UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2021-062
The Faculty of Sciences, Technology and Medecine

Dissertation

Defence held on September 16th, 2021 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Renaud Rwemalika
Born on 17th April 1987 in Bujumbura (Burundi)

On the Maintenance of System User Interactive Tests

Abstract

Many companies rely on software testing to verify that their software products meet their requirements. As such, software testing is a keystone of the quality process. It offers means to assess that both functional and technical requirements are met. However, scaling the number of tests while preserving their quality poses challenges, just as in any software artifact . This is particularly true in the case of complex tests interacting with the System Under Test (SUT) through its Graphical User Interface (GUI), i.e. System User Interactive Test (SUIT). The problem becomes challenging because as the SUT evolves, the SUIT suites need to adapt and conform to the evolved software. Indeed, because the user interface is a part of the system that tends to undergo rapid evolutions, SUITs are particularly prompt to break.

As the first contribution this dissertation, we aim to demonstrate the problem of test maintenance and overall improve the understanding of SUIT scripts evolution. To that end, we identify, collect and analyze test code changes across the evolution of an industrial test suite. We show that the problem of test maintenance is largely due to test fragility (most commonly performed changes are due to locator and synchronization issues) and bad practices (over 30% of keywords are duplicated).

To further investigate the question of bad test code practices such as test clones, we perform a multivocal study to identify which bad practices, i.e. SUIT smells, are already studied in both industry and academia. This process yields a catalog of 35 test code smells. For 16 of them, we derive metrics to analyze their diffusion across tests as well as potential refactoring actions removing the test code smells. Through an empirical study including both industrial and open-source test suites, we show that test code smells are largely present in SUITs, potentially contributing to SUIT fragility and hindering the maintenance process. Interestingly, refactoring actions remain rare during the lifespan of the tests. Yet, symptoms tend to disappear as old tests are discarded and new ones are introduced.

However, during the analysis of SUIT smells, we observe that bad practices impacting locators do not appear often in the test code. This observation contrasts with the analysis of the evolution of the SUIT presented in our first empirical study. This apparent contradiction arises from the limitation of property-based locators which rely on the structure of the representation layer of the SUT. Indeed, such approaches are sensitive to internal iso-functional changes occurring during the normal evolution of the SUT. To account for this limitation, we introduce a new way of expressing locators, HPath. Instead of relying on the internal representation of the presentation layer, HPath relies on the rendered characteristics of the GUI. Our results suggest that despite what is presented in the literature on locators, expressions relying on a smaller number of GUI elements to fully qualify a target do not necessarily lead to more robust locators. On the contrary, the choice of the GUI element properties seems to play a stronger role in the robustness to change.

Overall, this dissertation provides insights into how SUITs evolve and shows that SUIT fragility plays a major role in the associated maintenance effort. It also proposes techniques that effectively facilitate SUIT maintenance with the early detection of sub-optimal patterns and the introduction of a locator representation more robust to SUT evolution.

Acknowledgments

Throughout this thesis, I have received a lot of support and assistance.

First of all, I would like to thank Pr. Yves Le Traon who allowed me to pursue my PhD studies in his group and under his supervision. He believed in my research and provided me with valuable feedback. I am also grateful for the opportunities of teaching and the confidence he placed in me when organizing academic activities.

Of all the people in my thesis, I am especially grateful to my co-supervisors Mike Papadakis and Marinos Kintis for their patience, their advice, their training, as well as their support. They taught me how to conduct proper research and present my results to others. The frequent discussions we had allowed me to better understand the field, but also to become a better researcher altogether.

I am also thankful to my co-authors for their efforts and feedback that contributed to making this thesis stronger. I also want to thank all jury members for their interest in my research and the time invested in my dissertation.

I would like to express my gratitude to all my colleagues from SERVAL (SnT) for all the good discussions we had but also for their support during the darker times of my thesis.

I am thankful to the National Research Fund (FNR) of Luxembourg for trusting in the initial ideas that lead to this dissertation by granting the funding that supported my thesis. Equally, I would like to thank BGL BNP Paribas, and especially the test factory team, who was at my side for four years and offered me feedback and continuous trust.

Finally and more personally, I thank my partner Oksana Stolietnia, for her love, constant support, and patience during my thesis. I also thank my mother and my sister for their unconditional support and help throughout this adventure.

Contents

1 Introduction

With GUI-based applications becoming ubiquitous as a means for users to interact with software, the question of their correctness is of utmost importance. Yet, testing such applications still poses major challenges to practitioners. After introducing four approaches employed when creating SUIT, we introduce the challenges that come with them. Finally, we present the three contributions of this dissertation and scope the sub-challenges they tackle.

Contents

Today, companies need to produce software faster than ever in order to remain competitive. To do so, they rely mainly on two different strategies: the integration of a continuous deployment pipeline and the establishment of agile methodologies within the company. These strategies allow software producers to increase the release rate while maintaining a high quality for the software that is shipped.

One of the keystones of the quality process is based on the adoption of strategies to assess that both functional and technical requirements are met. Because of the complex dynamic runtime of most modern applications, static analysis techniques face limitations in revealing violations of the requirements. Thus, most teams heavily rely on testing to reveal faults in their software.

We define testing as a technical investigation to assess the quality of a SUT by executing the latter against a set of input values. Tests can further be classified through the scope they target into three main categories: unit, integration, and system tests. Unit testing covers the tests that are testing a unit in isolation, be it a class, a function, or any other software component, to validate each unit individually. In integration testing, related units are grouped together and tested as a group. Finally, system testing covers the tests evaluating the system as a whole to ensure its compliance with its requirements.

Because unit and integration tests target more restricted portions of the application, they offer better control to the tester who can isolate functionalities from its environment and tests specific behaviors of the system sub-parts. On the other hand, system tests, and especially those interacting with the user interface, target the system as a whole (usually as a black box). Therefore, they are sensitive to any change in the SUT and its interface. This property makes system tests more fragile, *i.e.* they may break as a result of non-functional changes that are not targeted by the test.

In this work, we analyze the evolution of those system tests interacting with the user interface to shed light on the reasons why these tests require costly maintenance [AFK16; CAM+20]. Then, we propose some solutions as to how to create more robust test suites, *i.e.* test suites that are less likely to break following non-functional evolutions of the SUT.

1.1 GUI Testing

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, [Mye94]. Indeed, as depicted by Myers [Mye95], user interfaces require the system to interact with complex graphical components, to provide multiple ways to provide the same command, to manage multiple asynchronous input devices through the use of asynchronous event loops, or to provide an interactive computer programming environment (*e.g.* read–evaluate–print loop).

To ease development, the literature introduces different architectural designs to decouple the code responsible for the user interaction from the rest of the system such as the Entity-Component-System model [RH18] or the Model-View-Controller [KP88].

Nevertheless, while these architectural solutions offer advantages by providing better isolation when testing individual components, in the case of system testing, the challenges remain open. Indeed, as the complexity of user interfaces and their associated code continues to grow, testing through the user interface for functional correctness remains challenging, but vital to help ensure the safety, robustness, and usability of the SUT.

Furthermore, today, GUI are becoming ubiquitous as a means for users to interact with a software application [MR92; Mye95; BRM09; MN10]. In this work, we define a GUI as a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events that result in graphical output [Mem07] and/or alter the state of the software [NRB+14]. A GUI-based application is a software for which the GUI is the main mode of interaction to the detriment of other types of user interfaces. As a consequence of the predominance of GUI-based applications, a lot of effort has been geared towards GUI automation, notably focusing on desktop applications [NRB+14; Adv18; PRZ18], web applications [MvD09; BSR+19], mobile applications [MTN13; GNA+13; MHJ16; SIA+19; YFF+19] or cross-platform applications [CPN20]. However, as we show later in this section, major challenges remain open.

The question of the correctness of a GUI-based application is of utmost importance since prototyping and developing the behavior of a system is harder than prototyping and developing the appearance of the application [MPN+08]. It is in this context that we introduce GUI testing which encompasses the techniques used to test user interface layout and its behavior. More precisely, GUI testing can be defined as "the process of testing a software application with a graphical front-end to guarantee that it meets its specification by performing a sequence of events against the GUI elements" [CPF+10; BNG+13; ISG12]. Thus, relying only on stimuli inputs and output reactions, a GUI test is considered as a black-box test. Additionally, we define a SUIT as an automated GUI test that exercises the SUT as a whole (system level). Note that in this work focusing on automated system-level test interacting with the SUT through its GUI, we use GUI test and SUIT interchangeably.

Strategies have been developed to exercise the user interface to ensure that functional requirements are satisfied. Different techniques have been proposed but all rely on the same principle. A test framework executes events belonging to GUI components and monitors the resulting changes to the program state [NRB+14]. In other words, each technique generates a sequence of test steps, that will be executed against the SUT (input) and captures some indication as to what is the current state of the system (output). Each test step consists of a triple: (1) an action to be performed, (2) the GUI element on which to perform the action, and (3) an optional value passed to the element. Consequently, GUI test generation can be classified using the way the sequence of test steps is generated. To do so, we use existing classification frameworks proposed by the literature [ABC+13; AAM+17] to reframe them in the context of GUI test generation and its maintenance.

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

1.1.1 Random GUI Testing

In random GUI testing, also known as monkey testing or random-walk, tests are automatically created by generating sequences of events at random in the hope of exploring the SUT to reveal failures. Because no requirements are passed as input such techniques rely on specific constraints that are application or domain invariant oracles [MvD09] to assess correctness. Thus, any sequence of events that causes the SUT to violate an invariant is considered as a fault revealing test case [BHM+15]. These properties make random GUI testing cheap to run on a large number of versions/configurations of applications.

Typically, such tools rely on the concept of fuzz testing which performs arbitrary or contextual events against the SUT. Tools like Monkey [Goo20], ATUSA [MvDR12] or Dynodroid [MTN13] rely on pseudo-random input events which include both user interface events and system events and assess the generated sequences against invariant oracles [AFT11].

One of the major challenges encountered when relying on random GUI testing resides in the exploration of the application states. Indeed, with modern applications, the infinite number of combinations of user inputs leads to a potentially large and sometimes infinite number of reachable states. Consequently, the search space cannot be covered exhaustively leading to potentially low state coverage [CBM+19]. Furthermore, in the case of more complex input, such as a chain of character, randomly generated input have often low chances of producing feasible test cases, thus requiring a large number of input generations to find interesting test cases (*e.g.* if a login page exists, only a combination of a valid user name and password would allow the test case to continue its exploration behind the login wall).

To alleviate these limitations, different approaches emerged introducing algorithms relying on heuristics to better explore the application state space and consequently generate more fault-revealing test cases. For example, EXSYST [GFZ12], EvoDroid [MMM14] and Sapienz [MHJ16] propose to use search-based algorithms to explore the space of possible sequences and generate tests for Android applications while Dynodroid [MTN13] exploits machine learning algorithms to achieve the same goal. The goal of the algorithm is to generate short sequences of input while improving coverage (code [GFZ12] and/or state [MTN13] coverage) to produce fault-revealing sequences. More recently, tools such as DIG [BSR+19] try to increase the coverage by specifically targeting the input generation problem. Finally, we can mention approaches that rely on behavioral invariants (*e.g.* ALARic [RAF18] builds upon the Android Activity lifecycle) to explore the space and find faults. While these approaches look promising, today, monkey testing requires the user to provide some inputs (*e.g.* login and passwords) to allow the algorithms to explore more states and only manages to partially cover the search spaces.

To summarize, 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. Nevertheless, because of the lack of knowledge of the functional requirements and the lifecycle of the application, exploring the application and covering hidden behaviors remains challenging. Thus, while companies do use random GUI testing, they use it in addition to other techniques that we present below.

1.1.2 Model-Based Testing

Model-Based Testing (MBT) is an approach encompassing the processes and techniques for the automatic derivation of abstract test cases from abstract models [UPL12]. Functional specifications of a system, with the assumption that they are precise enough, are used as input to design process models which in turn are used to generate automated test cases [GS11]. This process can be devised in the five below steps [UPL12]:

1. A model of the SUT is built from informal requirements or existing specification documents;

2. Selection criteria are chosen, to guide the automatic test generation;

3. Test selection criteria are transformed into test case specifications;

4. A set of test cases is generated with the aim of satisfying all the test case specifications;

5. Finally, the test cases are executed against the SUT.

Unfortunately, most of the current GUI-based applications are not derived from a model, thus, 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 [LBB$^+$15]. This limitation causes MBT to offer limited applicability which explains why practitioners shy away from the model-based solutions proposed in the scientific literature.

To tackle this shortcoming, researchers propose methods to automatically derive a model of the application by reverse-engineering [DFP$^+$02; MBN03; AAF$^+$15; CPN20]. This field, also referred to as Model Learning, relies on similar techniques to the ones presented in Section 1.1.1, when discussing Random GUI exploration. For example, Memon [Mem07] introduces an event-flow model to formally represent events and event interactions in a GUI-based application. However, because of the large space of events and interactions present in a given application, manually building the model is prohibitive. Thus, the author proposes to rely on reverse-engineering to automatically derive the models by automatically exploring applications (in a similar fashion as random GUI testing) in a process called GUI ripping [MBN03] using tools such as AndroidRipper [AFT$^+$12]. Unfortunately, models built that way do not guarantee completeness, leading to difficulties such as assessing if the coverage offered by the model is sufficient or choosing inputs during exploration. While the literature offers partial solutions such as improving the input generation [BSR$^+$19] by eventually relying on coverage [YCM07], or new ways to derive models from the GUI and its requirements [CPN20] the question of automatic exploration of modern GUI-based application remains an open problem.

1.1.3 Record & Replay

This leads us to the third category of GUI testing, Record & Replay which generates test cases from sequences of user interactions provided by the tester and not by exploring the model in a random fashion (Random GUI Testing) or by relying on a model (Model-Based Testing). As its name suggests, Record & Replay works in two phases: a record phase and a replay phase:

1. During the record phase, the tester manually interacts with the application, thereby generating events on the SUT. The tool records the interactions and a part of the SUT response state as specified by the tester.

2. During the replay phase, the recorded test cases can be replayed on subsequent versions of the SUT and the captured state of the application are used as test oracles.

The generated test cases require human intervention during the capture phase and can be rerun, reducing the overall effort of regression testing. Furthermore, because the test cases are generated by the framework, no particular skills are needed from the tester. Di Martino et al. [DFS$^+$21] 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 [HRT16]. Indeed, whenever the application naturally evolves, the generated test cases might break, requiring testers to regenerate the test. This limitation can be exacerbated by agile and continuous delivery practices where applications and their user interfaces are under constant evolution.

To address this drawback, the research community proposes different ways to tackle the problem of fragility. Some attempts have been geared towards generating more robust test cases [HRS16]. Other researchers focus their attention on the more efficient way to record the interaction [RD99]. Finally, instead of recording the interactions as sequences of actions, tools such as EventFlowSlicer [SC17] generate a model from these interactions. Thus, similarly to MBT, subsequent versions are tested against the model. Finally, Amalfitano et al. [ARA$^+$19] take the idea of relying on MBT further by allowing the tool to perform exploration of the GUI.

Therefore, we see that while the generation of test cases using Capture & Replay is efficient, the approach suffers from drawbacks in terms of the maintainability of the generated test suite. 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 such tests.

1.1.4 Test Scripting

Finally, the last technique covered in the section deals with tests that are manually crafted by testers themselves. Following the general definition, a test script can be defined as a piece of executable code that executes a test case against the SUT and generates a verdict. In this work, we focus on test scripts interacting with the SUT through their GUI, namely, SUIT scripts. SUIT scripts aim at separating test design from the technical implementation of the tests.

Indeed, 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 [PT15]. 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 [HF10]. Hence, acceptance tests are not concerned with the internal implementation of the SUT, but with its overall behavior. 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.

Practitioners propose design patterns to separate different concerns into different abstraction layers. For example, Humble and Farley [HF10] propose the three following layers adopted by open-source and commercial tools like Robot Framework [Rob20], Cucumber, JBehave, Finess, or TestComplete:

- **Acceptance Criteria.** It contains functional behavior, quality characteristics, scenarios, or business rules that the test is targeting. These are usually written in a form close to natural language. The goal of this layer is to express for all the stakeholders the requirements that are being automatically executed.

- **Test Implementation Layer.** It contains all the underlying implementation of the test called by the acceptance criteria layer. Because test implementations that refer directly to the application GUI elements are fragile, it is not uncommon to see large portions of acceptance test suites break when a single GUI element changes. While deferring the interactions with the application to the application driver layer does not make tests more robust to changes, it will make their maintenance easier with a better separation of concerns. Indeed, this layer follows the logic and vocabulary of the SUT, instead of the technical details of the implementation with its GUI elements.
- **Application Driver Layer.** It is the layer that understands how to interact with the SUT. While the two other layers use only the domain language of the business, the application driver layer is expressed in the domain language used by the drivers that communicate with the SUT. One of the consequences of a well-designed application driver layer is to improve test reliability. Moreover, because of the high degree of reuse that is implicit when relying on an application driver layer, complex interactions and operations can be written once and used in many tests.

One approach commonly used in industry that promotes this architecture is Keyword-Driven Testing (KDT). Indeed, KDT advocates that a separation of concerns allows tests to be written easier, helps to limit code duplication, and enables experts from different fields and backgrounds to work together, at different levels of abstraction, for the creation of the tests.

To do so, KDT [JXM08; HWZ12] aims at separating test design from technical implementation. Its goal is to limit the exposure to unnecessary details and avoiding duplication. Figure 1.1 shows an example of a KDT test. This test, named "Valid Login" (line 5, adopted from the official documentation of Robot Framework), is responsible for validating the correct behavior of the login form in an imaginary SUT. Lines 6–8 present the "steps" of the tests and, in KDT parlance, they are calls to *keywords*. In turn, these keywords are defined in the respective definition blocks between lines 10 and 28. Each keyword is itself decomposed into a series of steps. Keywords can have *arguments*. For instance, keyword "Open browser" (line 12) takes two arguments, "${LOGIN URL}" and "${BROWSER}". The use of arguments to call keywords allows to further extend the reuse of keywords.

As can be seen from the figure, most parts of this fully automated test are written in plain English. This enables unobstructed collaboration between different experts while creating a test. For instance, a business analyst can write the high-level part of the test (lines 4–8) and an automation expert can implement the

Figure 1.1: Example of Robot Framework test

```
*** 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 Text    username_id    ${username}
    Input Text    password_id    ${password}
    Click Button    validate_id

Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    0
    Login Page Should Be Open

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

Welcome Page Should Be Open
    Title Should Be    Welcome Page

Login Page Should Be Open
    Title Should Be    Login Page

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

remaining part of the test (lines 10-35), adding the technical details to automate the steps.

However, while manually generated test scripts present advantages in terms of communication, compared to the other techniques presented, they present a major drawback: they are expensive to create and incur a high maintenance cost as the system they are testing naturally evolves, especially when good practices are not respected.

1.2 Challenges of System User Interactive Tests

As depicted in Section 1.1 while different GUI testing techniques exist, they all come with advantages and limitations. When addressing the case of fully automated techniques such as random GUI testing and to a certain extent MBT, exploring the GUI space remains an open challenge. Thus, practitioners while relying, on these approaches tend to complete the test suites with manual test generation.

Unfortunately, manual test generation comes with its challenges as well. Indeed, because the GUI layer exhibits a lot of variation across versions and configurations [GLC+15], 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 [CMT19; DFS+21].

Better understanding why GUI test scripts break is crucial to understanding how to limit their fragility. The fragility of the test code is measured by its propensity to break following minor changes in the production code not affecting its functionalities [GF16; CMT19]. In the previous definition, by minor changes, we understand changes not presenting any trouble to a human. Moreover, we define a test breakage as the event that occurs when the test raises exceptions or errors that do not pertain to the presence of a bug [SYM18]. For example, a test exercising the login behavior of a system should not be impacted by the color of the `Login` button or its internal `id`.

Therefore, to create more robust tests, we first need to understand which are the properties that cause them to break, be it in their structure, the way they interact with the SUT or how they synchronize with it. By isolating these properties and the invariants present in the SUT onto which tests can rely upon research can suggest good practices and tools to help increase test robustness. As such, finding those properties and invariants is a prerequisite if we want to be able to introduce the use of SUITs at scale in fast pace development processes.

1.3 Overview of the Contribution and Organization of the Dissertation

This section presents the contributions of this dissertation to address the aforementioned challenges encountered when writing SUIT scripts as well as the organization of this dissertation.

1.3.1 Contributions

Following are the contributions of this dissertation:

- **An empirical study on the evolution of KDT in an industrial context (Chapter 3).** We conduct an empirical study on the evolution of KDT suites. Our aim is to investigate the problem of test maintenance, identify the benefits of KDT and overall improve the understanding of test code evolution (at the acceptance testing level). This information supports the development of automatic techniques, such as test refactoring and repair, and allows future research on test fragility. To this end, we identify, collect and analyze test code changes across the evolution of industrial KDT test suites for a period of eight months. We show that the problem of test maintenance is largely due to test fragility (most commonly performed changes are due to locator and synchronization issues) and test clones (over 30% of keywords are duplicated). We also show that the test design of KDT test suites has the potential for drastically reducing (approximately 70%) the number of test code changes required to support software evolution when compared to Record & Replay. To further validate our results, we interview testers from BGL BNP Paribas and report their perceptions on the advantages and challenges of keyword-driven testing.
- **An empirical study on the diffusion and refactoring of test smells in KDT (Chapter 4).** We conduct an empirical study on the diffusion and refactoring action observed in KDT suites. Following our analysis on test maintenance, we focus on bad test scripting practices, SUIT smells, and measure how often they are refactored. To this end, we collect and analyze SUIT smells and their refactoring across over two million test instances from BGL BNP Paribas composed of 48 test suites and 12 open-source projects. We show that the same type of smells tends to appear in industrial and open-source projects, but the symptoms are not addressed in the same way. Some smells tend to appear in most tests with diffusion of up to 90%. Yet refactoring actions are much less frequent with less than 50% of the affected tests ever undergoing refactoring. Interestingly, while refactoring actions are rare, some smells tend to disappear through the removal of old symptomatic tests and the introduction of new tests not presenting such symptoms.

- **An approach to generate more flexible locators (Chapter 5).** One of the main sources for test fragility is the evolution of GUI elements tests are interacting with. To reduce the impact of such evolution, we propose a novel approach, HPath, to provide more flexibility to Document Object Model (DOM)-based locators by exploiting the semantics offered in the Hypertext Markup Language (HTML)5 standards. We evaluate our approach on web elements mined from the test suites of two large open-source projects. Our results show that shorter locators are not necessarily more robust to change than longer ones. However, relying on better properties of the GUI elements can significantly increase their robustness. As such, when HTML5 semantics are present, HPath can exploit rendered properties of web elements to generate expressive locators for 73.35% of them. This reduces locator breakages from 64.99% when relying on implementation properties to 0.49% with rendered properties.
- **Ikora: Continuous Inspection for KDT (Chapter 6).** We introduce Ikora, an automated tool that statically analyzes Robot Framework test suites, enabling the continuous inspection of test codebase written using KDT. Ikora targets code written in the Robot Framework syntax and has been successfully deployed at BGL BNP Paribas, detecting issues otherwise unknown to the test automation engineers, such as the presence of duplicated test code, dead test code, dependency issues among the tests, and bad design practices.

1.3.2 Organization of the Dissertation

In the remainder of this dissertation, Chapter 2 presents the previous work that is related to the contributions presented in this dissertation. Chapter 3 presents an empirical study that evaluates the evolution of SUIT scripts written using KDT in an industrial setting. Chapter 4 presents an empirical study on the diffusion and refactoring of smells present in KDT scripts both in industrial settings and in open-source software. Chapter 6 describes the tools and frameworks built as the result of this work to contribute to the advances in good testing practices when developing GUI test cases. Finally, Chapter 7 concludes this dissertation and presents some potential future research direction.

2 Related Work

This chapter discusses the existing work related to the contribution of the dissertation. We present an overview of the existing literature on SUIT maintenance and the existing work aiming at reducing it.

Contents

As introduced in Chapter 1, SUIT scripts suffer from a high maintenance cost arising from the natural evolution of the SUT. In this chapter, we present the related work addressing specifically the problem of test evolution. Additionally, we present the work addressing the main cause leading to test evolution, namely, test fragility. Finally, we end this tour of the literature with approaches proposed to either address the problem of test fragility by making the test more robust or proposing to automatically repair the test. During this overview, we address the shortcomings of the existing work and position our contributions accordingly.

2.1 Test code evolution

In the literature, we find a large amount of work tackling the problem of test code evolution. Most studies present in the literature address the issue by analyzing each version of a test suite and extracting change patterns between two consecutive versions. These patterns are extracted with the help of fine-grained change extraction tools such as *ChangeDistiller* [FWP$^+$07] or by performing a coarser-grained level such as in the work of Zaidman et al. [ZVvD$^+$11] which restrict their analysis to the file level. The rules that are used to classify the changes to a particular pattern either originate from prior (expert) knowledge [MRZ14], are inferred from the analysis of changes occurring in the test code base [NCD$^+$14; LIH17] or reflect the influence of patterns observed in the production code on the evolution of test code [VD08]. Thus, the analysis of the evolution of test suites takes the form of mining changes between versions and categorizing them into patterns with the aim of aiding testers to reduce the maintenance cost of their test suites either by automatic maintenance [HZ12] or by exposing sub-optimal processes [LIH17].

In particular, a line of research has been dedicated to understanding the relationship (co-evolution) between the evolution and the maintenance of test code and production code [LD10; ZVvD$^+$11; MRZ14; LY17; VP18; AAA19]. The goal of this body of work is to perceive which are the type of production code evolution that trigger changes in test code. Other studies, take the test suite in isolation and perform systematic analysis of patterns to evaluate their root causes [PSO12]. To this end, the analysis of bad practices present in the test code, *i.e.* test smells, which is an active field of research with the detection of those anti-patterns [vDMvdB$^+$01; BHP$^+$17; TPB$^+$16], the analysis of their impact and diffusion [BQO$^+$15; TPB$^+$16; Kim20] and their automatic identification and removal [VDD$^+$07; RGD07; PAN$^+$20].

Interestingly, many studies focus on the detection of indications of low quality of the test suite resulting in low bug detection capabilities or higher maintenance cost. However, historically, fewer studies focus on test evolution caused by incorrect behavior in the test code leading to potentially wrong signal which can originate from bugs in the test code [VFM15], obsolete test cases [HLZ+13; TWM15] or non-deterministic behavior either in the test, the SUT or the infrastructure. However, with the work of Luo et al. [LHE+14] which points out the prevalence of tests which only fail (test flakiness), the non-deterministic behavior of tests begins to suscitate great interest among researchers.

All the work presented so far focuses on unit tests, which may have very different characteristics than SUITs. One key difference lies in the black box interaction with the SUT making their maintenance more difficult [BWK05]. To account for these differences, researchers need to investigate the specific evolution of SUITs. For example, Skoglund and Runeson [SR04] conduct an exploratory study on the evolution of SUITs showing that their maintenance is more costly than unit tests. This high maintenance cost associated with SUIT, which arises with each new version of the SUT, obliterates the benefits offered by test automation. To alleviate this problem we need a better understanding of the root causes of this fragility and to propose mitigation strategies.

To answer this question, not unlike the work presented earlier, studies investigate the co-evolution between production code and SUITs [SG10; CSD+14]. Further work focuses on external factors influencing the maintenance of test suite [AFO13; KWW+13; AFK16; LMM+17] providing insights on how to create and maintain SUIT suites. Findings include the analysis of the frequency at which tests should be updated [AFO13; AFK16], which type of bad practices appear and should be avoided [CW12; LMM+17] or on the contrary should be adopted [KWW+13].

Although much work has been conducted on unit test evolution, its root causes, and its impact, work targeting SUITs evolution remains scarce. Because of the particularities in terms of interaction with the code, structure of the test, and the portion of code exercised, SUIT evolution can rapidly lead to a higher maintenance cost. Thus, focusing on these peculiarities of SUIT and how they affect their evolution is necessary to address the problem of their high maintenance cost.

2.2 The Fragility Problem

As described in Section 2.1, the literature on co-evolution, suggests that there is a strong impact from evolution in production code on its associated test code, thus, requiring its maintenance. The fragility problem contrast with the promise of test automation that once test scripts are created, they can be used for efficient regression testing cycles [YTS+14]. It as been reported that up to 75% of SUITs break during GUI regression testing [MS03; GXF09a; CRT16] following minor changes in the SUT. This limitation constitutes a major obstacle for developer adoption where the diffusion of SUITs remains low (*e.g.*, only 8% [CMT17; CMT+19] in Android Application) when compared to unit tests (*e.g.*, only 14% [KTN+15] to 20% [CMT17; CMT+19] in Android Application) while some consider SUITs as a cost-ineffective investment and advocate for its replacement [VDP12; CJL+20].

To increase SUIT robustness, researchers first need to understand which changes are affecting SUITs. The literature proposes to classify changes originating from the SUT into three categories [CZV+11; YTS+14; CRT16]: (1) structural changes that change the representation of the GUI element that is being targeted (*e.g.* node attribute are changed in a webpage), (2) content changes affecting the content of a particular GUI element the SUIT is interacting with, such as the label of a field, the text from a button or the content of an image (3) build changes which alter the behavior of the application as the apparition of modal boxes requiring user consent for the usage of cookies in all websites to remain compliant to the General Data Protection Regulation (GDPR) imposed by the European Union. This classification later enabled researchers to focus on a subspace of the problem of test fragility to come up with more tailored solutions.

Moreover, studies have shown that different approaches may lead to different performances when accounting for fragility. Leotta et al. [LCR+13; LSR+14] perform a comparison between Capture & Replay and GUI Test Scripting showing that even though Capture & Replay is cheaper to produce, because of the high fragility of the tests that are generated, the cumulative cost of maintaining GUI Test Scripts becomes lower. These results show that more than the nature of the changes in the SUT, the structure of the SUITs may condition their fragility.

Indeed, the root cause of SUIT fragility often originates from their interaction with internal information of the SUT instead of solely relying on rendered elements [YTS+14]. As such, in a survey on smells in software test code conducted by Garousi and Küçük [GK18], the authors show that fragility can be exacerbated by bad practices in test code, such as the failure to use proper synchronization point when interacting with the GUI. Other studies focus on the representation of the identifiers, one of the major challenges being to identify GUI elements through the use of locators [HRT16] which allow to uniquely identify GUI elements. For example, in the domain of web testing, Leotta et al. [LSR+15b] proposes a fragility coefficient to qualify an XPath expression that uniquely describes a web element. Thus, the more specialized the XPath, the higher its fragility coefficient.

Hence, to reduce the impact of the SUIT fragility, the research community started investigating mitigation strategies. These strategies can be classified into two families: (1) strategies aiming at avoiding test breakage altogether, thus yielding tests more robust to changes, and (2) repairing the test whenever it breaks. We present the work addressing these two strategies in the remaining of this section.

2.3 Robust Test Code

To limit maintenance induced by the fragility of SUITs a straightforward approach is to render the test more robust to SUT evolution. To that end, Pirzadeh et al. [PSD14] propose a framework for creating *resilient* SUITs which are resistant to specific SUT refactoring by avoiding to rely on unnecessary details.

Often, the root cause of SUIT fragility is that to identify GUI elements, metadata depending on its internal (structural) representation are recorded [DLM+11; YTS+14; HRT16]. Thus, in the literature, we find a large amount of work focusing specifically on the generation of robust locators uniquely identifying a GUI elements. For example, some studies propose to generate more robust locators relying on the presentation layer of the SUT [MPR+11; LSR+14; LSR+15b; LSR+16; ZHH+18]. However, these structural locators all suffer from the reliance on the internal representation of the GUI, therefore, leaking internal details of the SUT. To overcome this limitation, different strategies have been developed to minimise the impact of structural changes, such as the use of multiple locators [LSR+15b; ZHH+18; LWC+20] or the use of robustness heuristics estimates [MPR+11; LSR+14; LSR+15b; LSR+15a].

In an effort to avoid altogether this limitation, Visual GUI Testing (VGT) [Bos14], uses computer vision to locate elements through the bitmap graphics shown to the user at runtime. Indeed, relying on computer vision allows VGT locators to be more flexible under structural changes, thus, leading to more robust locators [LCR+14]. Tools like Automating Test Automation (ATA) [TSS+12; TDS+13] or the approach proposed by Yandrapally et al. [YTS+14], extract contextual clues from the GUI to augment the information contained in the locator and avoid to rely too heavily on its internal representation. However, while this technique is more adaptive to underlying structural changes than structural locators, minor changes in the GUI representation (*e.g.* minor changes in the layout or in the color of GUI components) might break tests relying on the exact representation of GUI elements [AD17; AKR18].

One common problem in all these approaches is that there is a limit to robustness because predicting SUT evolution is difficult [KTN19]. Thus, in parallel to the creation of robust tests, researchers also investigate automatic repair ofSUITs exhibiting breakage.

2.4 Repairing Test Code

Aware of the high maintenance cost, and the limitation to SUIT robustness, other approaches concentrate on techniques to automatically maintain GUI test scripts and compare their performance to manual maintenance [GXF09a; GXF09b]. If we cannot avoid test breakages from occurring, then why not automatically repair the test. To that end, some tools such as GUITAR [Mem08], MobiGUITAR [AFT+15] SITAR [GCZ+16] or METER [PXP+20] propose model-based approaches to automatically repair SUITs when they exhibit a failure. By representing the test in a more abstract representation, these tools can detect unusable tests and then attempt to repair them by generating repair candidates. To validate the repair candidates some heuristics are used which ensure to some extent that the intent of the test is preserved. As such, if no candidates are available, the failure can be considered as fault revealing.

Besides model-based approaches, tools like WATER [CZV$^+$11], WATERFALL [HRS16] exploit the differences between the old and the new representation layer of the SUT to propose repair candidates. Similarly, white-box techniques have been proposed to repair GUI test scripts using differences between versions. For example, FlowFixer [ZLE13] instruments the GUI-based application to generate an execution trace from the old application. Then, it randomly generates new event sequences to match the method coverage of the previous version of the test. Other studies [GXF09a; FGX09], instead of comparing the traces, focus on the code underlying GUI change to guide the generation of new repair candidates.

Incidentally, we can observe an emerging schema. Using heuristics such as minimal deviation from the original graph[Mem08] or exercising previously executed methods [ZLE13], all these techniques create new tests by generating event sequences using different strategies to explore the search space (*e.g.* genetic algorithms [HCM10], exhaustively [Mem08] or randomly [ZLE13]). This guided search allows to preserve to some extent the original intent of the test that is being repaired. One major limitation to this kind of techniques comes from the oracle problem. When a new sequence of events is generated, even though heuristics are put in place to limit this effect, there is no guarantee that the intent of the test is preserved [LDO19]. Aware of this limitation, the tool proposed in the work of Gao et al. [GCZ$^+$16] requires human intervention during the validation process of the new tests.

In an attempt to reduce this effect, researchers have redefined the search space and only target a subset of the breakages. Thus, in a similar fashion as in Section 2.3, when generating repair candidates, many studies reduce the search space by only targeting one type of breakage, locator breakages. As such, different tools are proposed in the literature to leverage different properties of the page such as attribute properties in the [CZV$^+$11], constants in the GUI representation of the passing test [EMS18; KTN19] or relying on computer vision[SYM18] to regenerate broken locators while preventing any oracle violation.

2.5 Summary

This chapter presented the work in the literature that is related to the contributions of this dissertation. Overall, this chapter shows that the work addressing the causes of the high maintenance associated with maintaining GUI test scripts remains scarce. Furthermore, where some promising work showcase improvement to tackle the fragility problem, further reducing the brittleness of GUI test scripts remains crucial for a larger adoption by practitioners. The next chapters present our contributions to fill the shortcomings in the literature regarding those problems.

3

Evolution of System User Interactive Tests

The evolution of SUITs is only scarcely covered in the existing literature, most previous studies on test code evolution focusing on unit tests. However, SUITs present peculiarities that might affect their evolution, especially when considering the fragility problem. In this chapter presents an industrial case study on the evolution of KDT suites. Our aim is to demonstrate the problem of SUIT maintenance, identify the benefits of KDT and overall improve the understanding of SUIT evolution.

Contents

3.1 Introduction

In Section 1.1, we presented different ways to create SUITs, *i.e.* Random GUI Testing, Model-Based Testing, Record & Replay and test scripting. Among the test scripting approaches, KDT is a technique using keywords where each keyword describes a set of actions that are required to perform a specific step. Using keywords, testers can model concepts from the SUT with different levels of abstraction (*e.g.* "Login" and "Click button"); targeting various domains: Web (Selenium), Android (Apium), and Desktop (Sikuli); allowing different formalism such as data-driven tests or the gherkins syntax.

In this chapter, we attempt to answer a fundamental question about KDT: "*What are the practical benefits and challenges of adopting KDT?*". An answer to this question will have a direct impact on practitioners who want to make an informed decision about adopting a test automation technique and to researchers who want to understand the KDT evolution and automate test refactoring and repair. Our study sheds light on this manner by identifying and quantifying the practical gains and losses of this practice. We also look into the way KDT suites are maintained by identifying and categorizing the nature of changes performed during the KDT evolution. This information is essential to better understand KDT maintenance and forms the basis of automated test code refactoring and repair techniques.

3.1.1 Tree representation of KDT

KDT can be represented using a tree structure. Figure 3.1 shows this structure for the test of Figure 1.1. The root of the tree (purple rectangle) is the *Test Case* that is executed by calling all the keywords it contains. The intermediary nodes (white rectangles) are called *User Keywords* since they are created by the tester. Finally, the leaf nodes (green rectangles) are *Library Keywords. Library Keywords* are implemented by the system or an external library and responsible for either defining the control flow of the tests or interacting with the SUT.

We group keywords into seven categories based on their functionality and present them in Table 3.1. For instance, we define a *SYNC* keyword category for keywords dealing with the synchronization between tests and SUT; *e.g.* a keyword that waits 10 seconds for a GUI element of the SUT to become available. Note that where all the categories refer to *Library Keywords*, the category *USER*, on the other hand, target *User Keywords.* Focusing on the evolution of test code, *User Keywords* are the focus of this study, thus, we use the term keyword to refer to *User Keywords* unless stated otherwise.

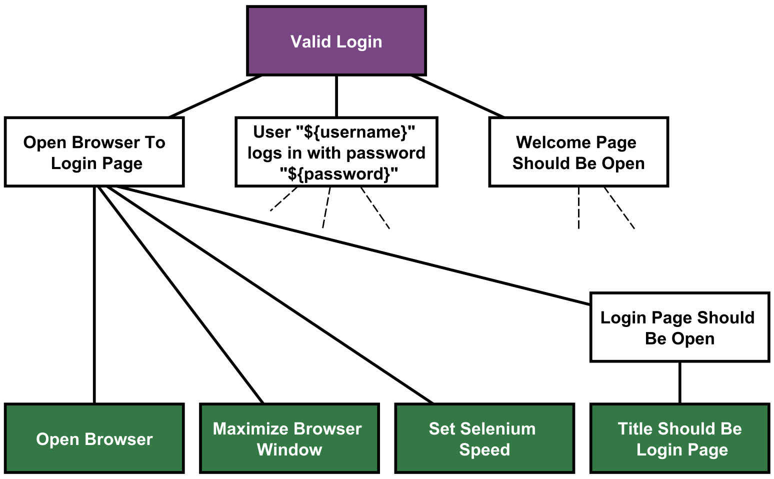

Valid Login
Open Browser To Login Page
User "${username}" logs in with password "${password}"
Welcome Page Should Be Open
Login Page Should Be Open
Open Browser
Maximize Browser Window
Set Selenium Speed
Title Should Be Login Page

Figure 3.1: Tree representation of the "Valid Login" KDT test.

Table 3.1: Keyword categories

Label	**Explanation**
ACTION	Keyword performing an action on the SUT capable of modifying its state.
ASSERTION	Keyword verifying that a predicate is true at a specific point of test execution
CONTROLFOW	Keyword allowing to modify the control flow of the test execution.
GETTER	Keyword allowing to extract an element from the SUT.
LOGGING	Keyword dumping logs during execution.
SYNC	Keyword relating to the synchronization between the SUT and the tests.
USER	Keyword created by a user.

3.1.2 Industrial Context

This chapter aims at investigating the evolution of KDT suites at the acceptance testing level based on the industrial practice. To this end, we work together with BGL BNP Paribas that has recently (one year prior to this study) adopted KDT and uses it in its daily software development work for acceptance testing.

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, at the end of 2016, the team started migrating from manual execution of the tests to the execution of automated tests written in Robot Framework. Robot Framework was chosen because of the ease it provides to develop tests targeting applications written in different technologies requiring different modes of interaction.

One other reason why our partner adopted KDT is that test cases at this testing level were created by different domain experts (business analysts and automation experts) and the adoption of a common language between the experts was imperative. All the tests used in our study have been created by a team of 3 testers and 2 business analysts working at BGL BNP Paribas using Robot Framework [Rob20], one of the tools used for the application of KDT (Section 1.1.4).

Robot Framework is a popular framework used world-wide by major companies, including Nokia, KONE, ABB. It is an open source tool originally developed by Nokia Networks and is mainly used for acceptance testing. The "Valid Login" KDT test of Figure 1.1 was written using this framework. One of the main advantages of Robot Framework is its high modularity. Indeed, Robot Framework is platform-agnostic and thanks to its driver plugin architecture, the core framework does not require any knowledge of the SUT. Another advantage of the framework lies in its simple syntax, which makes it easily accessible to testers, regardless of their background.

The project used in this chapter, hereafter referred to as *SubjectA* for confidentiality reasons, pertains to all the business activities of our partner. The front-end is a web application implemented in AngularJS, and, the back-end is composed of hundreds of services written in various programming languages. These services are managed by different teams, involving more than 100 developers. The KDT suite used in our study, referred to as *TestSuiteA*, is developed by 3 testers working at the Quality Assurance (QA) team of our partner and 2 business analysts.

Figure 3.2 shows the evolution of *TestSuiteA* across the eight-month period studied. The figure depicts the evolution of the number of *Test Cases* comprising the test suite, the number of *User Keywords* and the lines of code of the test suite. As can be seen, our analysis begins with a test suite of 39 test cases, 139 user keywords, and 1129 lines of code and ends with 117 test cases, 505 keywords, and 4732 lines of code.

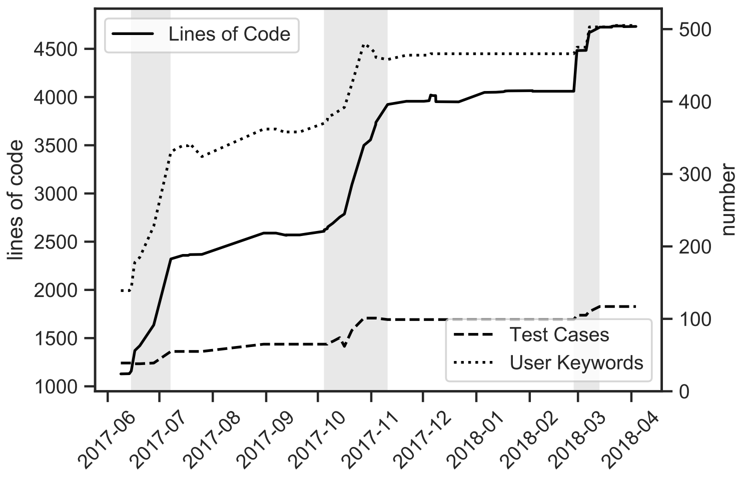


Figure 3.2: Evolution of TestSuiteA

In the time span depicted in Figure 3.2, we isolated three periods during which we saw an increased test creation activity (shown in grey). After discussing with the QA team, they corroborated that these periods were more focused on test creation and the remaining ones on test maintenance. Thus, we analyze separately these periods (greyed and non-greyed) and refer to them as "Creation" and "Maintenance".

3.2 Research Questions

In this chapter, we attempt to answer two main questions about KDT test suites at the acceptance testing level: "*What are the benefits and challenges of adopting KDT?*" and "*What kind of changes are performed during the evolution of a KDT test suite?*". Answers to these questions will enable practitioners to make more informed decisions about KDT and will improve our understanding of KDT test suite evolution. Thus, we pose the following research questions:

RQ1: *What types of test code changes are performed during KDT test suite evolution?*

Analyzing the changes performed by the testers during KDT test suite evolution forms the basis of any automated test refactoring and test repair technique. Although research presents such information in the case of unit testing [PSO12], no previous study has discussed such fine-grained changes in the context of KDT at the acceptance level, to the best of our knowledge.

RQ2: *How complex are the KDT test suites and how does this complexity affect their evolution?*

As mentioned in Section 3.1, one of the advantages of KDT is that it allows the separation of the technical implementation details of test code and its corresponding intention. This fact can lead to test suites having several “levels of abstraction” (Figure 3.1). To this day, it is not clear how complex the KDT test code is and how this complexity affects its evolution. Answering this question will provide us with a better understanding of the difficulties faced by practitioners when they try to apply KDT and can guide future research directions in ameliorating these problems.

RQ3: *Does code duplication exist in KDT test codebases? What is its impact on the evolution of the test code?*

Similar code fragments are known to exist in source code and test code alike [Bak95; RCK09; RBS13; LMM$^+$17]. In RQ3, we investigate whether KDT codebases contain duplicated test code and how these test clones affect the evolution of the test codebase. Answering this research question is important because if such test clones exist, we need to investigate appropriate techniques to detect them, analyze them and monitor their evolution.

RQ4: *What are the practitioners' perceptions of the benefits and challenges of KDT in practice?*

RQ4 pertains to documenting and analyzing the practitioners' opinion about the advantages and disadvantages of KDT. Such analysis can help other testers to adopt (or not) KDT. Additionally, this research question gives us the opportunity to ask the practitioners' opinion about our results, validating them and understanding them better.

3.3 Research Protocol

3.3.1 Definitions

- **Tree**: Keywords can be represented as trees, thus, we can define a *tree* T as an ordered, directed, acyclic graph with nodes $N(T)$ and edges $E(T) \subseteq N(T) \times N(T)$. The nodes of the tree denote keywords and each edge between two keywords denotes a “step”: the parent keyword has the child keyword as a *step*. For instance, in Figure 3.1, keyword “Open Browser To Login Page” has four steps: “Open Browser”, “Maximize Browser Window”, “Set Selenium Speed” and “Login Page Should Be Open”. As the tree is ordered, the execution of the steps will follow the order in the tree, from left to right. A node with no parent is a *root* node that should be defined in the *Test Case* block, while a node with no children is a $leaf$ node and should be a *Library Keyword.*

- **Keyword Level**: The *level* of keyword k, is the maximum number of edges that exist on the subpath(s) from k to a leaf keyword. In Figure 3.1, "Login Page Should Be Open" is a *level 1* keyword whereas "Open Browser To Login Page" is a *level 2. Library Keywords* at the leaves of the tree have a *level 0.*
- **Keyword Connectivity**: Connectivity is a metric of reusability among the keywords. A keyword can belong to several test cases represented as trees: let keyword k belong to trees $T_1, T_2, ...T_n$, i.e. $k \in N(T_1) \cup N(T_2) \cup ... \cup N(T_n)$, then we calculate the connectivity of k by counting the number of nodes (keywords) in the subpath(s) from the root nodes of $T_1, T_2, ...T_n$ to k.
- **Keyword Churn**: Keyword churn is the number of lines of code added, edited or deleted from one version to the next over a period of time.

The last 3 definitions correspond to metrics used in our study. The *keyword level* is used to group keywords having equal levels together. According to the philosophy of KDT, lower-level keywords should be more linked to the technical details of the SUT whereas higher-level keywords should be more abstract, expressing the functional requirements. The *connectivity* metric expresses the degree to which a keyword is reused and, as a consequence, the degree to which a change to this keyword can impact the test suite. Finally, the *churn* corresponds to the degree to which a keyword is changed during the evolution of the test suite.

3.3.2 Answering RQ1

To answer RQ1, we extract all the changes occurring in the test suite and group them per type of change. The types identified describe an action (insert, update, delete) performed on a code unit element (*User Keyword*, *Test Case*, *Variable*, etc.).

To this end, we extracted the 129 commits from the evolution of *TestSuiteA*. For each pair of consecutive commits, we gather the changes using a fine grain change algorithm.

The algorithm relies on previous, state-of-the-art studies [CRG+96; FMB+14; FWP+07; PSO12]. In these studies, the authors built Abstract Syntax Tree (AST) of Java classes and used tree edit distance algorithms to extract an optimal change path from one tree to the other, with each tree corresponding to a version of the codebase.

To detect the changes, the algorithm works in two phases:

1. Finding a match between elements of $v_1 \in V$ and $v_2 \in V$ where V is the set of versions – with one version corresponding to one commit – to come up with a mapping $e_{1n} \rightarrow e_{2n}$ where $e_{mn} \in E_n$ and E_n is the set of elements from v_n.

2. Finding a minimum edit script that transforms V_1 to V_2 given the computed mapping.

Algorithm 1 Element Matcher

```
Require: $E_1 \subset v_n$, $E_2 \subset v_{n+1}$
Ensure: final matching set: $M_{final}$
 1: $M_{final} \leftarrow \emptyset$
 2: $E_{1,unmatched} \leftarrow \emptyset$
 3: for $e_1 \in E_1$ do
 4:     if $findMatchFileAndName(e_1, E_2)$ then
 5:         $M_{final} \leftarrow M_{final} \cup (e_1, e_2)$
 6:         $E_2 \leftarrow E_2 - e_2$
 7:     else
 8:         $E_{1,unmatched} \leftarrow E_{1,unmatched} \cup e_1$
 9:     end if
10: end for
11: for $e_1 \in E_{1,unmatched}$ do
12:     if $findMatchFileAndContent(e_1, E_2)$ then
13:         $M_{final} \leftarrow M_{final} \cup (e_1, e_2)$
14:         $E_2 \leftarrow E_2 - e_2$
15:     else if $findMatchNameAndContent(e_1, E_2)$ then
16:         $M_{final} \leftarrow M_{final} \cup (e_1, e_2)$
17:         $E_2 \leftarrow E_2 - e_2$
18:     else
19:         $M_{final} \leftarrow M_{final} \cup (e_1, \emptyset)$
20:     end if
21: end for
22: for $e_2 \in E_2$ do
23:     $M_{final} \leftarrow M_{final} \cup (\emptyset, e_2)$
24: end for
```

Phase 1 is essential to the edit script since the more elements that can be matched, the better the minimum edit script will perform. Phase 2 produces an edit script detecting the basic edit operations *INSERT*, *UPDATE*, *DELETE* for each pair of matched elements.

Listing 1 presents the algorithm used for phase 1 to find an appropriate matching set $E_{1n} \rightarrow E_{2n}$.

- **Lines 3–10**: Search for two elements present in the same file with the same name. If no match is found from $e_1 \in E_1$, it is tagged as unmatched.
- **Lines 11–21**: The same operation is performed, relaxing the constraints. First, at line 12 the name is relaxed, to check if the element was renamed. Then at line 15 the file is relaxed to check if the element was moved to another file. If no suitable match is found for $e_1 \in E_1$, it is matched with a *null*

element and will be consider as a *DELETE* operation in phase 2.

- **Lines 22–24**: Check if there are elements from E_2 that weren't matched, in which case they will be considered as an *INSERT* operation in phase 2.

In phase 2, for each pair of matched elements, we extract the differences. In the case of *User Keyword* and *Test Case*, we use an edit distance algorithm on the sequence of steps which is a modification of the *String-to-String* algorithm presented in [Ukk85] using the Levenshtein edit distance.

3.3.3 Answering RQ2

For each keyword we extract its level and connectivity, using the tree structure of KDT presented in Section 3.3.1. We then cluster the keywords by each of these metrics. For each group, we analyze the number of changes performed and the keyword churn. In order to avoid skewing the churn results, we compute the churn during "Creation" and "Maintenance" separately.

Next, we attempt to provide an estimation of the number of changes saved due to the reusability offered by KDT. To answer this, for each tests, we create a sequence of steps executed during execution. Therefore, if a keyword is used twice, the steps from that keyword would appear twice in the sequence. We then compute the changes for each sequence of step execution from one version to the next. The sequences of steps obtained are similar to the ones generated by Record & Replay. While these results cannot be used to directly compare the maintenance cost of Record & Replay and KDT, it provides an estimation of the benefits of reusing keywords.

3.3.4 Answering RQ3

To answer this question, we extract similar keywords, also referred to as clones in the literature, and we analyze their evolution. To detect test clones in KDT test suites, we built a clone detection tool specifically designed for KDT test code. The tool is based on the fine-grained change algorithm presented in the previous section. We extract the differences between each pair of keywords $k_1, k_2 \in E_n$, ignoring changes related to documentation and *update name* (Table 3.2). For each pair k_1, k_2 we check whether they belong to one of the two types of clones analyzed in our work (definitions adopted from [LMM+17]):

- **Type I keyword clones**: identical keywords except for changes in white space, layout, and documentation. The clone detection tool tags a keyword pair as Type I clones only in the case of an empty set of differences.
- **Type II keyword clones**: keywords with a content syntactically identical except for step arguments. The clone detection tool tags a pair as Type II clones only if the set contains differences of type *update step arguments* and/or *update step return values* from Table 3.2.

Additionally, for each keyword, we extract the set of changes happening during the period under study. From this change list, we define 3 types of keyword evolution:

- **Keyword evolving**: If the change list of a keyword k is not empty, it is defined as evolving.
- **Keyword co-evolving**: Among the set of keywords evolving, keywords k_1, k_2 are defined as co-evolving if their changed list is identical.
- **Keyword not evolving**: Keyword k is defined as not evolving if its change list is empty.

Finally, we analyze the relationship between keyword evolution and keyword similarity by cross analysis of categories.

3.4 Experimental Results

3.4.1 RQ1: Types of Changes during KDT Evolution

This research question pertains to the types of changes performed by the testers during *TestSuiteA* evolution. The identified types and their total amount are presented in Table 3.2. The first column of the table shows the type of changes as extracted by our change algorithm. The next columns present the total amount of these changes during the *Creation* and *Maintenance* periods (as defined in Section 3.1.2) over the 8 months of the study.

We see that during *Creation*, the main activities in terms of the number of changes is “insert documentation”, “insert user keyword”, “insert variable”, and “update step”. The first three types of changes are naturally related to test creation, so this outcome is expected. A more interesting finding is that a lot of effort is devoted to documenting the keywords created. After discussing with the QA team, this effort is justified by the fact that this documentation will prove useful in the case of KDT test failures. “Update step” refers to modifications of steps of existing keywords. The specific kinds of modifications will be further investigated later in the section (Figure 3.3).

During *Maintenance*, the main types of changes performed are the “update step arguments”, the “update step”, “update documentation” and “insert user keyword”. After manually analyzing the changes to the arguments, we found two prevalent categories of commonly-changed arguments: arguments referring to *synchronization* between the SUT and the KDT tests, e.g. wait 3 seconds and arguments referring to *locators*, i.e., ways of locating elements in the GUI interface of the SUT. The arguments of the first category are typically used in the *SYNC* category of Table 3.1, and the latter at keywords of the *ACTION* and *ASSERTION* categories. Our results suggest that keywords belonging to these categories experience a high number of changes. Practitioners corroborate those results and motivate those results in RQ4.

Apart from “update step arguments”, “update step” constitutes one of the most

Table 3.2: Types and total amount of changes over the 8-months study

change type	Creation	Maintenance
insert documentation	**430**	2
insert step	135	62
insert test case	94	12
insert user keyword	394	80
insert variable	286	77
update documentation	106	96
update for loop body	0	0
update for loop condition	0	0
update name	45	6
update step	249	107
update step arguments	105	**144**
update step expression	7	6
update step return values	0	1
update step type	5	3
update variable definition	34	45
delete documentation	0	2
delete step	25	34
delete test case	26	2
delete user keyword	70	38
delete variable	6	23
Total	2017	738

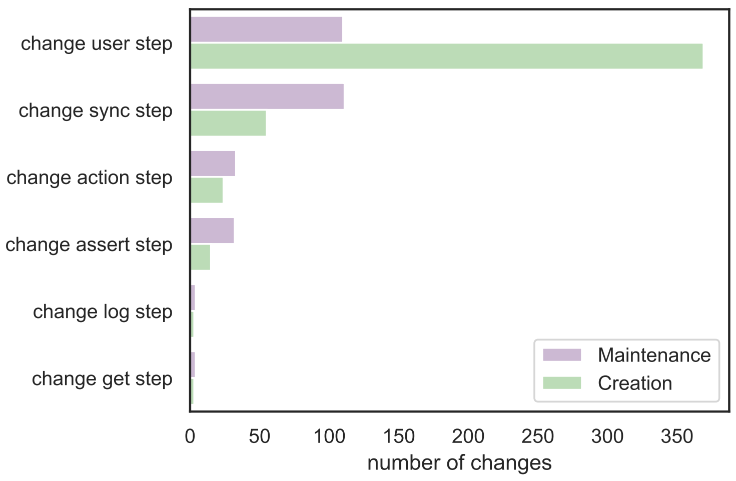
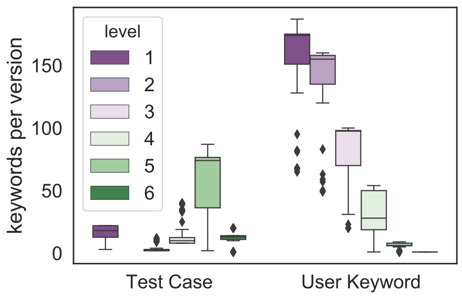
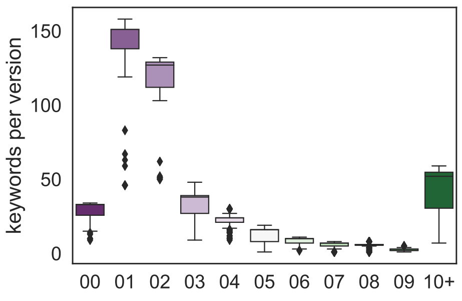


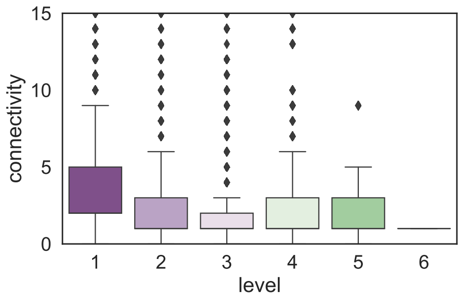
Figure 3.3: Total number of step changes per type



(a) Keyword Level



(b) Keyword Connectivity



(c) Connectivity per level category

Figure 3.4: Understanding KDT Test Suite Complexity

common change for both *Creation* and *Maintenance* periods. To further investigate the nature of these changes, Figure 3.3 plots the number of changes (x-axis) against the category of the enclosing keywords (y-axis), as presented in Table 3.1 with different colors for the periods studied.

As can be seen from the figure, *change user steps* is by far the greatest activity during creation, we see that changes in *synchronization steps* are equally important during maintenance. The interview conducted in RQ4 motivates that finding and explains it by the fact that many keywords are refactored during the creation of new tests to become more generic so they can be reused. Another trend is that except for the *user steps*, all other categories evolve more during maintenance. This is due to the same effect as mentioned earlier were changes in the application cause tests to break. *user steps* are less affected by that effect since they are more abstract and thus less sensitive to trivial application evolution.

3.4.2 RQ2: KDT Complexity and Evolution

The results for RQ2 are split into two parts: first, results about the complexity of KDT test suites are reported; and, second, the way this complexity affects its evolution is presented.

KDT complexity

To understand KDT complexity, we calculate the *keyword level* and *connectivity* metrics, defined in Section 3.3.1. The first metric refers to the different “abstraction levels” (moving from purely technical to requirements expression) of the test suite and the second one, to the reusability among the keywords. Figures 3.4a and 3.4b present the corresponding results.

Figure 3.4a depicts our results of the *keyword level* for *Test Cases* and *User Keywords*, with the y-axis referring to the number of keywords per version. Recall that *Test Cases* are the complete instantiation of a test - root node in the tree representation - and *User Keywords* are user-defined abstraction of the steps - intermediate nodes in the tree representation - (see also Section 1.1.4). As can be seen from the figure, most *Tests Cases* are relatively complex, with a level of 5, whereas most *User Keywords* are simple (levels 1 to 2). This indicates that most user-defined actions remain simple, in accordance with the philosophy of KDT.

Regarding the keyword reusability, Figure 3.4b plots the number of keywords per version (y-axis) with the *keyword connectivity* (x-axis). As can be seen, there is a high degree of reusability among *User Keywords.* More precisely, only 20.34% of the lines of code are used only once. Overall, the reused keywords amount to 51.56% of the total lines of code of *TestSuiteA*. As we will see next, this reusability is key to the decreased cost of the KDT maintenance.

Another interesting finding is the presence of dead test code, i.e., keywords not used anywhere in *TestSuiteA*; these keywords have connectivity of 0 in Figure 3.4b.

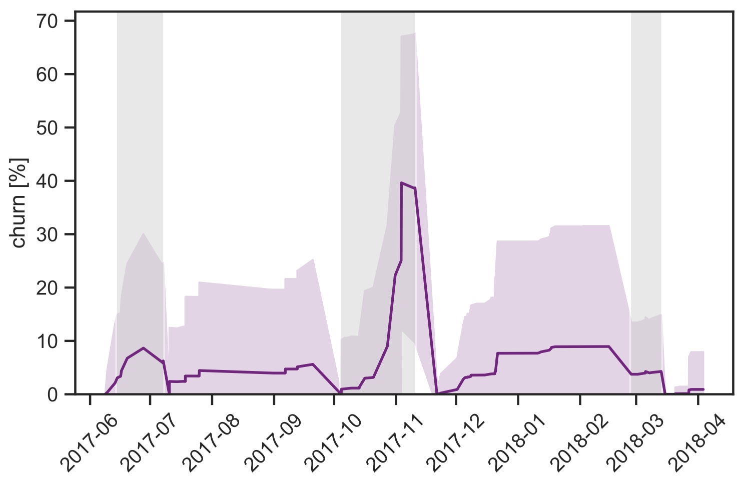


Figure 3.5: KDT test code evolution: Churn over time

In total, 5.58% of the keywords were not used, which amounts to 4.58% of the test codebase. When we presented our findings to the QA team, they were surprised and confirmed the existence of dead code, explaining that there is no tooling to support such analysis. Our tool solves this issue and has been integrated into the team's test code development processes (Chapter 6).

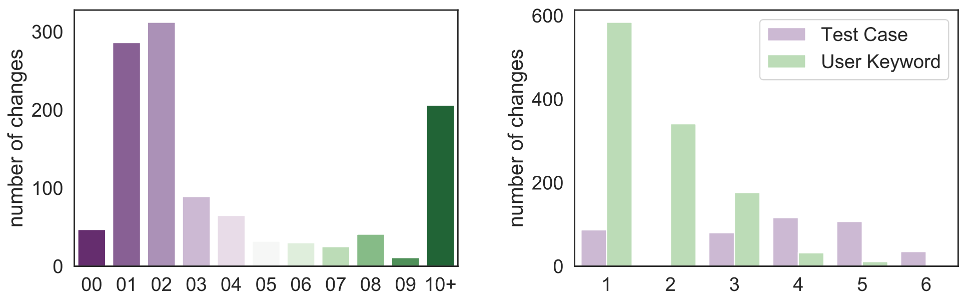
To investigate whether keywords of a particular level tend to be more reused than others, Figure 3.4c plots the keyword connectivity among the different keyword levels. By examining the figure, it becomes clear that keywords levels exhibit relatively high connectivity, indicating that the reusability of keywords is not restricted to a particular level with the exception of level 1 showing slightly higher connectivity.

KDT complexity and evolution

The second part of RQ2 refers to the evolution of KDT and how its complexity affects it. To better understand the number of changes performed during test code evolution, Figure 3.5 presents the test code churn (y-axis) over the eight-month period analyzed (x-axis), with a similar setup to Figure 3.2.

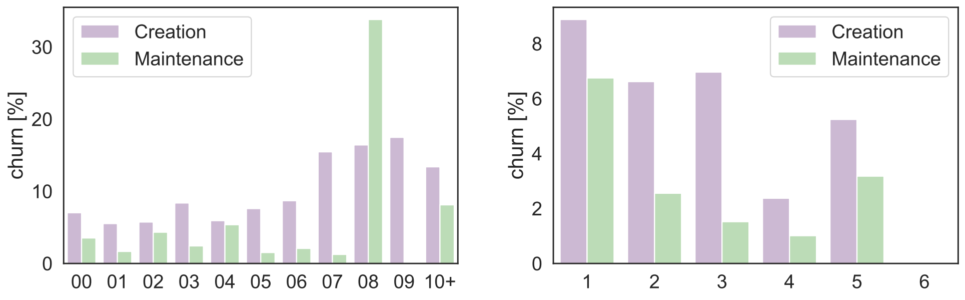
The purple line in the figure denotes the average churn across *TestSuiteA* evolution and the light purple, its variance represented here by the standard deviation. From the figure, it can be observed that during *Creation*, the churn is 8.13%, on average, whereas in the *Maintenance* period, its value is 3.61%. Overall, keywords are changed with a churn rate of 5.11%. This number suggests that keywords are not entirely rewritten, but localized modifications are performed.

To investigate further how the complexity of the KDT test code affects its



(a) Connectivity (b) Level

Figure 3.6: Changes distribution according to level and connectivity



(a) Connectivity (b) Level

Figure 3.7: Churn distribution according to level and connectivity

evolution, Figure 3.6 plots the number of changes, for the whole period studied, against the keyword connectivity and level and Figure 3.7 plots the churn against the same metrics.

After examining Figure 3.6a, it becomes clear that keywords reused one to three times are mostly changed. Keywords with higher connectivity do not change that often. Moreover, the figure shows that changes are performed on dead code (connectivity 0). This confirms that testers are unaware of the fact that these keywords are never executed, generating easy-to-avoid maintenance. Regarding the results for changes and level, depicted in Figure 3.6b, we can observe that the changes to *Test Cases* do not follow a specific trend, whereas for the changes to the *Users Keywords*, the lower level the keyword is, the more it is susceptible to be changed.

Regarding our findings on the relation between churn rate and connectivity, depicted in Figure 3.7a for the *Creation* and *Maintenance periods*, we can conclude that, during *Creation*, keywords that are reused often, i.e. higher connectivity, exhibit approximately 50%-60% increased churn rate, whereas, during *Maintenance* the opposite holds. Finally, regarding the results presented in Figure 3.7b about churn and keyword level, we can see that, during *Creation*, keywords with lower levels exhibit high churn values, whereas in *Maintenance* this only holds for keywords of level 1. These results suggest that low-level, highly reused keywords (basic action on the SUT), are evolving at a higher rate.

As we saw earlier, in *TestSuiteA* evolution, keywords changed with a churn rate of 5.11% but we also saw in the previous section that keywords are reused often. This raises the question: How many changes have been saved due to the reusability of the keywords? To answer this question, we compare the number of changes applied to *TestSuiteA* to the same suite without the keyword abstraction as explained in Section 3.3.3. We find that using KDT reduces the number of changes applied on *TestSuiteA* by 70.77% during "Creation", by 72.69% during "Maintenance" with an overall reduction during the entire period of 71.31%.

3.4.3 RQ3: KDT, Test Clones and Evolution

In RQ3, we explore whether KDT test suites contain test clones and how these clones affect *TestSuiteA* evolution. Table 3.3 presents the corresponding results. The table presents the total number of keywords that appear during the evolution of *TestSuiteA* (for all 129 versions) for each type of clone detected (first column – Type I keyword clones, Type II and non-clones("Others")) and each type of evolution (second column). The types of evolution studied are divided into three categories: keywords that are evolving strictly in the same way as others ("Co-evolution"), keywords that are evolving independently from others ("Evolution"), and keywords that do not evolve ("No change").

We can observe several interesting findings from the table. First, we see that Type I and Type II clones comprise 30.2% of the total amount keywords, indicating that almost one-third of the test code written is duplicated. This finding highlights the fact that practitioners applying KDT will benefit from tools and techniques that can assist them in managing test clones.

Secondly, our results suggest that approximately 50% of the Type I test clones evolve in the exact same way, indicating that the practitioners apply the same changes multiple times, wasting valuable effort. This is a high figure, especially when compared to the co-evolution of non-cloned keywords which is 4.9%. Taking these results into consideration and the fact that almost 10% of the keywords are evolving in the same way, it becomes obvious that automated refactoring techniques can reduce the maintenance effort of KDT suite evolution.

Finally, another interesting result exhibited in Table 3.3 concerns the overall evolution. We observe that only 7.4% of the keywords are not evolving. This shows that during the TestSuiteA's evolution more than 90% of the keywords are

Table 3.3: KDT Test Clones and Evolution

Keywords	Types of Evolution			Total
	Co-evolution	Evolution	No change	
Type I	3526	3599	412	7537 (13.7%)
Type II	171	8462	491	9124 (16.5%)
Others	1888	33433	3184	38505 (69.8%)
Total	5585	45494	4087	55166 (100%)
Percent	10.1%	82.5%	7.4%	-

modified.

3.4.4 RQ4: Benefits and challenges of KDT: The Practitioners' perspective

This research question pertains to the benefits and challenges of KDT as perceived by the practitioners. In the following, we present the main findings of our interviews grouped by the two main questions of our study:

What are the benefits and challenges of adopting KDT?

All interviewees agreed on two main benefits of KDT: the low learning curve and its simple syntax. Thanks to its syntax that is close to the natural language, new users can easily start being productive. This syntax is also well-suited for communication purposes with teams that may have different backgrounds and expertise. The layered structure of keywords (i.e. the different keyword levels) plays an important role in facilitating this by hiding the technical details at the lower levels of the test suite and exhibiting the more business-oriented at the higher levels.

The main challenges encountered by the practitioners reside in their interaction with the SUT. Even a small evolution of the SUT can easily break the tests. Additionally, the testers discuss that finding the elements of the SUT that will be used in the tests is challenging, especially in applications where testability was not the primary concern.

What kind of changes are performed on the test suite and why?

The testers report two main reasons for the changes: SUT evolution and keyword adaptation.

Regarding the SUT evolution, the testers reported that as the SUT evolves, its components evolve as well which will cause the tests to adapt. The testers focus on two types of changes that are in according to our findings regarding RQ1 (cf.

Section 3.4.1): *locators*, i.e., finding which GUI elements of the SUT should be used in the tests and *synchronization* issues between the tests and the SUT.

Regarding keyword adaptation, the testers said that they create keywords in a “best effort” approach to cover the current needs. As new features of tests are developed, keywords are modified to become more generic. This fact explains the results illustrated in Figure 3.3 where we observed many changes in *user steps* during *Creation*.

3.5 Threats to Validity

Threats to the external validity result from the generalization of our results outside the context of the study. Conducting the study with one industrial partner, the conclusions we draw may not be able to generalize to other companies using KDT. However, *SubjectA* is built using popular technologies, i.e., web frameworks and Java services, which are wide-spread across the industry. Secondly and most important, this study is the first one, to the best of our knowledge, that analyzes the evolution of KDT test suites based on real-world data. Of course, this does not preclude the need for other studies to investigate further our results. Another potential threat to generalization originates from the fact that we interviewed only 3 testers for RQ4.

Threats to the internal validity are due to the design of the study, potentially impacting our conclusions. The simple syntax of the test code allows for a robust model to be constructed. Our change algorithm presents some limitations: although phase 2 is based on the state of the art, it cannot detect *Move* operations, resulting instead in two operations *Delete* followed by an *Insert*. This limitation might have influenced our results during the accounting of the number of changes. However, the rather low number of the *delete step* operations (Table 3.2) indicates that this effect is marginal. Regarding the clone detection algorithm, as shown in [RCK09], the rate of false-positives is known to be low for Type I and Type II clones.

Threat to construct validity result from the non suitability of the metrics used to evaluate the results. The main threat lies in the division of our work in two periods: “Creation” and “Maintenance”. While empirical data motivated this separation, they lack of theoretical grounding. Further work on the test execution is needed to better motivate this decision.

3.6 Conclusions

Our results suggest that KDT test design is complex with several levels of abstraction and that this design favors reusability; more than 60% of the keywords are reused which has the potential of reducing the changes needed during evolution up to 70%.

Additionally, we find that keywords change with a relatively low rate (approxi-

mately 5%) indicating that after a keyword's creation only fine-grained, localized changes are performed by the testers. Our results suggest that the most common changes to KDT tests are caused by *synchronization* or element *location* changes between the SUT and the test suite and to the *assertions* of the tests. Our findings indicate that during evolution 90% of the keywords evolve and that test clones exist in KDT test suites; approximately 30% of the keywords are duplicated. Finally, we report on the practitioners' perception on the challenges and benefits of adopting KDT.

4

Smells in System User Interactive Tests

SUITs and their evolution can be hindered by sub-optimal decisions in their design referred to as smells. This chapter studies the smells that arise in SUIT scripts and the refactoring operation that are applied to remove their symptoms.

Contents

4.1 Introduction

Looking back at Chapter 3 we observe a large number of changes in the structure of KDT depicted in Figure 3.3. These changes are explained by the fact that many *Keywords* are refactored during the evolution of the test suite. Following the definition present in the literature, refactoring is a technique to improve the design of a system and enable its evolution [FBB+99]. Thus, in an effort to better understand the reasons leading to refactoring, we analyze the nature and diffusion of such sub-optimal designs in the test code leading to these iso-functional changes. As such, we rely on the concept of smell, which was first introduced by Fowler et al. [FBB+99], to define a poor design and implementation choice that hinder the system maintainability. Where a lot of work has been devoted to their analysis in production code, more recently, the research community started to focus on the bad practices present in the test code [vDMvdB+01; Mes07; RGD07; VDD+07; CW12; HHV+13; BQO+15; TPB+16; BHP+17; Kim20; PAN+20] and coined the term test smells to specifically refer to bad practices occurring in test code.

In this chapter, we focus our attention on those test smells present in SUITs that we will refer to as SUIT smells. Indeed, the interviews conducted in Chapter 3 and the analysis of the impact of sub-optimal decision such as the presence of clones in the test suite (Section 3.4.4) highlight the importance of better understanding how improving the quality of the code by avoiding anti-patterns can reduce the heavy maintenance cost of KDT tests.

More generally, where the literature explores extensively the question of bad practices and their refactoring in unit tests [vDMvdB+01; Mes07; RGD07; VDD+07; BQO+15; TPB+16; BHP+17; Kim20; PAN+20], fewer studies [CW12; HHV+13] target specifically the question at the system level. Nevertheless, SUITs present peculiarities in the way they interact with the SUT through its GUI, their black-box nature, or the way they are written to be understood not only by testers and developers but by business analysts as well.

Thus, in an effort to shed light on SUIT smells, we combine a multi-vocal literature review and an empirical study on the KDT test suites from BGL BNP Paribas as well as the ones present in 12 open-source repositories. Using this catalog of SUIT smells, we propose an automated approach for detecting their diffusion and refactoring.

4.2 Research questions

We begin our investigation by looking at how well SUIT smells are covered by both the grey literature and the academic literature. The goal behind this first question is to investigate whether or not there exists a gap between the knowledge present in the research community and among practitioners. Thus, we formulate our first research question as:

RQ1: *What are the SUIT smells studied in academic and grey literature?*

We continue our endeavor by assessing to which extent the symptoms of each smell are present in SUITs. To that end, we compute the number of occurrences of the symptoms for each smell in every SUIT. In other words, for each smell, we associate a metric and observe how that metric is distributed over the different tests. Therefore, we ask the following question:

RQ2: *How widespread are SUIT smell symptoms in SUITs?*

Finally, we focus on the refactoring actions performed by the maintainer of the test suites to remove the smell symptoms. However, because those symptoms can be removed by actions not related to the symptom removal itself but merely being the by-product of another (*e.g.* removing a feature, thus, the associated smelly code), we focus our analysis on fine-grain changes, specifically targeting the removal of the symptom itself. Hence, we formulate our third research question as follow:

RQ3: *How often do we observe refactoring actions removing SUIT smell symptoms?*

4.3 Research Protocol

4.3.1 RQ1: Identification of SUIT Smells

To perform a study on the impact of smells in SUIT, we first need to build a reliable and representative catalog of test smells. To this end, we start our investigation by collecting SUIT smells presented in both academic and grey literature. We classify in the academic or white literature papers that are published in peer-reviewed conferences or journals. On the other hand, we classify 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 [RS21]. 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 [GK18]. Figure 4.1 summarizes our adoption of these steps in the process of building a catalog of SUIT smells.

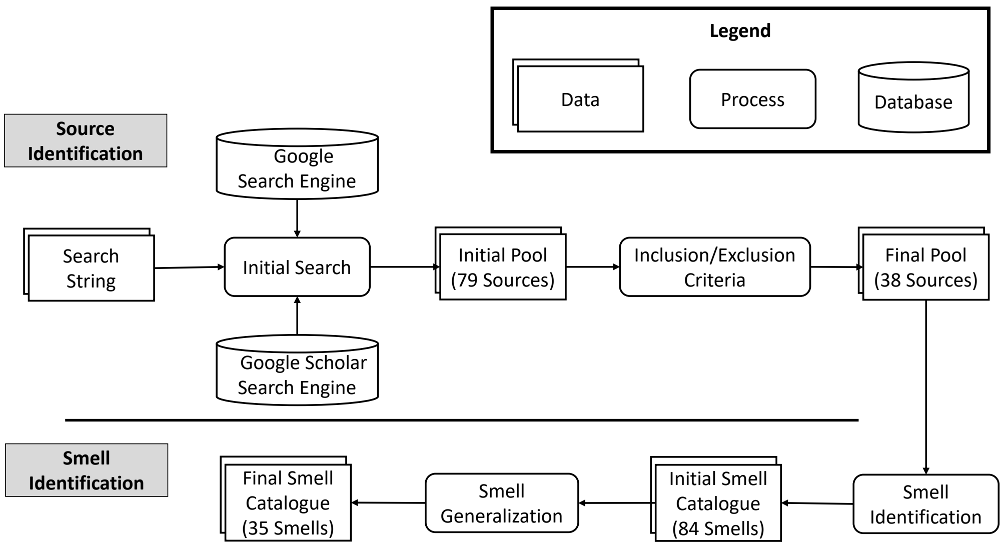


Figure 4.1: Overview of the process to establish the SUIT smell catalog.

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 that [GK18], hence, we limit our investigation to these two search engines. 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 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 [KC07; RS21]. Then, we proceed to a systematic 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, at which point the number of relevant sources per page drops to zero. From those, we read the title and abstract available to decide whether or not to further analyze the articles. This leads to a initial list composed of 47 sources in the grey literature and 32 sources in the academic literature for a total of 79 sources.

Inclusion/Exclusion Criteria. 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. Furthermore, some sources address testing issues that are not related to the code base 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* [Sta17] for which bugs making the functional test suite fail intermittently at low intervals are ignored. By excluding these irrelevant sources, 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[1].

Smell Identification. Relying on this final pool of sources, where each source contains at least one SUIT smell, we proceed to the smell identification where 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* [Sta17] 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 [Adv18]. 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* [Buw15] is grouped with *Comments and documentation instead of abstraction* [Kla14], 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. Consequently, we consider those two smells to present similar symptoms and effects and we group them in one smell named *Lack of encapsulation* [CW12; Kla14; Buw15; Eng16; Ren16]. Finally, we observe that some test smells are subsuming others. This is the case for the smells *Hardcoded Values* and *Using condition in test logic* that are both subsumed by the definition of *Obscure test* [HHV$^+$13; Gaw16; Sim19]. In this example we only keep the latter. Finally, 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 catalog of 35 SUIT smells[2] that can be detected statically.

[1] Available as a spreadsheet
[2] Available at https://github.com/UL-SnT-Serval/suit-smell-catalog

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 straightforward metric, avoiding false positives, could be constructed by analyzing the test code. Hence, in this work, we present a list of 16 SUIT smells. A comprehensive description of each smell is presented in Section 4.4.1.

4.3.2 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 4.1 summarizes the overall properties of these projects. In the following, we describe the projects and present the collection process.

Industrial project:

We leverage the codebase of our industrial partner, BGL BNP Paribas. Where the empirical analysis from Chapter 3 focuses on the test suite for one project, *SubjectA*, in this chapter, the entire test suite is considered. This test suite targets desktop, web, and mobile applications that are depending on services developed following a Service-Oriented Architecture (SOA). For instance, BGL BNP Paribas developed internal services using Java Swing which rely on either the Application Programming Interface (API) exposed by the Swing development or rely on computer vision, through the use of the Sikuli[3] library. BGL BNP Paribas also use the Robot Framework to test its mobile client application relying on the Appium[4] Library.

The goal of the team is to assess compliance to functional requirements by conducting acceptance testing. Today, the test suite consists of 559 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 a login to the ecosystem which is common to a large number 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 4.1 for the project *bgl*.

[3]http://sikulix.com/
[4]https://appium.io/

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 interacting with the SUT through its user interface. From the results of this first step, we filter out toy projects and tutorials, projects using Robot Framework for its robot process automation capabilities in production, and libraries extending the capabilities of Robot Framework. Following this approach, we gather 23 repositories from Github. Finally, to ensure that maintenance was performed on the test suite itself, we analyze the number of commits 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 4.1.

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. Among those tests, the majority both in industrial and in open-source projects target web applications. The SUITs interact with such applications through their webpages by interacting with the DOM or by relying on computer vision to isolate GUI elements. 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 for both industrial and open-source projects. Thus, we believe that this diversity allows avoiding biases observed in one type of project and improves the generalization of the observations.

4.3.3 RQ2: SUIT Smells Distribution

For each of the SUIT smells that we gathered, we compute a metric to automatically measure a smelliness score for the affected test. We rely on heuristics to construct these metrics. To this end, we apply the high-level investigation mechanism framed by Marinescu [Mar04] called “detection strategy”. As defined by the author, 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 associate the symptoms leading to the final rule for detecting the smells.

Note that step 3 of this approach describes the definition of a threshold. Even though different approaches have been proposed, *e.g.* using semantical properties

Table 4.1: Metrics for the 13 projects under study. *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.

Name	**LoC**	**#Commits**	**#TestCases**	**#Keywords**
bgl	57,030	309	559	5,000
apinf	1,068	345	74	138
bikalims	4,446	1,279	76	172
collective-cover	1,317	943	24	31
cumulus-ci	1,599	1,429	132	34
harbor	7,270	3,131	246	472
ifs	8,466	5,606	494	605
mms-alfresco	1,797	504	156	8
mystamps	2,361	353	212	111
ozone	2,636	271	230	90
plone	1,046	211	59	163
plone-intranet	2,225	1,759	213	134
rspamd	3,397	685	429	151

and statistical distributions [Mar04] or using Bayesian belief networks [KVG+09], 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 of a threshold [AMZ+16]. In 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 code base and not to classify tests as smelly or not.

To extract code metrics, we rely on a parsing engine presented in Section 3.1.1. The parser generates for each test a call tree, where the root is the entry point of the test and the leaf nodes are the actions generated on the system under test, or the application driver layer if we refer to the three-layer architecture described by Humble and Farley [HF10]. Each leaf node is annotated with information such as the type of action (assertion, getter, event, etc.) and each variable can be inferred a type based on the *Library Keywords* calling it. To create the metrics, we rely on heuristics exhibiting the symptom associated with the smell. While building the heuristics, we focus on high precision, even though this might lead to a low recall.

Where the detection of SUIT smells is typically performed using code metrics, in the case of one of the SUIT smells, namely *Using Personal Pronoun*, such metrics are not sufficient. Hence, we rely on natural language processing to extract information about the name of some *Keywords* and use text tagging to define whether or not the subject of the *Keywords* is the first-person pronoun or not.

When computing the metrics, we introduce two types of metrics: a count metric S and a density metric D. The count metric S counts the number of instances of a symptom observed in a test. On the other hand, the density metric D 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 locations for the symptom to be present, it appeared in the test.

Finally, to assess the diffusion of the symptoms across the test suite we present the total number of symptomatic tests $|t_s|$ over the total number of tests, $|t|$. Thus $|t_s|/|t|$ measures the percentage of tests presenting an instance of a symptom for a smell s. Table 4.2 presents a summary of all the metrics.

Table 4.2: 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.
% Symptomatic Tests	$\lvert t_s \rvert / \lvert t \rvert$	The proportion of tests that are presenting at least one symptom for a smell s for a version
# Refactoring	$\lvert R_s \rvert$	The total number of action refactoring nodes exhibiting a symptom R_s for a smell s counted across all tests in all versions.
% Refactoring	$\lvert t_{s,r} \rvert / \lvert t_s \rvert$	The proportion of symptomatic tests undergoing at least one refactoring action through their lifespan.

4.3.4 RQ3: SUIT Smells Refactoring

Refactoring can be defined as a technique to improve the design of a system and enable its evolution [FBB+99]. Relying on this definition, to answer our third research question, we conduct an analysis of the refactoring actions during the maintenance of the test suites. To do so, we collect every pair of subsequent versions and identify refactoring changes occurring in the test code base. A key component in this type of analysis is to properly identify what constitutes a valid refactoring action. Indeed, just observing a decreases 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 [TGS+13; SV17], we address this limitation by using heuristics based on static test code analysis. More specifically, we use the fine-grain change algorithm presented in Section 3.3.2 to extract instances of refactoring patterns. These patterns are derived from the definition of the symptoms present in the literature.

Consequently, for each SUIT smell s we store the set of nodes N that are exhibiting a symptom of the SUIT smell and the set of refactoring actions $R_{s,N}$ that when performed on one or more elements of N removes the symptom. We obtain a series of tuples $< n, r_{s,N} >$ where $r_{s,N} \in R_{s,N}$ and $n \in N$ for each pair of subsequent versions which describes the fine-grained refactoring actions that were performed on one or more nodes to fix a specific SUIT smell symptom, *i.e.* the refactoring actions.

To generate this list, first, we compile the set of potential actions $R_{s,N}$ that can fix a symptom of a SUIT smell s given a set of nodes N (the complete list is presented in Section 4.4.2). Then, for each tuple $< n, r_N >$ provided by the fine-grained change extraction tool and the knowledge of which nodes present a symptom in the previous version, we check if $r_N \in R_{s,N}$. If it is the case, then the fine-grained change $< n, r_N >$ is considered to be a refactoring action and added to the list of tuples $< n, r_{s,N} >$.

To assess to which extent practitioners refactor SUIT smell symptoms, we count the number of refactoring actions across all the versions, $\left|r_{s,N}\right|$. Thus, the higher this number, the more effort testers spend in removing the symptoms. To measure the prevalence of refactorings over the test suite, we present the number of refactoring actions over the total number of symptomatic tests: $\left|t_{s,r}\right| / |t_s|$ where $|t_s|$ is the number of tests t exhibiting a symptom s and $\left|t_{s,r}\right|$ is the number of symptomatic tests t_s undergoing at least one refactoring r removing a symptom of a smell s. The results of this analysis are presented in Section 4.4.4. Table 4.2 presents a summary of all the metrics.

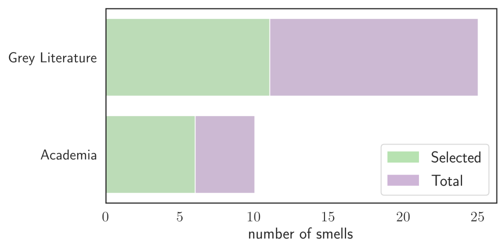
4.4 Experimental Results

4.4.1 RQ1: SUIT Smells Identification

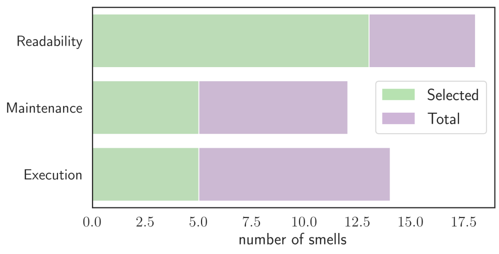
Figure 4.2a shows the number of test smells that were found in academic and grey literature. We show both the 35 initial SUIT smells extracted from the literature and the subset of 16 SUIT smells for which we could extract a metric in the test code.

In the figure, a SUIT smell is considered as covered by academia if at least one of the smells grouped in the generalization step (Section 4.3.1) is covered by a peer-reviewed article. Interestingly, all smells covered by academic literature appear in grey literature as well. 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 SUIT smells identified. Furthermore, 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* [HHV$^+$13], *Unsuitable Naming* [CW12], *Inconsistent Hierarchy* [HHV$^+$13] and *Data Creep* [ASM16]. While [HHV$^+$13] propose metrics for 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. [CW12] and [ASM16] on the other hand do not propose a metric to automatically extract symptoms for the smells they present.

To conclude this analysis of Figure 4.2a, 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 SUIT smells not previously studied in academia.



(a) Source



(b) Issues

Figure 4.2: Properties of the identified SUIT smells. Figure 4.2a displays the number of smells found in the academic and in the grey literature. Figure 4.2b displays the number of smells exhibiting a specific issue. Note that a smell can lead to one type of issue, hence the total is greater than the total number of SUIT smells.

Figure 4.2b 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 are the ones for which the highest number of 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 as it is the case for instance with the *Lifeless* smell [Buw15; Ren16; Buw19] where the test is not following the same lifecycle as the application. Thus, in both cases, the extraction of the symptoms requires information about the structure and lifecycle of the SUT.

4.4.2 SUIT Smells Catalog

The process described in section 4.3.1 leads to a final list of 35 SUIT smells presented in Table 4.3. Out of those 35 SUIT smells, we conduct a deeper analysis on 16 of them, the ones for which a metric can be automatically derived by observing Robot Framework test code.

Building on the tree model introduced in Section 3.1.1 we represent each test t as a rooted, ordered, directed, acyclic graph. The nodes, N_t, of the tree represent the set of calls to *Keywords*, C_t, and the set of of arguments A passed to the calls. A call, c_t, can be associated to a type namely: USER, INTERACTION, ASSERTION, CONTROLFLOW, GETTER, LOGGING and SYNC. For example, the set of

Table 4.3: Catalog of SUIT Smells and their origin. Smell in bold are associated with a metric and a complete description is presented in Section 4.4.2.

.

Name	**Sources**
Army of Clones	[CW12; HHV$^+$13; HEJ$^+$15; Kni19]
Complicated Setup Scenarios	[Sco15]
Conditional Assertions	[Gaw16]
Conspiracy Of Silence	[Gaw16; She20]
Data Creep	[ASM16; Sim19; Sha19]
Dependencies between tests	[Kla14; Adv18; Cri18; Bus19; Gol19]
Directly Executing UI Scripts	[Sco15]
Duplicate Check	[Buw19]
Eager Test	[Eng16; Ren16; Cri18; Sci19; Tem20]
Hardcoded Environment	[Gaw16; She20]
Hiding Test Data	[Jai07]
Implementation Dependent	[Jai07; Kap18; Gol19]
Inconsistent Hierarchy	[Cla14; Gaw16; Buw19]
Inconsistent Wording	[HHV$^+$13]
Lack of Encapsulation	[CW12; Eva12; Kla14; Buw15; Eng16; Ren16; Kni17a; Gol19; Kni19; Sha19]
Lack Of Early Feedback	[Dha17]
Lifeless	[Buw15; Ren16; Buw19]
Long Test Steps	[CW12; HHV$^+$13; Buw19]
Middle Man	[CW12]
Missing Assertion	[Kla14]
Narcissistic	[Eng16; Kni17b].
Noisy Logging	[Jai07].
Obscure Test	[HHV$^+$13; Gaw16; Sim19].
On the Fly	[Arc10]
Over-Checking	[Buw15; Ren16]
Pointless Descriptions	[Eng16]
Sensitive Locators	[Sco15; Kni19; Bat20; She20]
Sneaky Checking	[Kir14; Buw15; Ren16]
Stinky Synchronization	[Gaw16; Ren16; Bus19; Kni19; She20]
Test Data Loss	[Sim19]
Testing Data Not Code	[Dha17]
Unnecessary Navigation	[Arc10]
Unrealistic Data	[Gol19]
Unsecured Test Data	[Mor19]
Unsuitable Naming	[CW12; Gol19; Sha19; She20]

calls in a test performing an assertion is noted $C_{t,assertion}$. Finally, the definition of the *User Kyeword* called by a test t is noted K_t. Following this notation, we formally describe each smell with respect to its symptoms, its impact, and the metrics chosen to measure the prevalence of the symptoms following the protocol presented in Section 4.3.3. Finally, we present the refactoring actions removing the symptoms from the test code base.

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.
- **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.
- **Refactoring 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 refactoring pattern, $R_{AoC}(k)$, as follow:

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

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.
- **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).
- **Refactoring Action:** The symptom is considered refactored if the conditional assertion node is removed from the call graph. Thus, we accept the following refactoring pattern, $R_{CA}(c)$, as removing a symptom in a node c:

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

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.

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.
- **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:

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

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

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

- **Refactoring 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 refactoring pattern, $R_{HE}(a)$, for an argument a:

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

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.
- **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:

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

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

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

- **Refactoring 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 refactoring pattern, $R_{HTD}(c)$ performed on a *Library Keyword* call c as follow:

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

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

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.
- **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:

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

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

where $\left| C_t \cap C_{step} \right|$ is the size of the set of steps in the acceptance criteria of the test t and $\left| C_t \cap C_{step} \cap C_{driver} \right|$ 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*).
- **Refactoring Action:** The symptom is considered refactored if the direct call to a *Library Keyword* is removed from the acceptance criteria. Thus, we propose the refactoring patterns, $R_{LoE,1}(c)$ and $R_{LoE,2}(c)$, performed on a *Library Keyword* call c as follow:

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

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

In the first equation, $R_{LoE,1}(c)$, 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, $R_{LoE,2}(c)$, the call is removed.

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.
- **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:

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

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

  where $\left|C_t \cap C_{step}\right|$ is the number of steps in a test t and $\left|C_t \cap C_{step \geq L}\right|$ 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 [SAI+11]. Following this approach we find the knee point at 13 actions on the SUT for a step (quantile = 0.986, see Figure 4.3). Therefore, we set $L = 13$ and consider any step performing a sequence of actions on the SUT greater than 13 to be too long.
- **Refactoring 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 refactoring pattern, $R_{LTS}(c)$, where c is a long test step:

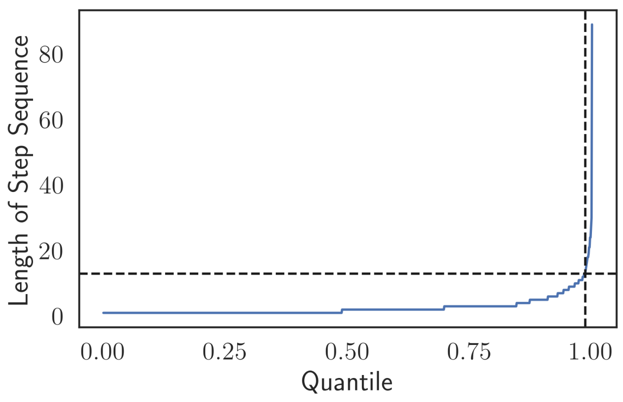


Figure 4.3: The blue curve represent the evolution of the length of the sequences of the step as the quantiles increase. The intersection of the black doted lines displays the knee point (0.986, 13) at which the values sequence lengths values start to increase dramatically.

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

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.

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.
- **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:

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

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

where $|K_t|$ is the number of *User Keywords* in a test t and $\left|K_t \cap K_{delegate}\right|$ 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.*

- **Refactoring 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 refactoring pattern, $R_{MM}(c)$ where c is a call to a *Delegate Keyword* as:

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

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.

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.
- **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:

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

- **Refactoring 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 refactoring pattern, $R_{MA}(t)$, where t is the test lacking any assertion as:

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

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

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.

- **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:

$$S_N(t) = \left|C_t \cap C_{step} \cap C_I\right|$$

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

  where $\left|C_t \cap C_{step}\right|$ is the number of "test steps" for a test t and $\left|C_t \cap C_{step} \cap C_I\right|$ is the number of "test steps" using a personal pronouns as the subject in there name.
- **Refactoring 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 refactoring pattern, $R_N(c)$, where c is a step of the acceptance criteria of a test t:

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

  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.

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.
- **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:

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

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

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

- **Refactoring 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 refactoring pattern, $R_{NL}(c)$, performed on a *Keyword* call c as follow:

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

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

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.
- **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.

- **Refactoring 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 refactoring pattern, $R_{OT}(a)$, where a is a *Keyword* call arguments as:

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

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

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.
- **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:

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

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

where $\left|A_t \cap A_{expected}\right|$ is the number of "expected" arguments in calls to *Library Keywords* annotated as "assertion" for a test t and $\left|A_t \cap A_{expected} \cap A_{computed}\right|$ 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.

- **Refactoring 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 refactoring action, R_{OtF}, addressing a symptom in an argument a:

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

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.

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 suggests the test is checking many different properties.
- **Impact on Maintenance:** The test may be too sensitive to the evolution of the SUT, verifying implementation properties instead of behavioral ones.
- **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".

- **Refactoring 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 refactoring pattern, $R_{OC}(c)$ where c is a call to an *Library Keyword* annotated with "assertion" as:

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

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.
- **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:

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

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

where $\left| A_t \cap A_{locator} \cap A_{|E|>1} \right|$ 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 $A_{locator}$.
- **Refactoring 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 refactoring pattern, $R_{SL}(l)$, as follow:

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

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.

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.
- **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).
- **Refactoring 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 refactoring pattern, $R_{SC}(k)$, where k is a *User Keywords* as:

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

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.
- **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:

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

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

where $\left| C_t \cap C_{sync} \right|$ is the size of the set of calls to *Library Keyword* annotated as "synchronization" in a test t and $\left| C_t \cap C_{sync} \cap C_{sleep} \right|$ 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".

- **Refactoring 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 refactoring patterns, $R_{SS,1}(c)$ and $R_{SS,2}(c)$, as follow:

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

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

4.4.3 RQ2: SUIT Smells Distribution

Building on the metrics derived in Section 4.4.2, we show for all projects the number of test exhibiting at least one instance of a symptom in Figure 4.4. The symptom *Hidden Test Data* does not appear neither in open-source project nor in industrial projects. Furthermore, symptoms *Noisy Logging*, *Sensitive Locator*, *Narcissistic* and *HardCoded Environment* manifest in less than 10% of the tests in open-source and industrial SUIT suites.

On the other end of the spectrum, three symptoms appear in the majority of the SUITs: the symptom *Hardcoded Values* appears in more than 90% in both type of projects, *Over Checking* between 75% (industrial) and 79.5% (open-source), and finally *Sneaky Checking* appearing in 70% of the test in both projects.

Finally, an interesting observation concerns the presence of significant difference between industrial and open-source projects. Notably, at BGL BNP Paribas, 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%). The results for the *Army of Clones* and *Middle Man* can be explained by the structure of the projects at BGL BNP Paribas. Symptoms for the SUIT smell *Army of Clones* are appearing because many repositories are interconnected, leading to code duplication

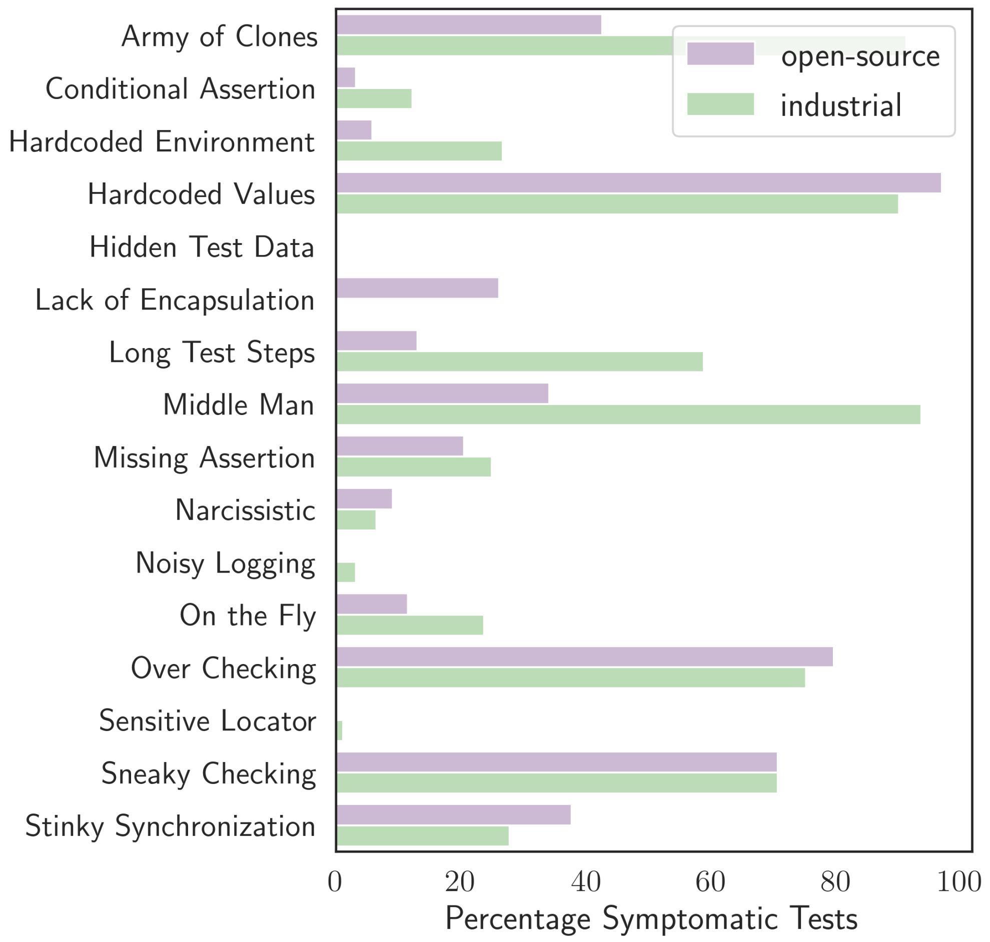


Figure 4.4: Percentage of tests exhibiting at least one instance of the symptom for open-source projects and industrial projects.

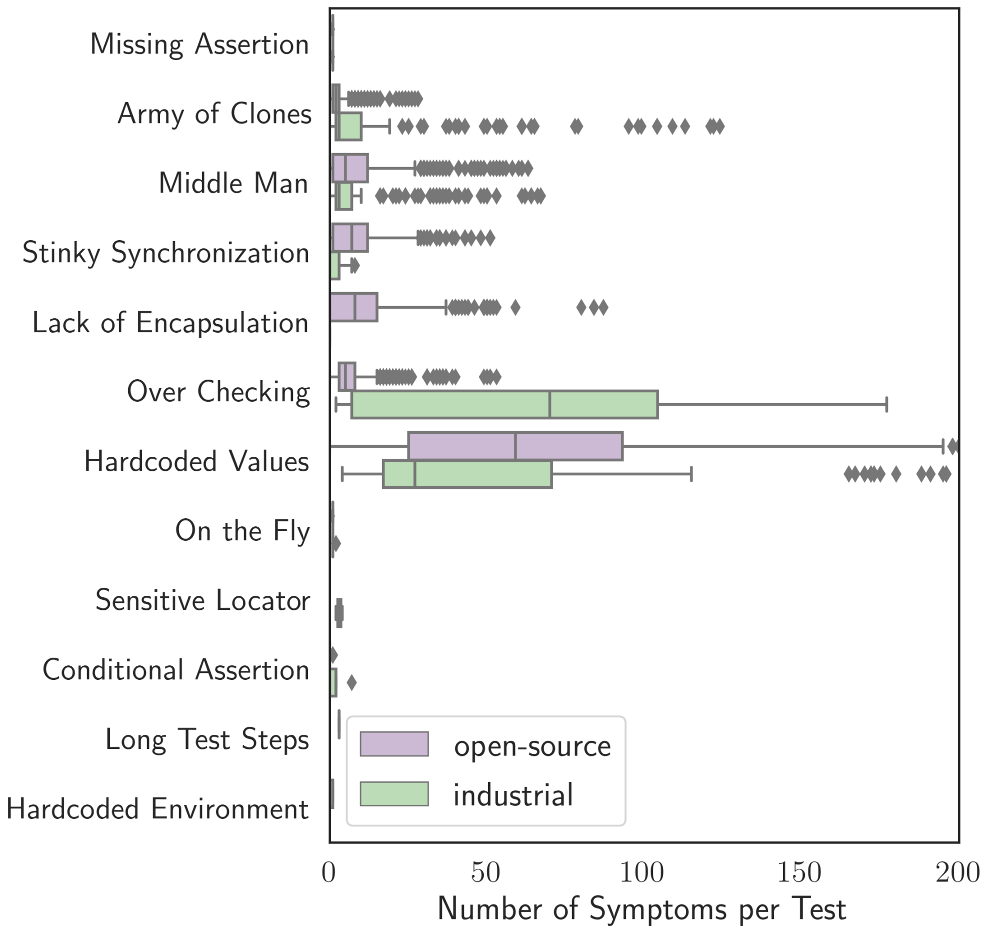


Figure 4.5: Distribution of the number of symptoms in symptomatic tests for all projects across commits.

across projects because testers working on different applications or different parts of the system test common functionalities over and over. As for the symptoms of *Middle Man*, their presence is explained by the use of *Keywords* that act as translation layers. While two test steps can perform the same concrete actions on the SUT, in different context they might operate on a different business logic. Thus, the 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. Finally, *Long Test Step* originates from the complex business logic that is expressed by the step in a test at BGL BNP Paribas and is more related to the SUT and the vocabulary employed by the business analysts than the test suite itself.

We continue our analysis with the evaluation of the number of symptoms appearing per tests (Figure 4.5). Note that only tests presenting at least one symptom are taken into account in our analysis, *i.e.* symptomatic tests. Looking at *Hardcoded Values*, we see that even both industrial and open-source projects exhibit the same proportion of test containing *Hardcoded Values*, the number varies significantly from a median of 27 in the case of industrial projects to 59 for open-source projects. A similar observation can be made for the case 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 test containing duplicated code (Figure 4.4) is higher, but the number of duplicated *Keywords* in symptomatic tests are higher in industrial projects (3 duplicated *Keywords* per test) than in open-source projects (2 duplicated *Keywords* per test).

To put the results of Figure 4.5 into perspective, we show the density metrics, $D_s(t)$, in Figure 4.6. 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 Figure 4.5), 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.

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.

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 Figure 4.5 *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.

Nonetheless, in the case of the industrial project, 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

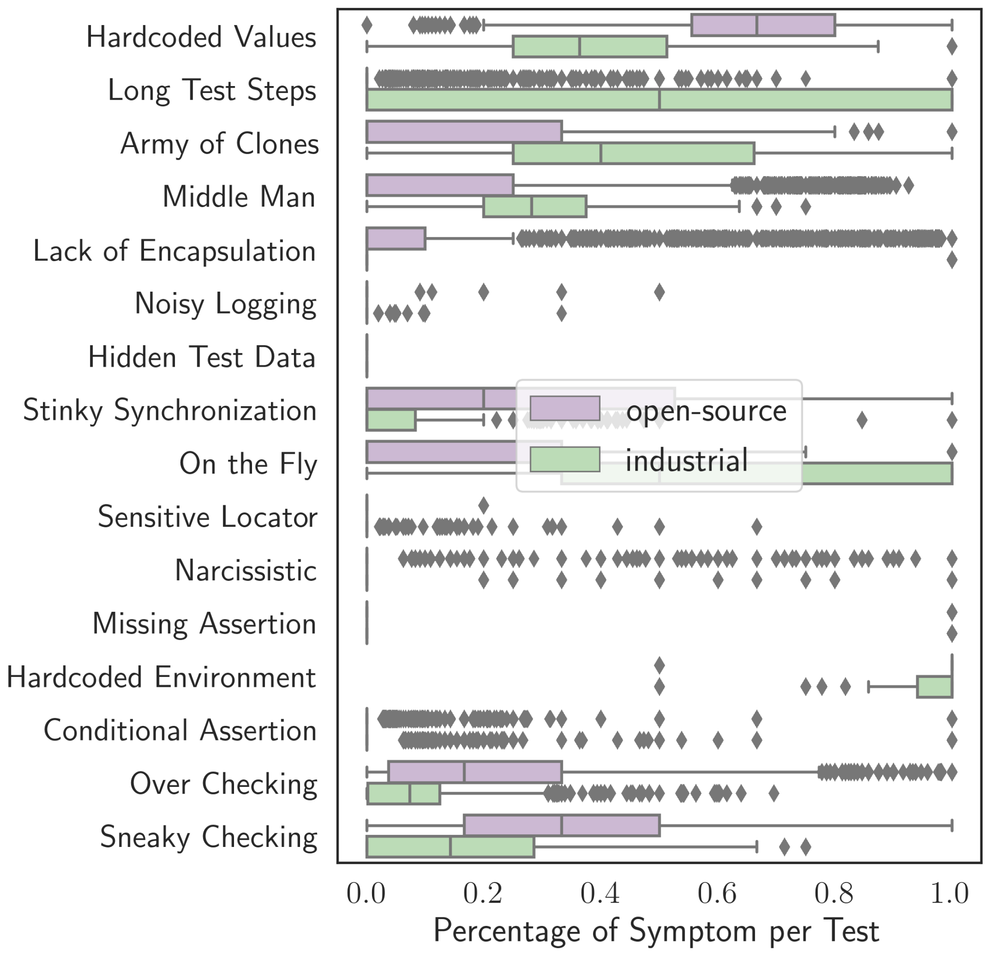


Figure 4.6: Distribution of the density of symptoms in each test for all projects across commits where the density is the percentage of locations that a symptom could appear at which actually exhibit the symptom.

the number of tests where the symptom could appear. Furthermore, the difference from the industrial project can explained by the hard policy in the QA team at BGL BNP Paribas, where explicit timeouts are considered as a major source of flakiness. Indeed, explicit timeouts are strongly discouraged by the QA team and are usually removed during code reviews.

BGL BNP Paribas and the open-source projects present similar trends despite their difference in terms of design and architecture. We observe that 7 out of the 16 smells symptoms do not typically appear in SUITs, whereas the smells *Hardcoded Values*, *Over Checking*, and *Sneaky Checking* are frequent. As for the difference observed between industrial and open-source projects, we note that the smells *Army of Clones*, *Long Test Step*, *Middle Man* and *On the Fly* often exhibit symptoms in the industrial project but not in the open-source projects and inversely symptoms of *Sticky Synchronization* tend to appear more often in open-source projects. Finally, the symptoms of *Hardcoded Environment* appearing aslmost systematically would suggest that the developers do not consider it as a bad practice.

Finally, we perform a ranking analysis between BGL BNP Paribas and the 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 the number of symptoms. These low scores suggest that the relative importance of the symptoms present in SUITs differs from industrial project and open-source projects.

To conclude, the symptoms *Hardcoded Values* (90% of the tests), *Over Checking* (between 75% and 80% of the tests), and *Sneaky Checking* (70% of the tests) are prevalent in both BGL BNP Paribas and open-source projects. These smells appear in large number and also exhibit a high density compared to their potential number of occurrences. Due to project and team structure, the industrial project is more prone to complexity smells such as *Army of Clones* and *Middle Man.* On the other hand, their code reviewing policies seem to prevent the symptoms of *Stinky synchronization.*

4.4.4 RQ3: SUIT Smells Refactoring

While Section 4.4.3 focuses on the prevalence of smell symptoms across SUITs, this section tackles the question of how often the symptoms are refactored by practitioners. Thus, relying on the definition of a symptom, we use the methodology described in Section 4.3.4 to extract the fine-grain changes removing a symptom

Table 4.4: Number of refactoring actions (*Count*) and percentage of symptomatic tests where at least one refactoring 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

from the test, *i.e.* refactoring actions.

Column *Count* in Table 4.4 shows the number of refactoring actions across all SUIT-modifying commits. Adding assertions to tests presenting the symptom *Missing Assertion* is the most common type of refactoring 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 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 refactoring actions see high values compared to the rest, namely, *Army of Clones* with 738 refactoring actions, *Hardcoded Value* with 226 refactoring actions, and *Middle Man* with 1,037 refactoring actions. Indeed, with the introduction of the tooling presented in Chapter 3, 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 refactorings 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 test code base. 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 code base 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, but ended up also addressing another smell, *Middle Man.* Finally, observing the number of refactoring actions addressing *Hardcoded Value* is mainly due to the fact that the symptom appears often. Indeed, when observing the column *Percent* from Table 4.4, we see only 18% of the test exhibiting the symptom are refactored through their lifetime.

Still focusing on the results for BGL BNP Paribas, for some symptoms, we never observe refactoring 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 code base. 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. One reason can be that to increase readability, assertions in the acceptance criteria are often encapsulated in a *User Keyword* with more meaningful names. Indeed, in Figure 1.1 (Section 1.1.4) we present an example of a KDT extracted from the official document of Robot Framework. We observe at line 9 a call to "welcome page should be open" which exhibit the symptom for *Sneaky Checking.* However,

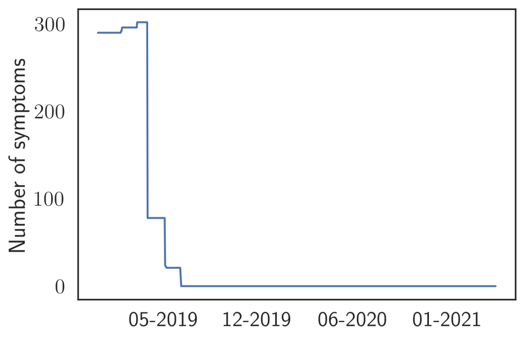


Figure 4.7: Evolution of the number of symptoms for the smell *Narcissistic* over time for the industrial project (BGL BNP Paribas).

calling directly the *Library Keyword* “Title Should be” would have decreased the readability of the test and introduced another symptom, *Lack of Encapsulation*.

Putting these results into perspective, Figure 4.7 shows the evolution of the symptom *Narcissistic* over time. However, when looking at the metric *Narcissistic* for BGL BNP Paribas, we observe no fine-grain refactoring. 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 refactoring action happening, the symptoms are removed from the test suite as new test are introduced and old tests deprecated.

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

Finally, similarly to Section 4.4.3, we perform a rank analysis using the normalized Levenshtein similarity. The value obtain 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 refactoring. These low values show that the majority of the refactoring are not ranked in the same way in both industrial and open-source projects. Taking into account the results of Section 4.4.3 this comparison suggest that when generalizing results obtained from open source projects, researchers should remain careful as their applicability in industrial context.

In conclusion, the results presented in this section show that the refactoring operation performed in industry and open-source projects present differences in term of which type of actions occurs the most often. However, in both cases, the proportion of symptomatic tests that are refactored remain low with the exception of the symptoms for *Missing Assertion*. Indeed, for half of the categories less than 5% of the symptomatic tests are ever subjects to refactoring. *Missing assertion* is a unique exception with between 70% (open-source) and 90% (industrial) of the symptomatic tests being refactored. Interestingly, while refactoring actions are rare, test smells like *Narcissistic* and *Middle Man* still disappear from the test code base as a side effect of the replacement of symptomatic tests by new tests not exhibiting the symptom.

4.5 Threats to Validity

Threat to construct validity results from the non-suitability of the metrics used to evaluate the results. To detect test smells we rely on heuristics based on code metrics. While we focus on metrics that offer good precision, we cannot ignore the fact that some of the smells are not detected by the metrics devised in this work.

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 presented in this chapter. 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, refactoring 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 refactoring actions. To account for this phenomenon, we manually analyzed a subset of the results to ensure the soundness of the process.

Finally, the threats to external validity, regarding the generalization of the results, concern mainly the choice of the projects analyzed. Indeed, by 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.

4.6 Conclusion

The goal of this chapter is to shed light on the smell occurring in SUITs. To do so, we have defined a catalog of SUIT smells based on the knowledge providing by both practitioners and academia through a multivocal literature review. This process leads to a catalog of 35 SUIT smells. For 16 of these smells, we derive metrics to characterize the presence of symptoms and their refactoring in the test code. Our results show that three symptoms *Hardcoded Values* (90% of the tests), *Over Checking* (between 75% and 80% of the tests), and *Sneaky Checking* (70% of the tests) are prevalent in both industrial and open-source projects. Furthermore, *Hardcoded Values* not only appear in most tests but when they are present tend to be in a large quantity with median values ranging between 50 and 70 occurrences per test. While these symptoms are largely present in the test code base, their refactoring, on the other hand, remains low where for half of the categories less than 5% of the symptomatic tests are ever subjected to refactoring. *Missing assertion* is a unique exception with between 70% (open-source) and 90% (industrial) of the symptomatic tests being refactored. Interestingly, while refactoring actions are rare, test smells like *Narcissistic* and *Middle Man* still disappear from the test code base as a side effect of the replacement of 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 more rigorous comparison between the two sets of projects, we observe significant differences. Indeed, both in terms of diffusion of the symptoms and refactoring operations both types of projects differ. These results can be explained 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 will have an impact on our partner development process which already started using our tooling to address some of the bad designs they observe in their test codebase. 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 test code.

5

Towards Robust Locators

When targeting web front end, test scripts rely often on DOM-based locators to identify elements in the web page to interact with. Introducing cosmetic or non-functional changes may break the locator, and consequently the test scripts depending on it. In this chapter we present HPath, a new technique to generate DOM-based locators exploiting the rendering engine to remove the reliance on internal structural details of the DOM, thus, generating locators more robust against SUT evolution.

Contents

5.1 Introduction

In Section 1.1, we define a GUI-based test as a sequence of test steps where each test step consists of a triple: an action to be performed, the GUI element on which to perform the action, and an optional value passed to the element. Consequently, for each step, the test has to be able to uniquely identify the GUI element it has to interact with. The literature identifies three locator strategies, *i.e.*, ways to locate GUI elements in order to interact with the SUT [Bos14; LSR$^+$]. Each locator strategy is referred to according to its generation, all raising different concerns in terms of fragility to SUIT evolution.

The first-generation locators are based on screen coordinates for interacting with the application. While this technique was good for Record & Replay, the high maintenance cost of this approach drove away most practitioners to the benefit of the two next generations.

The second-generation locators, also known as property- or DOM-based locators, rely on properties of GUI components in order to locate them. Here, each page is associated with a DOM offering a tree-like structure that can be navigated by means of XPath queries or Cascading Style Sheet (CSS) selectors. Exploiting the DOM structure instead of directly targeting the rendered page leads to locators not affected by the properties of the rendering artifacts, such as screen resolution. However, CSS selector and XPath share one pitfall: they rely heavily on internal properties of the elements they visit (e.g., its attributes or its position in the DOM tree and hierarchy of tree nodes). Despite being a reasonable strategy for most tasks they are used for (e.g., information retrieval, content formatting), this approach can be problematic in the case of automated GUI testing where the reliance on these attributes leaks structural details of the page that should not be present in the tests. Previous research [TDS$^+$13; HRT16] has shown that this leakage of structural details makes tests more sensitive to changes not related to the behavior of the application, resulting in test breakage.

While second-generation locators are wildly adopted in industry, a third-generation offers promising potential [AFR15]. This third-generation locators, adopted in VGT, uses computer vision to locate elements through the bitmap graphics shown to the user at runtime. This technique was born as an answer to the fragility of second-generation tools generating high maintenance costs under SUT evolution. Indeed, relying on computer vision allows third-generation locators to be more flexible under structural changes. However, while this technique is more adaptive to underlying structural changes than second-generation locators, minor changes in the GUI representation (*e.g* minor changes in the layout or in the color of GUI components) might break tests relying on the exact representation of GUI elements. Furthermore, relying on computer vision to identify GUI elements remains a non-trivial task, even though considerable progress has been made, notably with the use of machine learning [WFB19].

Hence, while second- and third-generation locators each offer their own advantages, they both still contribute to test fragility [AD17; AKR18]. As mentioned in Chapter 2, to overcome the test fragility problem, academia has investigated two families of approaches: generate tests more robust to SUT evolution [MPR$^+$11; TDS$^+$13; LSR$^+$14; YTS$^+$14; LSR$^+$15b; LSR$^+$16; ZHH$^+$18] or automatically repair tests following breakages [CZV$^+$11; SYM18; KTN19]. In this chapter, our contribution falls in the former line of research as we present an attempt at reducing test breakages resulting from SUT evolution.

To this end, we propose a novel second-generation locator, HPath, which is similar to the XPath specification but exploits rendering properties unique to HTML documents. The practical benefits of HPath can be measured via its capability to generate more robust locators than the current second-generation techniques. The idea behind it is to prune the DOM tree by computing which elements are rendered. Hence, while the algorithm works in a similar fashion as absolute XPath to traverse the tree, it works on a rendered tree instead of the original DOM. Furthermore, the remaining nodes only keep a subset of their properties, the ones that affect the display. Hence, by this tree transformation, HPath relies on similar attributes as third-generation locators (no reliance on structural details) while keeping the advantages of second-generation locators (tree representation which is easy to traverse).

Before presenting HPath, we formally introduce each of the elements from the presentation layer of a web application which are exploited by the algorithm and the current approach to traverse it.

5.1.1 Document Object Model

The DOM is a standard maintained by the World Wide Web Consortium (W3C) until 2004 and then took over by the Web Hypertext Application Technology Working Group (WHATWG). The goal is to provide a common API which can be used with different languages in a broad range of environments to generate well-formed eXtensible Markup Language (XML) documents. According to the definition provided by the W3C, the DOM defines the logical structure of documents and the way a document is accessed and manipulated. In this definition, a *document* is the representation of a set of information that may be stored in diverse systems which would traditionally be described as data [Con04]. In other words, the DOM formally defines the representation layer of web applications. This property is what makes the DOM an ideal target to generate locators.

5.1.2 Hypertext Markup Language

HTML is a standard markup language derived from the XML. It is maintained by the WHATWG and designed to display the DOM in a web page. In its current implementation, HTML5, it specifies a set of more granular content models which allow improving support for multimedia and while maintaining good machine parsing capabilities, ease its readability by humans. The specification of HTML5 can be found in the HTML Living Standard documentation [Gro21].

The HTML syntax consists in a tree differentiating two types of nodes: (1) encoded marker (tags) differentiating bits of information composing the document or defining anchors for multimedia content and referred to as elements E and (2) any other types of nodes, N, containing data[1] or lower-level structural properties[2] of the document.

The root of an HTML document tree is an *html* element, e_{html}. The standard enforces the presence of two unique elements as direct children of e_{html} that can appear only once in the document: a *head* element, e_{head} and a *body* element, e_{body}. e_{body} is the sectioning root that represents the content of the document. With HTML5 a lot of semantic has been added to children elements of e_{body}. In this work we follow the categories presented in the HTML elements reference [con20; Gro21] (Table 5.1).

A specificity of an HTML document is its focus on the presentation of the data. Thus, a mechanism to apply styles and positioning to elements under the sectioning root, e_{body}, was introduced under the form of the CSS. CSS defines a set of styling rules that are to be applied to target elements described by a CSS selector (see Section 5.1.3).

Finally, the HTML standard reserves two elements with no meaning by themselves: the div element, e_{div}, from the Content Sectioning category and the span element, e_{span}, from the Inline Text Semantics category. These two types do not hold any semantic meaning. Both these elements act as placeholders to mark up user-defined semantics common to their children through the use of global attributes, *e.g.* class, id, or name. This specific semantics can be useful when used with styling where a subtree can be targeted by a style rule or when interacting with a script to apply some processing to a portion of the document.

Figure 5.1 offers an example of a HTML5 Document. Line 1 defines the type of document, here notifying the parser that the document follows the HTML5 standards. The element e_{html} (Line 2-17) contains to children, e_{head} (Line 3-9) and e_{body} (Line 11-16). Within e_{head} we observe the presence of an element e_{style} from the *Document Metadata* category, which defines some CSS properties for the document. e_{body} contains two elements: e_{h1} (Line 12), from the category *Section Content* which is used to display a title, and e_{div} (Line 13-15) which does have any semantic in itself but is used to apply the style defined in the e_{style} thanks to its `class` attribute ("someStyle"). Within e_{h1}, we observe the presence of text

[1]text node, CDATA section node, comment node,.

[2]attribute node, processing instruction node, document node, document type node.

Table 5.1: Categories of elements defined by the HTML elements references. When the scope of our category differs from the HTML element references, the original merged categories are mentioned in footnote.

Category	Description
Document Metadata	Elements encapsulating data that are not present in the page, but rather, affects the way content is presented or provides additional information about the document. *e.g.* E_{link}, E_{meta} and E_{style}.
Content Sectioning	Elements providing landmark to organize the content of a document into logical pieces. According to the W3C recommendation, the structural information conveyed visually to users should be represented programmatically by the appropriate sectioning element defined by the ARIA landmark roles[Con14]. *e.g.* E_{aside}, E_{footer}, E_{nav}, etc.
Text Content[i]	Elements organizing blocks of content, typically text. *e.g.* E_{li}, E_{p}, $E_{blockquote}$, etc.
Inline Text Semantics[ii]	Elements formating text such as bold, italic, etc. There are also knowon as *inline* content. *e.g.* E_{em}, E_{strong}, E_{cite}, E_{small}, etc.
Embedded Content[iii]	Elements allowing to embedded multimedia resources and other content. They provide information about where to load the content from and how to display it. *e.g.* E_{img}, E_{object}, E_{embed}, E_{video}, etc.
Interactive[iv]	Elements with which a user can interact with. Typically it represents input elements. *e.g.* E_{input}, E_{button}, etc.
Scripting	Elements allowing to integrate scripting in order to create dynamic content and Web application. *e.g.* E_{canvas}, $E_{noscript}$ and E_{script}

[i] Text Content and Table Content. [ii] Inline Text Semantics and Demarcating Edits. [iii] Embedded Content, Image & Multimedia, SVG & MathML and Web Components. [iv] Interactive and Form.

Figure 5.1: Example of HTML Document

```
<!DOCTYPE html>
<html>
    <head>
        <style>
        .someStyle {
          border: 5px outset red;
          text-align: center;
        }
        </style>
    </head>
    <body>
        <h1>Section Header</h1>
        <div class="someStyle">
          <p id="myTarget">Text <span style="color:red">in</
    span> a div.</p>
        </div>
    </body>
</html>
```

node, n_{text} ("Section Header"). Similarly, e_{span} (Line 14) applies some style to the text within it ("in") but in this case the style is directly provided by the attribute `style`.

5.1.3 DOM-based Locators

A DOM-based locator can be defined as a query language for selecting a subset of targeted nodes $N_{target} \subseteq N_D$ where N_D is the set of nodes contained in the document D. Hence, a DOM-based locator is query on a tree D returning a set of nodes $N_{target} \subseteq N_D$ where the size $|N_{target}|$ varies between 0 and $|N_D|$. The HTML standard describes two ways of querying a set of nodes $N_{target} \subseteq N_D$ in a HTML document: XPath and CSS Selector.

XPath

XML Path Language also known as XPath came from an effort to provide a common syntax and semantics to identify parts of an XML document[Con99]. Thus, the query language is not bounded to HTML document but rather to any XML compliant document[3]. To query N_{target}, XPath relies on the concept of location path. A location path[GLS02] is a series of successive steps traversing the XML tree to uniquely identify a subset of its node. Thus the location path is a query expression to identify specific nodes from a tree. In the work, we use both location path and query expression interchangeably.

At each step, a set of nodes, referred to as context nodes, are uniquely identified. The direct left step of the context node describes the parent node and the right step describes child nodes. During the evaluation of a context node, predicates allow filtering a subset of nodes among their siblings (*e.g.* all nodes with a specific attribute). Hence during the location path resolution, the path is traversed from left to right, filtering out part of the tree until it reaches the right-most step and returns the result of the query. If the left-most step is the root of the tree, we call it an absolute XPath, otherwise, if the left-most step describes the root of one or many subtrees in the document, it is said to be a relative XPath.

Each step can be defined by a selection node criteria, an optional axis relation, and an optional set of predicates[BCD+03]. The selection node criteria defines the element tag from the HTML document. An axis represents a relationship to the context node, and locates relative nodes (parent, self, attributes, preceding, etc.) in the document. The predicate is an expression contained between brackets after the node criteria allowing more fine-grained selection during the evaluation of the step. Predicates can express logical expressions, arithmetic operations, or string manipulations [GKP+05].

[3]Note that the W3C describes a slightly modified version of the XPath 1.0 standard to interact with HTML documents.

Going back to the example presented in Figure 5.1, lets assume that N_{target} is e_p. Using an absolute XPath, the query takes the following form: `/html/body/div/p`. However, note that because e_p is unique in the HTML document, in this case, using a relative XPath can be easy as: `//p`. Unfortunately, the introduction of a subsequent e_p in the document would change the result of the latter query which would now returns more than one node. To circumvent this limitation, let's exploit the attribute `id` of e_p. Thus, by relying on this attribute, we can query the document using the following XPath: `//p[@id='myTarget']`. The last XPath can be translated to: retrieve all element from the document designating a paragraph (e_p) for which the value of the attribute `id` equals to the text "myTarget".

CSS Selector

Contrarily to XPath that relies on location paths, CSS Selector is a structure that determines which elements are matched in the document tree by a pattern described by the locator. Here, the locator is a chain of one or more sequences of simple selectors separated by combinators[Con18] represented by ">". The CSS standard defines 6 types of simple selectors supported by the standard: (1) type selector, (2) universal selector, (3) attribute selector, (4) class selector, (5) ID selector, and (6) pseudo-class. The type selector qualifies an element based on its tag, a class selector based on the class attribute of the element and an ID selector relies on the id attribute of the element. A universal selector is a wild card character (*) allowing to match any substring. The attribute selector is a generalization of the ID selector and the class selector where the name of the attribute and its values are defined. Finally, the pseudo-class concept is introduced to permit selection based on information lying outside the HTML document or that cannot be expressed using other simple selectors (*e.g.* a:visited which allows targeting an anchor element, E_a, that has already been visited).

In the example from Figure 5.1, we can observe an CSS Selector at Line 5. Indeed, `.someStyle` present in e_{style} is a pattern that allows to apply the style describe in Line 6-7 to all the nodes which match it, *i.e.* all nodes with a `class` attribute for which the value is equal to "someStyle". In the case of the example, this pattern is matched by one node, e_{div} at Line 13.

Both XPath and CSS Selector rely on the internal properties of the elements they target by exploiting the elements attributes (id, class, etc.) which is the cause of their limited flexibility to DOM evolution. In the following section, we present HPath aiming at alleviating this limitation of DOM-based locators.

5.2 HPath

Relying on the HTML standard described in section 5.1.2, we proposed HTML Path Language (HPath) a new query language to locate nodes in an HTML

Table 5.2: BNF of HPath.

`LocationPath`	`:=`	`RelLocationPath | '/' RelLocationPath?`
`RelLocationPath`	`:=`	`'/' Step | RelLocationPath '/' Step`
`Step`	`:=`	`NameTest Predicate? | NodeType '(' ')'`
`NameTest`	`:=`	`Literal`
`Predicate`	`:=`	`'[' PredicateExpr ']'`
`PredicateExpr`	`:=`	`Number | FunctionCall`
`FunctionCall`	`:=`	`FunctionName '(' ')' '=' '"' Literal '"'`
`FunctionName`	`:=`	`'label' | 'legend' | 'caption' | 'figcaption'`
`NodeType`	`:=`	`'text'`

document. As its name suggests, our approach is similar to the XPath specification but tailored for HTML documents. While the target is the same as the CSS Selector (both targeting HTML documents), our scope is different. Indeed, our query language aims at providing test automation engineers more expressive and more flexible locators for GUI-based testing. In other words, the goal of our approach is to provide locators that leak as little structural details as possible by relying on properties of the HTML nodes that are rendered on the page to perform the query.

The intuition behind HPath lies in the fact that, if only rendered properties are permitted in the query, then the resulting locators should be more resilient to iso-functional structural changes. As a consequence, our approach helps to reduce GUI-based test fragility [TDS+13; HRT16]. Moreover, relying on rendered properties (*e.g.* the label of an input field) makes the locator (and, by extension, the test script) easier to understand as the reliance on structural properties has been minimized.

Table 5.2 presents the grammar of HPath in Backus–Naur Form (BNF). HPath is expressed as a location path composed of a series of steps (in a XPath fashion, yet using different predicates). Thus, HPath navigates in the HTML documents based on the properties of the nodes it traverses. Each step comprises a *NameTest*, which defines the type (tag) of the context element associated to an optional *Predicate* or a node $N_D \notin E_D$. Note that only text nodes are supported since they are the only ones being rendered on the page. *Predicate* returns a boolean value filtering out parts of the tree during path traversal. HPath accepts a textual input for the resolution of the predicate (*Literal*) and returns whether or not the value matches the property of the context node.

In the remaining of this section we present the properties of HPath allowing it to generate more flexible location paths. We use the notation $E_{\texttt{type}}$ to describe a set of elements of type `type`, and $e_{\texttt{type}}$ to refer to a specific element of such a set.

Figure 5.2: Example of HTML Document with elements that are not affecting rendering

```html
<!DOCTYPE html>
<html>
    <head>
        <style>
        .someStyle {
          border: 5px outset red;
          text-align: center;
        }
        </style>
    </head>
    <body>
        <div id="for-robot">
            <h1>Section Header</h1>
        </div>
        <div class="someStyle">
          <p id="myTarget">Text <span style="color:red">in</span> a div.</p>
        </div>
    </body>
</html>
```

5.2.1 Node traversal

Not all nodes in an HTML document are used for rendering. Indeed, placeholders can be introduced in the DOM to make pages dynamic (animation, dragging, etc.) or to provide better readability for web robots (e.g., Search Engine Optimization). Thus, by definition, these nodes should not appear in the location path when creating flexible locators for GUI-base testing.

Therefore, the node traversal is done on a rendering tree, generated by the HPath engine, where all elements not affecting the rendering flow are pruned out. Referring to the semantic offered by the HTML standard, two types of elements, E_{span} and E_{div}, do not hold any intrinsic semantic by themselves and act as placeholders. Indeed, if no styling is applied to these elements, they are ignored by the rendering engine[Gri19]. Consequently, during the tree traversal, HPath ignores any e_{span} or e_{div} that will not affect the rendering flow.

Because, when performing GUI testing, the locator is computed after the processing of all styling and rendering operations. To avoid misclassifying any node as not rendered, HPath relies on the browser engine to prune any nodes not affecting the display of the page. This step leads to a new HTML document which only contains rendered node $N_R \subseteq N_D$. Thus, while HPath traverses the trees in a similar fashion as absolute XPath, it does so on a different tree.

Figure 5.2 shows an example of a HTML document containing an element (in red) not rendered by the web engines but is in any other aspect equivalent to the one presented in Figure 5.1. In the case of HPath, both document do end up generating the same working tree. Thus, lets assume we target e_{h1}, using absolute XPath, the query takes the form: `/html/body/div/h1`. However, in the case of HPath, the query takes the form: `/html/body/h1`.

5.2.2 Predicates

While computing predicates or any other node filtering mechanism, DOM-based locators usually rely heavily on implementation-dependent properties of the elements they visit (*e.g.* id or class). However, these attributes can change without affecting the functionality of a page or any of its visible parts. For example, modifying the id of a button, if not affected by a style rule, will not have any effect on its rendering. Thus, HPath defines a few selected predicates relying on rendered properties, which can take the following forms:

- **label:** Most of the interactive elements $E_{interactive}$ can be associated with a label element, E_{label} to provide a textual clue about their functionality. The HTML standard defines two strategies to link those elements: (1) e_{label} is a direct parent of $e_{interactive}$ or (2) e_{label} is a sibling of $e_{interactive}$ where the value of the attribute *for* of e_{label} equals the value of the attribute *id* of e_{input}. The **label** predicate extracts the textual content (see Section 5.2.3) of e_{label} associated with the context $e_{interactive}$. An example of the two forms is provided in Figure 5.3. Here, the query `//input[label()='City']` retrieves both e_{input}.

Figure 5.3: Extract of HTML document exhibiting input element

```
<form>
    <fieldset>
        <legend>Address 1</legend>
        <label for="city-1">City
            <input type="text" name="city-1" id="city-1">
        </label>
    </fieldset>
    <fieldset>
        <legend>Address 2</legend>
        <label for="city-2">City</label>
        <input type="text" name="city-2" id="city-2">
    </fieldset>
</form>
```

- **legend:** Related children elements of a form element (E_{form}) can be grouped together through the use of fieldset element ($E_{fieldset}$). Any $e_{fieldset} \in E_{fieldset}$

can be associated with a textual value defined in a legend element, e_{legend}. According to the standard, if e_{legend} is a direct child of $e_{fieldset}$, then it is associated to the parent $e_{fieldset}$. The **legend** predicate extracts the textual content of a e_{legend} associated to the context $e_{fieldset}$. An example is provided in Figure 5.3. In the example, as shown above, when only relying on e_{label} to select the input retrieve the two e_{input}. To refine the selection to the first address, the query can be expressed using e_{legend} as: `//fieldset[legend()='Address 1']/input[label()='City']`.

- **caption:** Table elements, E_{table}, can be assigned a caption through the use of a caption element $E_{caption}$. According to the standard, if a $e_{caption}$ is a direct child of a e_{table}, then it is associated to the parent e_{table}. The **caption** predicate extracts the textual content of a $e_{caption}$ associated to the context e_{table}. An example is provided in Figure 5.4. Here, the query `//table[caption()='Locator breakages']/tr[1]` retrieves the first row of the table.

Figure 5.4: Extract of HTML document exhibiting caption element

```
<table>
    <caption>Locator breakages</caption>
    <tr>
        <td>Name</td>
        <th>HPath</th>
        <th>XPath</th>
    </tr>
</table>
```

- **figcaption:** Figure elements, E_{figure}, can be assigned a caption through the use of a caption element $E_{figcaption}$. According to the standard, if $e_{figcaption}$ is a direct child of e_{figure}, then it is associated to the parent e_{figure}. The **figcaption** predicate extracts the textual content of $e_{figcaption}$ associated to the context E_{figure}. An example is provided in Figure 5.5. Here, the query `//figure[caption()='Length of the Locators']/img` retrieves the image.

Figure 5.5: Extract of HTML document exhibiting figcaption element

```
<figure>
    <img src="locator-length.jpg" alt="Locator␣Length">
    <figcaption>Length of the Locators</figcaption>
</figure>
```

If no predicate can be generated to compute the location path, then the resulting path resembles the one that would be generated by the XPath algorithm, i.e., with only positioning predicate and no axis. However, while both expression present the same syntax in such case, the difference is that HPath does not compute the path on the actual DOM but on the pruned DOM tree (as described in Section 5.2.1).

5.2.3 Content Extraction

In the current implementation of HPath, all predicates (except the positioning predicate) rely on textual information visible on the page. To extract the content of a text under an element, XPath extracts the content of the text node, n_{text}, which is a direct child of the context element. HPath goes beyond this procedure and analyzes styling information contained in the elements from the Inline Text Semantics category (see Table 5.1). HPath takes advantages of this semantic and filters out any element from this category when extracting text. The resulting n_{text} are then merged together to reconstruct the text that will be used by the predicates for evaluation. To query $e_i nput$ from Figure 5.6, one can write the following query: `//input[label()='City']`.

Figure 5.6: Extract of HTML document exhibiting inline text semantics elements

```
<div>
    <label for="city-2">
        <i class="fa␣fa-home"><b>City</b>
    </label>
    <input type="text" name="city-2" id="city-2">
</div>
```

5.3 Research Questions

HPath makes the assumption that some types of web elements are typically more targeted than others by SUITs and as such some elements can be removed from the DOM. Indeed, removing text formatting and non-displayed elements while being a reasonable assumption, might remove interesting elements from the DOM tree. Thus, we validate this assumption by analyzing which categories of elements are targeted by DOM-based locators in SUITs and ask:

RQ1: *Which categories of elements are predominantly targeted by GUI-based test scripts?*

Next, we evaluate our new approach, HPath, and compare its ability to exploit the properties of web elements through its predicates. The goal is to measure to which extend the query expression can be compressed but also to what extent HPath can generate meaningful predicates. To put these results in perspective, we compare it against two algorithms generating XPath, namely, *absolute XPath* and *Robula+*. This leads to the following research question:

RQ2: *Which element properties are exploited by the different location path strategies?*

Finally, we investigate the resilience against SUT evolution of HPath. The main goal of HPath being to generate query expression resistant to minor structural change in the SUT. Thus, to assess its robustness to changes, we compare it to the two XPath generation algorithms used in RQ2 and express the final research question as follow:

RQ3: *What is the resilience against SUT evolution of the different strategies?*

5.4 Research Protocol

This section presents the experimental setup for the case studies to answer our research questions. The goal of this chapter is to analyze the impact of using second-generation locators relying on rendered features. Thus, we rely on two main concepts: the properties of an element HPath can leverage to generate a query and its resilience to changes happening during the SUT evolution. In this study, the properties are measured by the length of the location path and the presence of predicates in any step of the location path. The resilience to change of a locator is evaluated by its propensity to avoid breakage following a change in the HTML document.

5.4.1 Projects Collection

To select adequate projects for a case study, we mine repositories from Github, using its search API which allows applying queries to filter the result from the Github database. Because of the restrictions of the mining process described in Section 5.4.2, we have to isolate projects with tests written in Java and exercising the Selenium API. Furthermore, both application code and test code need to be versioned in the same repository to be able to go back in time and run test suites for previous versions.

Moreover, because of the dynamic nature of our analysis, we need to be able to compile the projects and run them on successive versions. Unfortunately, despite our best effort, most of the projects we gathered could not be executed in our experimental setup (missing dependencies, outdated build tools). Thus, this process yields two large web-based open-source systems: OpenOLAT[4], a web-based e-learning platform, and MISO LIMS[5], a lab information management system designed for tracking next-generation sequencing experiments. Both projects rely on a Java backend that is communicating with a Relational Database Management

[4]https://github.com/OpenOLAT/OpenOLAT

[5]https://github.com/miso-lims/miso-lims

Table 5.3: Web Applications used in our experiments. Column *#Releases* presents the number of releases collected, *KLoC* is the number of thousand lines of code in the last release, *#Tests* is the rounded average number of integration tests per release and *#Locators* is the total number of locators recorded.

Project	**#Releases**	**KLoC**	**#Tests**	**#Locators**
OpenOLAT	8	1308	154	216424
MISO LIMS	57	326	8	15672

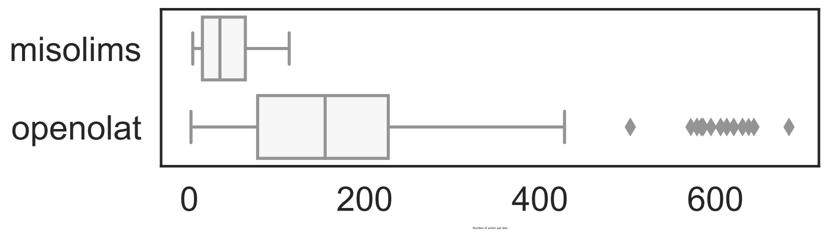


Figure 5.7: Number of actions triggered by test.

System (RDBMS). For the front end, both use HTML, CSS, and Javascript however MISO LIMS is relying on the HTML4 standard while OpenOLAT is using the HTML5 standards. Finally, both projects adopt a multi-tier architecture relying on different services working in concert.

Table 5.3 presents some key metrics of the projects under study. The 57 versions of MISO LIMS were collected between March 2019 and November 2020 and the 8 versions of OpenOLAT span from November 2018 to August 2020.

The low number of versions for the OpenOLAT project (8 versions) is due to the fact that earlier versions could not be compiled (Maven Plugin not supported). While both projects are quite large in terms of lines of code, we can see that the developers of the project MISO LIMS do not rely heavily on GUI-based testing, with an average number of 8 tests per version.

Figure 5.7 presents the number of actions triggered by the tests for each project under study. In average, the number of actions triggered by a test is 170 for the OpenOLAT project and about 42 for the MISO LIMS project. One intriguing point shown by the figure is the presence of tests not containing any interaction with the application through Selenium. These are tests not targeting the SUT through the user interface but rather directly interacting with the API of the application or the RDBMS itself, thus falling out of the purview of SUITs.

5.4.2 Mining Locator Breakages

We developed a tool called Mercator[6] to capture the state of the pages and the DOM-based locators used to interact with it in the test suites.

Selenium Instrumentation

More specifically, Mercator instruments method calls *findElement* and *findElements* from the class *RemoteWebDriver* offered by the Selenium API. Hence, at runtime, Mercator captures the following data every time a test exercises a DOM-based locator: (1) information about the state of the current web page *i.e.* complete dump of the DOM, size of the window and current Uniform Resource Locator (URL); (2) information about the test *i.e.* test name, call stack, inference on the previous call stack; (3) information about the locator *i.e.* locator strategy, locator value; (4) information about the project *i.e.* repository URL, commit id, previous commit id, commit date. This process is repeated for each version defined in the configuration of Mercator. For this study, we select every release of the projects. We restrict our study to releases because the projects rely on external dependencies which are not stable during the development cycle, thus, leading to code that cannot compile due to missing dependencies (SNAPSHOT). Furthermore, using releases increases the chances of observing genuine SUT evolution, thus, reduces noise.

HTML Dump

The most sensitive aspect of this process is collecting the DOM dumps. Indeed, as explained in Section 5.1.2, a HTML document can contain links to other resources such as multimedia and CSS style sheet stored externally. Thus, for the dump to be usable as a single HTML document file, during the instrumentation, a complete HTML document with no external dependencies needs to be computed. This is done by inlining all the CSS style sheets in the header, and providing placeholders for all multimedia content retaining their attributes, size, and position. Furthermore, all properties computed by JavaScript are inlined in the style attribute of the elements, and all script elements are removed from the dump. This is possible because pages are captured after the execution of all JavaScript routines which may alter the structure of the DOM tree. Finally, comment nodes and any formatting of the HTML document not affecting its content (line returns between to subsequent elements) are removed to limit syntactic differences between dumps.

One caveat encountered during instrumentation is the non-negligible amount of time (between 200ms to up to 1500ms during our experiments depending on the complexity of the page) required to create the dump. This delay causing tests to run slower might break some which are bounded by a maximum time budget (*timeout*).

[6] Available at `https://github.com/UL-SnT-Serval/mercator`

Table 5.4: Element selection strategies defined by the Selenium API.

Strategy	**Description**
class name	Locates elements whose class attribute contains the search value.
css selector	Locates elements matching the CSS selector.
id	Locate elements whose id attribute matches the search value.
link text	Locates anchor elements, E_a, whose inner text matches the search value.
patial link text	Locates anchor elements, E_a, whose inner text contains the search value.
tag name	Locates elements whose tag name matches the search value.
xpath	Locates elements matching the location path.

This leads us to use a strategy to monitor changes in the DOM tree to minimize the number of times the state has to be computed. Unfortunately, these changes failed to detect changes not altering the tree but modifying external properties of the elements (mostly hovering and selections in table) which lead to generating locators in our dataset that are not valid, 20.09% for MISO LIMS and 8.57% for OpenOLAT. We deem this tradeoff reasonable compared to the alternative where we observe about 35% of the tests failing due to our instrumentation.

With reasonable performance when instrumenting one version, the next step consists of devising a strategy to compute the lineage of a locator across versions. To achieve this objective, we rely on the changelog computed by the versioning system Git. Mercator enriches the data it records (locator and DOM) with information about the version and the test that is currently running. Then using the current stack trace and the changelog between the current version and the previous version (identified by their commit IDs) it is able to offer an approximation of the stack trace in the previous commit. To do so, it recomputes the stack trace in the previous version, taking into account line additions and deletions in the test code. This process is described in detail in Section 5.4.2.

Finally, the last step of this process is the extraction of the elements themselves. Each data point generated by Mercator contains a locator and its associated HTML document. Thus, by querying the DOM with the collected locators, we can retrieve the elements targeted by the tests relying on Selenium. The Selenium API offers different strategies to locate web elements that are shown in Table 5.4.

Pairs Creation

As shown in Figure 5.7, each test exercises a large number of actions, and simply relying on the test name does not offer the granularity required for this analysis.

Thus, we rely on *stack traces* to uniquely identify an element query in a version. The *stack trace* is the stack of subroutine state information from the test entry point to Selenium API call to locate an element. Each subroutine call is defined by a fully qualified class name, a method name, and a line number. Unfortunately, this method is sensitive to changes in the test code base from one version to another. Indeed, if lines are added or deleted before the call, then the line number is affected. Furthermore, when classes or methods are renamed, the subroutine information of the corresponding frame is modified. To minimize this effect, we compute the *previous stack trace* which is the stack trace that a call would have had in the previous version when accounting from the change extracted from the *unified diff* obtained through the version control system. We recompute the line number based on the line additions and deletions that occurred before the call in the file. If the target line is modified or there are class or method name changes, then, we consider that no mapping is possible. At this stage, each call to Selenium API can be identified by a *stack trace* and is associated with a *previous stack trace*. Thus, to create the pairs we link calls qualified by a tuple $< stack\,traces, version >$ to calls with a matching tuple $< previous\,stack\,traces, previous\,version >$. We will call $version_A$ and $version_B$, the first version and the next version of the pair respectively.

Breakage Detection

Once the pairs are created, we filter out any pair not exhibiting a SUT evolution. For each DOM we compute a hash code based on its string representation. Then, we filter out any pair where $hash_A = hash_B$ thus only accounting for pairs containing changes in the DOM. We consider a locator to break following a change in $< DOM_A, DOM_B >$, if querying DOM_B with $locator_A$ returns an empty set, *i.e.* fails to locate any element. We perform this operation on all pairs of the dataset where $hash_A \neq hash_B$. This process leads to a total of 30,436 pairs.

5.4.3 RQ1: Target Elements

The approach that we propose, HPath, makes the assumption that the interactions with the HTML document mainly targets visible elements, *i.e.* elements rendered on the web page.

Indeed, under our hypothesis, a test should interact with elements to navigate through the states of the web application then make assertions on visible elements to validate the proper execution of the test as a human would. Thus, not all elements of the HTML document hold the same value for the testers. To assess this hypothesis and answer RQ1, we extract the type (tags) of the elements targeted by the tests. Comparing this distribution with the one contained in the HTML document, we can assess which elements SUITs typically interact with.

Furthermore, previous work targets elements systematically[CDB15; LSR+15b;

AD17; EMS18] or generates test scripts[GXF09a; MPR$^+$11; KTN19] when a GUI test suite is not available. Unfortunately, studies relying on *in vivo* GUI test suites are usually conducted on industrial projects and the source code is not available[TDS$^+$13; YTS$^+$14]. While there exists studies tackling the question of the evolution and the characteristics of GUI tests [CSD$^+$14], they usually rely on static analysis and do not convey information about locators and their target elements. Thus, understanding which type of elements are targeted by testers would allow researchers to create benchmarks to evaluate their approaches by following the same distributions observed in other projects when no tests are available.

5.4.4 RQ2: Element Properties Used by Locators

To evaluate HPath, we compare it against two approaches to generate XPath: absolute XPath and Robula+. We select XPath algorithms because our approach is a derivative of XPath with the aim of addressing one of its limitations: its reliance on structural properties of the HTML document, making HPath more flexible to changes of the internal document structure.

The first algorithm, absolute XPath, builds an XPath where each step can only rely on a positional predicate. Thus, it does not rely on any attributes such as id, class, or name and can be seen as the most naive form of XPath generation. Note that when using the absolute XPath algorithm, the deeper an element is nested in a DOM tree, the longer the location path. Thus, this algorithm tends to leak the overall structure of the DOM, where any change in the node hierarchy of target elements will alter its absolute XPath representation.

Robula+ [LSR$^+$16] is a state-of-the-art XPath generation algorithm developed by the research community. The aim behind this algorithm is to generate robust XPath locators. It works with the assumption that the shorter the location path is, the less it leaks the overall structure of the DOM, thus making it more robust to changes in the DOM hierarchy. To generate the shortest location path possible, the algorithm starts with (`//*`), which selects every node, N_D, and then refine the expression in consecutive specialization steps until only the target candidate can be selected by the expression, $N_{target} \subseteq N_D$. Each refinement step is based on heuristics to improve the expressiveness and robustness of the final location path. During the refinement steps, the heuristics define the order in which Robula+ tries to select an attribute to build a predicate[7] and the attributes to ignore[8].

Note that other algorithms to generate robust XPath locators have been proposed in the literature, notably Montoto et al. [MPR$^+$11] and Thummalapenta et al. [TDS$^+$13]. The two alternative algorithms rely on similar properties as Robula+

[7]We follow the recommendations from the authors: *name*, *class*, *title*, *alt*, *value*.

[8]We follow recommendations from the authors: *href*, *src*, *onclick*, *onload*, *tabindex*, *width*, *height*, *style*, *size*, *maxlength*.

(id, name, class) but are less robust[LSR$^+$16]. As for other approaches relying on a combination of XPath and other strategies[LSR$^+$15b; AD17], they are considered out of scope since HPath could be integrated in such approaches. Furthermore, we consider third-generation approaches out of scope, because of their reliance on computer vision techniques which in itself bring a new set of challenges.

To characterize the properties exploited by the different approaches we proceed in two phases. First, we compute the length (number of steps) of the location paths generated by the three approaches. Indeed, previous work considers the addition of extra nodes in the location path as increasing the fragility of the path. For example, Leotta et al. [LSR$^+$15b] introduce the fragility coefficient as a function of the number of nodes traversed and in Chapter 4 we observe the SUIT smell sensitive locator which qualifies as bad practice to have any location path traversing more than one node.

Then, we conduct a fine-grain analysis of the predicates that are used by Robula+ and HPath (absolute XPath is omitted since it does not compute predicates). The goal here is to show which properties of the DOM elements can be exploited by the two approaches.

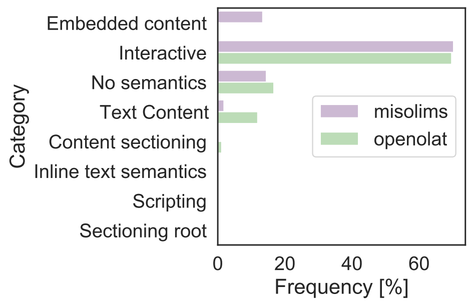
5.4.5 RQ3: Locators Resilience to SUT Evolution

Finally, we analyze which locators manage to successfully retrieve the element in the next version. In the cases of locator breakage, we perform a fine-grain analysis to isolate the cause of the breakage. This analysis is based on the structure of the location path where each step is composed of a node test and an optional predicate. Therefore, we compare $locator_A$ and $locator_B$ from the breaking pairs and extract their difference. The cause of the breakage is based on the type of changes observed between the two location paths. For example, if $locator_A$ is //div[@id="old"] and $locator_B$ is //div[@id="new"], then the cause of the breakage is classified as *predicate id attr.*. The list of all breakage causes is presented in Table 5.7.

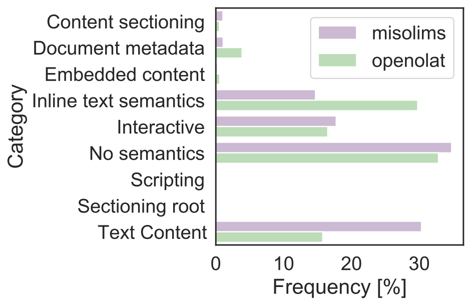
5.5 Experimental Results

5.5.1 RQ1: Targeted Elements

In this research question we evaluate the prevalence of each type of element targeted by GUI tests. Indeed, HPath makes strong assumptions about the type of elements being targeted by tests. For instance, elements from the Inline Text Semantics category and elements with no semantic attached (E_{div} and E_{span}) that are not rendered cannot be reached by our approach. Furthermore, two (*label* and *text*) out of the four predicates present in the current implementation specifically target elements of the Interactive category. The institution behind these assumptions is that GUI tests interact with the application in the same fashion as



(a) Targeted by GUI-based tests



(b) Present in DOM

Figure 5.8: Percentages of elements (a) targeted by GUI-based tests and (b) present in the DOM for MISO LIMS and OpenOLAT across all versions aggregated by category.

a user would, through visible GUI components.

Figure 5.8 presents the distribution of target elements grouped by their semantic categories. It contrasts the frequency of elements targeted by GUI tests (Figure 5.8a) to their overall frequency in the HTML document (Figure 5.8b). The discrepancy between the two shows that not all elements are of interest to testers. Indeed, the most prevalent categories in the HTML documents are No Semantics with 34.77% and 32.82% for MISO LIMS and OpenOLAT respectively, Inline Text Semantics with 29.76% for OpenOLAT and Text Content with 30.32% for MISO LIMS. However, Figure 5.8a shows a very different picture, where the overwhelming category is Interactive with 70.27% and 69.70% for MISO LIMS and OpenOLAT respectively, confirming our hypothesis.

However, looking at Figure 5.8a we observe a non-negligible amount of elements with no semantics accounting for 14.45% in MISO LIMS and 16.71% in OpenOLAT. After reviewing the tests it appears that some assertions and synchronization points target portions of the DOM defined by a e_{div}. Therefore, while they do not hold any intrinsic semantic in the HTML standard they do act as an anchor point that can be targeted by test scripts to assess the presence of an underlying group of elements. A more detailed discussion is presented in Section 5.5.2 regarding the impact on HPath.

Finally, two categories vary from one project to the other, namely, Embedded content and Text content. The presence of elements from the Embedded content in MISO LIMS (13.41%) is explained by clickable images under an anchor element, thus behaving like Interactive Elements. As for the elements from Text Content category present in OpenOLAT (11.91%), they are used when asserting that the SUT is in the expected state, thus, following our hypothesis.

To conclude this research question, we see that GUI tests typically target specific portions of the HTML document, namely, Interactive elements and visible components on the page (text and images). HPath, relies on relevant properties to specify these elements and offers good predicates when locating them. However, the proportion of target elements with no HTML semantic is not negligible, which could make HPath enable to locate these elements if they are not rendered on the screen. We investigate this possibility in the next research question.

5.5.2 RQ2: Element Properties Used by Locators

This research question investigates the relative performance in exploiting properties of target elements in the location path. HPath, absolute XPath and Robula+ formulating different hypotheses on which properties of an element can be used to locate it, we compare the three approaches.

Before being able to analyze which properties of the DOM HPath can leverage, we need to ensure that the target elements are reachable by the technique. Indeed, as shown in Section 5.5.1 around 15% of the target elements might not be rendered on the page. It appears that HPath fails to retrieve the locator in 11.58% of the cases for MISO LIMS and in 10.48% for OpenOLAT. These target elements not being rendered on the page, HPath is enabled to retrieve them. When observing the data, we see that assertions and synchronization points can target e_{div} to verify the presence of the underlying group of elements. However, in many cases, other elements could have been selected to yield the same result had the initial test automation engineer relied on HPath. For example, in the project OpenOLAT, a test waits for a calendar widget to be rendered on a page, thus waits for the containing e_{div} identified by its id (fc-view-container). The widget offering a series of named buttons (monat, tag, jhar) the test could rely on those to assess the availability of the widget. In the remaining of the discussion, only elements reachable by HPath are considered.

We can now start our discussion on the properties by analyzing how much of the tree hierarchy is leaked when generating the location path. Figure 5.9 displays the distribution of the length of the locators for the three strategies: absolute XPath, L_{AXPath}, Robula+, $L_{Robula+}$, and HPath, L_{HPath}. The figure shows a similar trend in both projects where absolute XPath yields the longest location paths with an average length of 8.6 and 18.7, Robula+ the shortest location path, with 1.6 and 1.0 and HPath lies between the two with 4.4 and 3.1, for MISO LIMS and OpenOLAT respectively. The lower performances of absolute XPath are to be expected since if no optimization can be done, both Robula+ and HPath generate a location path that is equal to the absolute XPath. Therefore $L_{HPath} \leq L_{AXPath}$ and $L_{Robula+} \leq L_{AXPath}$.

We compare the different distributions L_{AXPath}, $L_{Robula+}$ and L_{HPath} using

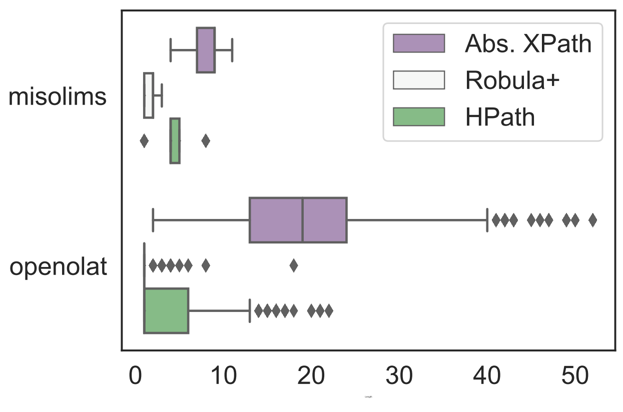


Figure 5.9: Length of absolute XPath, HPath and Robula+ for MISO LIMS and OpenOLAT across all versions.

statistical Wilcoxon signed-rank test[9] ($\alpha = 0.001$). The goal of this test is to assess whether or not there exists a statistical difference between the pairwise distributions. The results show that there is indeed a difference between all distributions, hence, the different strategies yield locators with different lengths. Then, we compute the pairwise effect size to determine how often lengths from one distribution are greater than in the other. The results show that in all the cases, the size effect is close to 1, *i.e.* all samples from one distribution are greater than in the other, with the exception of L_{HPath} and $L_{Robula+}$ for the OpenOLAT project where the size effect is 0.68. This is due to the fact that in many cases (58.52% for HPath and 97.72% for Robula+) both strategies manage to generate a location path of size 1, *i.e.* not leaking hierarchy details of the DOM.

Asides from the predicates, to other effects contributes to the longer length of HPath when compared to Robula+. First, in its current form, if no predicate is generated, HPath generates an absolute path. However, Robula+, always discards the root node e_{html} and often e_{body} since they are unique in the HTML document. Thus, HPath, can reduce its length the same way with no impact on its performance or changes in its assumption. Furthermore, if a *tag* is unique in the document, Robula+ creates a relative path from it, since it can directly be targeted. However, except for a few elements (e_{html}, e_{body}, e_{head}) the HTML standard does guarantee uniqueness. This can be seen as a limitation regarding robustness against change, where the addition of a second element of the same type can break the locator.

To conclude this discussion on the length of the location paths, we can see that strategies relying on different properties yield locators with different lengths.

[9]Shapiro-Wilk test ($\alpha = 0.001$) shows the non-normality of the distribution

Table 5.5: Properties of the context element leveraged by HPath and Robula+ predicates. *Count* is the number of locators using a property as a predicate and *%* is the percentage among all the locators collected.

Strategy	**Property**	**MISO LIMS**		**OpenOLAT**	
		Count	**%**	**Count**	**%**
HPath	*caption*	0	0.00	0	0.00
	figcaption	0	0.00	0	0.00
	label	0	0.00	31,048	16.03
	legend	0	0.00	22,354	11.54
	text	227	1.64	93,031	48.02
	total	**227**	**1.64**	**142,103**	**73.35**
Robula+	*id attr.*	5,161	32.93	143,334	66.23
	class attr.	412	2.63	48,483	22.40
	name attr.	207	1.32	150	0.07
	title attr.	0	0.00	3,094	1.43
	text	0	0.00	17751	8.20
	total	**5780**	**36.88**	**212558**	**99.80**

Relying on element attributes generates the shortest location path and HPath exhibits better performances for OpenOLAT than MISO LIMS, even though the tree is deeper (as shown by the length of the absolute XPath). This is a consequence of OpenOLAT using the HTML5 standard, where MISO LIMS implements HTML4, offering more opportunities for HPath to generate good predicates.

Therefore, we conduct a fine-grained analysis to understand which properties HPath and Robula+ are able to leverage to compute the location path. Table 5.5 shows which element properties are exploited in the predicates of both approaches. Where HPath relies on properties from the HTML5 semantics, Robula+ exploits mainly attributes of the context element to generate the predicate.

Rows *total* from Table 5.5 indicates how often a strategy is able to compute a predicate. We can see that in the case of OpenOLAT, both strategies often manage to compute predicate (73.35% for HPath and 99.80% for Robula+) but a lot less often in the case of MISO LIMS (1.64% for HPath and 36.88% for Robula+). Putting the results for MISO LIMS in contrast with the ones presented in Figure 5.9, we see that both strategies rely on other mechanisms to reduce the length. HPath relies on the rendering tree to prune e_{div} that are not rendered. On the other hand, Robula+ during each refinement step evaluates the current location path to see if it uniquely identifies the target element. Doing so, it is able to take advantages of the fact that some element types are unique in the document, leading to location path such as `"//img"` in documents containing a single e_{img} and manages to create a relative path 100% of the time. But again, relying on the uniqueness of an element type in a document can lead to locator breakages during SUT evolution.

The next point of our discussion addresses the type of predicate used by HPath. As expected, in the case of MISO LIMS, all predicates targeting HTML5 semantic elements cannot be leveraged and only the label property is used. In the case of OpenOLAT, two types of predicates are never used: *caption* and *figcaption*. Analyzing the HTML documents, it appears that none of them contain elements $e_{caption}$ or $e_{figcaption}$. The three remaining predicates *label*, *legend* and *text* are all related to forms (E_{legend}) and interactive elements (E_{label} and N_{text} child of Interactive element) and can be successfully used by the HPath predicate.

Robula+ on the other hands relies on the attributes of the element to compute the predicate. We see that almost all the elements of OpenOLAT offer at least one attribute that can be used (91.60%). However, in 8.20% of the cases, Robula+ relies on the text contained in the element. Note that the text extraction of Robula+ consists in retrieving the content the child n_{text} of the context element where HPath is relying on the process described in Section 5.2.3.

In conclusion, results show that HPath is able to compute a location path in 90% of the cases and when HTML5 semantics are available in the HTML document, HPath is able to successfully compute predicates for over 70% of target elements. In the cases where no HTML5 semantic is available HPath is still able to reduce the length of the location path by relying on the rendering tree it computes. Furthermore, even though relying on attributes properties offer more opportunities to generate predicate, we argue that those properties are less resistant to change in the SUT. Thus, we investigate this claim in the next research question.

5.5.3 RQ3: Locators Resilience to SUT Evolution

This third research question evaluates the impact of changes in the SUT on DOM-based locators. Relying on the methodology described in Section 5.4.2, we present in Table 5.6 the percentage of locators breakage following SUT evolution.

The results show that the two projects have very different profiles. Indeed, locator breakages depending on the type of the properties evolving in the SUT, they may vary from one development cycle to the next. However, note that the two projects are quite mature with over 15,000 commits in almost 10 years for OpenOLAT and over 4,200 commits in 9 years for MISO LIMS. This explains why no deep structural changes are observed over the periods we analyze as the low breakage count for absolute XPath suggests. Our goal being to spot iso-functional locator breakages, this stability ensures that when a breakage is observed, it is not due to a deep functional evolution of the page.

From Table 5.6, for MISO LIMS, Robula+ has the lowest number of breakages

Table 5.6: Locator breakages due to SUT evolution. *Count* is the number of locators that broke because of a change and *%* is the percentage among all the pairs collected.

Strategy	**MISO LIMS**		**OpenOLAT**	
	Count	**%**	**Count**	**%**
Abs. XPath	56	2.02	174	0.63
Robula+	10	0.36	17976	64.99
HPath	26	0.94	135	0.49

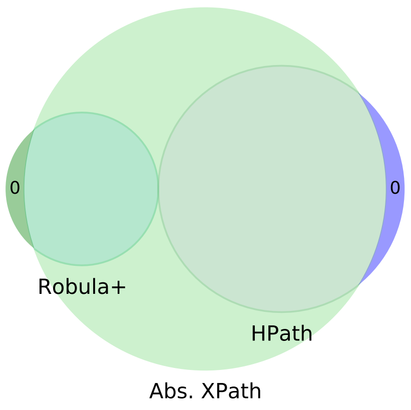
(10) and closely followed by HPath (26). Figure 5.10a shows the intersection of elements for which the locators break. It shows that absolute XPath is a superset of Robula+ and HPath suggesting that they only break following a change in the DOM hierarchy. Furthermore, there is no overlap between Robula+ and HPath, meaning that relying on different properties make the locators resilient to different types of SUT evolution (supporting the results from Leotta et al. [LSR+15b]).

On the other hand, the OpenOLAT project offers a very different picture. When constructing the HTML document to serve to the client, the backend of OpenOLAT automatically generates id, based on the version of the project, in order to ease communication with the database management system. HPath offers in this case the best resilience to change breaking only 0.49% of the time. Contrarily, Robula+, relying on element attributes, offers very poor performance breaking in 64.99% of the cases. Figure 5.10a shows that there exists an overlap between the different strategies but that in general they all react differently to changes.

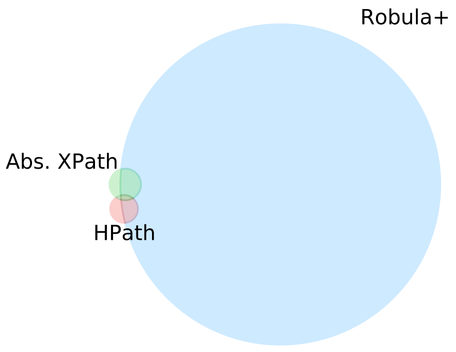
To better understand which properties of the elements causes the locators to break, Table 5.7 shows the origin of the failure of the locator. To compute the origin, we compared the breaking location path from $version_A$ to the passing one computed from $version_B$. The results of the fine-grain difference algorithm between the pair of locators point out the cause of the breakage.

In the case of MISO LIMS, most of the failures happened in the position predicate, this related to the hierarchy structure of the DOM. Again, in the case of OpenOLAT, the profile is more diverse. Robula+ is not affected by changes in the element types (NameTest) in comparison to the other approaches, but is sensitive to attributes changes, leading to poor resilience to the evolution of the SUT where breakage due to the id contributed to 97.61% of the breakage. Finally, when HPath is able to compute a predicate, it is resilient and never breaks in our dataset (Row *Any Predicate* from Table 5.7).

In conclusion, when HPath can generate predicates they lead to robust and flexible locators. However, it can be hard for the approach to extract the necessary



(a) MISO LIMS



(b) OpenOLAT

Figure 5.10: Overlap of locators having breakage for Absolute XPath (green), Robula+ (blue) and HPath (red).

Table 5.7: Properties causing locator breakages. *Count* is the number of locators breaking because of a property and *%* is the percentage among all the breaking locators for that strategy.

Strategy	**Cause**	**MISO LIMS**		**OpenOLAT**	
		Count	**%**	**Count**	**%**
Abs. XPath	*NameTest*	0	0.00	60	34.48
	Predicate position	56	100.00	114	65.52
Robula+	*NameTest*	0	0	11	0.06
	Predicate position	10	100.00	2	0.01
	Predicate id attr.	0	0.00	17,578	97.79
	Predicate class attr.	0	0.00	11	0.06
	Predicate title attr.	0	0.00	77	0.43
	Predicate text content	0	0.00	281	1.56
HPath	*NameTest*	0	0.00	119	88.14
	Predicate position	26	100.00	16	11.85
	Any Predicate	0	0.00	0	0.00

properties as shown in section 5.5.3. Therefore, while the results are promising, more research needs to be conducted to better exploit the rendering properties present in the HTML5 semantics. Furthermore, better reliance on the standard by developers would improve the performances of tool relying on the semantic of the language.

5.6 Threats to Validity

The internal threat to validity concerns the way evolution is computed. During our analysis of the HTML document pairs, we realized that some pairs only vary by a date or a timestamp unrelated to any of the targeted elements. While decreasing the overall percentage of breakage, because this effect is similar for all techniques, it does not have any effect on neither the ranking of the methods nor the number of breakages observed.

Furthermore, when we observe a breakage, we do not have the guarantee of the iso-functionality of the GUI element being targeted. Indeed, if the element is deeply modified or it might be expected for the locator to break. We limit the impact of this effect by using the Absolute XPath as a baseline and by ensuring that in the two versions the same test was executed against the SUT.

Finally, conducting our case study on two projects, the conclusions we draw may not hold true for other projects. This constitutes a major threat to the external validity for the generalization of our results outside the context of this study. However, the two projects have very different profiles, thus making the study covering boundary cases among the full range of possible projects. To further alleviate this limitation, we encourage other teams to replicate and extend our results.

5.7 Conclusions

We presented HPath, a novel DOM-based locator strategy to generate location paths more flexible for web testing. We have compared its potential to extract properties from the HTML document and its fragility under SUT evolution against state-of-the-art algorithm Robula+. Our results show that when HTML5 semantics are present, HPath can exploit rendered properties of web elements to generate expressive locators in 73.35% reducing locator breakages from 64.99% when relying on attribute properties to 0.49% with rendered properties. However, in its current form, it is not always able to extract rendered properties to create good location paths and leaks the hierarchical structure of the DOM for 41.48% of the elements.

While HPath offers clear advantages in its expressiveness (only relying on rendered properties) and flexibility (when compared to Robula+), it suffers from some limitations. Indeed, relying on very specific predicates, the approach is not always able to generate short and expressive location paths, which might expose more of the internal hierarchy of the HTML document. This effect is exacerbated when the targeted page is not relying on the current HTML5 standards. However, despite this limitation, our results show that relying on the structure of the rendered DOM still remains better than exploiting the internal attributes of the elements from the HTML document.

6

Ikora Inspector

Ikora Inspector is a static analysis tool for continuous inspection of KDT written using the Robot Framework syntax. We present the tool and the way it can be used to increase the quality of KDT code base. Then, we present a case study conducted at BGL BNP Paribas where the tool is introduced and used by QA automation engineers since 2019.

Contents

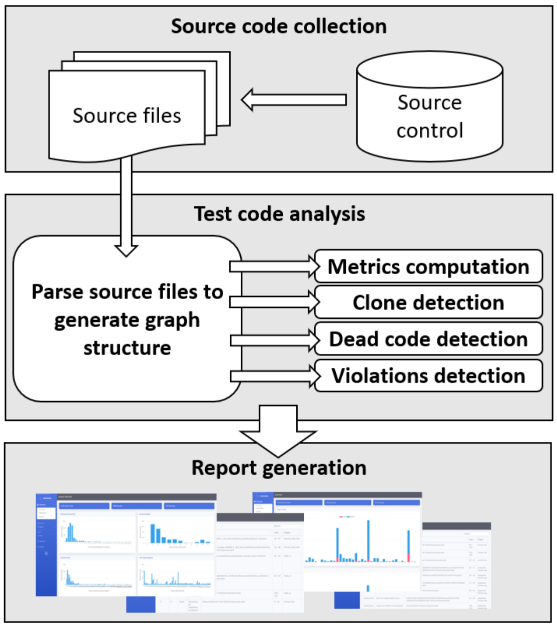


Figure 6.1: Main components of Ikora Inspector

6.1 Introduction

This chapter presents Ikora Inspector, a tool that aims at filling this gap. It inspects test code developed using Robot Framework [Rob20]. More precisely, Ikora Inspector aims at tackling the lack of static analysis tooling for KDT test suites that automation testers at BGL BNP Paribas, and at other companies, face. In Chapter 3, we analyze the evolution of such test suites and identify several challenges that impact test maintenance. Specifically, as the KDT test suites evolve, they become a complex software artifact, suffering from the same problems as typical codebases do, *e.g.* the presence of code duplication and dead code. Ikora Inspector addresses such challenges by continuously analyzing the test code for test smells and reporting Key Performance Indicator (KPI) about the project back to the testers.

6.2 Tool Description

Ikora Inspector [1] takes as input the Robot Framework test suites written in the Robot Framework syntax, see *Source code collection* component in Figure 6.1. Leveraging on the hierarchical structure of the keywords, each test can be represented as an ordered, directed, acyclic *tree* T with nodes $N(T)$ and edges $E(T) \subseteq N(T) \times N(T)$. The nodes $N(T)$ represent keywords and each edge $E(T)$

[1]Availble at `https://github.com/UL-SnT-Serval/ikora-inspector`

Table 6.1: Clone types

Label	Explanation
Type I	Identical keywords except for changes in whitespace, layout and documentation. The clone detection algorithm tags a keyword pair as Type I clones only in the case of an empty set of differences.
Type II	Keywords with a content syntactically identical except for step arguments and return values. The clone detection algorithm tags a pair as Type II clones only if the set contains differences of type *update step arguments* and/or *update step return values*
Type III	Superset of Type II clones, ignoring differences of type *update step*, *add step* and *remove step* if the step belongs to the category *logging.*
Type IV	Keyword performing the same sequence of actions on the SUT, regardless of the internal configuration of the tree. The clone detection algorithm tags a keyword pair as Type IV clones if the sequences of leaf nodes is strictly identical.

between two keywords denotes a "step": the parent keyword has the child keyword as a *step.* Since keywords can be used multiple times, by more than one tests, the test suite considered as a whole can be represented as a Directed Acyclic Graph (DAG), G, where each node is defined as one of:

- **Test Cases** Entry point into the graph, called by the framework to be executed. Each test case considered individually is the root of a *tree* T.
- **User Keywords** Internal nodes of the graph, they are created by the user, hence their name, to provide modularity and explainability to the test. Each User Keyword is composed of *steps*, which are calls to a User Keyword or to Library Keyword.
- **Library Keywords** Represent the exit point of the graph to perform concrete actions on the SUT, or the leaves of the tree representation of a single test. Provided by the framework or external libraries, they perform concrete actions, such as interactions with the SUT, logging or assertions.

We use the generated DAG as an intermediate representation of the test code to perform our analysis, see *Test code analysis* component in Figure 6.1. The tool supports multi projects analysis building a single graph for all the project while annotating each node with the project it belongs to. This representation allows the use of graph theory in order to generate metrics, detect violations, duplicated test code and dead test code. Note that while the tool targets the specific syntax of Robot Framework in this first version, extracting the parsing module would allow to support different syntax for KDT since all analysis are performed on a intermediate representation of the language.

The results of the analysis are presented in a static website composed of a series of dashboards providing information both at the inter- and intra-project level, see *Report generation* component in Figure 6.1. The content of the dashboards are defined with the help of our partners at BGL BNP Paribas, who are already using

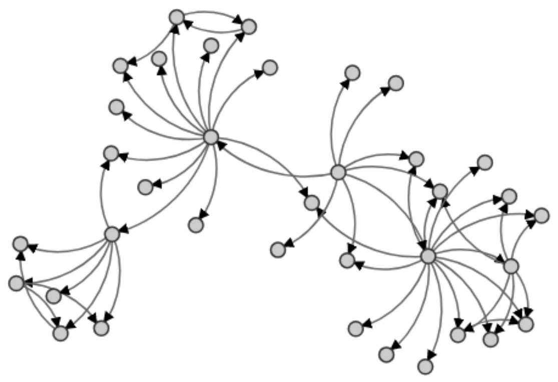


Figure 6.2: Projects Dependency Graph extracted from Ikora Inspector

the tool in production.

In the remaining of this section, we present the different type of analysis performed on the intermediate representation of the test code base.

6.2.1 Projects Dependency Graph

To improve modularity and re-usability testers build generic libraries containing low level functionalities that can be reused by different projects. The *project dependency* module of the tool provides a way to visualize dependencies between projects. The need for this visualization originated from what we consider as weakness of the Robot Framework language, namely, the way transitive dependencies are managed. Unlike popular languages like Java, C++ or Python, when a file is loaded by another one, all its dependencies are loaded and made visible as well.

Output The dependency graph page of the tool presents a complete project dependency graph, allowing the tester to assess the complexity of the test suites as well as detecting issues such as cyclic dependencies. Furthermore, as shown in Figure 6.2, the graph can become complex and difficult to analyze, therefore in each project page, we can observe a graph of its dependencies as well as projects depending on it.

6.2.2 Project Statistics

One of the criteria often linked to maintenance effort is the complexity of a project. The more complex a project and its components are, the harder it is to make it evolve and maintain the test code base. Detecting in time the growth of complexity of a project might prevent it from becoming to hard to manage. To this end, we defined a series of four metrics focusing on the complexity of the keywords composing a project.

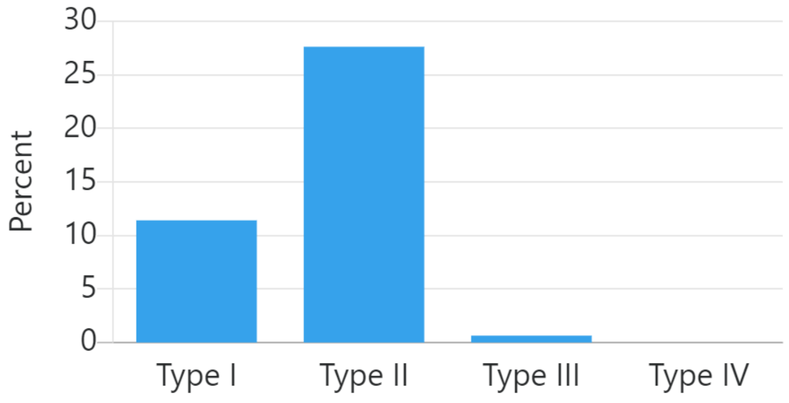


Figure 6.3: Percentage of duplicated lines extracted from the Summary Page of Ikora Inspector

Keywords Size

The *size* of keyword k (Test Case or User Keyword), is the number of nodes that exist on the subpath(s) from k to all the leaf keywords (Library Keyword).

Keywords Level

The *level* of keyword k (Test Case or User Keyword), is the maximum number of edges that exist on the subpath(s) from k to a leaf keyword (Library Keyword). According to the philosophy of KDT, the higher the value is, the more abstract the keyword. High-level keywords define business processes while low-level keywords define a concrete way of interacting with the SUT.

Keywords Connectivity

The *connectivity* of a keyword is a metric of re-usability among the keywords. Let keyword k belong to a *graph* G, then we calculate the connectivity of k by counting the number of nodes (keywords) in the subpath(s) from the entry points of G to k.

Test Cases Sequence Length

Let a test case TC be the root of a *tree* T. The *sequence length* of a test case TC is the number of concrete actions performed by the test, defined as the number of leaf nodes of the *tree* T.

6.2.3 Duplicate Test Code Detection

In Chapter 3, we observe a large amount of similar test code, *i.e.* clones. We observe that clones composed up to 30 percent of the total amount of keywords, indicating that almost one-third of the test code written is duplicated. Clones create several difficulties with inconsistencies during test code evolution generating bugs. Furthermore, we observed that about 50 percent of the strictly similar test code (clone Type I) evolve identically, indicating an added cost in the maintenance cost completely avoidable by refactoring.

The clone detection algorithm is based on the fine-grained changed algorithm defined in Section 3.3.2 which extracts the changeset between two keywords. The clone detection, ignoring changes impacting the documentation and the name of the keyword, detect the changes between the two keywords and assign a clone category defined in Table 6.1 for each pair of keywords. If a pair does not satisfy any of the definitions, then the keywords are not clones.

Output In the summary, see Figure 6.3, and in each of the project pages, statistics about the amount of duplicated test code are displayed in order for the tester to evaluate the level of duplication of the test codebase. Furthermore, a test code duplication page offers a table containing all the clones, clustered by groups of similar keywords. Each row contains information about the location of the keywords duplicated, the type of clones, and whether the clones are inter- or intra-application.

6.2.4 Dead Test Code Detection

During refactoring and evolution activities, the call graph might generate keywords not called by any other. In this work, we define dead test code any User Keyword or Variable never called in a *step*, in other words not connected to a Test Case, and therefore never executed. While building the graph representation of the test suite, the tool keeps track of all the callers and the callees for each keyword are variable. Therefore, detecting dead test code only consists of flagging User Keywords and Variables for which no callers were detected.

Output For each project, the percentage of dead test code is presented in terms of lines of code, see Figure 6.4. Furthermore, a dead test code page contains a table where each row contains information about the location of the unused User Keywords and Variables in order for the tester to be able to locate them and clean the test code.

6.2.5 Bad Practices and Error Detection

In its current version the tool allows to detect two types of bad practices specific to Robot Framework, 16 smells common to SUIT and two error types. However, conducting an analysis of the typical change patterns and the execution results, we intend to extend the set, specifically to address issues such as weak *locators* and weak *synchronization* points causing false positive and flakiness.

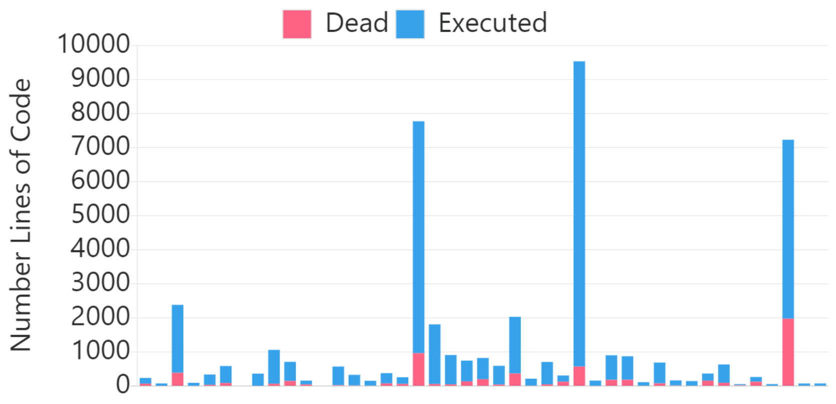


Figure 6.4: Number of lines of dead test code per project extracted from the Summary Page of Ikora Inspector

Duplicated Keyword Error

If two keywords have the same name in the same scope, it causes the program to crash during runtime. However, since keyword resolution is a dynamic process in Robot Framework, the error might be revealed after a long execution time, reducing the velocity of the team to detect and address such bugs. During the analysis, while resolving names, the tool checks for duplicate names.

Missing Definition

Identifies *steps* for which no keyword is associated or a missing variable, therefore generating an edge in the graph with no target node. Upon analysis of the results, we observe that we have false positives for keywords generated dynamically and therefore invisible during static analysis. This violation constitutes the only one generating false positive, however, relying too heavily on dynamic loading might hinder the readability of the test suite and therefore, the results are kept and displayed.

Transitive Dependency Warning

We call transitive dependency warning a call to a keyword not defined in the same file or in a direct dependency but in an indirect dependency as explained in Section 6.2.1.

Duplicated Variable Warning

During the resolution of variable names, if two variables have the same name and are defined in *Variable* blocks, the framework loads the first variable it encounters and discards all subsequent definitions. This behavior might lead to unexpected behavior, assigning wrong values to variables.

SUIT Smells

Relying on the tooling introduced in Chapter 4, Ikora Inspector is able to detect the number of symptoms present in a test. Relying on rules inserted by the user, whenever this number is exceeded, the test case is flagged as smelly. Because the nature of Ikora Inspector is to offer an overview of the project, the detection of symptoms for SUIT smells is also available as a SonarQube Plugin[2]

Output All these errors and warnings are grouped in a single table, giving the level of the violation (warning or error) as well as the location and its reason. Each row of the table contains information about the location of the violation, the cause, and its severity.

6.3 Case Study

In this section, we present the results generated by Ikora Inspector on the KDT code base at BGL BNP Paribas. The tool is integrated into their production pipeline where the tool runs nightly (via Jenkins), generating a report with its results. First, the tool checks out all the test projects defined in its configuration file, analyzes them, and generates a report for the automation testers. The report is then deployed on a server accessible by the members of the team. The KDT test code is composed of 43 projects accounting for 44,521 lines of test code, 452 test cases, and 4,448 keywords.

Figure 6.3 presents the percentage of clones across all projects. With values of 11.4% for Type I clone and 27.6% for Type II clones, we see that there is a high potential for refactoring. This result corroborates previous studies, e.g. [LMM$^+$17], once again highlighting the need for better tooling for automation testers. However, we see that Type III and Type IV clones remain low with values of 0.63% and 0.0% respectively. Interviewing the team and analyzing the results, we see that a large portion of the duplicated code is due to the lack of knowledge of the existence of the functionality in the code base, especially in the case of cross-project code duplication.

Figure 6.4 shows the number of test code lines never executed (dead code) for each of the 43 projects. The values go as high as 27.2% of dead code for a project of more than 7000 lines of test code. The main reason for this, as explained by the testers after analyzing the results, is due to refactoring activities. Indeed, in the absence of static analysis tools, during refactoring, deprecated keywords and variables are often kept in fear of breaking production test code.

Ikora Inspector has been successfully used by both project managers and automation testers. The former appreciate the clear KPIs provided by the report and the latter the information about the locations and causes of the issues reported

[2]Available at `https://github.com/kabinja/sonar-ikora-plugin`

allowing them to take action if needed. Ikora Inspector provides a central point of reference and communication between different roles at BGL BNP Paribas which is one of the main goals of every continuous inspection tool.

To conclude our evaluation, we asked the users of Ikora Inspector which were the main weaknesses or missing features of the tool. According to their answers, the main weaknesses reside in the static nature of the results. Indeed, while the tool provides a snapshot of the state of the code base, the lack of historical data makes it hard to control the evolution of the KPIs.

6.4 Conclusion

This chapter presents Ikora Inspector, a static analysis tool for continuous inspection of Keyword-Driven test suites written in the Robot Framework syntax. It analyzes the test code and provides information about its health and potential test smells introduced. The tool has been successfully deployed at BGL BNP Paribas and has been used in production for a month, providing valuable feedback to project managers and automation experts.

7 Conclusion

This chapter presents the overall conclusion of the dissertation and proposes potential research directions.

Contents

7.1 Summary of contributions

In this dissertation, we presented studies, techniques, and tools that contribute to reducing the maintenance cost that is associated with the evolution of SUITs. Indeed, with the increased adoption of Agile and DevOps methodologies, quick and reliable test feedback on every code change becomes essential, leading SUITs to evolve continuously alongside the system they exercise. However, the user interface is a part of the system that tends to undergo rapid evolutions, causing SUITs to be particularly prompt to break. In this context, this dissertation brings the following contribution: (1) an empirical study that evaluates how SUITs evolve in a large industrial system; (2) an empirical study that measures the prevalence of bad design practices potentially contributing to test fragility and their higher maintenance cost; and (3) a novel approach to render SUITs more resistant to SUT evolution by creating more robust locators.

The goal of the first contribution is to shed light on how and why SUITs undergo heavy maintenance. Our analysis reveals that test fragility (the sensitivity to SUT evolution) and test clones (keywords with similar test functionality) are the most important problems of KDT. On the positive side, our study provides evidence that following the good design practices of KDT (such as the separation of concern and the reusability of keywords) has the potential to reduce the required number of maintenance changes. Our analysis shows that this reduction is approximately 70% demonstrating major benefits of KDT. Our results also help to improve the understanding of the fine-grained changes performed during the evolution of KDT. We provide a taxonomy of test code changes and reveal the presence of test clones caused by the difficulty of selecting appropriate keywords. We believe that the main drawback of KDT lies in the absence of appropriate tooling, allowing to deal with test code growth, navigation, and comprehension. To address this concern, we introduce Ikora Inspector an automated approach to provide testers with visibility on their test codebase. Finally, we show that practitioners agree with the promises of KDT, on the advantages of the separation of concerns and the reusability of keywords.

With the second contribution, our objectives were to better understand which sub-optimal decision within the test code could have an effect on the maintenance by measuring the diffusion of bad practices and their refactoring from the test codebase. To achieve these goals, 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 build a catalog of 35 SUIT smells. For 12 out of 35 of the smells from this catalog, we propose an automated approach for detecting the introduction and refactoring of these SUIT smells. Leveraging on this approach, we evaluate the prevalence of SUIT smells and their refactoring in our industrial and open-source projects. Even though our empirical results suggest that most tests present the symptoms of bad practices, less than half of them ever experience refactoring during their lifespan. However, while refactoring actions are rare, SUIT smells like *Narcissistic* and *Middle Man* still disappear from the code base as a side effect of unintended maintenance and removal of symptomatic tests and the apparition of clean ones.

Finally, the third contribution tackles the breakages caused by second-generation locators. Because they rely heavily on internal properties of the elements they visit, second-generation locators can be problematic in the case of automated GUI testing as it leaks structural details of the SUT that should not be present in such tests. We propose HPath, a second-generation locator inspired by third-generation locators which avoid exposing internal details to the test by relying only on the rendered properties of the SUT. The practical benefits of HPath can be measured via its capability to generate more flexible locators than the current second-generation techniques. The results from our experiments suggest that HPath is able to reduce test breakages compared to classical approaches relying on internal properties. However, while the results are promising, for HPath to be useful when developing the user interface, developers need to rely on modern implementation of the HTML standard and follow good practices.

7.2 Perspectives

In the following, we discuss potential future research that follows the contributions and ideas presented in this dissertation:

- **Improvements on locators representation:** In Chapter 5, we propose an approach to exploit properties that make third-generation locators more robust to change and integrate them into second-generation locators. Unfortunately, many current frameworks do not allow for the extraction of the representation layer in a way that allows relying on second-generation locators. While relying only on VGT offers promising results, the model extracted by the current computer vision techniques remains unable to adapt to trivial changes in the rendering of the SUT. By generalizing the concept of rendering tree and proposing a common representation applicable to all GUI-based applications, the shortcoming of third-generation locators could be alleviated. We envision two possible paths leading to the generation of a general tree representation: (1) computer vision and similar tooling can be employed to extract the hierarchical structure of the GUI, therefore, leading to a partial rendering tree; (2) by exposing the rendering engines present in web browsers and other applications, testing tools can extract the intermediate representation of the GUI and interact with it. Again, each solution comes with advantages and limitations necessitating future research.
- **Test execution analysis:** In Chapter 4, we present an exploratory analysis

of the diffusion and the refactoring of bad practices in SUITs. In an attempt to better understand their impact on SUIT maintenance and derive solutions allowing to avoid it, we conduct a deeper analysis of the effects associated with two of SUIT smells: code duplication (Section 3.4.3) and fragile locator representation (Chapter 5). However, testers at BGL BNP Paribas refer to a third major cause leading to high maintenance, synchronization with the SUT. Indeed, when not carefully crafted (*e.g. Stinky Synchronization Syndrom* in Section 4.4.2), tests may fail in a non-deterministic manner because of the variability in factors outside the scope of the test. This happens because SUITs rely on the system as a whole and operate asynchronously from the SUT. Therefore, they are sensitive to variations and non-determinism occurring either in the SUT or the infrastructure it relies on. This leads us to define three root causes for SUIT failures: (1) test breakage resulting from a mismatch between the test and the SUT; (2) flaky failures resulting from variations in the SUT or in the infrastructure leading to false alarms; and (3) failures resulting from a fault present in the SUT which is the only type of test failure providing the signal intended by the test. Moreover, even when the failure is revealing a fault in the SUT, because of the size of the test, isolating the sub-system(s) responsible for the failure becomes challenging. Indeed, as shown in Chapter 3 and Chapter 2, a single test can exercise tens or even hundreds of interactions with the SUT. Consequently, we advocate for more research on the automatic analysis of test failures as the signal provided by the failure itself remains limited. Instead of mimicking the behavior of unit tests, SUIT report could be associated with more contextual information about the failure, its root causes, and potentially offer a triage mechanism to redirect the type of failure into categories or trace it back to the sub-system at fault.

Abbreviations

API Application Programming Interface.
AST Abstract Syntax Tree.

BNF Backus–Naur Form.

CSS Cascading Style Sheet.

DAG Directed Acyclic Graph.
DOM Document Object Model.

GDPR General Data Protection Regulation.
GUI Graphical User Interface.

HTML Hypertext Markup Language.

KDT Keyword-Driven Testing.
KPI Key Performance Indicator.

MBT Model-Based Testing.

QA Quality Assurance.

RDBMS Relational Database Management System.

SOA Service-Oriented Architecture.
SUIT System User Interactive Test.
SUT System Under Test.

URL Uniform Resource Locator.

VGT Visual GUI Testing.

W3C World Wide Web Consortium.
WHATWG Web Hypertext Application Technology Working Group.

XML eXtensible Markup Language.

List of publications and tools

Papers included in the dissertation

- Renaud Rwemalika et al. On the Evolution of Keyword-Driven Test Suites. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*, pages 335–345, Xi'an, China. IEEE, April 2019
- Renaud Rwemalika et al. Ukwikora: continuous inspection for keyword-driven testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 402–405, Beijing, China. Association for Computing Machinery, 2019

Papers not included in the dissertation

- Renaud Rwemalika et al. Can we automate away the main challenges of end-to-end testing? In *CEUR Workshop Proceedings*, volume 2361, 2018
- Matthieu Jimenez et al. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 695–705, New York, NY, USA. ACM, August 2019
- Renaud Rwemalika et al. An Industrial Study on the Differences between Pre-Release and Post-Release Bugs. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 92–102, Cleveland, USA. IEEE, September 2019
- Salah Ghamizi et al. Data-driven Simulation and Optimization for Covid-19 Exit Strategies. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3434–3442, New York, NY, USA. ACM, August 2020

Tools included in the dissertation

- Ikora Core: Robot Framework engine
  - `https://github.com/UL-SnT-Serval/ikora-core`
- Ikora Inspector: Continuous monitoring for Robot Framwork
  - `https://github.com/UL-SnT-Serval/ikora-inspector`
- Ikora SonarQube Plugin: Smell analysis for Robot Framwork

  - `https://github.com/kabinja/sonar-ikora-plugin`
- Mercator: Locator change miner
  - `https://github.com/UL-SnT-Serval/mercator`

List of figures

List of tables

Bibliography

[AAA19] Mamdouh Alenezi, Mohammed Akour, and Hiba Al Sghaier. The impact of co-evolution of code production and test suites through software releases in open source software systems. *International Journal of Innovative Technology and Exploring Engineering*, 9(1):2737–2739, 2019. ISSN: 22783075. DOI: `10.35940/ijitee.A4967.119119` (cited on page 14).

[AAF⁺15] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana. AGRippin: a novel search based testing technique for Android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 5–12, New York, NY, USA. ACM, August 2015. ISBN: 9781450338158. DOI: `10.1145/2804345.2804348` (cited on page 6).

[AAM⁺17] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software*, 125:322–343, March 2017. ISSN: 01641212. DOI: `10.1016/j.jss.2016.12.017` (cited on page 4).

[ABC⁺13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013. ISSN: 01641212. DOI: `10.1016/j.jss.2013.02.061` (cited on page 4).

[AD17] Inigo Aldalur and Oscar Diaz. Addressing web locator fragility. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 45–50, New York, NY, USA. ACM, June 2017 (cited on pages 17, 81, 96, 97).

[Adv18] Nikolay Advolodkin. Top 17 automated testing best practices (supported by data). 2018. URL: https://ultimateqa.com/automation-patterns-antipatterns/ (visited on 07/02/2020) (cited on pages 3, 45, 53).

[AFK16] Emil Alégroth, Robert Feldt, and Pirjo Kolström. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology*, 73:66–80, 2016. ISSN: 09505849. DOI: 10.1016/j.infsof.2016.01.012. arXiv: 1602.01226 (cited on pages 2, 15).

[AFO13] Emil Alegroth, Robert Feldt, and Helena H Olsson. Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pages 56–65. IEEE, March 2013. ISBN: 978-0-7695-4968-2. DOI: 10.1109/ICST.2013.14 (cited on page 15).

[AFR15] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. Visual GUI testing in practice: challenges, problemsand limitations. *Empirical Software Engineering*, 20(3):694–744, June 2015 (cited on page 80).

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th International Conference on Automated Software Engineering*, page 258, New York, New York, USA. ACM Press, 2012. ISBN: 9781450312042. DOI: 10.1145/2351676.2351717 (cited on page 6).

[AFT+15] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software*, 32(5):53–59, September 2015. ISSN: 0740-7459. DOI: 10.1109/MS.2014.55 (cited on page 17).

[AFT11] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261. IEEE, March 2011. ISBN: 978-1-4577-0019-4. DOI: 10.1109/ICSTW.2011.77 (cited on page 4).

[AKR18] Emil Alegroth, Arvid Karlsson, and Alexander Radway. Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 172–181. IEEE, April 2018 (cited on pages 17, 81).

[AMZ+16] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, June 2016. ISSN: 1382-3256. DOI: `10.1007/s10664-015-9378-4` (cited on page 49).

[ARA+19] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning. *Information and Software Technology*, 105(August 2018):95–116, January 2019. ISSN: 09505849. DOI: `10.1016/j.infsof.2018.08.007` (cited on page 7).

[Arc10] Matt Archer. How test automation with selenium can fail. 2010. URL: `https://mattarcherblog.wordpress.com/2010/11/29/how-test-automation-with-selenium-or-watir-can-fail/` (visited on 07/02/2020) (cited on page 53).

[ASM16] Emil Alegroth, Marcello Steiner, and Antonio Martini. Exploring the Presence of Technical Debt in Industrial GUI-Based Testware: A Case Study. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 257–262. IEEE, April 2016. ISBN: 978-1-5090-3674-5. DOI: `10.1109/ICSTW.2016.47` (cited on pages 51, 53).

[Bak95] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95, July 1995. DOI: `10.1109/WCRE.1995.514697` (cited on page 26).

[Bat20] Michael Battat. How do you simplify end-to-end test maintenance? 2020. URL: `https://dzone.com/articles/how-do-you-simplify-end-to-end-test-maintenance-au` (visited on 05/05/2021) (cited on page 53).

[BCD+03] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. Streaming XPath processing with forward and backward axes. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 455–466. IEEE, 2003 (cited on page 84).

[BHM$^+$15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015. ISSN: 0098-5589. DOI: `10.1109/TSE.2014.2372785` (cited on page 4).

[BHP$^+$17] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. How Good Are My Tests? In *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 9–14. IEEE, May 2017. ISBN: 978-1-5386-2807-2. DOI: `10.1109/WETSoM.2017.2` (cited on pages 14, 42).

[BNG$^+$13] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694, October 2013. ISSN: 09505849. DOI: `10.1016/j.infsof.2013.03.004` (cited on page 3).

[Bos14] Jan Bosch. *Continuous Software Engineering*. Springer International Publishing, 2014, pages 1–226 (cited on pages 17, 80).

[BQO$^+$15] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4):1052–1094, August 2015. ISSN: 1382-3256. DOI: `10.1007/s10664-014-9313-0` (cited on pages 14, 42).

[BRM09] Penelope Brooks, Brian Robinson, and Atif M. Memon. An initial characterization of industrial graphical user interface systems. *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*:11–20, 2009. DOI: `10.1109/ICST.2009.11` (cited on page 3).

[BSR$^+$19] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 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*, number 1, pages 142–153, Tallinn, Estonia. ACM Press, 2019. ISBN: 9781450355728. DOI: `10.1145/3338906.3338970` (cited on pages 3, 5, 6).

[Bus19] Yuri Bushnev. Top 15 ui test automation best practices. 2019. URL: `https://www.blazemeter.com/blog/top-15-ui-test-automation-best-practices-you-should-follow` (visited on 07/03/2020) (cited on page 53).

[Buw15] Hans Buwalda. Test design for automation: anti-patterns. 2015. URL: https://www.techwell.com/techwell-insights/2015/09/test-design-automation-anti-patterns (visited on 07/02/2020) (cited on pages 45, 52, 53).

[Buw19] Hans Buwalda. 8 test automation anti-patterns (and how to avoid them). 2019. URL: https://dzone.com/articles/8-test-automation-anti-patterns-and-how-to-avoid-t (visited on 07/03/2020) (cited on pages 52, 53).

[BWK05] Stefan Berner, Roland Weber, and R.K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th International Conference on Software Engineering*, volume 2005, pages 571–579. IEEE, 2005. DOI: 10.1109/ICSE.2005.1553603 (cited on page 15).

[CAM+20] Riccardo Coppola, Luca Ardito, Maurizio Morisio, and Marco Torchiano. Mobile Testing: New Challenges and Perceived Difficulties From Developers of the Italian Industry. *IT Professional*, 22(5):32–39, September 2020. ISSN: 1520-9202. DOI: 10.1109/MITP.2019.2942810 (cited on page 2).

[CBM+19] Alexandre Canny, Elodie Bouzekri, Célia Martinie, and Philippe Palanque. Rationalizing the Need of Architecture-Driven Testing of Interactive Systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Volume 11262 LNCS, pages 164–186. 2019. ISBN: 9783030059088. DOI: 10.1007/978-3-030-05909-5_10 (cited on page 4).

[CDB15] Joseph Paul Cohen, Wei Ding, and Abraham Bagherjeiran. XTreePath: A generalization of XPath to handle real world structural variation. *arXiv: Information Retrieval*, May 2015 (cited on page 95).

[CJL+20] Xin Chen, He Jiang, Xiaochen Li, Liming Nie, Dongjin Yu, Tieke He, and Zhenyu Chen. A systemic framework for crowdsourced test report quality assessment. *Empirical Software Engineering*, 25(2):1382–1418, March 2020. ISSN: 1382-3256. DOI: 10.1007/s10664-019-09793-8 (cited on page 16).

[Cla14] Josh Clayton. Acceptance tests at a single level of abstraction. 2014. URL: https://thoughtbot.com/blog/acceptance-tests-at-a-single-level-of-abstraction (visited on 07/02/2020) (cited on page 53).

[CMT+19] Riccardo Coppola, Maurizio Morisio, Marco Torchiano, and Luca Ardito. Scripted GUI testing of Android open-source apps: evolution of test code and fragility causes. *Empirical Software Engineering*, 24(5):3205–3248, October 2019. ISSN: 1382-3256. DOI: `10.1007/s10664-019-09722-9` (cited on page 15).

[CMT17] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Scripted GUI Testing of Android Apps. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 22–32. ACM, November 2017. ISBN: 9781450353052. DOI: `10.1145/3127005.3127008` (cited on page 15).

[CMT19] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Mobile GUI Testing Fragility: A Study on Open-Source Android Applications. *IEEE Transactions on Reliability*, 68(1):67–90, March 2019. ISSN: 0018-9529. DOI: `10.1109/TR.2018.2869227` (cited on page 10).

[Con04] World Wide Web Consortium. What is the document object model? April 2004. URL: `https://www.w3.org/TR/DOM-Level-3-Core/introduction.html`. (accessed: 15.01.2021) (cited on page 81).

[Con14] World Wide Web Consortium. Using aria landmarks to identify regions of a page. January 2014. URL: `https://www.w3.org/WAI/GL/wiki/Using_ARIA_landmarks_to_identify_regions_of_a_page` (visited on 01/16/2021) (cited on page 83).

[Con18] World Wide Web Consortium. Selectors level 3 - w3c candidate recommendation. January 2018. URL: `https://drafts.csswg.org/selectors-3/` (visited on 01/21/2021) (cited on page 85).

[con20] MDN contributors. Html elements reference. December 2020. URL: `https://developer.mozilla.org/en-US/docs/Web/HTML/Element` (visited on 01/16/2021) (cited on page 82).

[Con99] World Wide Web Consortium. Xml path language (xpath). November 1999. URL: `https://www.w3.org/TR/1999/REC-xpath-19991116/` (visited on 01/16/2021) (cited on page 84).

[CPF+10] Marco Cunha, Ana C. R. Paiva, Hugo Sereno Ferreira, and Rui Abreu. PETTool: A pattern-based GUI testing tool. In *Proceedings of the 2nd International Conference on Software Technology and Engineering*, volume 1, pages 202–206. IEEE, October 2010. ISBN: 978-1-4244-8667-0. DOI: `10.1109/ICSTE.2010.5608882` (cited on page 3).

[CPN20] Alexandre Canny, Philippe Palanque, and David Navarre. Model-Based Testing of GUI Applications Featuring Dynamic Instanciation of Widgets. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 95–104. IEEE, October 2020. ISBN: 978-1-7281-1075-2. DOI: `10.1109/ICSTW50294.2020.00029` (cited on pages 3, 6).

[CRG+96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montreal, Quebec, Canada. ACM, 1996. ISBN: 0-89791-794-4. DOI: `10.1145/233269.233366` (cited on page 27).

[Cri18] Lisa Cripsin. Keep your automated testing simple and avoid anti-patterns. 2018. URL: `https://www.mabl.com/blog/keep-your-automated-testing-simple` (visited on 07/02/2020) (cited on page 53).

[CRT16] Riccardo Coppola, Emanuele Raffero, and Marco Torchiano. Automated mobile UI test fragility: an exploratory assessment study on Android. In *Proceedings of the 2nd International Workshop on User Interface Test Automation*, pages 11–20. ACM, July 2016. ISBN: 9781450344128. DOI: `10.1145/2945404.2945406` (cited on pages 15, 16).

[CSD+14] Laurent Christophe, Reinout Stevens, Coen De Roover, and Wolfgang De Meuter. Prevalence and Maintenance of Automated Functional Tests for Web Applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 141–150. IEEE, September 2014 (cited on pages 15, 96).

[CW12] Woei Kae Chen and Jung Chi Wang. Bad smells and refactoring methods for GUI test scripts. *Proceedings of the 13th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*:289–294, 2012. DOI: `10.1109/SNPD.2012.10` (cited on pages 15, 42, 45, 51, 53).

[CZV+11] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. WATER. In *Proceedings of the 1st International Workshop on End-to-End Test Script Engineering*, pages 24–29, New York, New York, USA. ACM Press, 2011 (cited on pages 16, 18, 81).

[DFP$^+$02] G.A. Di Lucca, A.R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. WARE: a tool for the reverse engineering of Web applications. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 241–250. IEEE, 2002. ISBN: 0-7695-1438-3. DOI: `10.1109/CSMR.2002.995811` (cited on page 6).

[DFS$^+$21] Sergio Di Martino, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing. *Software Testing, Verification and Reliability*, 31(3), May 2021. ISSN: 0960-0833. DOI: `10.1002/stvr.1754` (cited on pages 6, 10).

[Dha17] Kumar Dharmender. Automation testing: anti-patterns. 2017. URL: `https://alisterbscott.com/2015/01/20/five-automated-acceptance-test-anti-patterns/` (visited on 07/02/2020) (cited on page 53).

[DLM$^+$11] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. Automated GUI refactoring and test script repair (position paper). In *Proceedings of the International Workshop on End-to-End Test Script Engineering*, pages 38–41, New York, New York, USA. ACM Press, 2011. ISBN: 9781450308083. DOI: `10.1145/2002931.2002937` (cited on page 17).

[EMS18] Hadeel Mohamed Eladawy, Amr E. Mohamed, and Sameh A. Salem. A New Algorithm for Repairing Web-Locators using Optimization Techniques. In *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, pages 327–331. IEEE, December 2018 (cited on pages 18, 96).

[Eng16] Theo England. Cucumber anti-patterns (part one). 2016. URL: `https://cucumber.io/blog/bdd/cucumber-antipatterns-part-one/` (visited on 07/03/2020) (cited on pages 45, 53).

[Eva12] Augusto Evangelisti. How to transform bad acceptance tests into awesome ones. 2012. URL: `https://mysoftwarequality.wordpress.com/2012/12/14/how-to-transform-bad-acceptance-tests-into-awesome-ones/` (visited on 07/02/2020) (cited on page 53).

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN: 0201485672 (cited on pages 42, 50).

[FGX09] Chen Fu, Mark Grechanik, and Qing Xie. Inferring types of references to GUI objects in test scripts. *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*:1–10, 2009. DOI: `10.1109/ICST.2009.12` (cited on page 18).

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 313–324, New York, New York, USA. ACM Press, 2014. ISBN: 9781450330138. DOI: `10.1145/2642937.2642982` (cited on page 27).

[FWP+07] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007. ISSN: 0098-5589. DOI: `10.1109/TSE.2007.70731` (cited on pages 14, 27).

[Gaw16] Maciej Gawinecki. Anti-patterns in test automation. 2016. URL: `http://nomoretesting.com/blog/2016/05/23/anti-patterns/` (visited on 07/02/2020) (cited on pages 45, 53).

[GCZ+16] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering*, 42(2):170–186, February 2016. ISSN: 0098-5589. DOI: `10.1109/TSE.2015.2454510` (cited on pages 17, 18).

[GF16] Vahid Garousi and Michael Felderer. Developing, Verifying, and Maintaining High-Quality Automated Test Scripts. *IEEE Software*, 33(3):68–75, May 2016. ISSN: 0740-7459. DOI: `10.1109/MS.2016.30` (cited on page 10).

[GFZ12] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis*, page 67, New York, New York, USA. ACM Press, 2012. ISBN: 9781450314541. DOI: `10.1145/2338965.2336762` (cited on pages 4, 5).

[GK18] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, April 2018. ISSN: 01641212. DOI: `10.1016/j.jss.2017.12.013` (cited on pages 16, 43, 44).

[GKP+05] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of XPath query evaluation and XML typing. *Journal of the ACM*, 52:284–335, 2005 (cited on page 84).

[GLC+15] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making System User Interactive Tests Repeatable: When and What Should We Control? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 55–65. IEEE, May 2015. ISBN: 978-1-4799-1934-5. DOI: `10.1109/ICSE.2015.28` (cited on page 10).

[GLS02] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, May 2002 (cited on page 84).

[GNA+13] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35th International Conference on Software Engineering*, pages 72–81. IEEE, May 2013. ISBN: 978-1-4673-3076-3. DOI: `10.1109/ICSE.2013.6606553` (cited on page 3).

[Gol19] Yoni Goldberg. Node.js and javascript testing best practices (2020). 2019. URL: `https://yonigoldberg.medium.com/yoni-goldberg-javascript-nodejs-testing-best-practices-2b98924c9347` (visited on 07/02/2020) (cited on page 53).

[Goo20] Google. Ui/application exerciser monkey. August 2020. URL: `https://developer.android.com/studio/test/monkey` (visited on 05/15/2021) (cited on page 4).

[GRC+20] Salah Ghamizi, Renaud Rwemalika, Maxime Cordy, Lisa Veiber, Tegawendé F. Bissyandé, Mike Papadakis, Jacques Klein, and Yves Le Traon. Data-driven Simulation and Optimization for Covid-19 Exit Strategies. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3434–3442, New York, NY, USA. ACM, August 2020 (cited on page iii).

[Gri19] Ilya Grigorik. Render-tree construction, layout, and paint. February 2019. URL: `https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction` (visited on 01/28/2021) (cited on page 87).

[Gro21] Web Hypertext Application Technology Working Group. Html living standard. January 2021. URL: `https://html.spec.whatwg.org/` (visited on 01/16/2021) (cited on page 82).

[GS11] Priya Gupta and Prafullakumar Surve. Model based approach to assist test case creation, execution, and maintenance for test automation. In *Proceedings of the 1st International Workshop on End-to-End Test Script Engineering*, pages 1–7, New York, New York, USA. ACM Press, 2011. ISBN: 9781450308083. DOI: `10.1145/2002931.2002932` (cited on page 5).

[GXF09a] Mark Grechanik, Qing Xie, and Chen Fu. Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts. In *Proceedings of the International Conference on Software Maintenance*, pages 9–18. IEEE, September 2009. ISBN: 978-1-4244-4897-5. DOI: `10.1109/ICSM.2009.5306345` (cited on pages 15, 17, 18, 96).

[GXF09b] Mark Grechanik, Qing Xie, and Chen Fu. Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts. In *Proceedings of the International Conference on Software Maintenance*, pages 9–18. IEEE, September 2009. ISBN: 978-1-4244-4897-5. DOI: `10.1109/ICSM.2009.5306345` (cited on page 17).

[HCM10] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing GUI Test Suites Using a Genetic Algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 245–254. IEEE, 2010. ISBN: 978-1-4244-6435-7. DOI: `10.1109/ICST.2010.39` (cited on page 18).

[HEJ$^+$15] Benedikt Hauptmann, Sebastian Eder, Maximilian Junker, Elmar Juergens, and Volkmar Woinke. Generating Refactoring Proposals to Remove Clones from Automated System Tests. In *Proceedings of IEEE 23rd International Conference on Program Comprehension*, volume 2015-August, pages 115–124. IEEE, May 2015. ISBN: 978-1-4673-8159-8. DOI: `10.1109/ICPC.2015.20` (cited on page 53).

[HF10] Jez Humble and David G. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley, Upper Saddle River, NJ, 2010. ISBN: 978-0-321-60191-9 (cited on pages 7, 49).

[HHV$^+$13] Benedikt Hauptmann, Lars Heinemann, Rudolf Vaas, and Peter Braun. Hunting for smells in natural language tests. In *Proceedings of the 35th International Conference on Software Engineering,* pages 1217–1220, San Francisco, CA, USA. IEEE, May 2013. ISBN: 978-1-4673-3076-3. DOI: `10.1109/ICSE.2013.6606682` (cited on pages 42, 45, 51, 53).

[HLZ+13] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. Is this a bug or an obsolete test? In *Proceedings of European Conference on Object-Oriented Programming*, pages 602–628, Berlin, Heidelberg. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-39038-8 (cited on page 15).

[HRS16] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. WATERFALL: an incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume 13-18-Nove, pages 751–762, New York, NY, USA. ACM, November 2016. ISBN: 9781450342186. DOI: `10.1145/2950290.2950294` (cited on pages 7, 18).

[HRT16] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do Record/Replay Tests of Web Applications Break? In *2016 IEEE International Conference on Software Testing, Verification and Validation*, pages 180–190. IEEE, April 2016 (cited on pages 7, 16, 17, 80, 86).

[HWZ12] Reinhard Hametner, Dietmar Winkler, and Alois Zoitl. Agile testing concepts based on keyword-driven testing for industrial automation systems. In *Proceedings of the 38th Annual Conference on Industrial Electronics Society*, pages 3727–3732. IEEE, October 2012. ISBN: 978-1-4673-2421-2. DOI: `10.1109/IECON.2012.6389298` (cited on page 8).

[HZ12] Victor Hurdugaci and Andy Zaidman. Aiding Software Developers to Maintain Developer Tests. In *Proceedings of 16th European Conference on Software Maintenance and Reengineering*, pages 11–20. IEEE, March 2012. ISBN: 978-0-7695-4666-7. DOI: `10.1109/CSMR.2012.12` (cited on page 14).

[ISG12] Ayman Issa, Jonathan Sillito, and Vahid Garousi. Visual testing of Graphical User Interfaces: An exploratory study towards systematic definitions and approaches. In *Proceedings of the International Symposium on Web Systems Evolution*, pages 11–15. IEEE, 2012. ISBN: 9781467330558. DOI: `10.1109/WSE.2012.6320526` (cited on page 3).

[Jai07] Naresh Jain. Patterns and anti-patterns: acceptance testing with fitnesse. 2007. URL: `https://blogs.agilefaqs.com/2007/08/25/patterns-and-anti-patterns-acceptance-testing-with-fitnesse/` (visited on 07/02/2020) (cited on page 53).

[JRP+19] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 695–705, New York, NY, USA. ACM, August 2019 (cited on page iii).

[JXM08] Jingfan Tang, Xiaohua Cao, and Albert Ma. Towards adaptive framework of keyword driven automation testing. In *Proceedings of the International Conference on Automation and Logistics*, pages 1631–1636. IEEE, September 2008. ISBN: 978-1-4244-2502-0. DOI: `10.1109/ICAL.2008.4636415` (cited on page 8).

[Kap18] Kostis Kapelonis. Software testing anti-patterns. 2018. URL: `http://blog.codepipes.com/testing/software-testing-antipatterns.html` (visited on 07/02/2020) (cited on page 53).

[KC07] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical report EBSE 2007-001, Keele University and Durham University Joint Report, 2007 (cited on page 44).

[Kim20] Dong Jae Kim. An empirical study on the evolution of test smell. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, number i, pages 149–151, New York, NY, USA. ACM, June 2020. ISBN: 9781450371223. DOI: `10.1145/3377812.3382176` (cited on pages 14, 42).

[Kir14] Jim Kirkbride. Testing anti-patterns. 2014. URL: `https://medium.com/jameskbride/testing-anti-patterns-b5ffc1612b8b` (visited on 07/02/2020) (cited on page 53).

[Kla14] Pekka Klarck. Robot framework dos and don'ts. 2014. URL: `https://slideshare.net/pekkaklarck/robot-framework-dos-and-donts` (visited on 07/02/2020) (cited on pages 45, 53).

[Kni17a] Andrew Knight. Bdd 101: writing good gherkin. 2017. URL: `https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/` (visited on 07/02/2020) (cited on page 53).

[Kni17b] Andrew Knight. Should gherkin steps use first-person or third-person? 2017. URL: https://automationpanda.com/2017/01/18/should-gherkin-steps-use-first-person-or-third-person/ (visited on 07/02/2020) (cited on page 53).

[Kni19] Andrew Knight. Bdd 101: writing good gherkin. 2019. URL: https://techbeacon.com/app-dev-testing/7-ways-tidy-your-test-code (visited on 05/04/2021) (cited on page 53).

[KP88] Glenn E Krasner and Stephen T Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal Of Object Oriented Programming*, 1(3):26–49, 1988. DOI: 10.1.1.47.3665 (cited on page 3).

[KTN+15] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the Test Automation Culture of App Developers. In *Proceeding of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, April 2015. ISBN: 978-1-4799-7125-1. DOI: 10.1109/ICST.2015.7102609 (cited on page 15).

[KTN19] Hiroyuki Kirinuki, Haruto Tanno, and Katsuyuki Natsukawa. COLOR: Correct Locator Recommender for Broken Test Scripts using Various Clues in Web Application. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 36 of number 4, pages 310–320. IEEE, February 2019 (cited on pages 17, 18, 81, 96).

[KVG+09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the 9th International Conference on Quality Software*, pages 305–314, USA. IEEE Computer Society, 2009. ISBN: 9780769538280. DOI: 10.1109/QSIC.2009.47 (cited on page 49).

[KWW+13] Hong Xing Kan, Guo Qiang Wang, Zong Dian Wang, and Shuai Ding. A method of minimum reusability estimation for automated software testing. *Journal of Shanghai Jiaotong University (Science)*, 18(3):360–365, 2013. ISSN: 10071172. DOI: 10.1007/s12204-013-1406-1 (cited on page 15).

[LBB+15] Valeria Lelli, Arnaud Blouin, Benoit Baudry, and Fabien Coulon. On model-based testing advanced GUIs. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10. IEEE, April 2015.

ISBN: 978-1-4799-1885-0. DOI: `10.1109/ICSTW.2015.7107403` (cited on page 6).

[LCR+13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281. IEEE, October 2013 (cited on page 16).

[LCR+14] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Visual vs. DOM-Based Web Locators: An Empirical Study. In *International Conference on Web Engineering*. Volume 8541, pages 322–340. 2014. DOI: `10.1007/978-3-319-08245-5_19` (cited on page 17).

[LD10] Ahmed Lamkanfi and Serge Demeyer. Studying the Co-evolution of Application Code and Test Cases. *Proceedings of the 9th Belgian-Netherlands Software EvoLution Seminar)*, 2010:1–4, 2010 (cited on page 14).

[LDO19] Xiangyu Li, Marcelo D'Amorim, and Alessandro Orso. Intent-Preserving Test Repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 217–227. IEEE, April 2019. ISBN: 978-1-7281-1736-2. DOI: `10.1109/ICST.2019.00030` (cited on page 18).

[LHE+14] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, New York, NY, USA. ACM, November 2014. ISBN: 9781450330565. DOI: `10.1145/2635868.2635920` (cited on page 15).

[LIH17] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*:821–830, 2017. DOI: `10.1145/3106237.3106288` (cited on page 14).

[LMM+17] Thierry Lavoie, Mathieu Mérineau, Ettore Merlo, and Pascal Potvin. A case study of TTCN-3 test scripts clone analysis in an industrial telecommunication setting. *Information and Software Technology*, 87:32–45, July 2017. ISSN: 09505849. DOI: `10.1016/j.infsof.2017.01.008` (cited on pages 15, 26, 29, 114).

[LSR+] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Pesto: automated migration of dom-based web tests towards the visual approach. *Software Testing, Verification and Reliability*, (4):30. DOI: `10.1002/stvr.1665` (cited on page 80).

[LSR+14] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Reducing Web Test Cases Aging by Means of Robust XPath Locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 449–454. IEEE, November 2014 (cited on pages 16, 17, 81).

[LSR+15a] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Meta-heuristic Generation of Robust XPath Locators for Web Testing. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*, pages 36–39. IEEE, May 2015. ISBN: 978-1-4673-7079-0. DOI: `10.1109/SBST.2015.16` (cited on page 17).

[LSR+15b] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *Proceedings of IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, April 2015 (cited on pages 16, 17, 81, 95, 97, 103).

[LSR+16] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Robula+: an algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, March 2016 (cited on pages 17, 81, 96, 97).

[LWC+20] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. WebRR: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1498–1508, New York, NY, USA. ACM, November 2020. ISBN: 9781450370431. DOI: `10.1145/3368089.3417069` (cited on page 17).

[LY17] S. Levin and A. Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–46, September 2017. DOI: `10.1109/ICSME.2017.9` (cited on page 14).

[Mar04] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 350–359. IEEE, 2004.

ISBN: 0-7695-2213-0. DOI: 10.1109/ICSM.2004.1357820 (cited on pages 47, 49).

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, volume 2003-Janua, pages 260–269. IEEE, 2003. ISBN: 0-7695-2027-8. DOI: 10.1109/WCRE.2003.1287256 (cited on page 6).

[Mem07] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, September 2007. ISSN: 09600833. DOI: 10.1002/stvr.364 (cited on pages 3, 6).

[Mem08] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology*, 18(2):1–36, November 2008. ISSN: 1049-331X. DOI: 10.1145/1416563.1416564 (cited on pages 17, 18).

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. eng. Addison-Wesley, 1st edition. Edition, 2007 (cited on page 42).

[MHJ16] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, New York, NY, USA. ACM, July 2016. ISBN: 9781450343909. DOI: 10.1145/2931037.2931054 (cited on pages 3, 4).

[MMM14] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 599–609, New York, New York, USA. ACM Press, 2014. ISBN: 9781450330565. DOI: 10.1145/2635868.2635896 (cited on page 4).

[MN10] Atif M. Memon and Bao N. Nguyen. Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End. In *Advances in Computers*. Volume 80, pages 121–162. Elsevier Inc., 1st edition, 2010. DOI: 10.1016/S0065-2458(10)80003-8 (cited on page 3).

[Mor19] Peter Morlion. Software testing anti patterns. 2019. URL: https://www.enov8.com/blog/software-testing-anti-patterns/ (visited on 07/02/2020) (cited on page 53).

[MPN+08] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. How designers design and program interactive behaviors. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 177–184. IEEE, September 2008. ISBN: 978-1-4244-2528-0. DOI: `10.1109/VLHCC.2008.4639081` (cited on page 3).

[MPR+11] Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. Automated browsing in AJAX websites. *Data & Knowledge Engineering*, 70:269–283, March 2011. ISSN: 0169023X (cited on pages 17, 81, 96).

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, New York, USA. ACM Press, 1992. ISBN: 0897915135. DOI: `10.1145/142750.142789` (cited on page 3).

[MRZ14] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying Fine-Grained Co-evolution Patterns of Production and Test Code. In *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, September 2014. ISBN: 978-1-4799-6148-1. DOI: `10.1109/SCAM.2014.28` (cited on page 14).

[MS03] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 28 of number 5, page 118, New York, New York, USA. ACM Press, 2003. ISBN: 1581137435. DOI: `10.1145/940071.940088` (cited on page 15).

[MTN13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, page 224, New York, New York, USA. ACM Press, 2013. ISBN: 9781450322379. DOI: `10.1145/2491411.2491450` (cited on pages 3–5).

[MvD09] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, pages 210–220. IEEE, 2009. ISBN: 978-1-4244-3453-4. DOI: `10.1109/ICSE.2009.5070522` (cited on pages 3, 4).

[MvDR12] Ali Mesbah, A. van Deursen, and Danny Roest. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, January 2012. ISSN: 0098-5589. DOI: `10.1109/TSE.2011.28` (cited on page 4).

[Mye94] Brad Myers. Challenges of HCI design and implementation. *Interactions*, 1(1):73–83, January 1994. ISSN: 1072-5520. DOI: `10.1145/174800.174808` (cited on page 2).

[Mye95] Brad A. Myers. User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995. ISSN: 15577325. DOI: `10.1145/200968.200971` (cited on pages 2, 3).

[NCD+14] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, number 1, pages 803–813, New York, NY, USA. ACM, May 2014. ISBN: 9781450327565. DOI: `10.1145/2568225.2568317` (cited on page 14).

[NRB+14] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, March 2014. ISSN: 0928-8910. DOI: `10.1007/s10515-013-0128-9` (cited on page 3).

[PAN+20] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsDetect: an open-source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, New York, NY, USA. ACM, November 2020. ISBN: 9781450370431. DOI: `10.1145/3368089.3417921` (cited on pages 14, 42).

[PRZ18] Mauro Pezzè, Paolo Rondena, and Daniele Zuddas. Automatic GUI testing of desktop applications. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, number i, pages 54–62, New York, NY, USA. ACM, July 2018. ISBN: 9781450359399. DOI: `10.1145/3236454.3236489` (cited on page 3).

[PSD14] Heidar Pirzadeh, Sara Shanian, and Farzin Davari. A Novel Framework for Creating User Interface Level Tests Resistant to Refactoring of Web Applications. In *Proceedings of the 9th International Conference on the Quality of Information and Communications*

Technology, pages 268–273. IEEE, September 2014. ISBN: 978-1-4799-6133-7. DOI: `10.1109/QUATIC.2014.43` (cited on page 16).

[PSO12] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding Myths and Realities of Test-suite Evolution. *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 1:33:1–33:11, 2012. DOI: `10.1145/2393596.2393634` (cited on pages 14, 25, 27).

[PT15] Pallavi Pandit and Swati Tahiliani. AgileUAT: A Framework for User Acceptance Testing based on User Stories and Acceptance Criteria. *International Journal of Computer Applications*, 120(10):16–21, June 2015. ISSN: 09758887. DOI: `10.5120/21262-3533` (cited on page 7).

[PXP$^+$20] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering*, 5589(c):1–20, 2020. ISSN: 0098-5589. DOI: `10.1109/TSE.2020.3007664` (cited on page 17).

[RAF18] Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino. Is this the lifecycle we really want? In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 68–77, New York, NY, USA. ACM, July 2018. ISBN: 9781450359399. DOI: `10.1145/3236454.3236490` (cited on page 5).

[RBS13] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: a systematic review. *Inf. Softw. Tech.*, 55(7):1165–1199, 2013. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2013.01.008` (cited on page 26).

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009. DOI: `10.1016/j.scico.2009.02.007` (cited on pages 26, 38).

[RD99] Michiel Ronsse and Koen De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999. ISSN: 07342071. DOI: `10.1145/312203.312214` (cited on page 7).

[Ren16] Josiah Renaudin. Software testing anti patterns. 2016. URL: https://www.slideshare.net/JosiahRenaudin/antipatterns-for-automated-testing (visited on 07/03/2020) (cited on pages 45, 52, 53).

[RGD07] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based Assessment of Test Quality. *The Journal of Object Technology*, 6(9):231, 2007. ISSN: 1660-1769. DOI: 10.5381/jot.2007.6.9.a12 (cited on pages 14, 42).

[RH18] Thibault Raffaillac and Stéphane Huot. Application du modèle Entité-Composant-Système à la programmation d'interactions Applying the Entity-Component-System Model to Interaction Programming. In *Proceedings of the 30th on l'Interaction Homme-Machine*, pages 42–51. ACM, 2018. ISBN: 9781450360784 (cited on page 3).

[RKP$^+$18] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, and Yves Le Traon. Can we automate away the main challenges of end-to-end testing? In *CEUR Workshop Proceedings*, volume 2361, 2018 (cited on page iii).

[RKP$^+$19a] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. An Industrial Study on the Differences between Pre-Release and Post-Release Bugs. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 92–102, Cleveland, USA. IEEE, September 2019 (cited on page iii).

[RKP$^+$19b] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. On the Evolution of Keyword-Driven Test Suites. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*, pages 335–345, Xi'an, China. IEEE, April 2019 (cited on page iii).

[RKP$^+$19c] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. Ukwikora: continuous inspection for keyword-driven testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 402–405, Beijing, China. Association for Computing Machinery, 2019 (cited on page iii).

[Rob20] Robot RobotFramework. Introduction. 2020. URL: http://robotframework.org/ (visited on 02/15/2021) (cited on pages 7, 24, 108).

[RS21] Filippo Ricca and Andrea Stocco. Web Test Automation: Insights from the Grey Literature. In *Proceedings of 47th International Conference on Current Trends in Theory and Practice of Computer Science 2021*, pages 472–485, Bolzano, Italy, 2021. ISBN: 9783030677305. DOI: `10.1007/978-3-030-67731-2_35` (cited on pages 43, 44).

[SAI+11] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171. IEEE, June 2011. ISBN: 978-1-4577-0384-3. DOI: `10.1109/ICDCSW.2011.20` (cited on page 58).

[SC17] Jonathan A. Saddler and Myra B. Cohen. EventFlowSlicer: A tool for generating realistic goal-driven GUI tests. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 955–960. IEEE, October 2017. ISBN: 978-1-5386-2684-9. DOI: `10.1109/ASE.2017.8115711` (cited on page 7).

[Sci19] Anthony Sciamanna. What are the anti patterns of automation with selenium? 2019. URL: `https://anthonysciamanna.com/2019/10/20/avoiding-automated-testing-pitfalls.html` (visited on 07/02/2020) (cited on page 53).

[Sco15] Alister Scott. Five automated acceptance test anti-patterns. 2015. URL: `https://alisterbscott.com/2015/01/20/five-automated-acceptance-test-anti-patterns/` (visited on 07/02/2020) (cited on page 53).

[SG10] Yuri Shewchuk and Vahid Garousi. Experience with Maintenance of a Functional GUI Test Suite using IBM Rational Functional Tester. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering*, pages 489–494, Redwood City, San Francisco Bay, CA, USA, 2010. ISBN: 1891706268 (cited on page 15).

[Sha19] Liraz Shay. Bdd cucumber features best practices. 2019. URL: `https://www.linkedin.com/pulse/bdd-cucumber-features-best-practices-liraz-shay/` (visited on 07/02/2020) (cited on page 53).

[She20] Himanshu Sheth. 16 selenium best practices for efficient test automation. 2020. URL: `https://www.lambdatest.com/blog/selenium-best-practices-for-web-testing/` (visited on 05/04/2021) (cited on page 53).

[SIA+19] Ibrahim-anka Salihu, Rosziati Ibrahim, Bestoun S Ahmed, Kamal Z Zamli, and Asmau Usman. AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing. *IEEE Access*, 7(c):17158–17173, 2019. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2019.2895504` (cited on page 3).

[Sim19] Alex Siminiuc. What are the anti patterns of automation with selenium? 2019. URL: `https://www.quora.com/What-are-the-anti-patterns-of-automation-with-selenium` (visited on 07/02/2020) (cited on pages 45, 53).

[SR04] Mats Skoglund and Per Runeson. A case study on regression test suite maintenance in system evolution. In *IEEE International Conference on Software Maintenance, ICSM*, pages 438–442, 2004. ISBN: 0-7695-2213-0. DOI: `10.1109/ICSM.2004.1357831` (cited on page 15).

[Sta17] StackExchange. What are anti-patterns in test automation? 2017. URL: `https://sqa.stackexchange.com/questions/8508/what-are-anti-patterns-in-test-automation` (visited on 04/06/2021) (cited on page 45).

[SV17] Danilo Silva and Marco Tulio Valente. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 269–279. IEEE, May 2017. ISBN: 978-1-5386-1544-7. DOI: `10.1109/MSR.2017.14`. eprint: `1704.01544`. URL: `http://ieeexplore.ieee.org/document/7962377/` (cited on page 50).

[SYM18] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 503–514, Lake Buena Vista, FL, USA. ACM, October 2018. ISBN: 9781450355735. DOI: `10.1145/3236024.3236063` (cited on pages 10, 18, 81).

[TDS+13] Suresh Thummalapenta, Pranavadatta Devaki, Saurabh Sinha, Satish Chandra, Sivagami Gnanasundaram, Deepa D. Nagaraj, Sampath Kumar, and Sathish Kumar. Efficient and change-resilient test automation: An industrial case study. In *Proceedings of the 35th International Conference on Software Engineering*, pages 1002–1011, San Francisco, CA, USA. IEEE, May 2013 (cited on pages 17, 80, 81, 86, 96).

[Tem20] Jane Temov. Want to speed end-to-end testing? don't send in the clones. 2020. URL: `https://techbeacon.com/app-dev-testing/want-speed-end-end-testing-dont-send-clones` (visited on 07/02/2020) (cited on page 53).

[TGS+13] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 132–146, Ontario, Canada. IBM Corp., 2013 (cited on page 50).

[TPB+16] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 4–15, Singapore, Singapore. ACM Press, 2016. ISBN: 9781450338455. DOI: `10.1145/2970276.2970340` (cited on pages 14, 42).

[TSS+12] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering*, pages 881–891. IEEE, June 2012. ISBN: 978-1-4673-1066-6. DOI: `10.1109/ICSE.2012.6227131` (cited on page 17).

[TWM15] Xinye Tang, Song Wang, and Ke Mao. Will This Bug-Fixing Change Break Regression Testing? In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, volume 2015-Novem, pages 1–10. IEEE, October 2015. ISBN: 978-1-4673-7899-4. DOI: `10.1109/ESEM.2015.7321218` (cited on page 15).

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, January 1985. ISSN: 00199958. DOI: `10.1016/S0019-9958(85)80046-2` (cited on page 29).

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, August 2012. ISSN: 09600833. DOI: `10.1002/stvr.456` (cited on page 5).

[VD08] Bart Van Rompaey and Serge Demeyer. Estimation of Test Code Changes Using Historical Release Data. In *Proceedings of 15th Working Conference on Reverse Engineering*, pages 269–278. IEEE, October 2008. ISBN: 978-0-7695-3429-9. DOI: `10.1109/WCRE.2008.29` (cited on page 14).

[VDD$^+$07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering*, 33(12):800–817, December 2007. ISSN: 0098-5589. DOI: `10.1109/TSE.2007.70745` (cited on pages 14, 42).

[vDMvdB$^+$01] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Simulation of corrugated horns and exciters for circular corrugated waveguides. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pages 92–95, September 2001. DOI: `10.1134/S106422690609004X` (cited on pages 14, 42).

[VDP12] Raynor Vliegendhart, Eelco Dolstra, and Johan Pouwelse. Crowdsourced user interface testing for multimedia applications. In *Proceedings of the Workshop on Crowdsourcing for Multimedia*, page 21, New York, New York, USA. ACM Press, 2012. ISBN: 9781450315890. DOI: `10.1145/2390803.2390813` (cited on page 16).

[VFM15] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE, September 2015. ISBN: 978-1-4673-7532-0. DOI: `10.1109/ICSM.2015.7332456` (cited on page 15).

[VP18] Laszlo Vidacs and Martin Pinzger. Co-evolution analysis of production and test code by learning association rules of changes. In *Proceedings of IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation*, pages 31–36. IEEE, March 2018. ISBN: 978-1-5386-5920-5. DOI: `10.1109/MALTESQUE.2018.8368456` (cited on page 14).

[WFB19] Thomas D. White, Gordon Fraser, and Guy J. Brown. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 307–317, New York, NY, USA. ACM, July 2019. ISBN: 9781450362245. DOI: `10.1145/3293882.3330551` (cited on page 81).

[YCM07] Xun Yuan, Myra Cohen, and Atif M. Memon. Covering array sampling of input event sequences for automated gui testing. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, page 405, New York, New York, USA. ACM Press,

2007. ISBN: 9781595938824. DOI: 10.1145/1321631.1321695 (cited on page 6).

[YFF$^+$19] Shengcheng Yu, Chunrong Fang, Yang Feng, Wenyuan Zhao, and Zhenyu Chen. LIRAT: Layout and Image Recognition Driving Automated Mobile Testing of Cross-Platform. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 1066–1069. IEEE, November 2019. ISBN: 978-1-7281-2508-4. DOI: 10.1109/ASE.2019.00103 (cited on page 3).

[YTS$^+$14] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. Robust test automation using contextual clues. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 304–314, New York, New York, USA. ACM Press, 2014. ISBN: 9781450326452. DOI: 10.1145/2610384.2610390 (cited on pages 15–17, 81, 96).

[ZHH$^+$18] Yu Zheng, Song Huang, Zhan-wei Hui, and Ya-Ning Wu. A Method of Optimizing Multi-Locators Based on Machine Learning. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 172–174. IEEE, July 2018 (cited on pages 17, 81).

[ZLE13] Sai Zhang, Hao Lü, and Michael D. Ernst. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 45–55, New York, NY, USA. ACM, July 2013. ISBN: 9781450321594. DOI: 10.1145/2483760.2483775 (cited on page 18).

[ZVvD$^+$11] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, June 2011. ISSN: 1382-3256. DOI: 10.1007/s10664-010-9143-7 (cited on page 14).