA: A static-dynamic model generation strategy for mobile Apps testing. IEEE Access 7(c):17158–17173. https://doi.org/10.1109/ACCESS.2019.2895504

Satopaa V, Albrecht J, Irwin D, Raghavan B (2011) Finding a kneedle in a haystack: detecting knee points in system behavior. In: Proceedings of the 31st IEEE international conference on distributed computing systems workshops, IEEE, pp 166–171. https://doi.org/10.1109/ICDCSW.2011.20

Sciamanna A (2019) What are the anti patterns of automation with selenium? https://anthonysciamanna.com/2019/10/20/avoiding-automated-testing-pitfalls.html

Scott A (2015) Five automated acceptance test anti-patterns. https://alisterbscott.com/2015/01/20/five-automated-acceptance-test-anti-patterns/

Shay L (2019) Bdd cucumber features best practices. https://www.linkedin.com/pulse/bdd-cucumber-features-best-practices-liraz-shay/

Sheth H (2020) 16 selenium best practices for efficient test automation. https://www.lambdatest.com/blog/selenium-best-practices-for-web-testing/

Silva D, Valente MT (2017) RefDiff: detecting Refactorings in version histories. In: Proceedings of the 14th international conference on mining software repositories, IEEE, pp 269–279. arXiv:1704.01544. https://doi.org/10.1109/MSR.2017.14

Siminiuc A (2019) What are the anti patterns of automation with selenium? https://www.quora.com/What-are-the-anti-patterns-of-automation-with-selenium

Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: Proceedings of the international conference on software maintenance and evolution, IEEE, pp 1–12. https://doi.org/10.1109/ICSME.2018.00010

Spadini D, Schvarcbacher M, Oprescu AM, Bruntink M, Bacchelli A (2020) Investigating severity thresholds for test smells. In: Proceedings of the 17th international conference on mining software repositories, ACM, New York, NY, USA, pp 311–321. https://doi.org/10.1145/3379597.3387453

StackExchange (2017) What are anti-patterns in test automation? https://sqa.stackexchange.com/questions/8508/what-are-anti-patterns-in-test-automation

Temov J (2020) Want to speed end-to-end testing? don't send in the clones. https://techbeacon.com/app-dev-testing/want-speed-end-end-testing-dont-send-clones

Tsantalis N, Guana V, Stroulia E, Hindle A (2013) A multidimensional empirical study on Refactoring cctivity. In: Proceedings of the conference of the center for advanced studies on collaborative research, IBM Corp, Ontario, Canada, pp 132–146

Tufano M, Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D (2016) An empirical investigation into the nature of test smells. In: Proceedings of the 31st international conference on automated software engineering, ACM Press, Singapore, pp 4–15. https://doi.org/10.1145/2970276.2970340

Utting M, Pretschner A, Legeard B (2012) A taxonomy of model-based testing approaches. Software Testing Verification and Reliability 22(5):297–312. https://doi.org/10.1002/stvr.456

Van RB, Du Bois B, Demeyer S (2006) Characterizing the relative significance of a test smell. In: Proceedings of the 22nd IEEE international conference on software maintenance, IEEE, pp 391–400. https://doi.org/10.1109/ICSM.2006.18

Van RB, Du Bois B, Demeyer S, Rieger M (2007) On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans Soft Eng 33(12):800–817. 10.1109/TSE.2007.70745

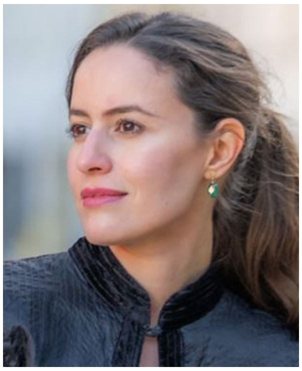
Virginio T, Martins LA, Soares LR, Santana R, Costa H, Machado I (2020) An empirical study of automatically-generated tests from the perspective of test smells. In: Proceedings of the 34th Brazilian symposium on software engineering, ACM, New York, NY, USA, pp 92–96.https://doi.org/10.1145/3422392.3422412

Yu S, Fang C, Feng Y, Zhao W, Chen Z (2019) LIRAT: layout and image recognition driving automated mobile testing of cross-platform. In: Proceedings of the 34th international conference on automated software engineering, IEEE, pp 1066–1069 https://doi.org/10.1109/ASE.2019.00103

Yuan X, Cohen M, Memon AM (2007) Covering array sampling of input event sequences for automated gui testing. In: Proceedings of the 22nd international conference on automated software engineering, ACM Press, New York, USA, pp 405

Zhang S, Jalali D, Wuttke J, Muşlu K, Lam W, Ernst MD, Notkin D (2014) Empirically revisiting the test independence assumption. In: Proceedings of the international symposium on software testing and analysis, ACM Press, New York,USA, pp 385–396. https://doi.org/10.1145/2610384.2610404

Renaud Rwemalika is a Research Associate at the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) at the University of Luxembourg. His career interests revolved around software quality, technical debt in system testing, static test code analysis, and generally test maintenance define his career interest. He earned his Ph.D. from the University of Luxembourg with a specialization in software engineering in September 2021.



Sarra Habchi is a Research and Development Scientist at La Forge, Ubisoft Canada. Her research interests lie primarily in the area of software quality and reliability. She received her Ph.D. from the University of Lille, France.

Mike Papadakis is a senior research scientist at the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) at the University of Luxembourg. He received a Ph.D. in Computer Science from the Athens University of Economics and Business. His research interests include software testing, static analysis, prediction modelling and search-based software engineering. He is best known for his work on Mutation Testing for which he has been awarded an IEEE TCSE Rising Star Award 2020.

Yves Le Traon is a professor at the University of Luxembourg where he leads the SERVAL (SEcurity, Reasoning, and VALidation) research team. His research interests include (1) innovative testing, debugging and repair techniques, (2) mobile security using static code analysis, machine learning techniques and, (3) design of robust machine-learning based systems. His research is inspired from and applies to several industry partners (IoT, Fintech, Smartgrid, Industry 4.0). He was elevated IEEE Fellow in 2022.

Marie-Claude Brasseur is the Manager of the quality assurance team at BGL BNP Paribas, Luxembourg. She is interested in the continuous improvement and monitoring of end-to-end tests deployed in a cloud environment.