POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico di Torino

Master's Degree Thesis

Enhance robustness of test cases by linting bad practices

Supervisors

Dott. Riccardo COPPOLA

Prof. Maurizio MORISIO

Prof. Ardito LUCA

Candidate

Davide CALARCO

03/2022

Summary

In software testing, GUI test cases are software functions that test the quality of the application under test (AUT) only passing through the graphical user interface (GUI). GUI test cases are well-known to be fragile, namely they do not pass anymore as soon as an element in the GUI gets modified. The thesis therefore aims at investigating the sources of fragility in GUI-based test cases and how to decrease it. For the purpose, first it is presented the overall background about software testing with particular attention to web applications. Then, a statistical collection and analysis on real projects aims at giving an objective overview about how testers face fragility and the forms under which fragility appears. A second data collection and analysis, this time on community-crafted wikis, produces a direct and practical set of good practices as countermeasures for fragility. The analysis of these wikis suggests to implement a linter, that is a static analyzer, to enforce such good practices in test cases; after an assessment about the deployment environment of the linter and its design, the thesis presents the results: test cases taken from real projects differ from the ideal model where the aforementioned rules apply; fragility has multiple sources, categorized in a taxonomy. Subsequently, the threats to validity criticize the methodologies followed to collect data and test the linter. Finally, the thesis suggests the possible future improvements about the present study and the linter.

Acknowledgements

ACKNOWLEDGMENTS

I would like to praise the doctor Riccardo Coppola that guided me along this journey.
I would like also to thank professors Maurizio Morisio and Ardito Luca for the opportunity to take this interesting thesis subject.
Last but not least, special gratitude goes to my family and friends that supported me from the very beginning of this chapter of my life.

Table of Contents

List of Tables

List of Figures

Acronyms

AUT
Application Under Test

POP
Page Object Pattern

GUI
Graphical User Interface

DOM
Document Object Model

CSS
Cascading Style Sheet

IDE
Integrated Development Environment

AOP
Aspect Oriented Programming

E2E
End-to-End

SE
Software Engineering

TDD
Test-Driven Development

BPMN

Business Process Model and Notation

BDD

Behaviuor-Driven Development

OS

Operating System

Chapter 1

Introduction

Software engineering borrows the concept of fragility from metalworking, like also rigidity, viscosity and so on. As commonly known, an object can respond in two ways when it gets subjected to some force: it morphs by bending itself, or it tries to keep a shape. In this last case, the object may either endure as it is against the force, or break in pieces; the more fragile the body is, the more breaking points occur within the volume of the object. In computer science, the software devoted to test an application, a.k.a. test cases, behave similarly: as soon as the AUT (Application Under Test) gets changed by a developer, test cases tend to break and need to be corrected accordingly; this phenomenon goes by the name of fragility of test cases. The more fragile a test case is, the more defects are induced by a modification in the AUT. Within the scope of this study, one step further has been done by widening the definition of fragility, as described in subsequent chapters.

Fragility is a property that affects quality of test cases, that is their maintainability. Nowadays, several tools try to improve quality of test suites: frameworks (i.e. Pyccuracy, Jasmine), libraries (i.e. Selenium), as well as dashboards (i.e. TestQuality), wrapper libraries (i.e. FluentBuilder), plugins (i.e. fastlane-plugin-test_center), static analyzers (Sonar). However, the aforementioned techniques only address test suites' fragility indirectly, that is as part of a whole, or partially. The purpose of the present study is therefore to face the fragility problem tête-à-tête, first by defining its boundaries and then implementing an automated tool that, given the results of the study, can aid testers in developing a less fragile test suite.

Fragility is an issue that affects most of the possible software artifacts in computer science, since change is an intrinsic property of software, namely it tends to evolve. For the purpose of the present study, the scope of possible software artifacts has been shrunk to GUI testing, where each test case emulates the behaviour of final users in order to find bugs in the AUT. This kind of technique is widely adopted because it allows to check the alignment of the application with the functional requirements agreed with the customer or the end user.

According to some heuristics, test cases would be flaky because users demand too much from them and from browser automation tools [1]. This statement, from one hand, highlights that visual testing is not a panacea, which is the reason why several testing techniques exist. On the other hand, however, the present study wants also to underline that it exists a gap from what is the ideal correct usage of visual testing and what testers actually know and write down while coding GUI test cases.

The remainder of the thesis is structured as follows: chapter 2 describes the state of the art about software testing, with particular attention to web applications (section 2.2), GUI test cases (section 2.3), testing tools (section 2.4), the definition and characteristics of fragility (section 2.5) and countermeasures against fragility (section 2.6). After that, chapter 3 aims at providing a different view on fragility by first collecting data from the field (section 3.1 and section 3.2), and then analyzing them to provide a taxonomy on the different causes of fragility (section 3.3). Then, chapter 4 discusses and shows the design of a linter to help testers in decreasing fragility. Subsequently, chapter 5 criticizes the linter characteristics by subjecting a series of real software test cases to the linter. Finally, chapter 6 ends with the possible future paths of the present study.

Chapter 2

Background and Related Work

2.1 Software testing

SE (Software Engineering) is the branch of engineering that tries to model software processes and facets. Indeed, programs have a complex lifecycle which therefore SE splits in separate but coherent steps. The overall goal of such steps is to produce a program, given in input a set of requirements. SE models each step of a software process as a function that, receiving a certain resource like documents or source code, returns a new artifact with added value. The following list summarizes the standard steps of a general software process:

Requirement analysis is the step where a software company listens to customer's desires and formalize them in the so-called 'requirement document'. This document may change along time, depending on which software process the firm opts for. A requirement document typically partitions the set of desired features in functional requirements and non-functional requirements; the former ones establish what the software should do, while the latter ones state how the software must satisfy the former ones. For instance, in case of an online library, the application must allow users to skim the next page of the catalogue in less than one second;

High-level design is the task of thinking the architecture of the software that is going to be created. In this step, designers envision, evaluate and write down the various modules of the tool and their mutual connections at a high level of abstraction, in accordance with the requirement document discussed above;

Detailed design is the process step where designers choose the low-level options

available to implement the modules of the program;

Implementation is the core of software processes, as it provides for the developer team writes the working source code in compliance with the requirement document, the architecture and the detailed design;

Unit testing belongs to the testing sub-process, whose general aim is to validate the program against the detailed design. In particular, unit testing focuses on the modules of the tool considered separate from each other;

Integration testing checks the validity of the modules of the tool, when they work together; the behaviour of the modules must mirror the high-level design presented earlier;

System testing (or E2E (End-to-End)), finally, checks how much the software product, seen as a black box, suits the requirements established during requirement analysis.

The aforementioned steps of software processes do not have any intrinsic order of application; depending on the specific software process chosen by the developing firm, such phases appear in a specific order. For instance, a traditional process is the V-model shown in Figure 2.1.

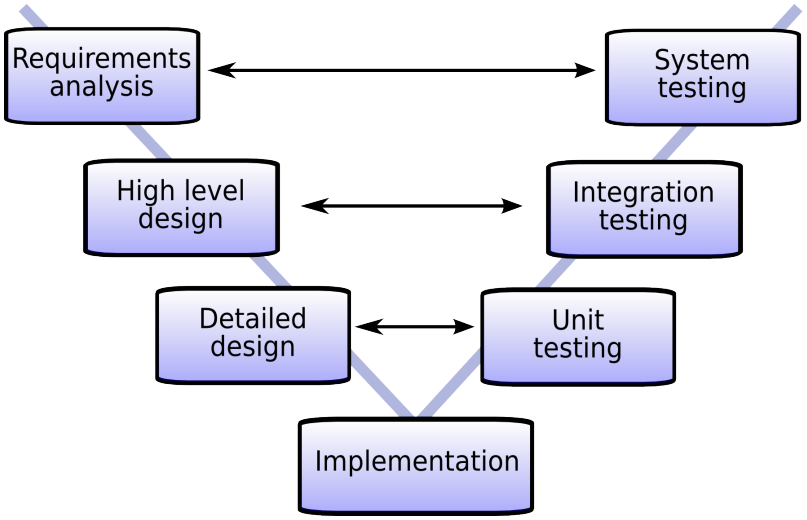


Figure 2.1: The V model software process follows a standard development pattern: first design, then implement and finally test.

Other software process models, like TDD (Test-Driven Development), put system testing in place from the very beginning of the development: for each requirement defined in the requirement document, the tester/developer first writes

the related test and then the source code. Figure 2.2 formalizes the typical TDD development/testing process.

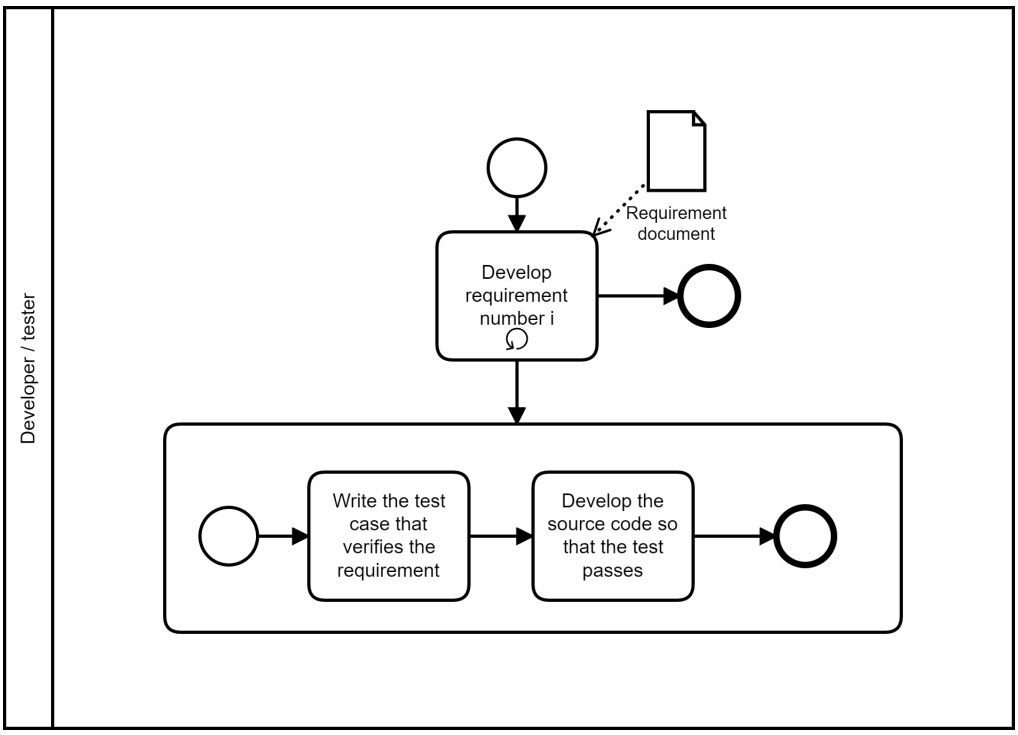


Figure 2.2: This BPMN chart models the developer, who also acts as a tester, that for each requirement first writes the related test case and then the pertinent source code.

Some software processes stress some phases at the expense of other ones. For instance, Agile methodologies put emphasis on working code, that is on sources and tests, whereas documents must remain very lightweight.

This thesis does not restrict the research net on any specific software process.

2.2 Testing of web applications

As for standalone desktop programs or mobile software, web application testing is a branch of the wide topic about software testing. Consequently, it inherits from general software theory diverse concepts, common findings and observations pertaining any aspect of test-related tasks, from psychological to economical; for instance, testing has the purpose of finding errors in an application or demonstrating its correctness. However, some peculiarities make web application testing diverge

from the other techniques. A presentation of the main concepts on web application testing is therefore necessary to achieve the goal of the study, as this chapter does.

Nowadays, people expect that every organization, whether a private firm or a social association, exposes its services on a web page [2]. Depending on the mission of the organization, Internet applications are more or less complex: schools and churches devote small focus on their web application which then result to be quite small and poorly maintained; instead, retailers tend to develop a full-fledged web portfolio. [2]

Users have a high-level standard about web applications today, which must satisfy a specific set of properties, here listed by decreasing order of importance for the service's perception of the user: aesthetics, responsiveness, precision, availability, usability, privacy, reliability and so on. If the web application doesn't match the expectation, a user may decide to check for another trustworthy company that offers the same service with better quality. In case of products in the flesh, people are instead willing to accept an unsatisfactory program bought on the shelf of a store, as long as it fulfills their basic needs, since they have paid for that program. Web applications have become the new first impression of companies, in the sense that users perceive the image of a firm on the basis of its website [2].

In order to ensure a high-tech quality, on the company side there must be a certain effort on testing the web application before any deployment or upgrade. Such effort amounts to half of the time and more than half of the cost in the context of a project, since the very beginning of the project itself [2].

If the development of a web application presents specific facets with respect to other types of software, the same can be stated about testing web applications. The resulting challenges stem from the great set of failure points that can occur in such applications, which cannot be controlled; since testing an application means detecting its failures, a test suite must take into account the aforementioned challenges, here partially summarized: [2]

1. User-related issues: a user can be more or less skilled, employ different browsers, operating systems and devices; the connection speed may also be fast or slow;

2. Business issues: the web application may interact with third-party hyperlinks, so testers must understand the system behind them for the sake of accuracy;

3. Internationalization issues: some data like timezone, currency and language may vary depending upon the user's culture;

4. Security issues: a company must seriously perform a risk analysis of its network and establish what are the security countermeasures to deploy against the considered attacks, like authentication, authorization, privacy and so on; these features must be thoroughly tested;

5. Testing environment issues: the test infrastructure must be identical to the real production ecosystem, which is made up of specific web servers, databases, firewalls and so on.

2.3 GUI testing

One strategy to test a web application is GUI testing: this term refers to all the techniques that pass through the visual interface exposed by the AUT to check its behaviour. GUI testing can in turn be further split in minor categories, depending on the method employed by test cases to find web elements at execution time:

- Coordinate-based locators: elements rendered by the AUT are historically found by test cases through their coordinates within the graphical interface (see Figure 2.3);
- DOM-based locators: test cases interact via the DOM data structure. The DOM (Document Object Model) data structure is a representation of a web page, automatically built by browsers; it conveys the hierarchical organization of elements that are rendered on the web page and their attributes like id, name and so on (see Figure 2.4);
- Visual-based locators: test cases interact with the AUT through algorithms for image recognition; the tester collects some screenshots of the graphical state, which are then used by these algorithms to assert the presence or absence of elements or attributes in a certain application state.

GUI testing concentrates on the front-end tier of the client-server architecture. As a consequence, it probes bugs in the client-side scripts, in the presentation layer to some extent and in the business rules compliance; it can be employed also for performance testing, although this practice is discouraged since visual test tools introduce an unknown overhead to operations [1].

Given the domain where graphical testing sheds light in order to find bugs, it is a high-level testing technique (a.k.a., end-to-end testing). Therefore, testers must formerly create lower-level tests for separate independent components, so as to easier the task of detecting which is the code snippet causing the failure [2]. As explained later in the thesis, the order of creation of test cases may follow different rules.

Graphical testing belongs to the category commonly called black-box testing, since it models a web application as a set of widgets and events working as a whole to accomplish a purpose. In other words, these tests don't depend upon how an actual functional requirement has been implemented.

GUI testing is a difficult task, for diverse reasons [3]:

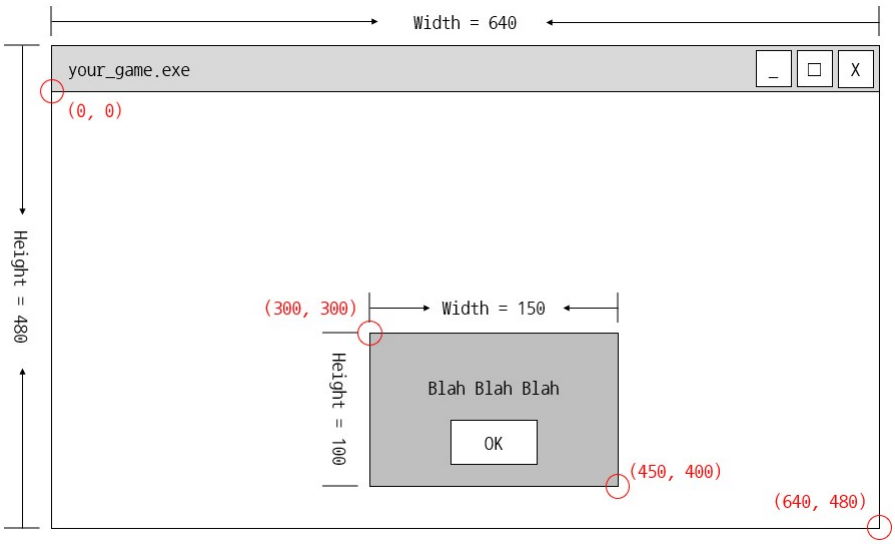


Figure 2.3: The widget at the center has coordinates (300, 300), which are relative to the container. The coordinates in this case refer to the top-left corner of the widget.

1. Interfaces are designed to be human-friendly rather than approached by a computer program;
2. Isolating graphical components is useless since they are strongly coupled. This is also the reason why the GUI testing cannot belong to the unit testing category;
3. GUIs are event-driven.

A GUI test can be further classified according to this taxonomy:

1. Functional test: the purpose is checking whether the web application works as expected and without issues;
2. Acceptance test: this is a subcategory of functional testing; it checks the application compliance against the customer's expectation;
3. Performance test: the goal is measuring how the application performs under different workloads; the load is simulated through multiple users that activate a certain feature, whereas performance measures like latency are gathered through other tools;
4. Interface-only test: what is being tested is not the whole AUT, quite just its mere interface, or better, the behaviour of the GUI in response to a business logic that was already tested through unit tests [3];

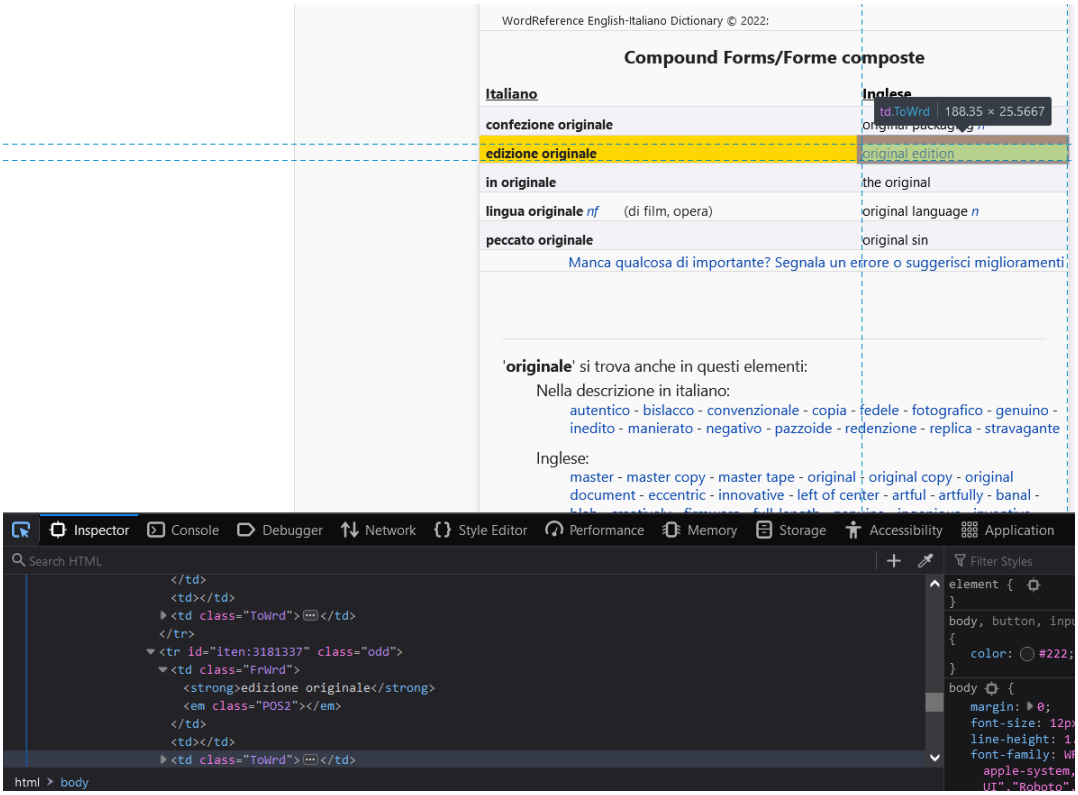


Figure 2.4: The element highlighted in green on the top right of the GUI corresponds to an element with tag 'td' and class attribute equal to 'ToWrd'. The inspector tab of the browser shows the HTML code from which the DOM is built and rendered, and in particular it highlights the widget. At the bottom-left corner, the inspector shows that the highlighted text box is child of the 'body' element, which in turn is child of the 'html' root element of the web page.

The creation of GUI test cases can be:

manual : this is the basic procedure;

partially automated : the testing tool translates the actions of a tester in test cases;

completely automated : an algorithm autonomously builds test cases.

Among the fully-automated creation testing approaches, the event-driven one has been studied previously [4]. Such paradigm models the GUI of the AUT as a graph, where each node represents an event, while each edge is a possible action between two events. The first observation that come out from this method is that

the focus is on events rather than widgets; secondly, the GUI gets modeled in a state machine useful to formalize and, at the same time, enumerate all the possible actions of the end user; considering all the possible paths is in turn an important task in testing. Figure 2.5 is an example of GUI model produced by event-driven algorithms.

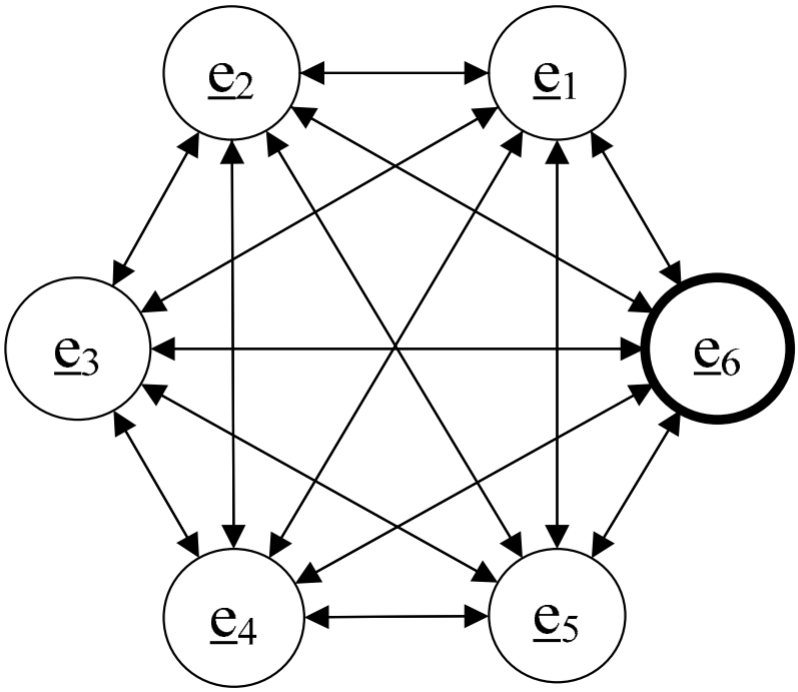


Figure 2.5: This graph represents all the possible actions that can follow an event, namely the graph is *complete*. The underlying algorithm will trim those edges according to some policy, otherwise the number of resulting test cases would be exponential [4].

Given the predominant monetary and time costs to develop a software project, along with the impact on user acceptance, testing a web application is a critical process. As such, the development team (testers included) apply the regression testing technique: they reuse tests written for a previous version of the web application so as to avoid creating tests from scratch at each upgrade. However, the aforementioned technique is not enough, as tests may not be compatible with the upgraded web application. The present study investigates why tests break when a web application changes and which patterns developers should follow to reduce the chance of breakage in a test suite, under the same conditions.

Anyway, testers must first consider that automating tests is not always the case:

if the application is going to change considerably in the near future, or the project management didn't devote so much time for testing, then the tester should just manually operate on the application without writing any programmatic test suite. Functional tests, that is GUI tests or graphical tests, require much resources to be run. The developer must therefore first wonder about he should instead build other types of tests, like unit tests or even manual tests. They must also take into account an important benefit GUI tests, namely they cover large portions of the AUT, whose features are often mission-critical for the web application. [1]

The following code snippet is an example of GUI test case; the language used is JavaScript, while the test tool is Selenium WebDriver:

```
1 it('Should be on the home page', function(done) {
2     var element =
3         driver.findElement(selenium.By.className('banner-text'));
4     element.getText().then(function(text) {
5       expect(text).toBe('Statistical Business Register');
6       done();
7     });
8 });
```

The purpose of the above test case is to check that the banner in the header of the web page shows "Statistical Business Register". The first statement searches for the first element in the GUI whose className attribute is 'banner-text'. The second statement wraps two inner statements, executed as soon as the text of the element becomes available: the former, at line 5, asserts (that is checks) whether the text is equal to 'Statistical Business Register'. In the affirmative case, the test case is considered passed (line 6); otherwise the test case reports a logical failure because the text does not match the expected value. Instead, if the driver cannot find any element having the specified className selector, the test case throws an exception, which is an error.

It is important to notice the difference between failure and error: while the former one represents a semantical gap between the expected outcome and the actual one, the latter has only to do with a wrong test execution.

2.4 Technologies for web application testing

As web applications are characterized by a range of software properties, like performance and security, there exist several frameworks which enable developers to enhance a subset of them. Figure 2.6[5] gives a quantitative outline on the usage of testing tools. The bar chart shows that projects typically adopt a Continuous Integration tool, a framework to automate testing and an actual testing platform.

Free tools like Selenium and Appium are generally preferred to proprietary applications such as HP and UFT. Given the relevance of the Selenium testing tool demonstrated by Figure 2.6, along with the desired target platform of the thesis which is web applications for computers only, the present study investigates test suites written for the Selenium tool. In order to better understand the context, the next section first of all gives an outline of the Selenium portfolio.

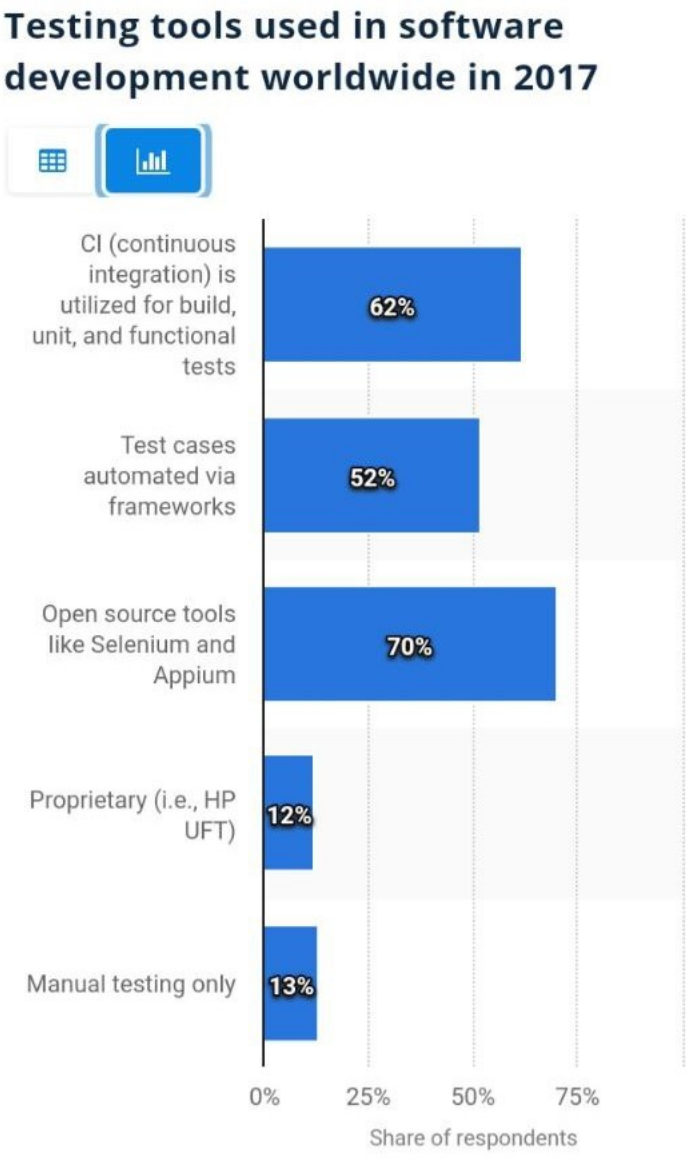


Figure 2.6: Testing tools usage in 2017.

The tools that will be described in this chapter are not meant for a specific direct research step within this study, quite just to give a theoretical background on some existing tools and set the stage for subsequent findings based on them.

2.4.1 Selenium

The present study has chosen Selenium as the target tool for research. Here follows a brief description thereof, intended to justify this choice and present the software capabilities.

In order to launch Selenium test cases, the tester has to either use Selenium IDE, which is actually an extension for Firefox, or the Selenium WebDriver library. Selenium IDE has the so-called capture and replay capability: it records the actions performed by the tester and automatically translate them into statements, which become test cases. Although more promising thanks to the capture and replay capability, Selenium IDE has some limitations as pointed out by previous studies. Indeed, it does not provide classical features that are available in the traditional programming paradigm, like conditional statements and logging; additionally, Selenium IDE produces too much duplicated code. [6]

Selenium is more in general a software suite, as shown in Figure 2.7.

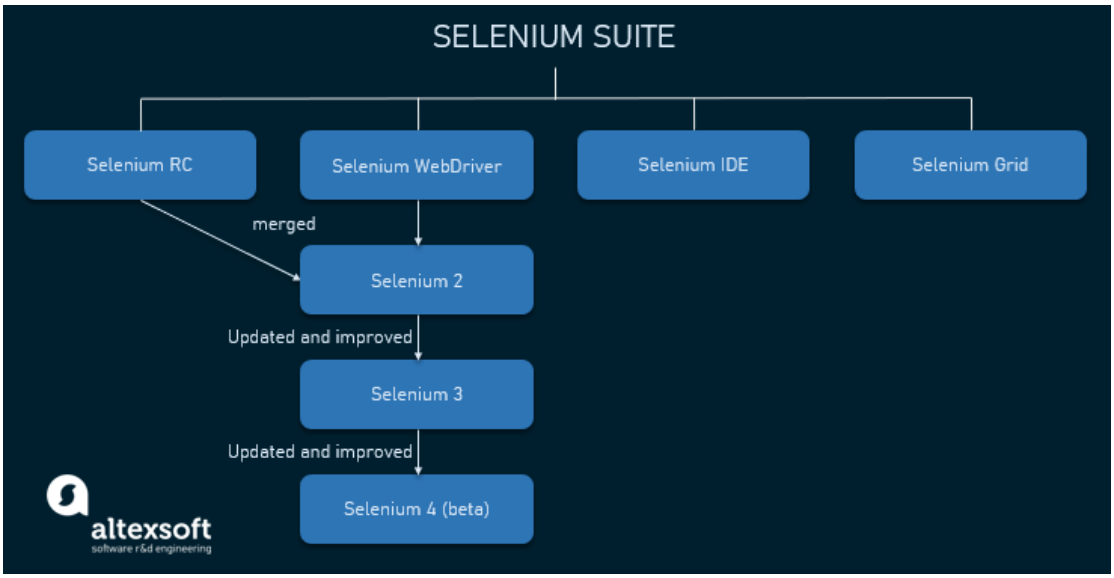


Figure 2.7: The Selenium suite as of 2021.

Selenium Grid tool allows to split the execution of a test suite on different machines, so as to run them in parallel and so decrease the time needed to complete. Indeed, the execution of GUI testing is a long-standing problem in general, due to its slowness.

Although the Selenium project started in 2004, it is still widely used by companies for a range of reasons:

1. Selenium is free, fact that encourages novices on trying it; on the business side, companies acknowledge that Selenium is one of the few testing tools that is simultaneously free and powerful at the same level of pay-for-use tools;

2. Selenium is compatible with a range of browsers, the most used nowadays: Chrome, Firefox, Edge, Safari and so on; this kind of support with browsers is native, in order to overcome the limitations of the sandbox in browsers;

3. It allows to write test cases in several programming languages, the most common as of today: JavaScript, Java, C#, etc;

4. Integrated from the start with Agile, DevOps and Continuous Delivery workflow; these three concepts indeed have great relevance in the modern software development paradigm;

5. Selenium supports mobile testing through specific tools, like Selendroid and Appium;

6. Testers can run Selenium on Windows, Linux and Mac.

It is essential to observe that all these features are not so commonly supported by other testing software, overall.

However, Selenium leads some disadvantages that must be taken into account: [7]

1. Steep learning curve: novices and non-skilled people like managers will find Selenium quite complex to learn since good programming skills are required;

2. No support for desktop apps: Selenium does not provide any direct way to test stand-alone applications designed for offline and local usage;

3. No built-in image comparison: Selenium has to be integrated with other tools when doing visual testing, a.k.a. image comparison;

4. No reporting capabilities: Selenium has to be used in combination with dashboards that record the execution measures.

The present study concentrates on test cases integrated with the Selenium WebDriver and its evolutions, although no particular attention is put on distinguishing the different versions thereof.

2.4.2 Cypress

Though the present study focuses on test suites written for Selenium, it may be useful to compare it with another tool for better comprehension. The selected tool is Cypress, whose developing team explicitly sheds light on its differences and advantages w.r.t. the well-established Selenium suite [8].

Cypress runs inside the browser, that is in the same event loop of a web application; Selenium is instead designed as an outside driver that issues commands to the browser through the network. Cypress test cases therefore benefit from a *native* access to the DOM, to the AUT and to the underlying network traffic. Testers in other words have greater control over the testing environment by arbitrarily setting its properties; this is a great advantage, given what section 2.2 explained.

As this study discusses later on, one disadvantage of GUI tests is the synchronization between the driver statements and the state of the AUT. Cypress runs inside the browser, so it can automatically coordinate the test suite execution and the AUT state without explicit management from the developer. The relevance of this point is highlighted by the remainder of this study.

Unlike Selenium, Cypress automatically takes screenshots without requiring third-party support. This feature is useful both for visual testing, if needed, and to debug a test case. Test cases written for Selenium are indeed difficult to debug.

Finally, the other relevant difference w.r.t. Selenium is the built-in dashboard that enables testers to enjoy their work. Indeed, GUI testing is known to be a repetitive task: even medium-sized web applications contain tens of GUI screens and hundreds of GUI actions [9], so testers find this task tedious; the direct consequence is a lower level of attention from developers, that unintentionally inject several bugs in tests. This is a kind of psychological fragility, as defined by the taxonomy discussed later on in section 3.3.

2.4.3 Other tools

Another technique for web application testing, which is pretty close to GUI testing, is BDD (Behaviuor-Driven Development). This technique is an extension of TDD (Test-Driven Development), applied to acceptance test cases. Indeed, customers typically provide a set of test cases that the AUT must pass in order to be considered 'accepted'. Tools that focus on this strategy try to express acceptance test cases in natural language, so that they can be understood even by non-expertise people on programming like managers. One of these tools is Pycurracy, a BDD-styled tool written in Python that aims to make it easier to create automated acceptance tests [10]. Here follows another example, where the behaviour of a test case is encoded in a specification language, whose syntax is based on XML:

```
<TestCase>
    <Type>JButton</Type>
    <Order>7</Order>
    <Action>
        <Push>yes</Push>
    </Action>
    <LogMessage>push button (1)</LogMessage>
</TestCase>
```

The test case gets then automatically translated into a GUI test case in the flesh [11]. A significant advantage of BDD, which is in common with other test-first development methods, is that one can write test cases even if the AUT is not fully implemented yet.

There is a plethora of tools that help during the testing process. Some of them can be classified as dashboards, which group test cases in minor categories like 'To-Do', 'Passing', but not only. For instance, Percy [12] automatically collects snapshots of the whole application page set; when a change in the AUT affects the GUI, Percy notifies the developers about it.

Sikuli [13] is a record-and-playback tool, whose main feature is to translate the actions of the tester within the GUI in test statements. Anyone can use these kind of tools; however, the playback feature keeps the machine's cursor busy for a quite a long time, since the playback speed is limited.

Vista [14] is a prototype tool for the automatic repair of a test suite. The description of such kind of tools is discussed later in subsection 2.6.1.

Vista leverages computer vision to correct breakages occurring in test cases.

TestQuality [15] measures fragility of test cases by inspecting which of them still work for different versions of the web application. This analysis is a kind of prediction: the more a test case has survived without any breakage, the greater chance that it will keep working for the next release of the application.

TestQ [16] is a static analyzer that inspects the whole source code of test suites in a project in order to detect code smells and present them in a GUI through hierarchies, graphs, measures and charts. TestQ concentrates on overall maintenance, but some considered code smells pertain fragility more or less directly.

Other platforms rather concentrates on the self-healing process, where test cases that present a breakage get repaired. This technique is thoroughly explained later in the thesis.

2.5 Fragility of tests

Following the ISO/IEC-25010 standard, the fragility property, sometimes called brittleness, belongs to the wide category 'Maintainability' and, more in detail, to its

subcategory 'Modifiability'. Here is indeed reported the definition of modifiability according to the standard: «Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality». It is essential to stress that the present study investigates fragility about test cases, rather than about the AUT.

Fragility of test cases depend upon many factors, like the testing tool, the developer knowledge and the testing technique. Concerning this last one, several strategies can be adopted as discussed in section 2.2. The three techniques respond differently to modifications in the AUT:

- Coordinate-based testing: as the target element is moved across the graphical interface, all the referencing test cases break. On the other hand, if the element is just lifted or lowered w.r.t. the DOM hierarchy, no direct breakage occurs in the test suite;

- DOM-based testing: if an element gets moved w.r.t. the GUI in the same page, no test case directly breaks as the underlying DOM structure has not been modified accordingly. Instead, whenever an element is moved to another container, web page or its attributes are changed, so test cases may break depending on the employed locator;

- Visual testing: even if an element gets moved according to either the GUI or the DOM, visual tests don't break for direct reasons, since the element's appearance in a certain state did not change. However, similar objects in the GUI may disturb the recognition of the intended element and produce false positives or disrupt the flow of a test case which may succeed without accomplishing its real purpose.

As the previous list points out, no technique is universal and fragility-free. The three ways elements on the web page are located, called locators, are fragile against a certain type of modification and robust toward another one.

Previous studies, like [17], built a taxonomy on the various fragility causes in mobile apps. Although some variations exist due to the deployment environment, most of these causes are in common with desktop applications; subsequent chapters will demonstrate this. Figure 2.8 shows a two-level taxonomy [17] that categorizes each change to a wider abstract change concept.

Here follows a brief description of the taxonomy:

Test code change only pertains changes in the test source files;

Application code change is about the evolution of the business logic of the AUT;

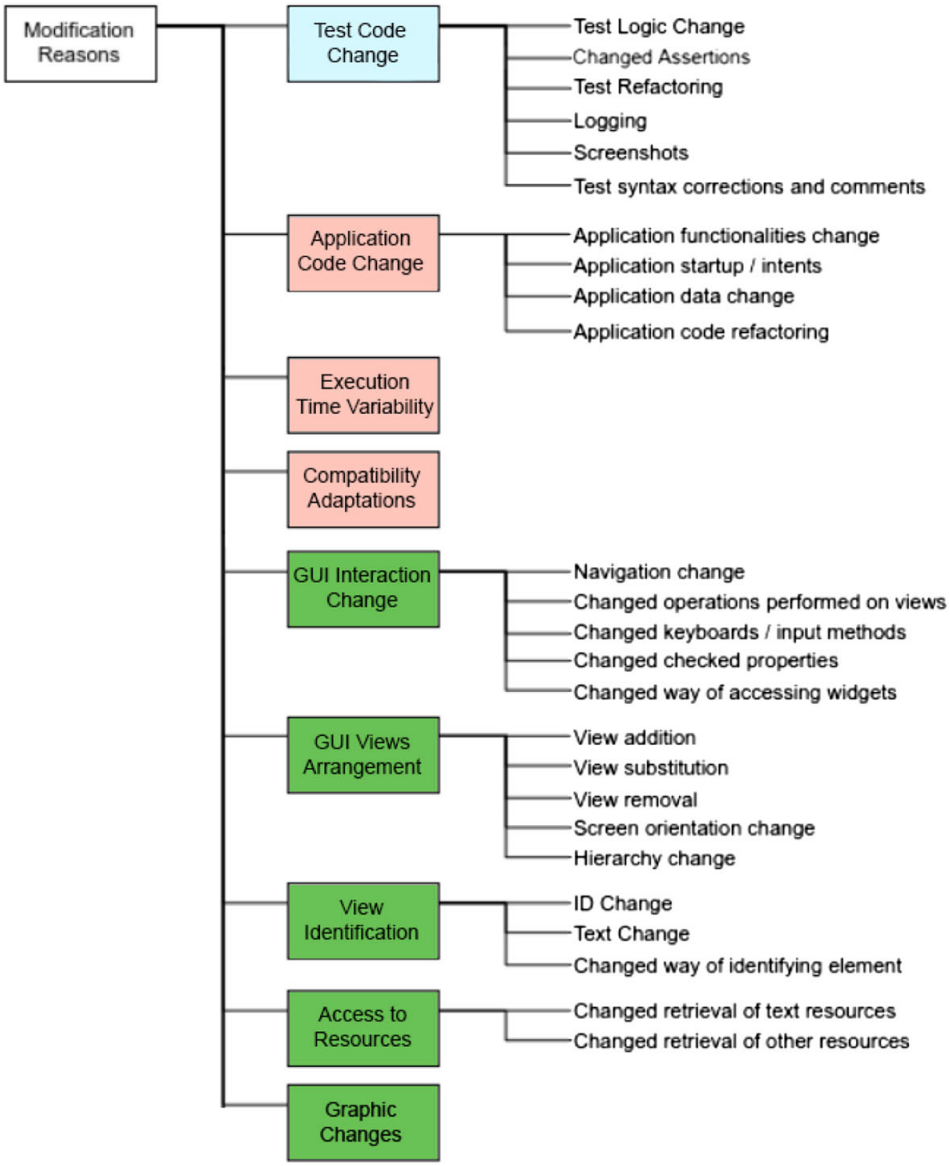


Figure 2.8

Execution time variability models the overheads due to network latency and host machine workload;

Compatibility adaptations are changes in the code that take care of selecting the right approach to solve a problem w.r.t. the host Android OS (Operating System);

GUI interaction change have the purpose to align the test code with the upgraded way of accessing / checking graphical widgets;

GUI Views arrangement considers any addition / modification / deletion of the GUI of the AUT;

View identification is about locator changes, that is how the test code retrieves an element of the AUT;

Access to resources categorizes all the modifications on the methods used to fetch resources of the project, like properties files;

Graphic changes , finally, embrace all those modifications in the appearance of widgets, like screen size or textboxes width.

The present study will propose a one-level taxonomy with the same intent as the aforementioned one.

2.6 Approaches to reduce fragility

The fragility of test suites is a problem that can be addressed also by enhancing one property of theirs: robustness, resiliency or antifragility.

What make the three properties differ is their approach toward change:

1. Robustness: the purpose of improving such a property of a test suite is to avoid any future modification at all; in other words, this technique struggles with the concept of change by shielding the test suite against any modification. Robustness is a kind of fault prevention, which is enforced by following best practices [18];

2. Resiliency: methods that enhance this characteristic of a test suite still try to refuse change, but, if that happens, they adapt the test suite by repairing it, namely by correcting it automatically;

3. Antifragility: this modus operandi totally embraces change, since antifragile software learns from bugs how to improve itself.

2.6.1 Resiliency

Also called 'Self-healing' or 'Auto-repairing', resiliency is the traditional alternative to reduce fragility beside robustness. In essential terms, a repairing tool automatically solves breakages; when the repairing tool cannot determine a suitable patch for the broken code, a human tester intervene to solve the issue manually.

A previous study distinguishes three types of breakages [9]:

Direct breakages cause the test to fail in the same statement where the source of the failure resides;

Propagated breakages provoke a failure of the test case some statements after the real source of the failure;

Silent breakages do not cause the test to fail, although the test behaviour semantically diverges from the intent of the test case.

Healing tools can only probe direct or propagated breakages as their direct consequence is a test failure, whereas silent breakages remain undetected. That's why repairing tools cannot totally replace human maintainers. Moreover, even in case of direct or propagated breakages, the repairing tool may find more than one admissible solution, so a human operator must come into play also in this case.

The paramount idea behind the repair strategy is that most actions that testers perform in order to align breakages can be automated. Indeed, as pointed out previously in this study, GUI testing is a repetitive task, fact that often enables automation.

subsection 2.4.3 already presented some self-healing tools.

2.6.2 Antifragility: a new way to engineer errors

The concept of antifragility is traced back to the book 'Antifragile' of Nassim Nicholas Taleb. Most test frameworks nowadays pretend that what the developer wants is a robust or resilient test suite. However, the real opposite of fragility goes by the name of antifragility, whereas robustness and resiliency are just on the way between the two concepts [19].

A resilient system is not antifragile, either; an adaptive resilient system is. In other words, state and behavioural repairs described in subsection 2.6.1 subtend to an antifragile process if no human intervenes. In final analysis, an adaptive resilient system is a learning system which actively changes its behaviour in response to breakages; this process recalls Artificial Intelligence and Machine Learning techniques studied by Data Science [18]. Although this approach is promising, the present thesis does not follow it since it requires skills proper of data scientists. The study therefore just presents antifragility without the willingness to provide an actual implementation.

Since antifragile software enhances when a failure (breakage) occurs, it should self-inject faults as well. This is the reason why antifragile software is said to embrace change. For instance, Netflix deliberately introduces stress like delays and breakdowns in their servers to test them; the Netflix team calls this technique as 'chaos engineering'.

This approach, that is antifragility, shapes the maintenance process to be reactive to spontaneous changes stemming from business evolution; this point is important and has industrial relevance, as business is in continuous ferment. Additionally, the induced changes may reveal some solutions that innovate the system [19]. The benefit is that antifragile software is a source of innovation for projects.

In general, antifragility triggers a revolution of the current noosphere about testing (design principles, mindset of engineers and so on) [18]. Traditionally speaking, bugs are evaluated as unwanted by developers due to their disruptive effects. Antifragility, besides the implementation requirements, asks first of all for a change of mindset, where the fault is seen as a kickstart for improvement. This task is more psychological than technical and demonstrates that testing is not about know-how only. The present study will underline this observation in multiple occasions in subsequent chapters.

2.6.3 Robustness, resiliency and antifragility in comparison

The effectiveness of the three approaches described so far is shown in Figure 2.9.

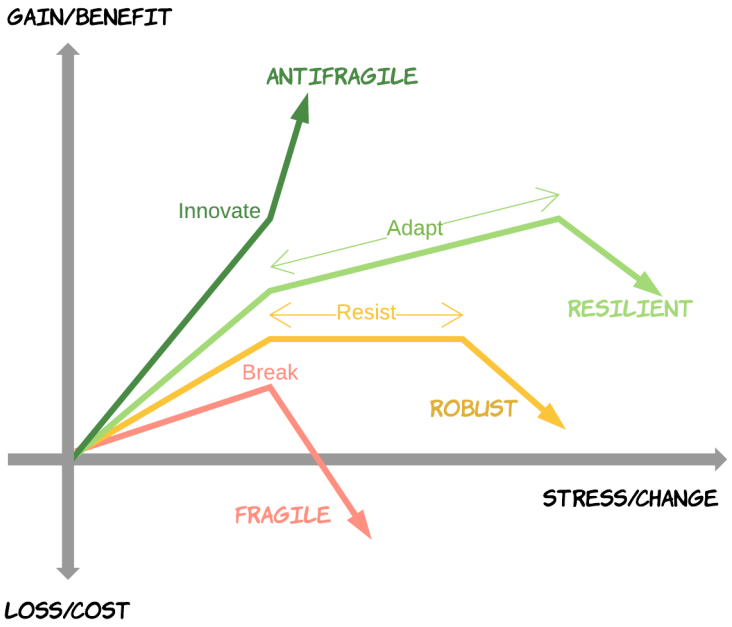


Figure 2.9: Comparison of fragility countermeasures in terms of benefit.

The figure underlines the effectiveness of each countermeasure in qualitative terms. Each technique responds differently to induced or natural stress, which

comes more or less spread along time.

In the basic form ('Fragile' curve), a fragile application just breaks when a change occurs; in this scenario, developers did not devote any particular attention to software fragility, either with or without purpose.

Robustness ('Robust' curve) is the worst strategy intended to address the fragility problem. It works by delaying to some extent the time where software breaks, but that instant will return anyway later on.

Resiliency ('Resilient' curve) is a better policy than robustness. However, both robustness and resiliency tend to zero when the stress tends to infinite. This common problem is the kickstarter for a better solution, which is not just theoretical. As pointed out later on, developers unconsciously or voluntarily feel the need for a solution such that no further human effort will be required to maintain software when a change in the AUT happen.

Antifragility ('Antifragile' curve) is the best strategy to respond to stress in modern Software Engineering. Among the benefits of antifragility briefly described in subsection 2.6.2, the chart above underlines its capability to benefit from stress. Indeed, the curve tends to infinite when change tends to infinite, unlike the other curves.

2.6.4 A further countermeasure: the Page Object Pattern

Another existing technique to reduce fragility is the POP (Page Object Pattern), which has been catching on in the last years. According to its basic principle, testers must separate the behaviour of tests from their actual implementation: a test case is just a series of human-friendly abstract actions, like "Do login"; each action, in turn, calls the underlying low-level functions to carry out the task. Here is an example of test that follows the POP pattern:[1]

```
1     @Test
2     public void testLogin() {
3         SignInPage signInPage = new SignInPage(driver);
4         HomePage homePage = signInPage.loginValidUser("userName", "
    password");
5         assertThat(homePage.getMessageText(), is("Hello userName"));
6     }
```

As it can be observed from the code, each page of the related web application is modelled as a class according to the Object-Oriented Programming. Each class implements the actions needed to perform one abstract action.

Test cases benefit from this pattern under the fragility point of view since, if the AUT gets changed, they remain as is due to their behavioural design. What will be modified is the implementation, that is the page classes. In other words,

the POP moves the fragility problem from test cases to more fine-grained tasks which face small issues. The consequence is that the sum of maintenance effort needed by each implementation function is strictly lower than the maintenance effort required by a test suite that does not adopt the POP.

It is worth to mention here that the page object pattern helps in the fragility problem depending on the definition of fragility; previous studies [6] argued that POP don't improve robustness of test suites, since page objects still belong to test cases. However, according to the definition of fragility within the present study, the POP helps because it reduces by a factor of three the time needed to realign the test suite and by a factor of eight the LOCs involved in the realignment, as pointed out by the same studies. These maintenance indicators in turn affect whether the test team is willing to devote time for testing or proceeds in deleting tests to reduce the maintenance effort; but deleting test cases is a kind of modification which increases fragility.

Chapter 3

Conceptualization of the fragility issue

This chapter first gives the definition of fragility within the scope of the thesis. Secondly, there is a bottom-up data collection process with the purpose of gathering in-the-flesh data on the most common fragility issues. Thirdly, another data collection task is performed where the source are community-crafted wikis. Finally, it follows the analysis of the collected data to infer a model that tries to fit those data and, simultaneously, leads to decrease fragility in test suites.

Let S be an arbitrary snippet of code, like a statement or a test case. Then S is fragile against a modification M if M can be applied to S. The measurement of the degree of fragility of S increases as the number of times M occurred in the wild enhances.

The above definition is wide, since M may or may not take place in the AUT. For instance, M can be a comment modification or even a macro definition. It is also wider than some previous related works which considered breakages only as a response to changes in the AUT. Concerning the measurement modality of the degree of fragility given a snippet S, it is necessary to count the occurrences of M that took place in projects developed in the past. Incidentally, this way to measure fragility of a currently-available test case is a kind of prediction and has to do with quantitative data mining.

In any case, the above definition considers a modification relevant only when it happened at least a certain amount of times in the wild: for example, evident style modifications that occurred in $< 0.1\%$, w.r.t. a sample of test cases taken from the wild, induce a negligible level of fragility. Instead, if a modification has never occurred, then, according to the above definition, it induces fragility with null level, which is equivalent to state that the modification doesn't produce any fragility at all.

The upper definition has been actually built incrementally, as the data collection process was going on. However, the desire to enlarge the scope of the definition to every applicable modification was established from the very beginning of the present study.

3.1 Data collection from test suites history

3.1.1 Design

The first step on data collection consists of inspecting a test case, or better its modifications between successive commits, along its lifetime in a project. The task was performed on GitHub, thanks to its user-friendly graphical interface. Incidentally, no script has been leveraged for the purpose described in this section.

In order to accomplish the aforementioned collection process, the search key ‘extension:java filename:*test* language:Java selenium‘ was applied in the GitHub search bar. GitHub groups results by categories like ’Repositories’, ’Code’, ’Wikis’ and so on; this section leveraged the category ’Code’, in order to work on test cases; the other collection process, described in section 3.2 instead selected ’Wikis’. The GitHub utility then shows a series of test cases as result, sorted by best match, and split up in 10 test cases per web page, as shown in Figure 3.1.

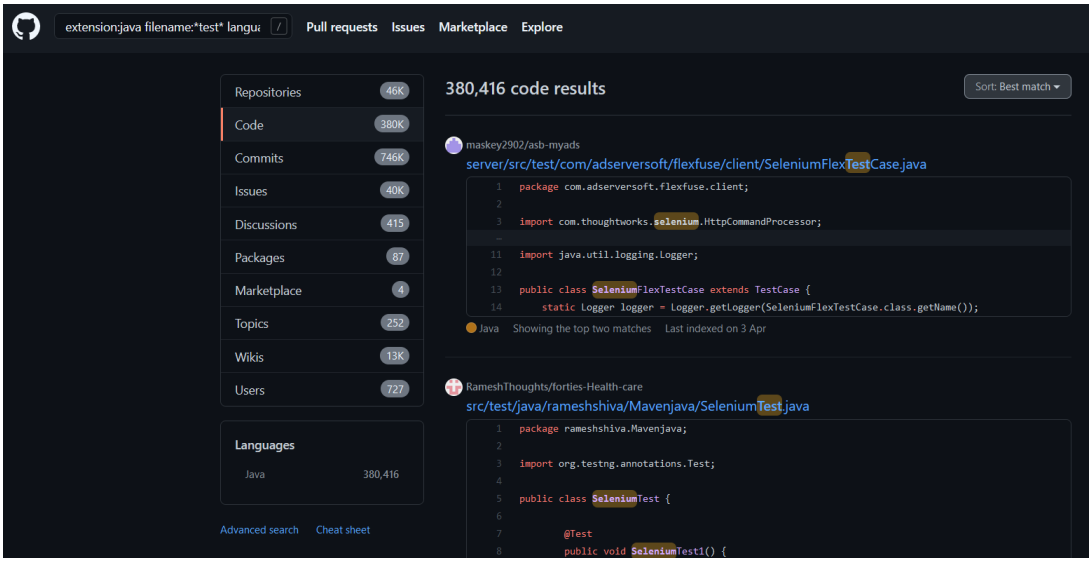


Figure 3.1: The collection process: scanning different test cases.

For each test case, GitHub allows to land on the pertinent web page, where the button currently called ’History’ lists all the commits of a repository to which the test case has been involved, that is modified, as shown in Figure 3.2.

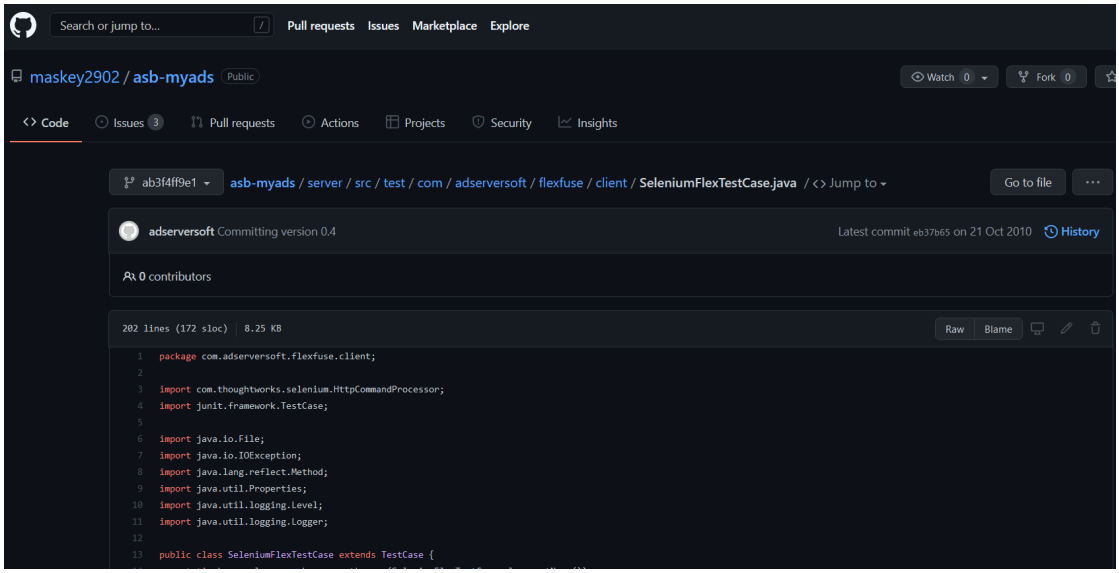


Figure 3.2: The collection process: clicking on the test case's history, at the center-right edge of the image.

For each commit, it is enough to click on its title, which acts as a link, so as to vision the modifications applied to files w.r.t. the previous commit, as shown in Figure 3.3 and in Figure 3.4.

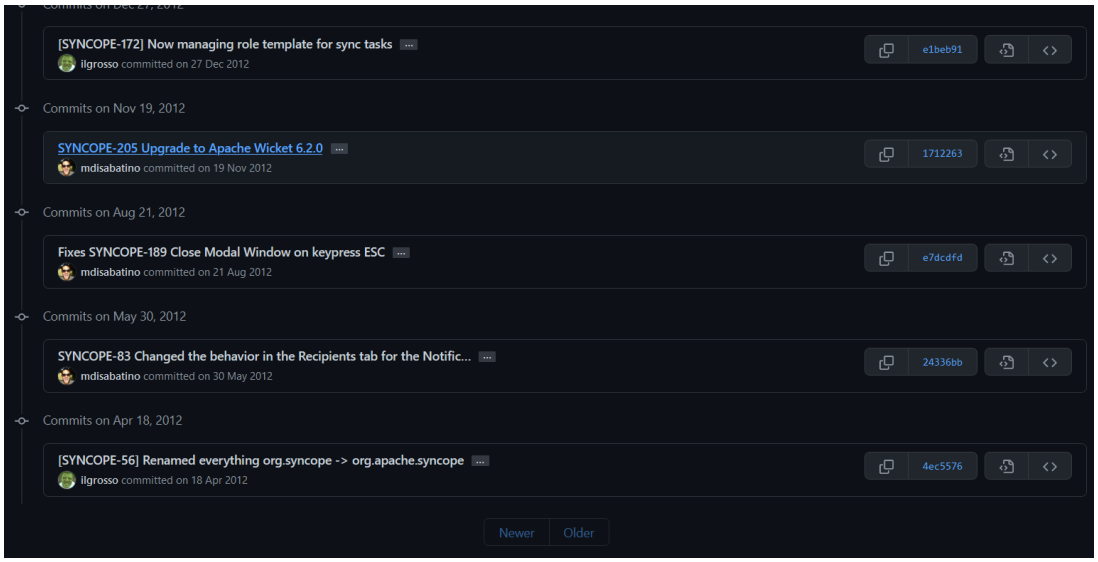


Figure 3.3: The collection process: listing test case's commits.

At this point, the collection process starts (or continues) and data are written

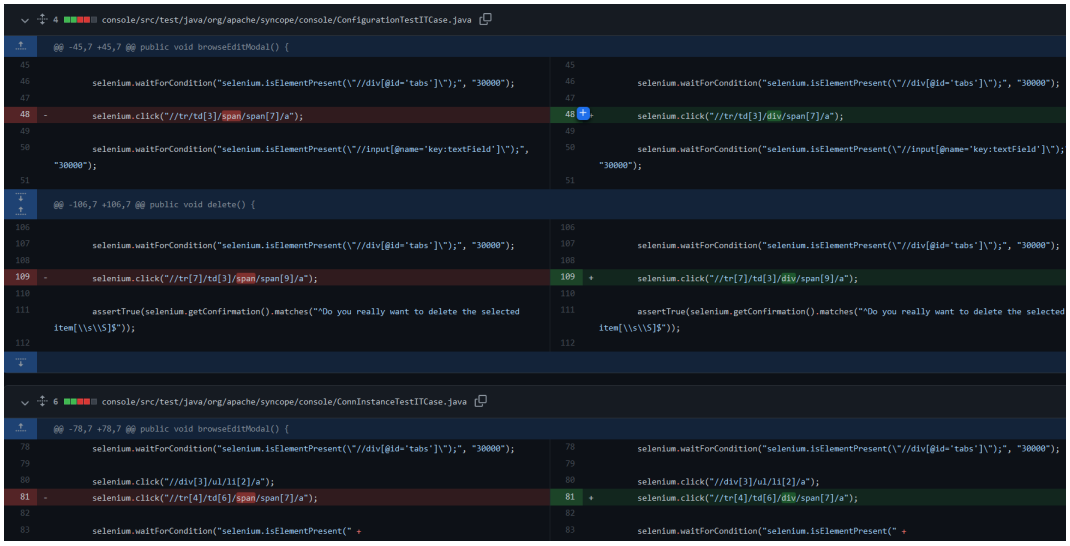
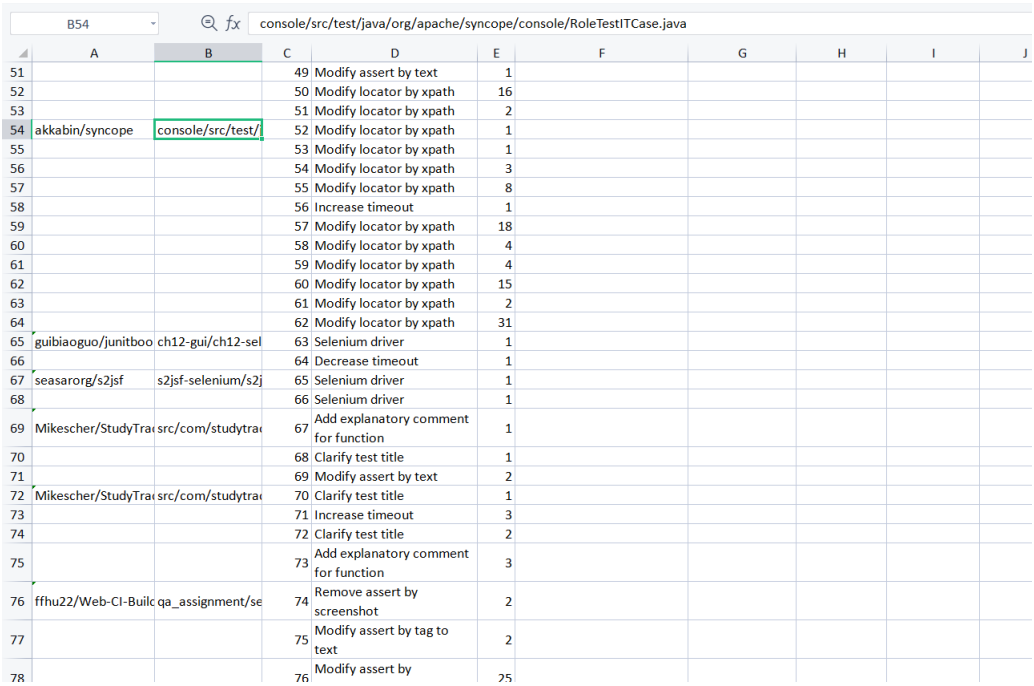


Figure 3.4: The collection process: inspecting test case's changes in a commit.

down on a spreadsheet, as shown in Figure 3.5.



B54 — console/src/test/java/org/apache/syncope/console/RoleTestITCase.java

	A	B	C	D	E	F	G	H	I	J
51			49	Modify assert by text	1					
52			50	Modify locator by xpath	16					
53			51	Modify locator by xpath	2					
54	akkabin/syncope	console/src/test/	52	Modify locator by xpath	1					
55			53	Modify locator by xpath	1					
56			54	Modify locator by xpath	3					
57			55	Modify locator by xpath	8					
58			56	Increase timeout	1					
59			57	Modify locator by xpath	18					
60			58	Modify locator by xpath	4					
61			59	Modify locator by xpath	4					
62			60	Modify locator by xpath	15					
63			61	Modify locator by xpath	2					
64			62	Modify locator by xpath	31					
65	guibiaoguo/junitboo	ch12-gui/ch12-sel	63	Selenium driver	1					
66			64	Decrease timeout	1					
67	seasarorg/s2jsf	s2jsf-selenium/s2j	65	Selenium driver	1					
68			66	Selenium driver	1					
69	Mikescher/StudyTra	src/com/studytra	67	Add explanatory comment for function	1					
70			68	Clarify test title	1					
71			69	Modify assert by text	2					
72	Mikescher/StudyTra	src/com/studytra	70	Clarify test title	1					
73			71	Increase timeout	3					
74			72	Clarify test title	2					
75			73	Add explanatory comment for function	3					
76	ffhu22/Web-CI-Build	qa_assignment/se	74	Remove assert by screenshot	2					
77			75	Modify assert by tag to text	2					
78			76	Modify assert by	25					

Figure 3.5: The collection process: writing down data on a spreadsheet.

The spreadsheet is subdivided in rows, each storing an N-uple with signature:

(project, test case path, type of modification, number of occurrences).

'project' is a string that uniquely identifies a repository on GitHub. If this identifier is copied and pasted into the GitHub search bar, the result indeed shows one repository only;

'test case path' is the relative path of the test case within the repository; this metadata helps in locating the test case given the repository; notice that a test case is a test file in this context, not just a test function;

'type of modification' is a label that categorizes a modification for future statistics; see Figure 3.5 for an example;

'number of occurrences' normally refers to how many changes of the same type occur in the same test case within a commit. As a consequence, multiple instances of the same identical modification in a test file are counted as distinct modifications, so all of them concur to the final measure.

Since the GitHub search outcome is grouped by test case, storing the repository identifier and the test case relative path avoids to consider the same project multiple times.

It has been decided to stop the collection process at page 72, having a reasonable database at hand. Nothing prevents future collection effort in order to increase the corpus. As previously pointed out, since the test cases and their relative project identifiers are saved into the spreadsheet, it is possible to avoid starting from scratch.

The number of scanned test cases is 720, since the process stopped at page 72 and GitHub shows 10 test cases per page. However, test cases that traced back only to one commit were discarded, since only differences between different versions of the same test case are relevant. However, having just one commit doesn't imply that developers didn't find any fragility issue during maintenance, quite that they have not been traced back on GitHub. From the number of total test cases and the actual entries in the spreadsheet, it comes out that the actual number of test cases involved in multiple commits is 66 (9.2%).

The purpose of this collection is, as described apart in the study, to underline a general and objective overview of the modifications that developers apply to test cases. The general data roll-up is shown in Figure 3.6.

The total number of probed modifications is 1291, each one belonging to a category as shown in Figure 3.6. The various categories are ranked by decreasing number of modifications, beside the last one about unlabeled type of modifications. A category is the semantic generalization of various types of modification from the spreadsheet.

The successive sub-sections discuss each category shown in Figure 3.6 with more detailed data and description.

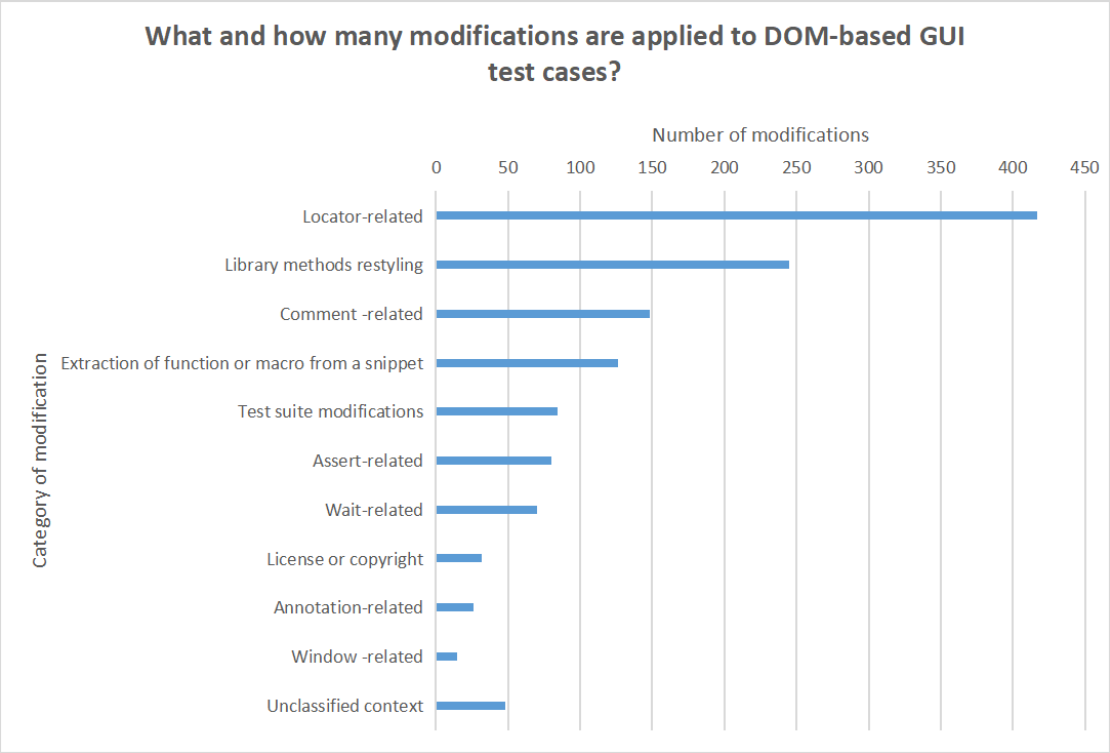


Figure 3.6: Data have been rolled up and categorized.

3.1.2 Fragilities related to locators

The most relevant category is about locator modifications, the traditional main issue on GUI testing (32%). Figure 3.7 shows the general trend about locator-related modifications.

An XPath is a kind of locator that identifies a web element through the hierarchical DOM path of it. If the path starts from the root of the web document, the XPath is said absolute, otherwise it is called relative. Here is an example of absolute XPath of the submit button of a probable web page:

```
1 "/body/div[@name=\"container\"]/items[3]/input[type=\"submit\"]"
```

Returning to the study, the longer the XPath, the greater the probability that the XPath is involved in a change. A change in the AUT may trigger other XPaths to be modified, beside the directly-interested ones. For instance, the addition or deletion of an item in a collection causes a correction not just to the pertinent XPath, quite also to the subsequent ones in the list that get shifted upwards or downwards. Imagine to delete the item at index 2 in the 'items' collection of the

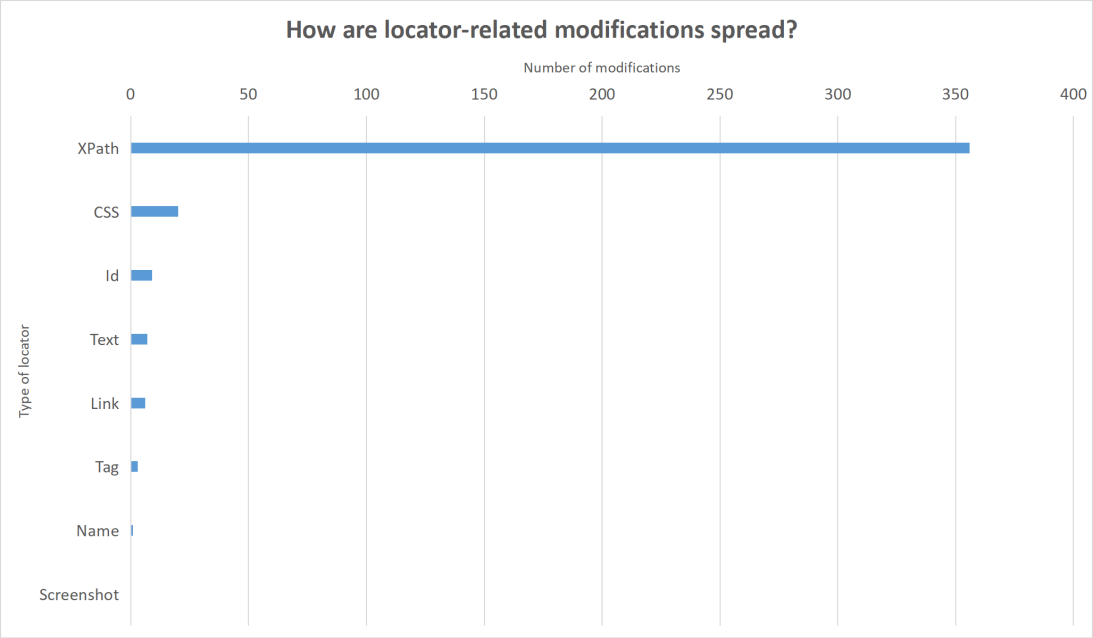


Figure 3.7: Data have been rolled up and categorized according to the locator type.

XPath shown above; the consequence is that the XPath would (its logical reference is incorrect), but other XPaths that eventually refer to any item at index greater than 3 would break as well.

XPath locators reveal their strong fragility given that 28% of the total modifications belong to this category alone, whereas their occurrence w.r.t. the locator-based modifications is 85%.

CSS (Cascading Style Sheet) locators identify one or multiple web elements through the properties defined in their 'style' attribute; these properties convey data about aesthetics of web elements. Here follows an example of CSS locator inside an assert statement; the locator should find an element with tag 'a' and 'lang' attribute equal to 'en':

```
1    verifyEquals(selenium.getText("css=a:lang(en)"), "this is the
     first element");
```

Figure 3.7 ranks CSS locators at second position (4.8%) w.r.t. the total number of modifications. Their modification frequency is however much lower than XPaths, like also the subsequent locators. This finding suggests that all the remaining kinds of locator, CSS included, would be robust. The shown data however also depend upon the preference of testers, which opt for a kind of locator rather than another

one for some reason. section 3.2 will clarify this point by using another kind of data source.

An id locator references a web element through its 'id' attribute. Such attribute is a plain string, like 'search-bar' or 'sort-feature'. It appears to be rarely modified by testers, since it represents just the 2.2% of the total modifications within the shown sample.

Text is a common attribute for web elements, since it is one of the most basic forms to represent data. Consequently, text locators are eligible to be a universal kind of locator. However, text locators are tightly bound to the presentation layer of the AUT, causing their great fragility. Figure 3.8 shows an example of modification on a text locator.

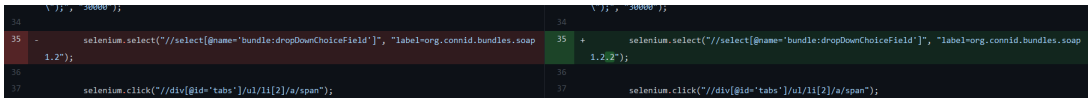


Figure 3.8: A slight modification of the text of the dropdown label (at the end of the string) causes the statement to fail.

At fifth position, with a percentage of 1.4%, there are modifications of link locators. This type of locator embraces hyperlinks of a web page, which typically convey the landing page of the link and the text that is shown to the user.

Tag locators refer to elements sharing the same tag, which in web applications define the nature of widgets across a page. A tag is a predefined keyword, so it can only assume a certain built-in static value. As such, tag locators are typically useful to select multiple elements in once. Their frequency of change is 0.7% w.r.t. the total modifications.

Name locators refer to an element through its 'name' attribute. This attribute must not be confused with the 'class' attribute used to assign a CSS style to the element. Name locators are typically employed in forms as unique identifiers, like the following statement demonstrates:

```
1    driver.findElement(By.name("password")).sendKeys("123");
```

Name attributes are occasionally modified, since just one occurrence has been recorded. Unlike web applications, desktop applications instead widely employ names to identify items; indeed, local applications have a GUI which in most cases is a composition of forms (a.k.a. windows).

No locator-related modification pertain screenshots, as far as the statistics report. This number is justified by the fact that the present study concentrates on GUI testing rather than visual testing, although it would have recorded such modifications in any case. Image recognition indeed requires additional modules

installed together with the testing tool in order to find an element, thought as data structure, given its appearance on the GUI.

Screenshots can also be employed to do assertions, as subsection 3.1.7 will underline better. In fact, asserting a visual statement does not require to infer the data structure associated to an element, quite just a sort of graphical matching capability.

3.1.3 Fragilities related to library restyling

The second most relevant type of modification, as shown by Figure 3.6, is about library restyling, namely the task of developers that adapt the library methods to their actual needs.

Although the task does not functionally affect test cases, the definition of fragility defined within the scope of this research embraces the effort to write such utility methods.

The task is performed in 5 test cases out of the 66 total maintained test cases (7.5%). The most relevant test case (94% of the restyle modifications) has been entirely re-ported to a corrected API of the testing tool. Mapping the whole existing tool API is famous to be a tedious task, which partially justifies why it is delivered so rarely. The remaining 4 test cases have got instead spurious mappings.

What encourages developers to write such functions comprises clearness, readability or convenience, as shown in Figure 3.9.

The task is strictly related to the extraction of functions or macros from frequent snippets, shown later on in subsection 3.1.6. The difference is that library restyling may take place ahead of need, while data generalization typically is issued when duplicated snippets start to appear.

An alternative motivation may consist of organizing a test case in logical parts, like steps of a use case, especially when the test is getting longer and longer.

The importance of underlining the category 'Methods restyling' is that it can be considered an intermediate form of POP. This property is in common with comments modifications, as discussed in subsection 3.1.4.

3.1.4 Fragilities related to comments

The third most relevant type of change pertains comments. They are spread at different scope levels, as shown in Figure 3.10.

Comments belong to the set of metadata of test files, like also code licenses. As such, they do not affect the functionality of test cases and so they do not concur directly in increasing fragility. However, since the present study treats test files as text documents, comments become relevant in the testing ecosystem.

```
137 +     protected void clickIncreaseFontSizeButton() {
138 +         selenium.click("inspector-button-font-size-increase");
139 +     }
140 +
141 +     protected void clickDecreaseFontSizeButton() {
142 +         selenium.click("inspector-button-font-size-decrease");
143 +     }
144 +
145 +     protected void clickToggleFontUnderlineButton() {
146 +         selenium.click("inspector-checkbox-font-underline");
147 +     }
148 +
149 +     protected void clickToggleFontItalicButton() {
150 +         selenium.click("inspector-checkbox-font-italic");
151 +     }
152 +
153 +     protected void clickToggleFontBoldButton() {
154 +         selenium.click("inspector-checkbox-font-bold");
155 +     }
```

Figure 3.9: Example of methods that wrap more fine-grained functionalities.

Testers sometimes prefer commenting code rather than deleting it directly. The deletion may be postponed to a future commit or it just works as a toggled functionality that the tester activates at will.

Comments sometimes reflect the need to summarize the user behaviour behind a certain snippet; this practice can be considered as a primitive POP (Page Object Pattern), explained in subsection 2.6.4. Indeed, comments related to a code block have the purpose to summarize the behaviour of that source code in order to speed up reading.

3.1.5 Fragilities related to licenses

Another kind of modification that testers apply regards licenses. They establish restrictions, terms and conditions to which the final user has to comply with [20]. License strategies depend upon organizational goals and, as such, they cannot be controlled. Anyway, from the practical point of view, a license is a kind of metadata that does not affect code in the flesh. Moreover, copyright declarations are out-of-scope within the present study, beside the fact that they anyway involve testers and test case files.

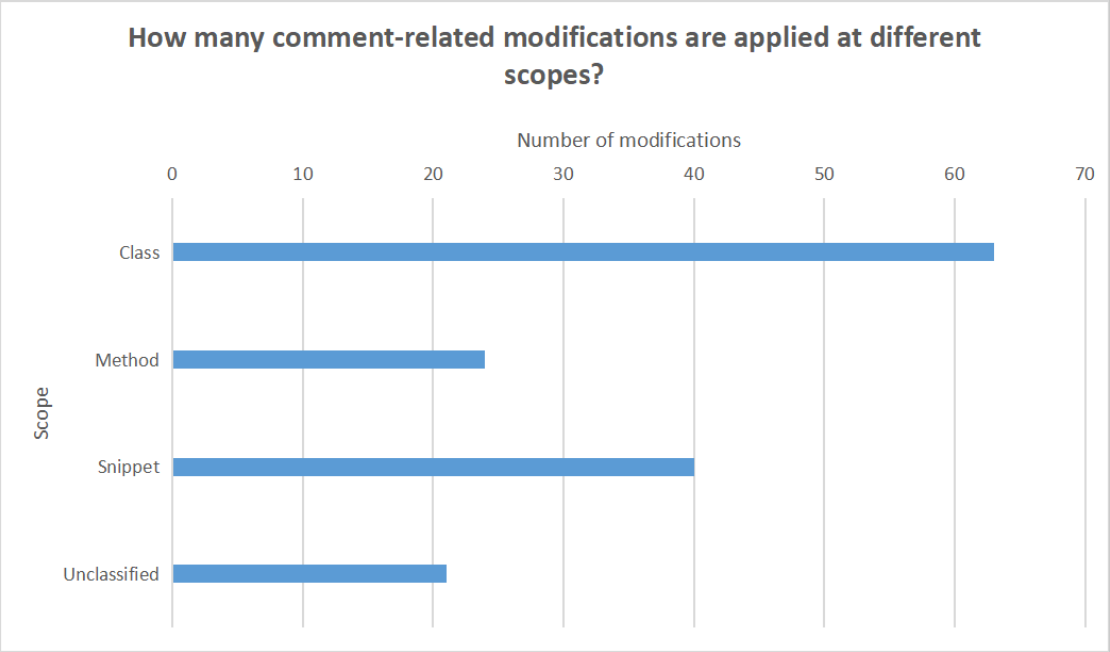


Figure 3.10: Comments are spread at different scopes.

3.1.6 Fragilities related to generalizations

As test cases become longer and longer, some snippets or literals repeat along code. Developers tend to extract a function or a macro respectively from frequent / duplicated snippets and constants. Extracting a code snippet means creating a function or constant that can be invoked or referenced by other statements that require the same behaviour or information. Figure 3.11 shows an example of extraction of a macro (that is constant) from a literal.

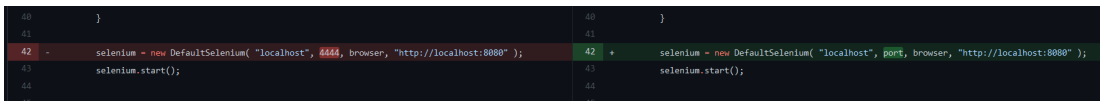
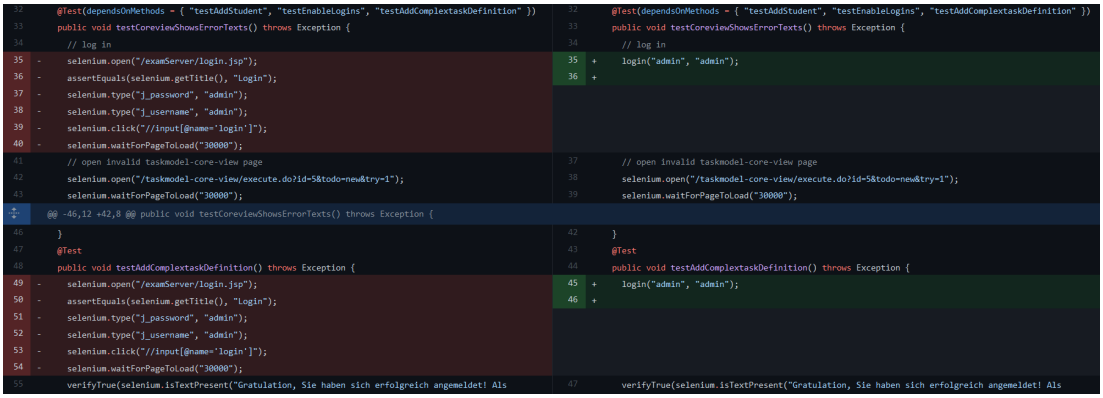


Figure 3.11: The literal 4444 gets generalized in the constant named 'port'. Now the constant can be re-used everywhere in the same source file.

Figure 3.12 instead gives a flavour of the extraction of a function (that is a method) from a duplicated snippet.

Figure 3.13 gives a quantitative overview on generalizations, grouped by target object which can be either a function or a constant. The chart also underlines that there are no other kinds of generalizations, within the sample recorded during the collection step.



```
@Test(dependsOnMethods = { "testAddStudent", "testEnableLogins", "testAddComplextaskDefinition" })
public void testCoreviewShowsErrorTexts() throws Exception {
  // log in
-   selenium.open("/examServer/login.jsp");
-   assertEquals(selenium.getTitle(), "Login");
-   selenium.type("j_password", "admin");
-   selenium.type("j_username", "admin");
-   selenium.click("//input[@name='login']");
-   selenium.waitForPageToLoad("30000");
+   login("admin", "admin");
  // open invalid taskmodel-core-view page
  selenium.open("/taskmodel-core-view/execute.do?id=5&todo=new&try=1");
  selenium.waitForPageToLoad("30000");
@@ -46,12 +42,8 @@ public void testCoreviewShowsErrorTexts() throws Exception {
}
@Test
public void testAddComplextaskDefinition() throws Exception {
-   selenium.open("/examServer/login.jsp");
-   assertEquals(selenium.getTitle(), "Login");
-   selenium.type("j_password", "admin");
-   selenium.type("j_username", "admin");
-   selenium.click("//input[@name='login']");
-   selenium.waitForPageToLoad("30000");
+   login("admin", "admin");
  verifyTrue(selenium.isTextPresent("Gratulation, Sie haben sich erfolgreich angemeldet! Als
```

Figure 3.12: The tester, manually or through the IDE, replaces the duplicated snippet with the invocation to a function called 'login'.

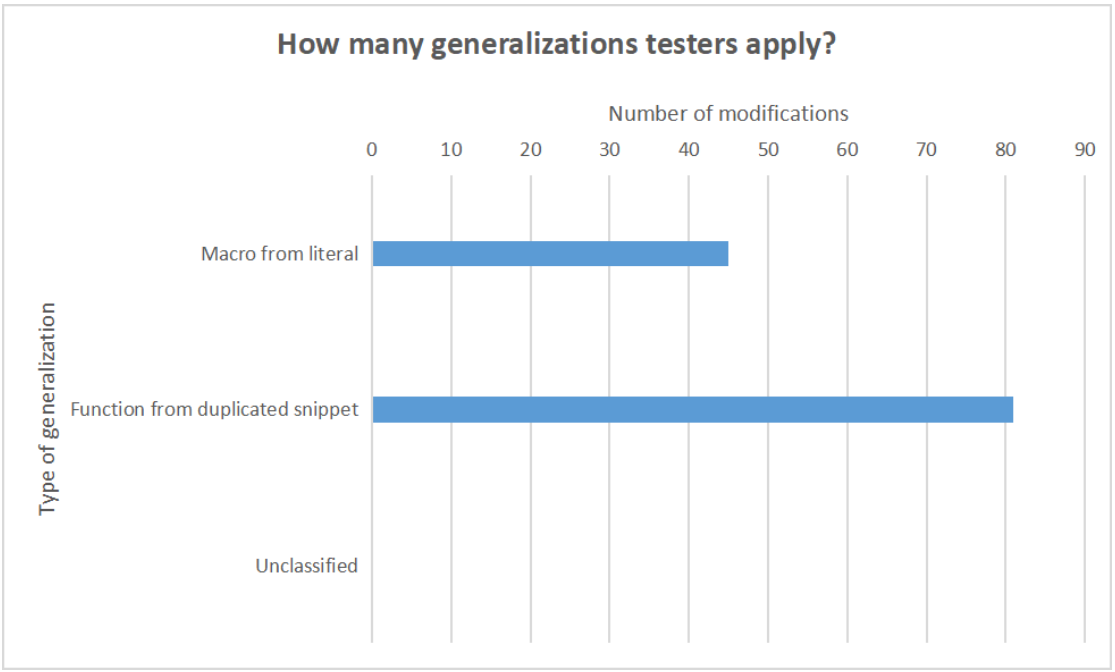


Figure 3.13: Testers apply two kinds of generalizations, especially when the test case gets longer and longer.

3.1.7 Fragilities related to assertions

One typical kind of statement in test cases is the assertion. Asserting two values means testing whether they satisfy a certain relation, often equality. The nature of an assertion is determined by the type of values to compare. For instance, the following instruction asserts that the current web page has the same appearance of

the 'login.png' image:

```
1    Assert.assertTrue(SeleniumUtils.verifyScreenshot(driver, "login.
     png"));
```

Instead, the assertion on text can take this form, where no image recognition is employed:

```
1    Assert.assertTrue(SeleniumUtils.isTextInInput(driver, "first_name
     ", "Amber"));
```

Figure 3.14 shows the frequency of modifications related to assertions.

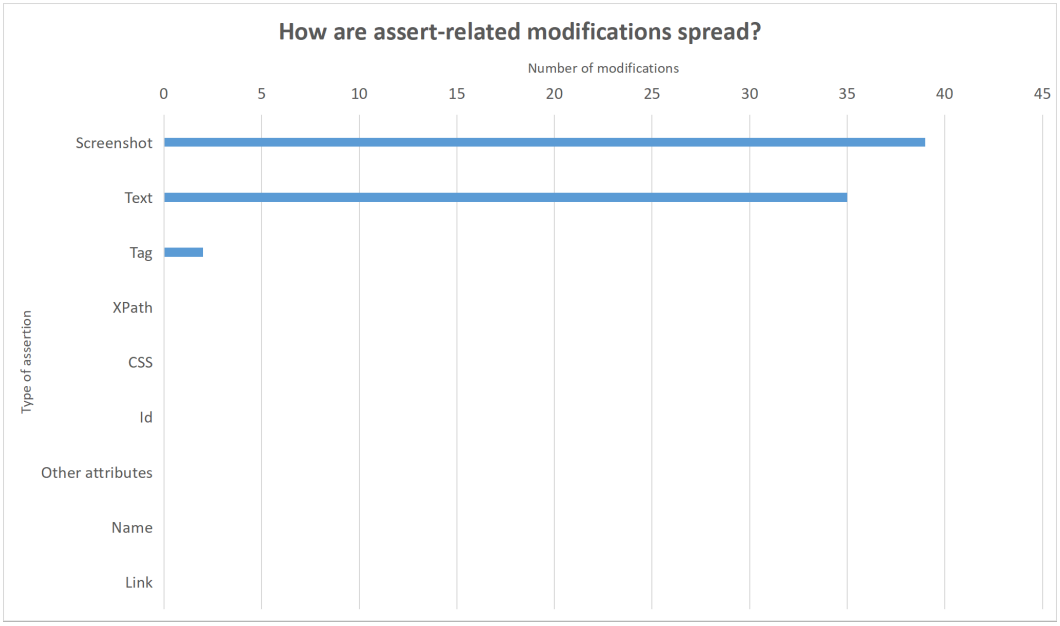


Figure 3.14: The chart is the result of filtering and cleaning assertion modifications recorded from diff files.

The two most modified kinds of assertions pertain screenshots and text. The other attributes are instead typically used to locate an element rather than to assert on themselves.

3.1.8 Fragilities related to wait strategies

Another traditionally-relevant issue that affect GUI test suites is synchronization. The purpose of such feature is to coordinate the test case execution with the

pertinent state of the AUT, since it is not automatically enforced neither by the AUT nor by the test tool. For instance, given the statement:

```
driver.getElementById("nav").click()
```

It doesn't implicitly wait that the target element has been rendered or even attached to the DOM, before clicking. This lack of synchronization make test cases dependent upon the infrastructural delays, which are in turn affected by the load of the host machine. As a consequence, testers add wait statements between visual actions; the associated test case, therefore, consistently interact with web elements that now are in place at the right moment.

As shown in Figure 3.15, there are several wait strategies that testers apply.

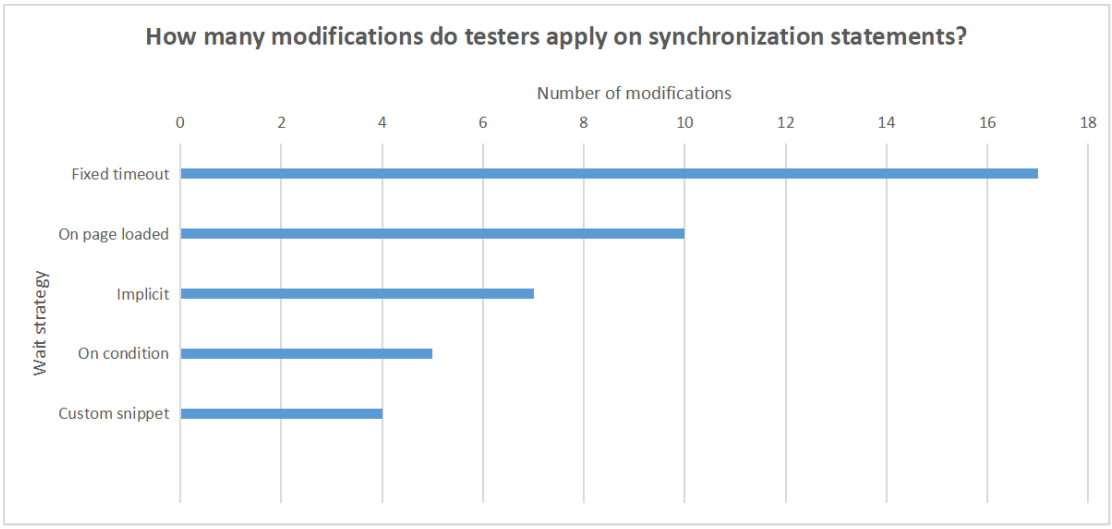


Figure 3.15: Testers can synchronize test statements with the AUT through different kinds of strategies.

Here follows a description of the aforementioned wait strategies:

Fixed timeout means that the wait statement makes the execution of the test case suspend for a certain number of milliseconds; Figure 3.15 ranks this policy at first position, in accordance with successive results of this study. Here is an example of fixed timeout statement in Java, where the timeout is 3000 ms:

```
Thread.sleep(3000);
```

On page loaded is useful to wait that the switch operation from the current tab of the web page where the test case is working to another one has delivered. This statement is an example, with timeout equal to 30000 ms:

```
selenium.waitForPageToLoad("30000");
```

Implicit automatically adds a fixed timeout wait before each GUI test statement starts acting, like an aspect of AOP programming. The following statement, for instance, sets a default wait timeout of 10 seconds:

```
driver.manage().timeouts()
    .implicitlyWait(10, TimeUnit.SECONDS);
```

On condition enables the test flow to sleep until a certain condition has been met; this kind of wait policy will come up again in the second data collection process of the present thesis (see subsection 3.2.2). Here is a common example, where the test case does not continue until the element denoted by the XPath has become visible:

```
wait.until(
    ExpectedConditions.visibilityOfElementLocated(
        By.xpath("//div[contains(text(),'COMPOSE')]")
    )
);
```

Custom snippet is a tailored wait approach, that the tester develops on purpose. The following custom wait snippet polls every second if the element denoted by the XPath has text equal to "3"; if after a minute the text has not appeared yet, the test case fails:

```
for (int second = 0; ; second++) {
    if (second >= 60) SeleneseTestBase.fail("timeout");
    try {
        if ("3".equals(selenium.getText("//tbody[@id='table-
management']/tr[2]/td[5]"))) break;
    } catch (Exception e) {
    }
    Thread.sleep(1000);
}

```

Recommendation R.D.0

This section establishes a recommendation that stemmed from subsection 3.1.8. The meaning of 'recommendation' is described later in subsection 3.2.2, since that research phase has found many more recommendations.

Source: subsection 3.1.8

Waiting for a fixed timeout increases fragility of test cases. Developers typically tune the timeout:

- By increasing it: the reason is that the last execution of the test case threw a NotFound-like exception, because the waiting time for an element to appear was too short;
- By decreasing it: the developer tries to gain execution time from tests; indeed, GUI tests in general need much time to deliver.

The preceding two countermeasures are weak. As an alternative solution, the developer can turn the target fixed-time wait into a condition-based wait. Indeed, the latter kind of wait allows the test case to continue as soon as the condition is met, without remaining idle until the timeout has expired. On the other hand, waiting on a condition never fails due to lacking elements on the GUI, and this fact is deterministic.

Skimming the diff files from GitHub, it turns out that a fixed-time wait typically assumes the form of a 'Thread.sleep' call in Java and 'setTimeout' call in JavaScript.

Contract: Turn fixed-time waits into condition-based waits.

State: implemented

3.1.9 Discussion

Although performing data mining by collecting test cases modifications along their history is a reasonable and direct source of information, other channels have been considered. Indeed, developers feel the implicit need to establish rules that hopefully guarantee stronger robustness and try to follow them. These rules can be still found in the wild, but under the form of wikis (as called on GitHub) rather than code. Additionally, authoritative organizations like Google, the Selenium team, Node.js and so on have published guidelines on the Net with the same purpose over the years. Taking into account what developers experienced about fragility so that to write down their own guidelines enrich the range of rules that a tool is capable to enforce or suggest; the developer would rely upon a tool that summarizes the knowledge of hundreds of peers from the wild.

A more formal justification of the motivation that encouraged this study to search for other sources of information is that data collection based on statistics is

a quantitative approach. Nowadays automating data analysis through algorithms is catching on a wider and wider audience. Nevertheless, experts of big data are at the same time more convinced that these techniques, like association rule recognition, cannot reveal every facet of the truth starting only from numbers. To prove this, it's enough to observe that even firms that analyze data coming from social networks or mobile cells still leverage the traditional focus groups to shape their product, which in turn is a qualitative approach. Here going deep into these topics is not of interest, but it is still important to highlight what are the advantages of both quantitative and qualitative techniques: while the former gives a general and objective overview, the latter obtains hands-on information directly from end users. In fact, the number of recommendations found through wikis is much greater than that stemmed up from diff files. As a consequence, combining both techniques has been deemed more profitable for the purpose of this study.

3.2 Data collection from wikis

A wiki is here a statement or a group of statements, sourced by a member or a group on the online community; a wiki is worded in plain language. The modern search engines facilitate the work of finding pertinent wikis, both on GitHub through vertical searching and on the wide Net through horizontal searching.

3.2.1 Methodology

Regarding the methodology that has been followed to find and record recommendations, it is identical to that one described in subsection 3.1.1, applied for diff files. The only difference is that, rather than opening the 'Code' tab, the selected tab here is 'Wiki'. In this case, the search net has been widened to the whole Net.

No statistics have been gathered in this collection process, since it wants to get in touch with practical findings about GUI testing. The result of the process is indeed a series of recommendations, shown in subsection 3.2.2.

Cleaning and filtering wikis

Wikis, as found in the wild, appear in a raw format. Therefore, this section describes the task of formalizing the wikis and normalizing them in a common format.

The language in which these wikis are written is not well-finished: there are sentences with no independent clause. This issue triggers a work of textual analysis to shed light on what is the real message of a wiki. Additionally, another essential task have been carried out and that commonly goes by the name of internal consistency check.

Since multiple rules may declare opposite statements to each other, they have been checked also against external consistency.

Regarding the methodology, recommendations have been evaluated incrementally in order to easier the process: given the corpus of already-scanned and consistent recommendations, a further rule is compared with them and admitted in case of coherence.

The intermediate result of the analysis process has been filtered out by fragility-pertinent rules: guidelines that establish a recommendation just for the sake of clearness, for instance, have been dropped. Instead, those good practices that enforce clearness or other properties for the name of fragility have been kept.

The result of the collection and filtering process is presented in subsection 3.2.2.

3.2.2 Recommendations from the wikis

The final result of the process is a set of recommendations, as follows. Each recommendation contains a summary of the source text which, in its original form, normally covers more ample topics. The rule that the developer should follow is labeled as the contract that the rule recommends. It may follow a further discussion that solves or explains some issues stemming from the summary of the contract. Finally, a couple of headers, 'state' and 'reason', summarize whether the rule has been implemented and, eventually, why not.

First of all, the UML dictionary shown in Figure 3.16 establishes the definition of recommendation within this study.

A recommendation may have multiple reasons (Motivation) to exist, each one traced back to a more or less authoritative source. The more motivations, the stronger is the recommendation. A recommendation may be related to another one. A given code snippet (Snippet) may match a given pattern (Recommendation::pattern) for a certain reason (Diagnostic:reason). For instance, the reason could be 'Use of setTimeout function'.

Table 3.1 lists the recommendations and, for each one, its state of implementation. Indeed, as discussed in chapter 4, testers can follow these rules with the aid of a tool; the tool do not implement all the rules, for different reasons like complex heuristics or huge implementation effort. Out of 25 recommendations, 23 are eligible to be implemented, since those labeled as 'Not needed' are aggregations of other ones. Among the 23 remaining recommendations, 15 are implemented (65%), excluding those partially enforced.

Recommendations have ids that are not strictly incremental; for instance, the possible id 'R.W.2' is not used. The motivation is that the list of recommendations is a living document, where each row may get merged with other rules or declared part of another one; compacting ids would be confusing: keeping the same id for a rule is clearer in the long run.

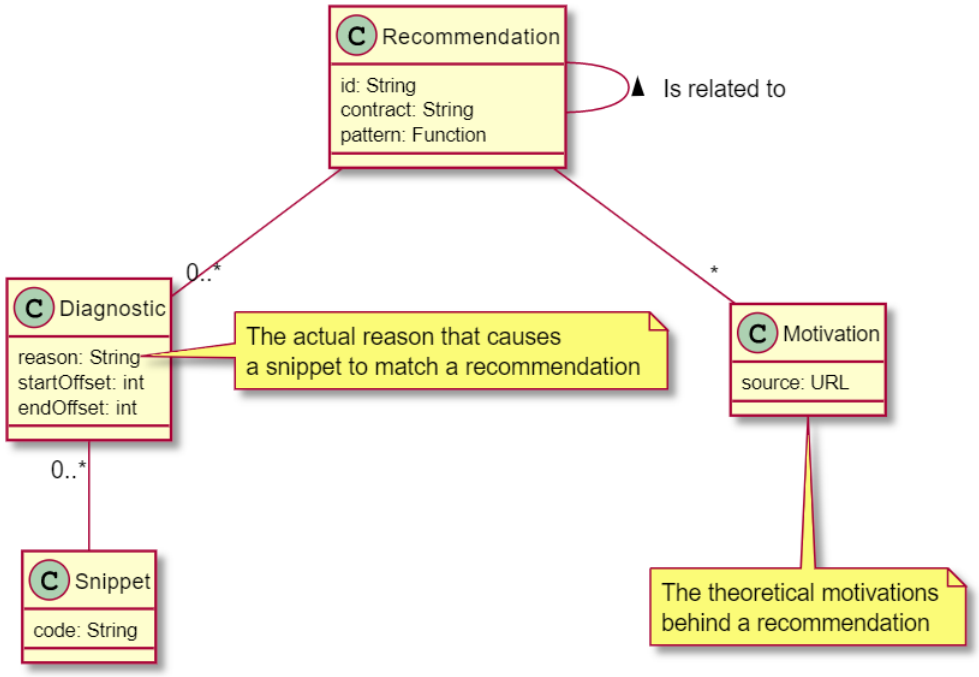


Figure 3.16: What a recommendation is within this study.

Recommendation R.W.0

Summary: the testing pyramid (Figure 3.17) [21] shows the recommended proportion in the number of end-to-end test cases w.r.t. lower-level tests. The reason is that, moving upwards in the pyramid test plan, there are issues like:

- Test fragility (tests that break easily and unexpectedly, even when changes shouldn't have influenced the test);
- Longer feedback time;
- Increased effort levels;
- Higher costs to implementation;
- More specialized knowledge required.

Contract: keep the number of unit tests greater than the number of end-to-end tests.

State: non implemented.

Recommendation id	State	Type of faced fragility
R.D.0	Implemented	Synchronization
R.W.0	Non implemented	Effort
R.W.1	Implemented	Data
R.W.3	Implemented	Data
R.W.4	Non implemented	Cognitive
R.W.5	Non implemented	Cognitive
R.W.6	Implemented	Cognitive
R.W.7	Implemented	Effort
R.W.12	Not needed	/
R.W.12.0	Non implemented	Effort
R.W.12.1	Implemented	Effort / psychological
R.W.12.2	Non implemented	Effort / cognitive
R.W.12.3	Implicitly implemented	Effort
R.W.12.4	Implemented	Data
R.W.12.5	Partially implemented	Data
R.W.12.6	Implemented	Data
R.W.12.7	Implemented	Execution plan
R.W.13	Not needed	/
R.W.14	Implemented	Effort / cognitive
R.W.15	Non implemented	Effort
R.W.16	Implemented	Effort
R.W.17	Non implemented	Execution plan
R.W.19	Implemented	Data
R.W.20	Implemented	Data
R.W.21	Implemented	Effort

Table 3.1: The table summarizes the list of recommendations formalized within this study, along with their state of implementation.

Reason: projects typically store GUI tests in separate folders than other tests; additionally, folders inside a project often does not follow a standard hierarchy, as well as names of folders are arbitrary.

See also: R.W.12.0

Recommendation R.W.1

Summary: XPath locators relative to an element found by id come up to be more robust than absolute ones: for instance, //*[@id="fox"]/a. [22]

Contract: use relative XPath locators in place of absolute XPath locators.

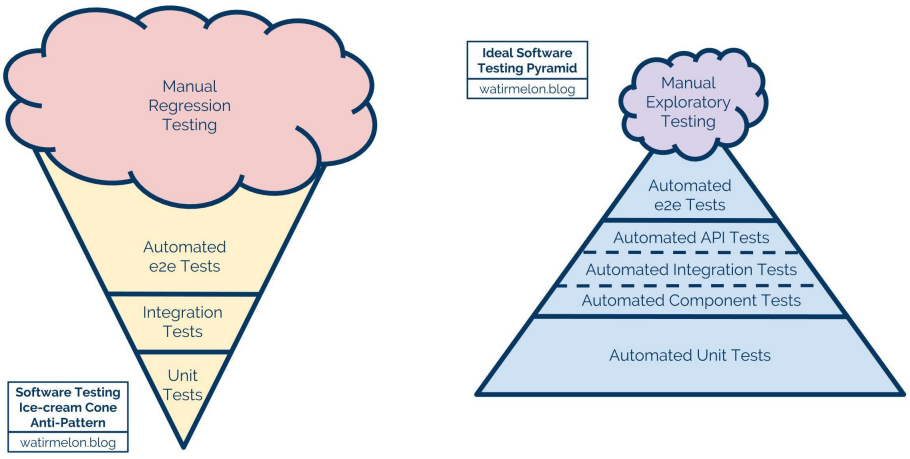


Figure 3.17: On the left, the ice-cream test plan. On the right, the pyramid test plan.

State: implemented.

Recommendation R.W.3

Summary: Id locators lead to high readability. They are the fastest locators, since they are implemented by calling the 'getElementById' method, at browser level. [22]

Additionally, predictable locators by id help in writing tests for dynamically-populated lists, whose tests are typically hard to maintain.[23]

XPath locators are more vulnerable to UI changes than ids, fact that augments test maintenance. XPath locators are slow and so they may break test cases that make use of timeouts, increasing their fragility. [23]

Id locators are the most robust choice; when they are not available, CSS locators should be selected; as last resort, XPaths can be chosen. [24]

Contract: prefer locators by id.

Discussion: it is natural wondering about why locators by id are more robust than their XPath and CSS counterparts. A justification for this behaviour could be that CSS identifiers, beside being placeholders, also encode a certain style attribute; therefore, modifying them is more likely than ids which don't carry any other meaning for the AUT. The same reason subsists for XPaths: beside identifying an element, they are bound to the structure of the DOM, whereas ids are transparent to any structural and presentational change.

State: implemented.

Recommendation R.W.4

Summary: ids and names of elements should reflect their functional purpose so as to lower the probability they get changed. Ids must be meaningful and should not convey a presentational purpose. Additionally, they would be more readable. If an element is not directly involved in a use case, like containers, their ids or names should be generic.[25]

Contract: give to an element an id that mirrors its functional purpose. When an element has no particular meaning, give it a generic id.

State: non implemented Reason: this rule requires natural language processing to put in place some heuristics.

See also: R.W.5

Recommendation R.W.5

Summary: no discussion is present in the source wiki.[25]

Contract: give to elements a name that mirrors their functional purpose. When an element has no particular meaning, give it a generic name.

State: non implemented Reason: this rule requires natural language processing to put in place some heuristics.

See also: R.W.4

Recommendation R.W.6

Summary: variable names concur in filling up the working memory of the developer, which is quite limited. As a consequence, their meaning must be immediate and clear. [26]

Contract: keep names of variables clear to everyone.

State: implemented.

Recommendation R.W.7

Summary: do not concatenate words and abbreviations in selectors by any characters (including none at all) other than hyphens, in order to improve understanding and skimming. [25]

Contract: separate words in ids and class names by a hyphen.

State: implemented.

Recommendation R.W.12

Discussion: Node.js recommendations are curated and edited by the respective authors and by the Node community, so they did not require neither cleaning effort

nor additional analysis. Each Node.js best practice is sometimes further explained in a web page apart, which is not reported here for the sake of brevity.

State: not needed Reason: this is an aggregation rule.

Recommendation R.W.12.0

Summary: integration tests must be developed before unit tests along the lifecycle of a test suite, due to their wider coverage applying a given effort. Unit tests, instead, due to their narrow coverage given the same effort, lead teams to abandon automatic testing, especially at the beginning of the project. [27]

Contract: first write end-to-end tests, then unit tests.

Discussion: this rule seems to struggle with section 3.2.2, but it does not. The reason is that section 3.2.2 comes into play when the project development is at steady state; instead, the current rule acts at the very beginning of the project development. This rule has to do with fragility because, if not respected, may cause the test team to delete a subset of the test cases, which is a modification.

State: non implemented Reason: projects typically store GUI tests in separate folders than other tests; additionally, folders inside a project often does not follow a standard hierarchy, as well as names of folders are arbitrary.

See also: R.W.0

Recommendation R.W.12.1

Summary: if the name of a test case doesn't describe the starting scenario, the developer should infer it by reading the function's body. Additionally, clearly stating what's the expected output in advance helps in overcoming the psychological bias that may lead the tester to define by mistake an assertion so as to make the test find no bugs. [27]

Contract: give test cases a name with three sections: what is being tested, under which circumstances and what's the expected result.

Discussion: when testers write a new test case or when the test suite is getting big, they try to establish a rule to name test cases in a clear and consistent way; this task implies modifications in the test cases and in particular in their name. This recommendation establishes an effective naming rule from the very beginning, avoiding subsequent changes.

State: implemented.

Recommendation R.W.12.2

Summary: arranging each test case in a uniform way saves effort since the developer's mind recognizes the same pattern along the test suite. Saving effort, in turn, decreases the probability that the test case gets abandoned. [27], [28]

Contract: arrange each test case in three successive sections: setup, act and assert.

State: non implemented Reason: heuristics of this rule are numerous and none of them is compulsory. This recommendation is therefore complex to recognize.

Recommendation R.W.12.3

Summary: linters help in recognizing anti-patterns early. Run them before any test and before the commit so as to minimize the time needed to review code issues. The recommendations of this document may be indirectly enforced by linting the code against other types of good practices. [27]

Contract: run linters to detect any anti-pattern.

Discussion: the exact point in the build process where linters should be activated is not established as part of the contract.

State: implicitly implemented Reason: this rule is implicitly enforced.

Recommendation R.W.12.4

Summary: test cases that rely upon global variables are fragile. Indeed, these can be changed unexpectedly due to their wide scope. The issue is even worse in JavaScript, where their scope may be the whole project. [29]

Contract: do not use global variables in test cases.

Discussion: the rule embraces also global constants, since they may be changed statically by the programmer causing unexpected errors.

State: implemented

Recommendation R.W.12.5

Summary: test cases must not access the same data in the test DB. This avoids that tests break because they share data. [27], [30]

Contract: devote separate DB data to each test case.

State: partially implemented Reason: the linter can indirectly recognize this rule, but a complete heuristic recognition is possible only if the linter recognizes databases, which is out of the scope of the linter.

Recommendation R.W.12.6

Summary: creating a new web driver instance per each test case ensures test isolation and makes parallelization more easy. [30], [31]

Contract: create a separate web driver per each test case.

Discussion: creating and configuring a separate web driver instance per test case surely leads a huge of duplicated code. However, this drawback is mitigated thanks to a couple of observations:

- The linter is running on an IDE: as already done in other parts of the present study, one hypothesis is that the developer is working with an IDE, which in turn most likely has a replace feature to quickly modify snippets that are identical;
- The recommendation forbids the test suite fixture methods, while those applied around each test case are allowed: test frameworks typically offer a couple of methods, like 'beforeEach' and 'afterEach', that act as fixture and cleanup methods respectively, executed according to the Aspect Programming [32].

State: implemented

Recommendation R.W.12.7

Summary: running only a subset of the test cases save effort. Tagging tests having a common attribute and running only those tests lower the test run time. [27]

Contract: tag related test cases in order to run just a subset of them.

State: implemented.

Recommendation R.W.13

Summary: a project is said rigid when a modification takes more and more effort to be applied since it implies the correction of many other consequent issues. This is a typical scenario when modules are strongly coupled. Fragility grows when a project is rigid. A developer may choose to avoid applying the modification from the start, but it is not always possible: in this situation test cases and the AUT break, showing up a relevant set of bugs and errors. Keeping test cases decoupled from each other and from the context reduces fragility against any type of modification. [33]

Contract: keep test cases decoupled from each other, under every point of view.

Discussion: this rule is actually the general case of other recommendations, each one taking care of a specific point of view.

State: not needed Reason: this is an aggregation rule.

See also: R.W.12.4, R.W.12.5, R.W.17

Recommendation R.W.14

Summary: sections on data setup, actions and assertions must be as short as possible. This approach minimizes the fragility of test cases. Long test cases

instead are expensive to run and poorly debuggable since the fault is more difficult to trace back. Moreover, the developer has difficulty to follow the whole plan of a test case due to working memory limitations. [28], [26]

Contract: keep test cases as short as possible.

State: implemented

See also: R.W.12.2

Recommendation R.W.15

Summary: the test setup should not perform visual actions; the scenario must instead be initialized by calling APIs and performing DB queries that the AUT exposes. [28], [34]

Contract: do not perform visual actions to setup the test case scenario. Instead, use APIs of the AUT and direct DB queries.

State: non implemented Reason: locating the setup phase of a test is complex due to the wide range of possible statements, most of which are optional.

See also: R.W.12.2

Recommendation R.W.16

Summary: third-party libraries and services decrease the stability of tests. [35]

Contract: minimize the number of external libraries.

Discussion: external modules increase the wall-clock time of a test case to deliver, which is a critical issue in GUI testing.

State: implemented

Recommendation R.W.17

Summary: no test case should continue the workflow of other tests; instead, when the tester decides to split a use case in many test cases, each test case but the first must rather stub the preceding scenario with a proper test setup. [36]

Contract: test cases must not directly continue the workflow of other test cases.

State: non implemented Reason: grasping whether a certain test case is the continuation of a previous test is complex.

See also: R.W.13

Recommendation R.W.19

Summary: link locators only work on link elements. Moreover, they are translated to XPath selectors under the hood, so they inherit all their drawbacks. [24]

Contract: do not use link locators.

State: implemented

See also: R.W.3

Recommendation R.W.20

Summary: locators by tag are unpredictable in case multiple elements are admissible to be picked up but the target is just a specific one. They instead are useful when the goal is selecting multiple elements. [24]

Contract: use tag locators to pick up multiple elements.

State: implemented.

Recommendation R.W.21

Test cases should adopt the Page Object Pattern, in order to decouple the test behaviour from the underlying implementation. In most cases, one or two operations per section (data setup, actions or assertion sections, n.d.r.) are enough. Test cases compliant with the POP must not contain any visual statement; they should contain assertions. Page objects should contain visual statements; they should not contain any assertion, beside those for checking that the page has loaded. [28]

Contract: adopt the Page Object Pattern.

State: implemented

3.3 Data analysis

After having collected and formalized data from projects and wikis, it comes natural to build a taxonomy on the different kinds of fragility encountered so far.

In terms of methodology, the taxonomy has therefore been built a-posteriori. According to some studies, one approach to produce a taxonomy is to delay as much as possible the act of reading the related and preceding works on the same subject. [17]. The following taxonomy has been built straining for this methodology on the basis of the collected data. However, for the sake of completeness, it is necessary to draw fully from the results of other studies. In final analysis, the classification stems both from a-priori considerations, data collections on the wild and other studies.

Figure 3.18 summarizes the different natures of fragility.

Fragility can be classified according to its nature:

- Data fragility: test cases are run on some input test data, whose eventual modification automatically affects tests;
- Execution plan fragility: the execution plan of a test case is the order and the dependencies with which a certain subset of a test suite is run;

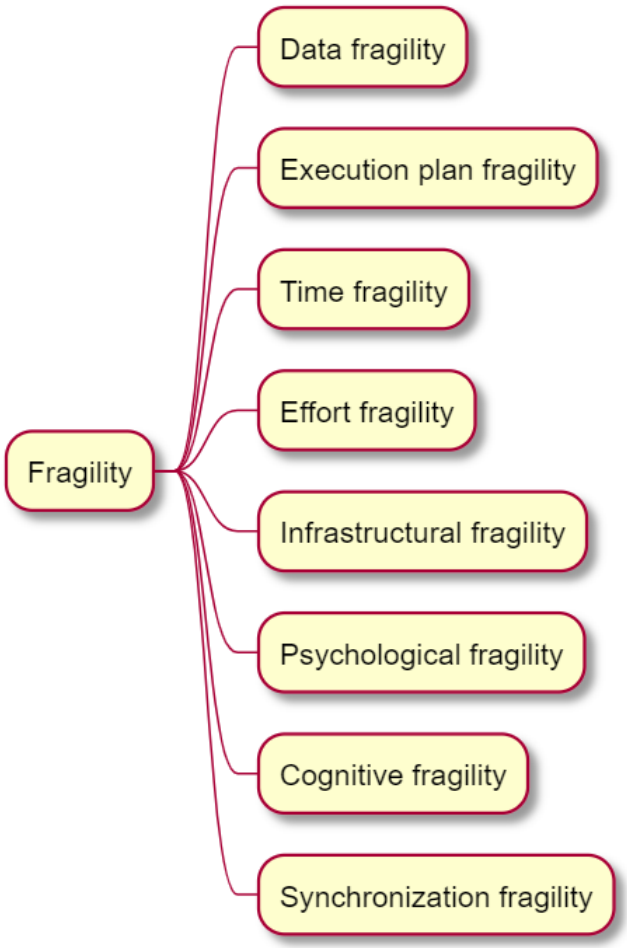


Figure 3.18: Mindmap that enumerates the various causes of fragility.

- Time fragility: the test may depend upon the circumstantial time it is executed. Input data like time and dates may produce breakages, especially when compared to the current timestamp;
- Effort fragility: the effort needed to read/correct/run a test case may induce the programmer to remove or comment it. The more a test case is fragile, the more breakages come out, which in turn imply a greater effort for fixing them;
- Infrastructural fragility: a breakage may occur due to random delays in the network or in the host machine; the symptoms include unexpected test case failure that typically disappears if the test case gets run multiple times. In

other words, test case execution is not idempotent;

- Psychological: anxiety leads testers to write test cases that work, rather than trying to build tests that discover new bugs. GUI testing is a repetitive task and therefore tedious, fact that lead developers to lose attention; as a result, they unintentionally inject faults while writing test cases, for instance by copying and pasting code snippets;
- Cognitive fragility: the developer has limitations in terms of working memory [26]. This kind of fragility deals with physical limitations of human beings, whereas psychological fragility interests feelings and thoughts;
- Synchronization fragility: test statements are not automatically coordinated with the execution of the AUT, resulting in breakages due to statements that seek for components not yet loaded.

Chapter 4

Tool design

As discussed in section 2.6, there are three kinds of countermeasures to reduce fragility. Although their separate effectiveness is different, the three techniques are not mutually-exclusive: a project can benefit from all them. For instance, a robustness-like tool can suggest more sturdy developing patterns, whereas a repairing tool can come into play whenever the same proposed pattern has broken; at the same time, testers can adopt an antifragile business model to encourage change and innovation. This observation is the justification that led the present study to develop a robust-like tool rather than a repairing tool which is more promising in terms of resulting quality. The tool is independent from other techniques but, simultaneously, it still works with them.

The task adopted for data collection, discussed in chapter 3, suggests to develop a static code analyzer that probes bad practices in test suites and propose more robust patterns to testers.

The first architecture that has been evaluated for the development of the tool is based on Maven plugins. Maven is a free product that organizes the different steps of the build process in a coherent and complete lifecycle; it also allows programmers to write plugins that extend the basic building features for precise build steps. However, a Maven plugin would inform the tester about fragility issues and their location in a separate file, like an XML report; the consultation of such a report would force the test developer to continuously switch between the test sources and the report.

Another infrastructure that has been evaluated to develop the code analyzer is based on Sonar plugins. Sonar is a static code analyzer that, given a project, searches for code smells, bugs and vulnerabilities; it additionally has extension capabilities to customize the underlying rules matched against the code. Unfortunately, Sonar doesn't provide an immediate result, so the test developer typically runs it only a-posteriori, when the code has already been written.

The alternative solution, which is also the adopted one, is writing an extension

for an IDE. After all, most developers nowadays leverage an IDE to speed up the editing tasks; these programs, however, are designed to be extended so as to support a wider variety of activities. Indeed, IDEs already offer several utilities for the most common actions within the development environment. The code analyzer therefore takes the shape of a linter. The additional benefit of leveraging this kind of infrastructure is that the linter acts while the developer is working and shows clearly where the bad practice is located by highlighting it.

The choice about the infrastructure of the linter is summarized and modeled by the deployment diagram shown in Figure 4.1.

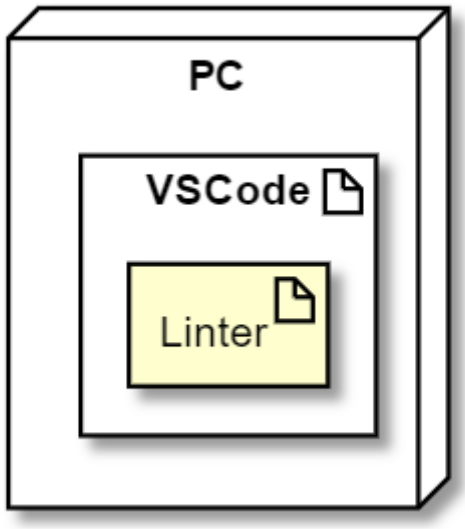


Figure 4.1: UML deployment diagram

VSCode is a famous IDE at the moment, whose extension capabilities are regularly upgraded and maintained by Microsoft. The related official documentation is also well-finished and the community is active and participating. These factors facilitate a lot the development of an extension, as chosen for this study. The tool, called FragilityLinter, is written in Javascript on top of the Node.js framework, as required by the VSCode IDE.

The tool can be further classified as a static predictor, since it suggests an alternative to the choice of the developer based on past modifications happened in a range of projects. It is not a dynamic predictor because it does not scan the project's GitHub history.

After having located the environment where FragilityLint works, here follows the design thereof.

As shown in the UML class diagram, the linter architecture is scattered along four packages: a graphical tier, a logical tier, a data tier and the external dependencies. The architecture follows the three-tier pattern due to the presence of the graphical, logical and data tiers and their specific interconnection.

The data tier is in this case a simple static array of recommendations; additionally, this makes the tool stateless, since no data is stored and retrieved dynamically.

There has been a strain to keep modules as decoupled as possible: the recommendation module is not aware that it is called during a parsing process; however, the way the core module probes bad practices is hardwired in the module itself, due to parsing concerns. Indeed, the Java language is more constrained than Javascript overall: for instance, a Java routine is never a function quite always a method inside a class; this fact shaped the module interconnection.

The tool leverages two external free libraries, Acorn and Java-Parser, that can be downloaded as node modules (see [37] for information about node modules). They parse a given string in tree data structures, which are then traversed by the Core module. Acorn accepts Javascript code and produces an Abstract Syntax Tree, while JavaParser receives Java code as input and returns a Concrete Syntax Tree. Formally, the Core module is a semantic analyzer, whereas Acorn and JavaParser are parsers; in this study, however, the two concepts are interchangeable.

It has been chosen to leverage a separate parser for each language rather than only one along with a Java-to-Javascript transpiler. The translation indeed distorts information about the location of the bad practice.

Figure 4.3, Figure 4.4 and Figure 4.5 show the linter working in practice.

The design goals of the linter can be classified and summed up according to this listing:

- Performance: the tool works while the developer is coding, unlike TestQ or, overall, code-review tools. The latter ones indeed are well-known for the time they need to deliver and, indeed, code review is a separate task. Moreover, since the linter accesses the document to scan in read-only mode, it may diagnose each recommendation in a parallel job in order to increase the performance significantly. However, FragilityLint is written in JavaScript, a language that is inherently single-threaded; as a consequence, parallel programming cannot be fully exploited. Anyway, overall performance of the tool remain fast enough when scanning a single file during development, if the host machine belongs to a mid-range category price at the moment where the present study has been written;

- Decoupling: this property is considered essential in software engineering, as decoupled modules benefit from high maintainability and flexibility, along with enhanced readability; the various modules are shown in Figure 4.2;

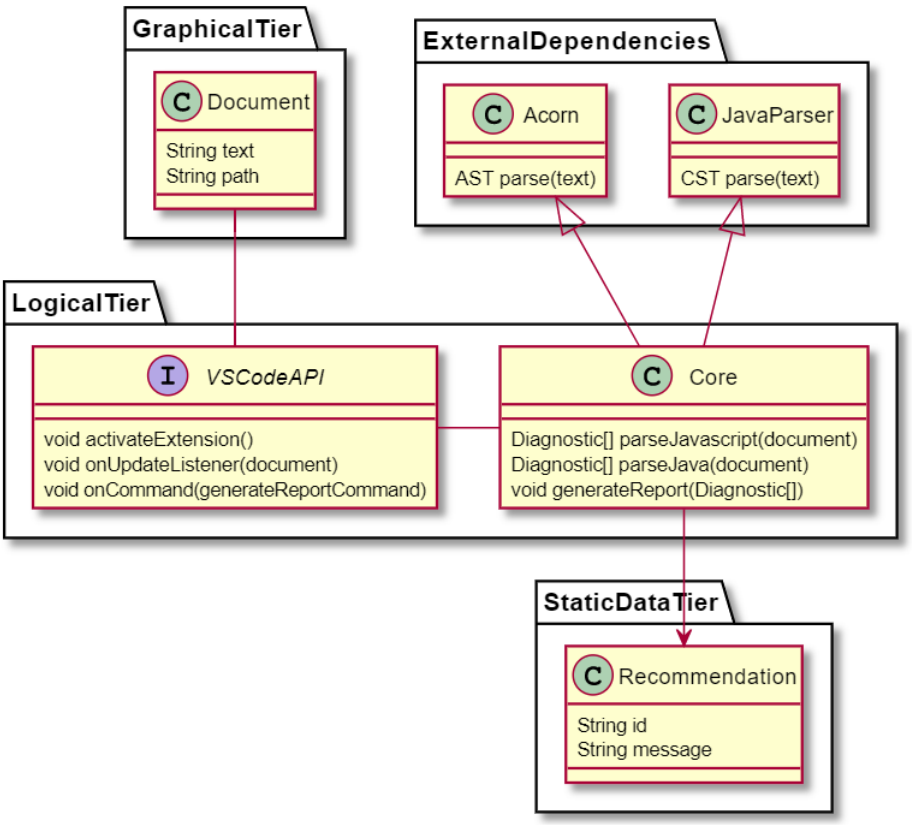


Figure 4.2: UML class diagram

- Ready-to-use: no configuration is needed for the tool to work, so as to free the tester from looking after another step in the build process; in future, the linter may however expose properties to disable some rules from being probed; for other possible future improvements, see chapter 6;
- Transparent to the test suite: the linter is linked to the development environment only, since it does not modify in any way the scanned test cases.

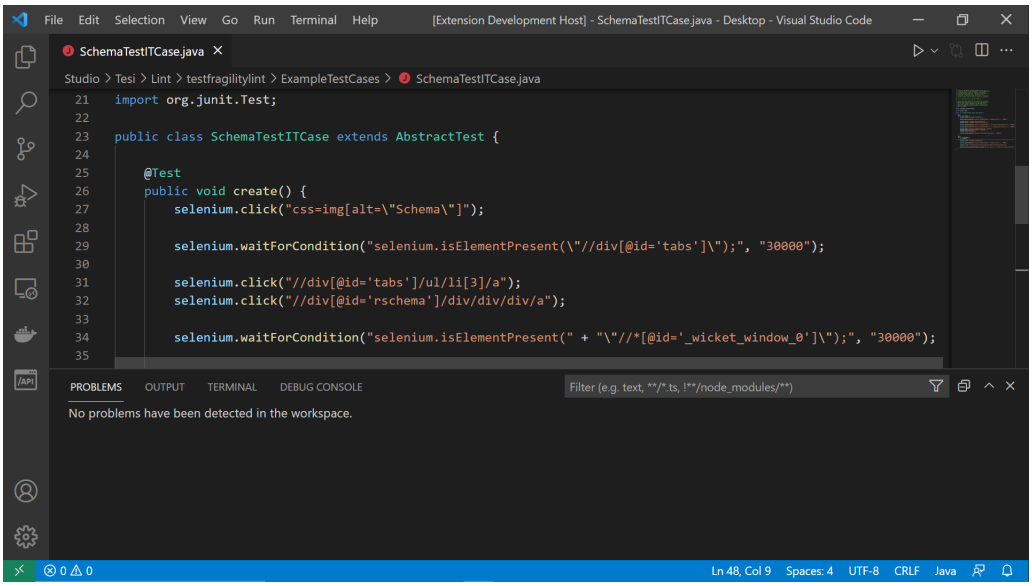


Figure 4.3: The linter is analyzing the test case.

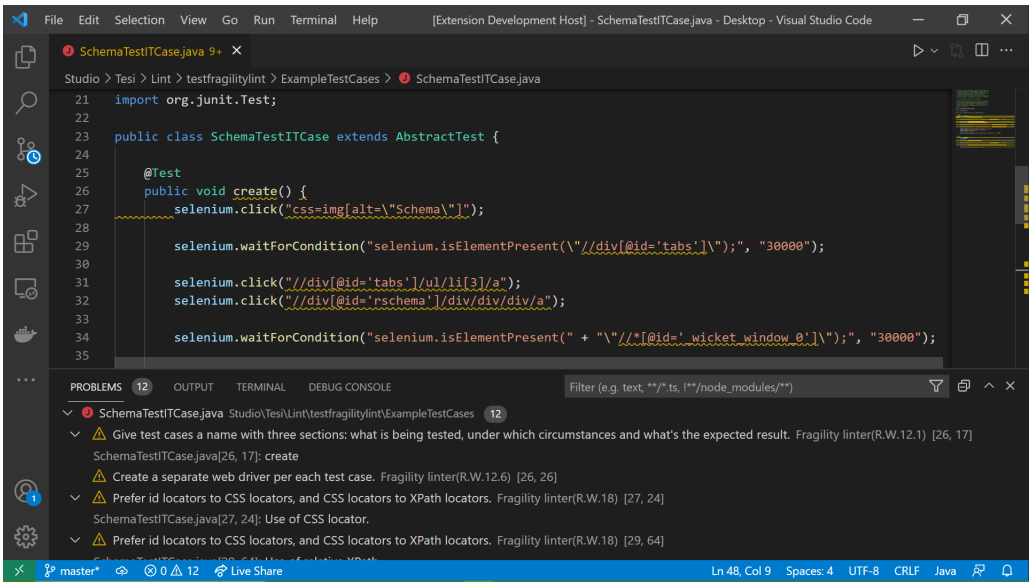


Figure 4.4: The linter underlines the code smells.

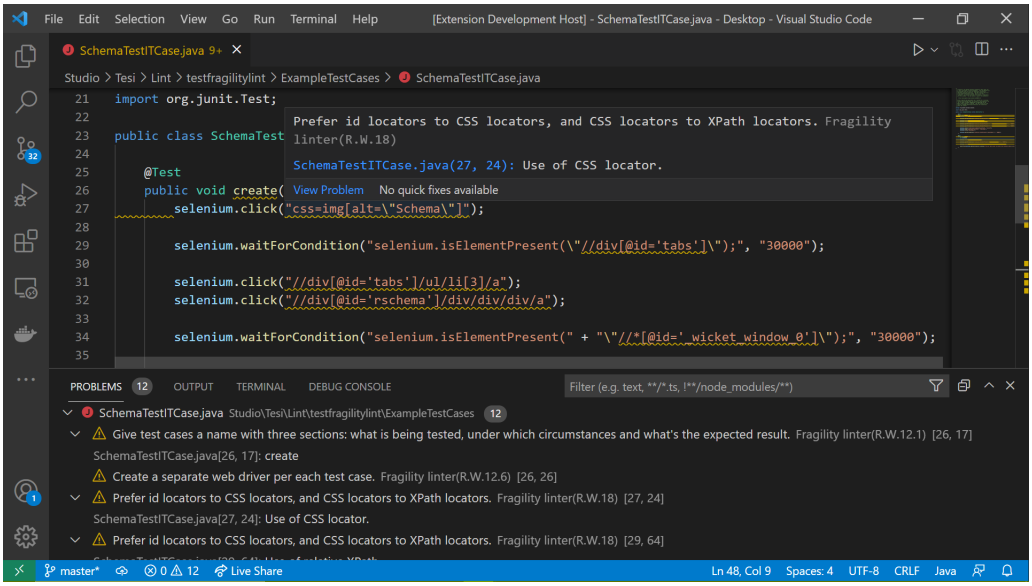


Figure 4.5: Each code smell is enriched with an informational tooltip, shown as soon as the mouse hover over it.

Chapter 5

Tool validation

This chapter first presents the validation of the linter by applying it on real test cases. At the end, a more wide discussion analyzes the possible threats to validity of the tool and, overall, of the whole study.

5.1 Methodology

The tool needs some validation, namely whether the rules gathered previously are effectively enforced or not by developers in test suites.

The validation process has been split up in three stages:

1. Selection of the software artifacts, as discussed in section 5.2;

2. Implementation of a report feature: the linter can serialize the detected rule violations into a JSON file; however, this is a kind of raw format which is hard to read and understand. As a consequence, rule violations can be automatically rendered in charts; indeed, graphs typically donate a wide and user-friendly view on data, unlike tables and XML or JSON report files;

3. Generation of chart reports: the linter has been requested to build four chart reports on the selected software artifacts.

In order to produce results that can be rendered in charts, the lint has been enriched with a command in the context menu of a resource that dynamically builds a chart based on the collected diagnostics. A resource can be either a local file or a local folder. For a flavour about the feature, see images in section 5.3.

5.2 Selected Software Artifacts

The tool has been applied to a set of real test cases, so as to donate greater industrial relevance to the present study [38]. Test cases have been selected from the list built during data collection discussed in section 3.1, beside some extra in order to reach the established threshold of 30 projects. This means that the lint has been partially tested on the data collected from diff analysis, while the lint itself is built on top of the rules gathered from wikis. This analysis therefore connects the two collection processes.

In order to produce the results, the lint has been requested to generate a chart report for a couple of folders. One folder is named 'Java' and contains a set of 15 sub-folders, each one being a test suite of a project; the test files are written in Java language. The other folder, 'Javascript', has the same structure as the former one, but the test files are written in Javascript language. Each sub-folder contain an unspecified number of test files, depending on the project; this quantity ranges from 1 to 20. It is worth to note that the linter only recognizes test files whose name follows a certain pattern; Java test file names have to contain the word 'test' or 'Test'; Javascript test file names have to contain the word 'test' or 'spec'. Some test files do not respect this pattern, so they are ignored by the linter.

5.3 Results

The figures beneath show the result of the linting process. The target resources are the 'Java' and the 'Javascript' folders that contain the selected software artifacts. The report of each folder shows occurrences grouped by either recommendation or test file.

Each chart can be classified as a histogram. Data are sorted by decreasing frequency, in order to help understand which are the most common issues in test suites.

Overall, the charts demonstrate that the actual test cases differ from the model established with the recommendations discussed in section 3.2. Comparing Figure 5.1 and Figure 5.3, the recommendation which is more violated is about global variables. This rule has actually a wide scope in software engineering: it is well-known indeed that global variables have drawbacks and must be avoided as much as possible. The recommendation about locators is one of the most violated. This rule is historically important in the context of GUI testing, but developers for some reason do not follow it. This result alone justifies the creation of the linter as a tool to minimize the usage of bad practices in these kind of test cases. The same comparison also points out that developers prefer to describe the behaviour of a certain snippet (function, block, statement or whatever else) by commenting it

Figure 5.1: Chart report on Java folder, grouped by recommendation.

Figure 5.2: Chart report on Java folder, grouped by test file.

rather than adopting the POP (Page Object Pattern). Indeed, commenting can be seen as a primitive and quick form of POP. Testers also do not tag their test cases, fact that would enable them to run just a subset of the test cases in order to decrease the overall execution time.

As expected, the results are essentially independent from the specific language in which test suites are written. A corollary of this finding is that peculiarities of a certain language does not help in writing more robust test cases.

Regarding the other two charts, Figure 5.2 and Figure 5.4, they have no relevance

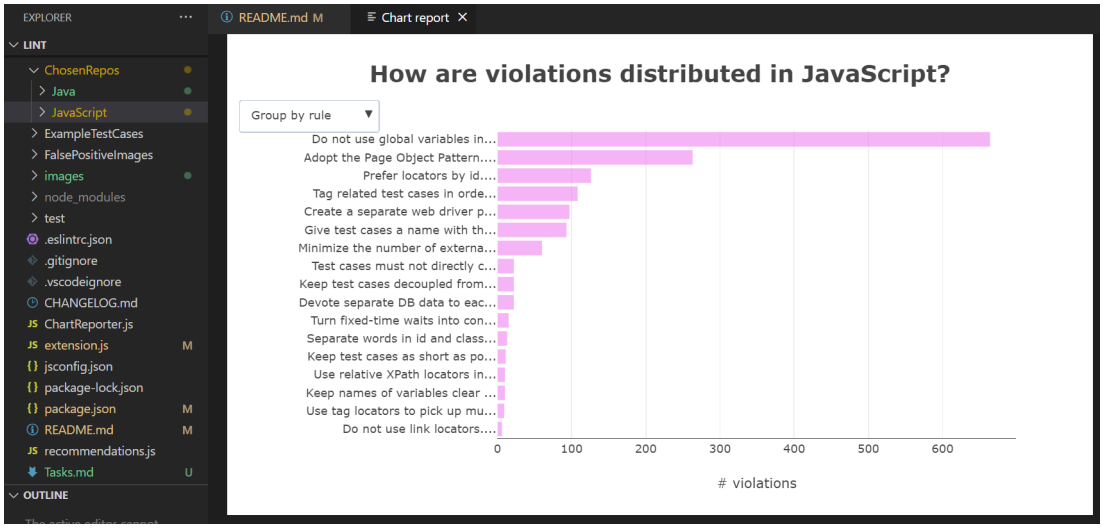


Figure 5.3: Chart report on Javascript folder, grouped by recommendation.

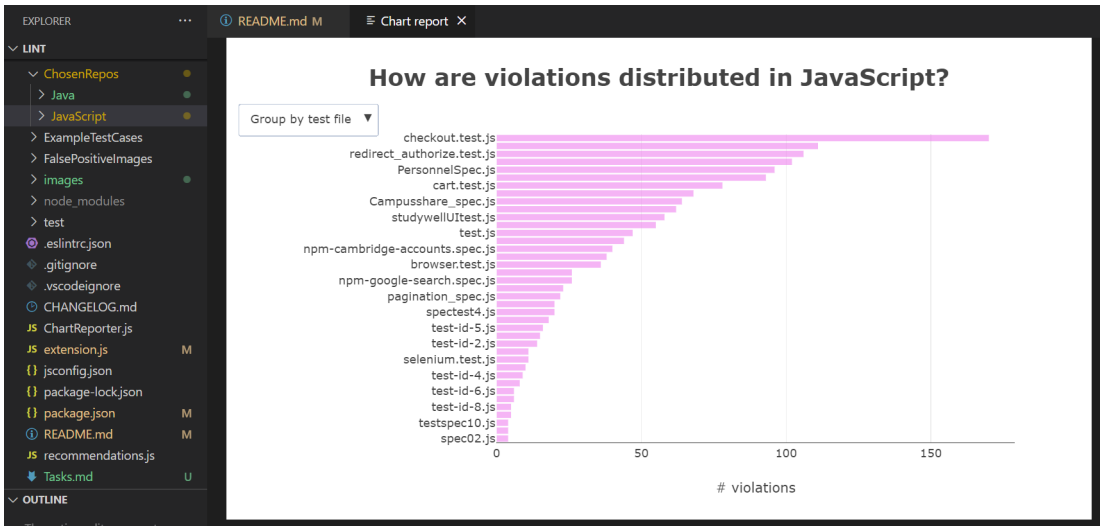


Figure 5.4: Chart report on Javascript folder, grouped by test file.

here since what is shown is a mixture of violations coming from different projects. The two images are anyway an example of what the linter can do. They are mostly useful for the final user that may want to rank the test cases by number of violations in order to select where to put effort.

The four charts above, from the theoretical point of view, also represent a measurement of fragility of test cases. The way chart reports measure fragility is compliant with the definition of fragility given in chapter 3.

5.4 Threats to Validity

The lint enforces rules that are as independent as possible from the testing tool used. However, since most test cases from the selected ones exploit Selenium, it is expected that the lint cannot actually recognize the same recommendations when other testing frameworks like Cypress are used.

Some test cases are written for Protractor rather than Selenium, a test framework developed specifically for AngularJS (for instance /ChosenRepos/JavaScript/JsRepo12/wkExclude/spectest3.js). They have been kept because the syntax for locator-related statements is similar to those exposed by Selenium.

The application has been tested against a set of test cases. As known from testing theory, grasping when software quality has achieved a good threshold is hard. The lint can certainly be enhanced in terms of accuracy, precision, number of recommendations, capability to be configured, number of supported IDEs and so on. This fact certainly impacts the results with a degree that cannot be null.

Validation of empirical results is an important research step in software engineering. [38] This paragraph discusses consequently the threats to validity of the conducted study.

Here follows a list of threats to validity of the present study, grouped according to a previous study that classified them into this taxonomy [38]:

1. Internal validity threats: *threats which may have affected the results and have not been properly taken into account.* During test collection from test cases, the project's latest commit date has not been considered; in other words, no speculation has been done upon how much the considered projects are old. This may induct a gap between the target overview of the collection process and the present one. In more detail, the selenium test suite may have not been aligned to the current application version, which is sometimes still alive or just maintained. However, the Selenium library that is available nowadays expose the same essential methods, although they follow a more fancy code style.

2. Construct validity threats: *threats about the relationship between theory and observation(s).* According to ISO/IEC-25010, software quality can be external (expressed at runtime), internal (expressed by the code structure) or in-use (linked to the developer and the context); since a lint is formally a static analyzer, it can only probe internal quality and quality in use. This last one has been considered too during the present study, given the classification of psychological fragility and the motivations of some recommendations;

3. Conclusion validity threats: *possibility to derive inaccurate conclusions from the observations.* No significant threat belonging to this category has been found;

4. External validity threats: *threats that affect the generalization of results.* Code smells (bad practices) cannot be generalized by definition, indeed their formal signature include the context under which they apply; some testers consider a certain code smell as a design benefit, indeed [16]. Some teams keep test cases deliberately fragile, so as to detect a certain type of modification. This observation only partially harms the results of the proposed linter, since its rules assume that the context regards visual testing of web applications.

Chapter 6

Conclusions

The following study has faced the problem of fragility in test cases, in the context of DOM-based End-to-End Visual test suites.

1. The first step was studying the related background in order to figure out the main observations and relationships between test cases, fragility and testers;

2. Secondly, the study concentrated on gathering actual data to outline the hands-on behaviour of programmers toward their tests, along the lifetime of projects. Data take the form of either modifications in the test code or recommendations;

3. The results suggested to build a linter that could help testers in writing more robust test cases by respecting the compliance with recommendations;

4. The linter has been tested against different real test cases;

5. The output of the linter has been redirected to generate report files, that have been analyzed to provide an overview of the distance between what programmers should do and what they actually do while writing test code.

The proposed linter aims to increase the robustness of test cases, which is one of the possible countermeasures against fragility. In this way, programmers learn how to write better test cases without inventing homemade personal practices.

The tool is currently compliant only with one IDE. In future, it is planned to extend its compatibility with other editors, like IntelliJ Idea and Visual Studio. The suggested method to accomplish this task is to migrate the current VSCodeAPI module to a Language Server Protocol API, a standard intended for extensions that enables the same back-end source code to be exploited by different IDEs on client side.

The study has been conducted for Selenium test suites; although the recommendations have a certain degree of tool-independence, further studies against other tools would confirm this type of independence. The suggested tool as of today is Cypress, briefly presented in subsection 2.4.2.

Rules currently have a uniform gravity. In future, the linter should manage with rules having a different relevance. The suggested way to measure the importance of a rule is given by the number of reasons behind the rule, where 'reason' is defined in the dictionary shown in Figure 3.16.

From the theoretical point of view, collecting data should remain a continuous or at least periodical process. Since the present study has thoroughly specified the source of each recommendation / diff analysis, further data mining doesn't have to start from scratch.

Appendix A

GitHub references

The GitHub URL of the thesis contains the spreadsheet about diff files: `https://github.com/Thefolle/ThesisFragility`

The linter root folder is located in the same URL of the thesis, inside the folder '/Lint/testfragilitylint'.

Bibliography

[1] *Selenium guidelines*. URL: http://www.selenium.dev/documentation/test_practices/ (cit. on pp. 2, 7, 11, 22).

[2] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, Incorporated, 2011 (cit. on pp. 6, 7).

[3] Alex Ruiz and Yvonne Wang Price. «GUI Testing Made Easy». In: () (cit. on pp. 7, 8).

[4] Yongjie Xu, Xiaodong Zhu, and Yigang Wang. «Towards GUI Test Based on Interactive Event-Flow Model». In: () (cit. on pp. 9, 10).

[5] *Testing tools used in software development worldwide in 2017*. URL: https://www.statista.com/statistics/673467/worldwide-software-development-survey-testing-tools/ (cit. on p. 11).

[6] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. «Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study». In: *IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (2013) (cit. on pp. 13, 23).

[7] *The Good and the Bad of Selenium Test Automation Software*. URL: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-selenium-test-automation-tool/ (cit. on p. 14).

[8] *Key Differences of Cypress*. URL: https://docs.cypress.io/guides/overview/key-differences (cit. on p. 15).

[9] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. «Visual Web Test Repair». In: () (cit. on pp. 15, 19).

[10] *About Pyccuracy*. URL: https://github.com/heynemann/pyccuracy (cit. on p. 15).

[11] M. Assem, A. Keshk2, N. Ismail3, and H. Nassar4. «Specification-Driven Automated Testing of Java Swing GUIs Using XML». In: () (cit. on p. 16).

[12] *Percy*. URL: https://percy.io/ (cit. on p. 16).

[13] *Sikuli*. URL: http://doc.sikuli.org/ (cit. on p. 16).

[14] *Vista*. URL: https://github.com/saltlab/vista (cit. on p. 16).

[15] *Test Quality*. URL: https://www.testquality.com/ (cit. on p. 16).

[16] Manuel Breugelmans and Bart Van Rompaey. «TestQ. Exploring Structural and Maintenance Characteristics of Unit Test Suites». In: *Lab On REengineering, University of Antwerp* () (cit. on pp. 16, 64).

[17] Riccardo Coppola, Maurizio Morisio, Marco Torchiano, and Luca Ardito. «Scripted GUI testing of Android open-source apps: evolution of test code and fragility causes». In: *Empirical Software Engineering* (May 2019) (cit. on pp. 17, 50).

[18] Martin Monperrus. «Principles of Antifragile Software». University of Lille and Inria, France. Jan. 2017 (cit. on pp. 19–21).

[19] Russ Miles. «An Introduction to Designing and Building Antifragile Microservices with Java». In: June 2014 (cit. on pp. 20, 21).

[20] *Source Code License Definition*. URL: https://www.lawinsider.com/dictionary/source-code-license (cit. on p. 33).

[21] *Improving test quality*. URL: https://github.com/uselagoon/lagoon/discussions/2613 (cit. on p. 42).

[22] *Why would you use ID attributes*. URL: https://github.com/manoelcyreno/test-samples/wiki/Why-would-you-use-ID-attributes (cit. on pp. 43, 44).

[23] *Is adding ids to everything standard practice when using selenium*. URL: https://sqa.stackexchange.com/questions/6326/is-adding-ids-to-everything-standard-practice-when-using-selenium (cit. on p. 44).

[24] *Order of preference of selectors*. URL: https://www.selenium.dev/documentation/webdriver/locating_elements/#tips-on-using-selectors (cit. on pp. 44, 49, 50).

[25] *HTML best practices*. URL: https://google.github.io/styleguide/htmlcssguide.html#ID_and_Class_Naming (cit. on p. 45).

[26] *Working memory limitations*. URL: https://github.com/howard8888/pycon-ca-2018/wiki (cit. on pp. 45, 49, 52).

[27] *The official Node best practices*. URL: https://github.com/goldbergyoni/nodebestpractices (cit. on pp. 46–48).

[28] *Overview of Test Automation*. URL: https://www.selenium.dev/documentation/test_practices/overview/ (cit. on pp. 46, 49, 50).

[29] *Global variables induce fragile tests*. URL: https://github.com/freudgroup/freudcs/wiki/Javascript-Namespace-Declaration (cit. on p. 47).

[30] *Global state should not be used.* URL: https://www.selenium.dev/documentation/test_practices/encouraged/avoid_sharing_state/ (cit. on p. 47).

[31] *Start the webdriver per each test.* URL: https://www.selenium.dev/documentation/test_practices/encouraged/fresh_browser_per_test/ (cit. on p. 47).

[32] *A definition of Aspect-Oriented Programming.* URL: https://study.com/academy/lesson/aspect-oriented-programming-definition-concepts.html (cit. on p. 48).

[33] *Signs your software project is rotting.* URL: https://github.com/jopheno/CleanArchitecture/wiki/Signs-your-software-project-is-rotting (cit. on p. 48).

[34] *Test setup.* URL: https://www.selenium.dev/documentation/test_practices/encouraged/generating_application_state/ (cit. on p. 49).

[35] *Mocking external services.* URL: https://www.selenium.dev/documentation/test_practices/encouraged/mock_external_services/ (cit. on p. 49).

[36] *Test independency.* URL: https://www.selenium.dev/documentation/test_practices/encouraged/test_independency/ (cit. on p. 49).

[37] *About node modules.* URL: https://nodejs.org/api/modules.html (cit. on p. 55).

[38] Serdar Dogan, Aysu Betin-Can, and Vahid Garousi. «Web application testing: A systematic literature review». In: *The Journal of Systems and Software 91* 91 (2014), pp. 174–201 (cit. on pp. 60, 63).