POLITECNICO DI TORINO
Repository ISTITUZIONALE

Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection

Original
Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection / Fulcini, T., Garaccione, G., Coppola, R., Ardito, L., Torchiano, M.. - ELETTRONICO. - (2022), pp. 17-24. (A-TEST 2022: 13th Workshop on Automating Test Case Design, Selection and Evaluation Singapore November 17-18, 2022) [10.1145/3548659.3561306].

Availability:
This version is available at: 11583/2971256 since: 2022-09-13T08:52:39Z

Publisher:
ACM

Published
DOI:10.1145/3548659.3561306

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript, con Copyr. autore

(Article begins on next page)

26 November 2026

Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection

Tommaso Fulcini
tommaso.fulcini@polito.it
Department of Control and Computer Engineering, Politecnico di Torino
Turin, Piedmont, Italy

Giacomo Garaccione
giacomo.garaccione@polito.it
Department of Control and Computer Engineering, Politecnico di Torino
Turin, Piedmont, Italy

Riccardo Coppola
riccardo.coppola@polito.it
Department of Control and Computer Engineering, Politecnico di Torino
Turin, Piedmont, Italy

Luca Ardito
luca.ardito@polito.it
Department of Control and Computer Engineering, Politecnico di Torino
Turin, Piedmont, Italy

Marco Torchiano
marco.torchiano@polito.it
Department of Control and Computer Engineering, Politecnico di Torino
Turin, Piedmont, Italy

ABSTRACT

Context: GUI Test suites suffer from high fragility, in fact modifications or redesigns of the user interface are commonly frequent and often invalidate the tests. This leads, for both DOM- and visual-based techniques, to frequent need for careful maintenance of test suites, which can be expensive and time-consuming.

Objective: The goal of this work is to present a set of guidelines to write cleaner and more robust test code, reducing the cost of maintenance and producing more understandable code. Based on the provided recommendations, a static test suite analyzer and code linter has been developed.

Method: An ad-hoc *grey literature* research was conducted on the state of the practice, by performing a semi-systematic literature review. Authors' experience was coded into a set of recommendations, by applying the grounded theory methodology.

Based on these results, we developed a linter in the form of a plugin for Visual Studio Code, implementing 17 of the provided guidelines. The plugin highlights test smells in the Java and Javascript languages.

Finally, we conducted a preliminary validation of the tool against test suites from real GitHub projects.

Conclusions: The preliminary evaluation, meant to be an attempt of application of the plugin to real test suites, detected three main smells, namely the usage of global variables, the lack of adoption of the Page Object design pattern, and the usage of fragile locator such as the XPath.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

GUI Testing, Software Testing, Software Engineering, Test smells

ACM Reference Format:
Tommaso Fulcini, Giacomo Garaccione, Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2022. Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '22), November 17–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3548659.3561306

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
A-TEST '22, November 17–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9452-9/22/11…$15.00
https://doi.org/10.1145/3548659.3561306

1 INTRODUCTION

Software Testing is one of the key Software Engineering sub-disciplines: it aims at detecting failures and misalignments with the software requirements, allowing to discover faults and bugs during software development, saving costs and effort. Different approaches (manual, capture and replay, or automated and hybrid testing) can be adopted to perform Graphical User Interface (GUI) testing activities, and at different levels (from small code units to the complete system). GUI testing identifies all the testing techniques that send input to the System Under Test (SUT) through its GUI. Interacting with a system through its GUI is a useful way of testing if the application meets the requirements since the test sequences mimic the interactions of a end user of the SUT.

Although automated GUI testing brings many positive aspects such as saving time, boosting customer satisfaction, and enhancing collaboration among developers and testers [1], it is known to be one of the most fragile ways of testing an application, since existing test cases may break when some visual changes are performed on the SUT.

In the context of automated GUI testing of web applications, we can distinguish three main methods employed by test cases to find a web element at execution time [13]:

(1) Coordinate-based locators: elements are identified by their coordinates in the graphical interface;
(2) DOM-based locators: elements are accessed via the Document Object Model (DOM) structure;
(3) Visual-based locators: test cases use algorithms for image recognition to detect the elements starting from screenshots.

DOM (or layout-) based locators are still the most frequently adopted in both practice and related literature. With a focus on

this family of GUI testing tools, we drove our research to practice by conducting an effort-bounded ad-hoc research on open-source Github projects. We specifically considered projects in which the web GUI testing tool Selenium was used or cited for scripting test cases. The present research aims to extract guidelines from the shared knowledge and experience of practitioners to provide a reference for developers that will approach scripted GUI testing techniques in the future with suggestions to be followed of bad habits and test smells to avoid.

The remainder of the paper is organized as follows: Section II provides the needed background information to understand the guidelines; Section III explains the method followed to extract the guidelines deepening the reasons for the recommendations; Section IV provides an overview of the work-in-progress tool; Section V explains the preliminary evaluation carried out on the linter; Section VI concludes the paper by providing suggestions for future expansions of the guidelines and the linter.

2 BACKGROUND

GUI test suites are often considered complex to design and implement due to the execution overhead which characterizes them, but once built, they are often reused adopting the regression testing technique to test the SUT iteratively. Although reusing an existing test suite is an effort-saving practice, the test code needs to be maintained to keep it consistent with the new version of the system: this intrinsic fragility causes test failure not representing actual bugs, but test fragility deriving from its design or implementation. ISO/IEC-25010 standard classifies fragility to the 'Modifiability' sub-category of 'Maintainability' as the *"degree to which a product or system can be effectively modified without introducing defects or degrading existing product quality"*. In this context, we can define a test case as *robust* whether, following an evolution of the SUT, the test case does not require any adjustment. A test which is passing in a certain version is considered *fragile* if, as a result of the evolution of the SUT, it fails requiring external intervention to fix it [5]. For the rest of the manuscript, we will refer to the fragility of test cases simply as fragility.

In GUI testing activities, to perform an action against the SUT's interface, it is necessary to utilize *locators*, i.e., mean to identify an element on-screen to target. There are three existing categories of locators: on-screen coordinates of the widget in the SUT (or *first generation* locators); DOM-based (or *second generation locators*) utilizing information from the document object (e.g., HTML tags, ids, CSS properties); finally, visual (or *third generation* locators) which consists of screen captures of the target element that is searched within the running SUT's GUI by using image recognition techniques.

In the context of DOM-based locator tools for web applications, tests are bound to the SUT recurring to the hierarchical structure of the HTML page [9]. These kinds of test cases are subject to an intrinsic fragility that derives from the way elements are identified: using the DOM to select which elements should be tested makes test cases tightly coupled to SUT's pages. This tight coupling leads to binding test cases to specific visual elements which are proxies of functions, rather than binding to the specific feature under testing. As a result, most of the evolution of the system, especially when updating visual aspects, causes test cases to break and testers are required additional effort at fixing the test case [2].

One of the most used practices to reduce the fragility of test suites is the adoption of the Page Object (PO) pattern [8]. Page Object is a design pattern that consists in modeling all the web pages that a test case will involve as objects using the native programming language of the test case. This representation exposes all the required functionalities of a web page in the shape of a function (or method) called in the implementation of a test case: this architectural pattern allows the creation of a reusable infrastructure, as all the test cases which require a specific function to be performed will simply call the function implemented by the page object. A second advantage in using the PO pattern is the encapsulation of the implementation details inside the page object, this allows testers implementing test cases to work on a high level of abstraction. Overall, PO allows reducing the burden on testers when repairing a test suite after a break, reducing the time and lines of code (LOC) needed to repair.

Besides all the positive aspects brought by the adoption of POs, this pattern requires a non-negligible effort in its implementation, due to the overhead of setting up objects and their methods. For this reason, PO is well-suited in contexts where GUI testing suites are reused a considerable number of times in different iterations and discouraged for testing simple GUI operations.

In scripted GUI testing, during the implementation of test code, as it happens when developing source code, it is not infrequent to introduce *test smells*, i.e., poorly designed test code whose presence may negatively affect some aspects of the test suite such as quality, maintainability, understandability, etc.. Although test smells might not influence the proper functioning of a test suite in some specific releases, their presence can result in breaking test cases in a subsequent release, negatively influencing the fragility of the whole test suite, as reported by Mathew et al. [11]. Another problem directly related with test smells, which deteriorates the maintainability of test cases when a breakage occurs, is the low visibility of the relationship between the fixture point and the outcome of the test. Therefore, following the guidelines from the existing literature is one important step toward robust and well-designed test suites. In addition to having a taxonomy of bad practices, it is very important to keep it up to date, whenever new architecture, design, and technology emerges.

Among all the automated element-based testing tools, Selenium is one of the most flexible since it allows the creation of test suites in two different ways: using the capture and replay technique via Selenium IDE or programmatically via Selenium WebDriver [7]. The first is a browser extension that allows recording all the actions performed by the tester, which are then translated into statements and assembled in test cases. Although it is considered promising thanks to the capture and replay capability, Selenium IDE has indeed some limitations, such as the absence of conditional statements and logging features. Selenium WebDriver, instead, drives a browser natively through an object-oriented API for accessing the DOM of a web page. It allows the interaction with the SUT (e.g., clicking, typing elements, opening URLs) using also verification statements, allowing the specification of expected values and behaviors directly in the preferred object-oriented programming language.

3 GUIDELINES EXTRACTION

Investigating the state of the practice led us to develop guidelines based on practitioners' experience to avoid bad habits and limit their diffusion in future projects. In this section, we will deepen the research method followed to identify the pool of sources from which guidelines were extracted, along with their discussion.

3.1 Research Method

To conduct an analysis based on the state of the practice we decided to investigate which are the best habits stated by GitHub maintainers. Hence, we decided to search among the grey literature sources to find tangible examples of experience-based recommendations.

The search was driven using the Google search engine, with an *effort bound* on the first one hundred results (i.e., the first ten pages of results), to include only the most relevant contributions. The employed search string was meant to include the GitHub platform, the widely used tool for GUI test scripts Selenium and a term associated with the recommendation aspect, in this case, we chose both the 'recommendation' term and 'best practice'. The resulting search string was '`github selenium (best practices OR recommendations)`'.

Results from the direct search were filtered by firstly applying Inclusion and Exclusion Criteria (from now on, respectively IC and EC) and secondly evaluating the quality of the retrieved information.

We defined a set of IC and EC, each result item had to fully meet all the IC and none of the EC to be included in the study. The inclusion criteria were defined as follows:

- **IC1**: The source is an item of grey literature dealing with problems directly related to fragility of scripted GUI testing based on the authors' experience or authoritative recommendation from GUI testing tools developers.
- **IC2**: If the grey literature item is a GitHub project, the source code is publicly available for anyone (i.e., open-source).
- **IC3**: The item is written in a language that is directly comprehensible by the authors: English or Italian.

Whilst the applied exclusion criteria are the following:

- **EC1**: The source is not directly related to the topic of fragility of scripted GUI test cases for web applications.
- **EC2**: The source is a GitHub project whose code is partly or fully private.
- **EC3**: The source neither provides any evidence of the stated recommendation nor do the authors consider the recommendation a valid guideline.

To evaluate the quality of the sources in our pool, we could only rely on AACODS (Authority, Accuracy, Coverage, Objectivity, Date, and Significance) evaluation methodology, which is most suitable for quality assessment of grey literature items [15]. All the authors evaluated all sources independently, and discussions were conducted where no consensus was obtained in the first round of evaluation. The developed questionnaire's score was normalized for each source on a 0-5 interval: only the sources which had a score above 2.5 were kept in the final pool.

After the quality assessment, a process of backward level one snowballing was performed from the initial pool of sources: all the reference links contained in the original set of contributions were included in the preliminary pool, and their quality was evaluated, using the same approach above.

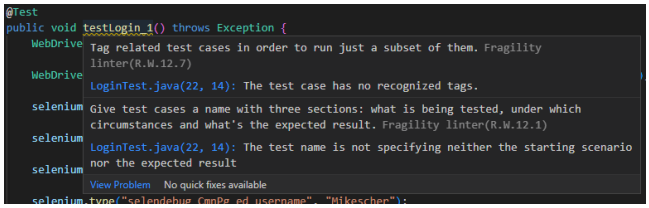

Figure 1: Screenshot of the tool: the underlined test code line has reported the corresponding guidelines violated

A total of 23 grey literature items composed the final set of sources, from which the guidelines were extracted and formalized. To extract the guidelines we followed the principles of Straussian Grounded Theory for evidence-based research [10]. The *Open Coding* [16] phase was performed over the analysis of all the collected sources, with the following steps: (i) a new guideline was added to the final set when no semantically equivalent guidelines have already been elicited; (ii) a guideline is synthesized along with semantically equivalent existing ones; in this case, the most generalizable text is kept in the final set of guidelines. All the authors of the manuscript participated to the coding phase. During the guidelines extraction, an internal consistency check was performed for each new coded guideline. In case of contradictions, the contrasting rules were compared and assessed based on of the relevance of the grey literature source and the interpretation of the authors of the present manuscript.

3.2 Guidelines discussion

The result of the synthesis process is a set of guidelines that extend the existing classification developed by Garousi et al. in their secondary study [6] with the addition of hints for the specific GUI testing context. Some reported guidelines are compatible with test smells already classified and discussed in the literature: we decided to keep them in our list to stress their importance and to highlight the fact that in the four-year time period which separates the present study from Garousi's literature mapping, some of the recommendations to avoid test smells remain still valid. We also motivate the inclusion of repeated guidelines because they are specifically applicable to the GUI testing discipline.

The extracted guidelines were examined and grouped into four different categories following the axial coding method, as defined by Strauss and Corbin [16]: the full list is displayed in Table 1. The categories are:

(1) *General Purpose*: those rules that do not depend on any particular test suite implementation aspect;
(2) *Locator-related*: those rules that harden the produced test cases that rely on DOM locators;
(3) *Name-related*: conventions referred to common testing habits improving readability, management, and repairing effort;
(4) *Implementation-related*: recommendations that suggest a cleaner, safer, and more robust way to implement test cases;

Table 1: Summary of the extracted guidelines

Id	Guideline description	Mentions
R0	Keep the number of unit tests greater than the number of end-to-end tests	[S01], [S21]
R1	End-to-end tests and integration tests should be developed before unit tests	[S07], [S21]
R2	Keep test cases as simple and short as possible	[S06], [S08], [S16], [S17], [S22]
R3	Prefer to use locator by Id, CSS locators and Xpath when not available, in that order	[S22], [S02], [S03], [S04], [S17], [S18], [S19]
R4	Use relative XPath in place of absolute XPath	[S02]
R5	Do not use link locators	[S04]
R6	Use tag locators only for multiple elements	[S04]
R7	Use relative URLs locator instead of absolute ones	[S19]
R8	For elements and variables use name that mirrors functionalities, use Ids when there is no specific functional purpose	[S05]
R9	Keep names of variables clear to everyone	[S06], [S18]
R10	Test cases name should contain three sections: what is being tested, the circumstances and expected results	[S07], [S18]
R11	Separate words in Ids and class names by a hyphen	[S05]
R12	Do not use global variables in test cases	[S09]
R13	Create a separate web driver for each test case	[S10], [S11], [S24]
R14	Devote separate database data to each test case	[S07], [S10], [S21], [S19]
R15	Minimize the number of external libraries	[S13]
R16	Tag test cases to run just a subset of them when necessary	[S07]
R17	Do not perform visual actions to setup the test case scenario. Use APIs of the AUT and direct DB queries instead	[S08], [S12]
R18	A test case must not continue the workflow of other test cases	[S16], [S22], [S14]
R19	Run linters to detect anti-pattern	[S07]
R20	Adopt Page Object Pattern	[S08], [S15], [S16], [S18], [S19], [S20], [S22], [S23]
R21	Avoid *Thread.sleep* statements by turning fixed-time waits into condition-based ones	[S19]
R22	Run test cases with multiple browser's driver	[S21], [S19]

General purposes guidelines are three rules that come from the Testing Pyramid concept [4], which pinpoint unit testing as the fundamental building block of maintainable testware. The three resulting recommendations are a natural consequence of this pattern, requiring to produce of a higher number of unit tests than end-to-end tests to verify thoroughly with simpler code snippets functional requirements, additionally, unit tests should cover those requirements that may not be covered by higher level tests, for this reason, they should be developed subsequently. Keeping test cases short is essential since each test case should verify only one particular aspect.

Locator-related guidelines are applicable only to DOM-based locators and they are suggestions on how to identify an element to be tested in the test code. These guidelines provide a ranking of preference based on the experience coming from the practice, such as the preference of using mainly Ids to identify elements, CSS locators in second place, and finally XPath. Ids appear to be faster and safer since Selenium and other tools provide direct access by calling the *'getElementById'* method at browser level. CSS locators are preferred to the XPath counterpart since their independence from DOMs, which is considered a highly mutable structure, still, it is not the recommended choice, since CSS identifiers' main purpose is to encode a particular style.

Considering XPath locators, relative ones are considered more robust than the absolute one, since the path between two elements appears to be less mutable than the complete path, which relies on the whole DOM. Link locators are discouraged, as they work only on link elements and under the hood they are translated to XPath selectors, inheriting all the aforementioned drawbacks.

Tag locators are also discouraged when used to target one specific element while multiple elements are admissible to be picked; this kind of locator should be instead used in case of need to select multiple elements of the same kind.

Speaking of locators, also URLs belongs to this category as they allow to specify which is the web page the automated script is targeting: the usage of an absolute URL is discouraged in favor of a relative one. For this purpose, an abstract method that processes a relative URL into an absolute one should be implemented to convert from one to another. This guideline should however be reconsidered when carrying out security testing: in fact, the absolute URL contains the specifications of the protocol used, the correct implementation of which should be ensured and evaluated by security tests by explicitly specifying the complete URL each time.

Name-related guidelines define useful conventions that improve firstly readability of the code produced and secondly the maintainability, since they allow to reduce the burden of getting familiar with the code and the workflow during the maintenance phase, by making clear the purpose of variables and methods. Naming variables with clear names that state their purpose helps in understanding the code flow before even running it; additionally, a well-defined naming convention keeps consistency in the variable names limiting redundancy and unused variable smells.

In particular, assigning a good name to a test case that refers to the testing conditions allows the tester to infer the starting scenario avoiding reading the function's body. Additionally, clearly stating the expected output in advance helps in overcoming any psychological biases that may lead the tester to define mistaken assertions.

Implementation-related guidelines provide practical suggestions that may improve the resiliency of the test suite or reduce maintenance effort and execution time. Isolation of test cases makes them more robust, for this reason, the usage of global variables, and in particular shared web drivers, is discouraged since it can contain spurious values when different test cases are ran. For the same principle we can derive the recommendation of keeping independent test cases, avoiding continuing the workflow of other test cases. Shared data are considered fragile also in a test database, which should contain separate data for each test case. One more cause of

fragility is the use of third-party libraries, since they may increase the wall-clock time of a test case to deliver, causing synchronization problems.

Test cases should also avoid performing any visual action to set up the test scenario, this duty should be left to SUT's API or Database queries as encouraged by Selenium's official website [14]. Albeit visual interaction should be avoided at all when scripting GUI interaction, some interactions may depend on previous ones: this requires a sort of synchronization to be sure that SUT's state is the one foreseen. The coordination of these operations should avoid the suspension of the running thread for a fixed amount of time, turning these statements into condition-based wait such as preferring *waitForPageToLoad* or *wait.until* to *Thread.sleep*. The aforementioned code is inherently fragile as the timing of web driver operations is by definitions aleatory.

Indeed, one of the most useful and effective good practice the guidelines encourages is the adoption of PO pattern, as the high number of mentions suggest: it allows to create far more robust test cases concerning the counterpart not implementing PO, as documented by Leotta et al. [8]. Another guideline specifically applicable only to the GUI testing technique is the usage of multiple browsers for running tests. The rationale is that, although many browsers implement similarly the appearance of web pages, each one may have a specific scripting implementation making the web application dependent on the browser type and version: this allows to reach a higher coverage over the browser fragmentation phenomenon. Furthermore, the implementation of each web driver highly influences how test case method calls are performed.

To conclude, guidelines R16 and R19 represent hints related to the development and running environment: tagging test cases allows to execute just the necessary subset, instead of the full suite, and using a linter inside an IDE is the most common way of detecting and removing test smells in the produced code.

4 THE PROPOSED TOOL

Based on the described guidelines, we started developing a linter that supports testers by detecting test smells in a test suite. The tool was conceived as a static code analyzer that probes bad practices that could cause fragilities, shaped as a plugin for the Visual Studio Code IDE (from now on, VS Code). The tool, called *FragilityLinter*, has been written in Javascript, using Node.js framework.

The choice of VS Code was driven by its wide usage by the developer community, which makes it one of the top three IDE [3]. Whilst the two target programming languages were chosen basing both on the experience of the authors and on Stack Overflow's developer survey, whose 2021 iteration put Javascript and Java respectively in the first and fifth place of programming languages by usage [12].

Figure 2 shows the logical architecture of the project, divided into four packages: a graphical tier, a logical tier, a data tier, and the external dependencies.

The data tier contains a static array of recommendations, associating the hint's message to an id representing the fragility. The external dependencies are two free libraries (i.e., Acorn for Javascript and Java-Parser for Java) that parse the code in their specific programming language creating an Abstract Syntax Tree

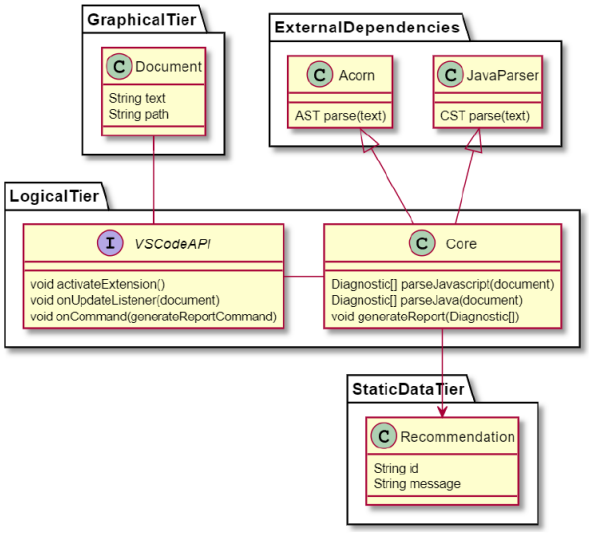

Figure 2: UML class diagram representing the structure of the linter

(AST) structure in the Javascript case and a Concrete Syntax Tree (CST) structure for the Java language. The input structure obtained using the proper parser is afterward analyzed by the core module, which operates as a semantic analyzer, to detect anti-patterns. The usage of two separate libraries makes the linter flexible, as it could be extended to any other object-oriented programming language by adding a suitable parser for the new programming language, to extract an AST or CST-compatible structure to be provided to the core module as an input. The logical tier includes the recognition of fragilities on the code analyzed and a module with the implementation for the specific IDE, VS Code in this case.

The tool, whose development is in progress, implements sixteen out of the twenty-three proposed recommendations. For time and effort reasons, the presented version has some limitations due to the prototypal nature of the linter: in particular, R0 and R1 have not yet been implemented since they respectively require an overview on the whole project structure to compute and compare the number of unit and end-to-end test cases, and the access to the full version control history of all the files to determinate which tests were written first.

Guideline R8 requires natural language processing to distinguish if the names match functionalities: this requirement constitutes a processing overhead that should be empirically assessed to establish its compliance with the real-time nature of the linter which is executed directly during the editing of the test case.

Guideline R14 is partially implemented since the linter can indirectly recognize the rule, but a complete heuristic recognition is possible only by analyzing databases.

R17 and R18 have not yet been implemented because of the high degree of complexity that characterizes them: the former due to the large variety of possible statements that may constitute a setup phase, most of which are optional; the latter because of the strong connection with the SUT which characterize the rule and

the level of complexity in grasping whether a certain test case is the continuation of a previous one. The last guideline which has not been implemented on account of the complexity is R20: detecting the adoption of PO pattern is a highly complicated task, which requires also some implementation assumptions. In particular, all the definitions of Page classes, which should be defined in different files than the one containing the actual tests, should be parsed to collect all the pages under test with their functionalities, and the test file should ensure that the correct method is called to perform the action against the SUT. For the same reason mentioned for R0, parsing all the files in a non-well-defined project structure has been considered out of the scope of the prototypical version of the linter.

A report function was embedded in the linter to show the distribution of the different guidelines violations in the different test files or folders. The report can compute the total count either for each rule, or for each file: this allows the assessment of the distribution of fragilities per single test file, or per rule violated.

5 PRELIMINARY EVALUATION

A preliminary evaluation was performed to assess the actual operational state of the linter, whilst at the same time characterizing existing open-source projects based on their adoption of the elicited guidelines.

To that extent, a small number of public repositories was selected and statistics about the fragilities addressed by the guidelines were computed.

5.1 Methodology

We selected open-source projects where all the test code was fully accessible to verify if the prototype of the linter was successfully recognizing test smells identified by our guidelines. We browsed GitHub using an ad-hoc search string allowing us to narrow the resulting pool to projects containing in the first case Java and in the second case Javascript code with a file name containing the word test and using Selenium, with the following search string:

```
extension:java filename:*test* language:Java selenium
```

We required the usage of the web GUI testing tool Selenium mainly because of its large diffusion among open-source projects with GUI testing suites. GitHub's results were then grouped by Code and a total of 15 projects per language were included to be used as a target for the linter. Test files were grouped in folders firstly by language and secondly by repository, the report features were then operated on all the files. The detected rule violations were organized and rendered in charts showing the diffusion of the different violations grouped by guideline.

5.2 Results

Considering the case of java test suites, the most violated guideline among those implemented, as Figure 3 shows, was by far the usage of global variables (i.e. R12) with more than a thousand occurrences detected. Considering the distribution of smells in Javascript, reported in Figure 4, a similarity in the distribution of R12 can be noticed within the two programming languages. The second guideline

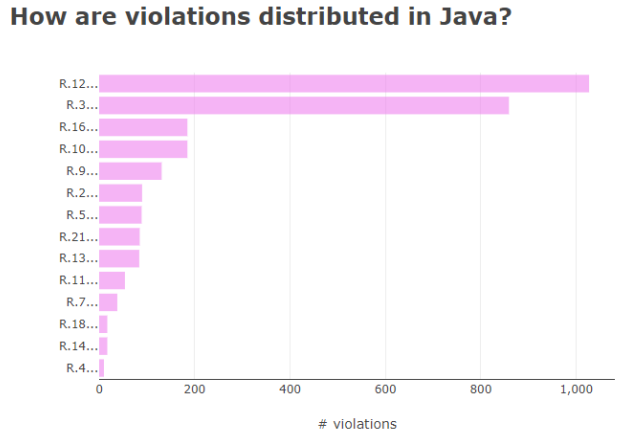

Figure 3: Diffusion of the guidelines violations over the retrieved Java project

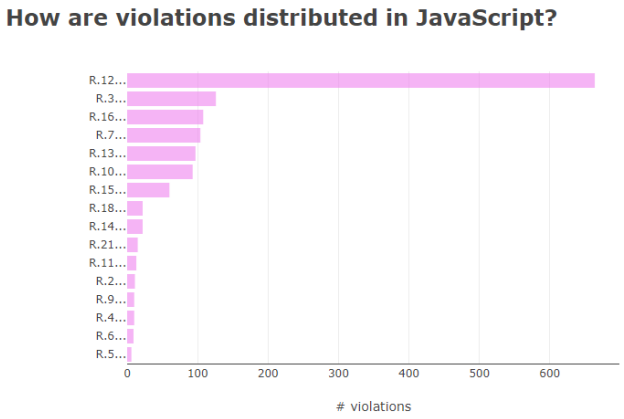

Figure 4: Diffusion of the guidelines violations over the retrieved Javascript project

by number of detected test smells in both programming languages was the usage of discouraged locators such as XPath in place of the more robust id locator (i.e. R3). A significant disparity of proportions characterizes this rule since more than eight hundred occurrences were verified in Java, while just about a hundred in the case of Javascript.

The third most common violation is again the same in both languages. Recommendation R16 denotes a lack of scheduling in the projects, which could be improved by correctly tagging the test cases to run just the needed subset, saving a considerable amount of time in each run. Although the usage of absolute URLs instead of relative ones is the fourth more frequent violation in Javascript, it has a different diffusion in Java, as it has been counted more than a hundred times in the first case, while just thirty-eight occurrences were counted in the second case. Another common guideline occupying the fourth place for Java and the sixth for Javascript is the naming convention which proposes to name test cases by stating the subject, circumstances, and expected result of

the test case, with the goal of making its usage clear to everyone. The remaining noticeable violations concern the usage of the same web driver for different test cases (i.e. R13) in Javascript, which could also be considered as a special case of guideline R12, and the usage of unclear variable names which undermine the understandability of the test code (R9) in the case of Java.

Looking at the distribution of violations we observed almost no difference between the two considered programming languages, except for R3 which seems to be significantly more widespread in Java. A further noticeable statistic is the number of violations contained in the different testing files: in Java, the first and the second files per violations included respectively 374 and 202 smells, while in Javascript they were 118 and 106. In this respect we acknowledge that the resulting statistics may be biased by the quality of the retrieved Github repositories and the incomplete state of the tool. A systematic and more in-depth project selection will be required to provide robust and reliable evidence on the state of diffusion of test smells that we plan to perform once the linter's coverage of guidelines is complete.

6 THREATS TO VALIDITY

The main caveat about this work is that represents an initial work aimed to reduce test smells for scripted GUI testing and to improve test robustness. A few threats may affect the validity of the results.

6.1 Construct Validity

Construct validity refers to the concept that a novelty actually mimics what it intends to mimic, by direct or indirect objective standards. For this study, the main biases can be derived from the process of selection of the primary source items and the synthesis process. The guidelines were meant to reflect the state of the practice, for this reason, grey literature items were deemed suitable for the analysis. The selection of relevant sources in the grey literature context is highly dependent on two factors: the employed search string and the time in which the search has been carried out since the Google search engine has algorithms that mutate very frequently. We also argue that the quality of the retrieved grey literature items degraded rapidly with the page progression: in fact, the backward snowballing process was more useful than expanding the effort bound to include more direct results.

6.2 Internal Validity

Internal validity is the degree to which a study establishes the cause-and-effect relationship between the treatment and the observed outcome. In this study we presented a linter which, as already mentioned, is still in a prototypal form: this fact undeniably has influenced the obtained results. We are aware that the subset of implemented suggestions may not be representative of the total since their different nature, but we argue that the more operative hints about the way the testing code is written have been successfully implemented. The guidelines which remain unimplemented are those which we can consider more abstract and design-related, hence less likely to identify and localize at the code level by automated linting. One more internal validity threat affecting the results could be the fact that testing utilities, unless they contain the "*final*" keyword, are counted as smells whether declared as global variables. We acknowledge that the provided preliminary distinction between smelly global variables and side-effect-free utilities might have slightly offset the gathered data.

6.3 External Validity

External validity refers to the extent to which inferences drawn from a study's sample apply to a broader or other target populations, therefore external validity threats are related to how the presented results can be generalized. This study focused on scripted GUI testing of web apps, in particular, those cases using the Selenium tool. All the provided guidelines remain valid for any other scripted GUI testing disciplines, regardless of the tool used; some of the recommendations can also be applied to the general testing activity.

Concerning the actual implementation of the linter, we already suggested the possibility to extend the core of the plugin with libraries providing the extraction of AST or CST from the starting code, which would make the linter compatible with any other programming language with the aid of the provided library. The current implementation is specific for VS Code IDE, we can estimate a low effort to adapt the plugin as it stands to other IDEs which support Javascript built on top of Node.js framework. The development of the plugin for different IDEs which are not compatible with Node.js will certainly require significant effort, which is currently out of the scope of the research.

Other validity threats are represented by the method used for the selection of repositories to test the linter against. A more rigorous and thorough study should take into consideration a statistically significant sample of repositories to provide a complete overview of the state of the practice. The employed search string should also be assessed and enhanced iteratively, to ensure that the most relevant and representative open-source projects are included and more reliable statistics are collected.

7 CONCLUSION AND FUTURE WORK

With the present work, we provided an investigation into the state of the practice of web applications locator-based GUI testing, from the point of view of GUI test fragility.

We analyzed and classified the most common test smells affecting scripted GUI testing techniques, as reported by practitioners. The provided classification enriches the taxonomy of test smells already existing in the literature – as of the systematic mapping from Garousi et al. [6] – with some novel GUI-related issues. We provided a discussion for each test smell, along with suggestions and guidelines about the potential drawbacks of smelly code and guidance on how to avoid anti-patterns. On the basis of the presented guidelines, we built a prototypal static code analyzer in the form of a plugin for VS Code. The tool is able to detect test smells in Java and Javascript languages and notify them to the tester.

We evaluated the linter on 30 different open-source Github projects, the results showed that the most violated guideline both in Java and Javascript projects was by far the usage of global variables in test cases, followed by the usage of the discouraged Xpath locator, instead of the recommended locators based on ids or CSS selectors.

Short term improvements to the tool include the completion of the linter: we will work toward the recognition of the unimplemented test smells and the inclusion of Python, which is the second most used general purpose programming language [12]. Once the tool will be completed in these respects, we plan to carry on a systematic execution against a larger number of open-source repositories – not limited to the GitHub platform – to further validate the tool's capabilities and investigate the actual diffusion of GUI test smells.

REFERENCES

[1] Emil Alégroth, Arvid Karlsson, and Alexander Radway. 2018. Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 172–181. https://doi.org/10.1109/ICST.2018.00026

[2] Luca Ardito, Morisio Maurizio, and Huang Shijie. 2019. Test Fragility: An exploratory assessment study on an Open-Source Web Application. (2019). https://webthesis.biblio.polito.it/14471/1/tesi.pdf

[3] Pierre Carbonnelle. 2022. Top IDE index. https://pypl.github.io/IDE.html Accessed: 2021-11-14.

[4] Mike Cohn. 2010. *Succeeding with agile: software development using Scrum*. Pearson Education.

[5] Riccardo Coppola, Maurizio Morisio, Marco Torchiano, and Luca Ardito. 2019. Scripted GUI testing of Android open-source apps: evolution of test code and fragility causes. *Empirical Software Engineering* 24, 5 (2019), 3205–3248.

[6] Vahid Garousi and Barıs Kucuk. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software* 138 (2018), 52–81. https://doi.org/10.1016/j.jss.2017.12.013

[7] A. Holmes and M. Kellogg. 2006. Automating functional tests using Selenium. In *AGILE 2006 (AGILE'06)*. 6 pp.–275. https://doi.org/10.1109/AGILE.2006.19

[8] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. 2013. Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 108–113. https://doi.org/10.1109/ICSTW.2013.19

[9] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-Based Web Locators: An Empirical Study. In *Web Engineering*, Sven Casteleyn, Gustavo Rossi, and Marco Winckler (Eds.). Springer International Publishing, Cham, 322–340.

[10] Fahad M. Alammar, Ali Intezari, Andrew Cardow, and David J. Pauleen. 2019. Grounded theory in practice: Novice researchers' choice between Straussian and Glaserian. *Journal of Management Inquiry* 28, 2 (2019), 228–245.

[11] Delin Mathew and Konrad Foegen. 2016. An analysis of information needs to detect test smells. *Full-scale Software Engineering/Current Trends in Release Engineering* (2016), 19.

[12] Stack OVerflow. 2021. Stack Overflow's 2021 survey. https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies Accessed: 2022-04-9.

[13] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2014. PESTO: A Tool for Migrating DOM-Based to Visual Web Tests. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 65–70. https://doi.org/10.1109/SCAM.2014.36

[14] Selenium Development Team. 2021. Generating application state | Selenium. https://www.selenium.dev/documentation/test_practices/encouraged/generating_application_state/ Accessed: 2022-04-13.

[15] Jess Tyndall. 2010. The AACODS Checklist. https://dspace.flinders.edu.au/xmlui/bitstream/handle/2328/3326/AACODS_Checklist.pdf.

[16] Maike Vollstedt and Sebastian Rezat. 2019. *An Introduction to Grounded Theory with a Special Focus on Axial Coding and the Coding Paradigm*. Springer International Publishing, 81–100. https://doi.org/10.1007/978-3-030-15636-7_4

OTHER SOURCES

[S01] B. Williams. Improving code quality, 2021. URL https://github.com/uselagoon/lagoon/discussions/2613. Accessed: 2022-02-23.

[S02] M. F. P. S. Cyreno. Why would you use id attributes, 2017. URL https://github.com/manoelcyreno/test-samples/wiki/Why-would-you-use-ID-attributes. Accessed: 2022-02-23.

[S03] ejunker. Is adding ids to everything standard practice when using selenium?, 2013. URL https://sqa.stackexchange.com/questions/6326/is-adding-ids-to-everything-standard-practice-when-using-selenium. Accessed: 2022-02-23.

[S04] S. Developer. Finding web elements, 2022. URL https://www.selenium.dev/documentation/webdriver/elements/finders/. Accessed: 2022-02-23.

[S05] G. Team. Google html/css style guide, 2021. URL https://google.github.io/styleguide/htmlcssguide.html. Accessed: 2022-02-23.

[S06] H. Schneider. pycon-ca-2018, 2018. URL https://github.com/howard8888/pycon-ca-2018/wiki. Accessed: 2022-02-23.

[S07] S. K. Kyle Martin, Kevyn Bruyere. Node.js best practices, 2022. URL https://github.com/goldbergyoni/nodebestpractices. Accessed: 2022-02-23.

[S08] S. Developer. Overview of test automation. https://www.selenium.dev/documentation/test_practices/overview/, 2021. Accessed: 2022-02-23.

[S09] C. Naranjo. Javascript namespace declaration, 2013. URL https://github.com/freudgroup/freudcs/wiki/Javascript-Namespace-Declaration. Accessed: 2022-02-23.

[S10] S. Developer. Avoid sharing state. https://www.selenium.dev/documentation/test_practices/encouraged/avoid_sharing_state/, 2021. Accessed: 2022-02-23.

[S11] S. Developer. Fresh browser per test. https://www.selenium.dev/documentation/test_practices/encouraged/fresh_browser_per_test/, 2021. Accessed: 2022-02-23.

[S12] S. Developer. Generating application state. https://www.selenium.dev/documentation/test_practices/encouraged/generating_application_state/, 2021. Accessed: 2022-02-23.

[S13] S. Developer. Mock external services. https://www.selenium.dev/documentation/test_practices/encouraged/mock_external_services/, 2021. Accessed: 2022-02-23.

[S14] S. Developer. Test independency. https://www.selenium.dev/documentation/test_practices/encouraged/test_independency/, 2021. Accessed: 2022-02-23.

[S15] D. Zivanovic. Automation in selenium: Page object model and page factory, 2016. URL https://www.toptal.com/selenium/test-automation-in-selenium-using-page-object-model-and-page-factory. Accessed: 2022-02-23.

[S16] N. Advolodkin. Automation best practices w/ java workshop, 2021. URL https://github.com/nadvolod/automation-best-practices-java/blob/main/README.md#local-environment-setup. Accessed: 2022-02-23.

[S17] M. W. Docs. Setting up your own test automation environment. https://github.com/mdn/content/blob/main/files/en-us/learn/tools_and_testing/cross_browser_testing/your_own_automation_environment/index.md, 2021. Accessed: 2022-02-23.

[S18] G. Karadas. Selenium best practices, 2017. URL https://github.com/previousdeveloper/Selenium-best-practices. Accessed: 2022-02-23.

[S19] A. BM. Selenium best practices, 2015. URL https://gist.github.com/arjunbm13/42f8a1fcf9b2f8ca8599. Accessed: 2022-02-23.

[S20] D. Garg. Selenium code practice, 2021. URL https://github.com/DipanGarg/Selenium-Code-Practice. Accessed: 2022-02-23.

[S21] devonfw. Selenium best practices, 2020. URL https://github.com/devonfw/mrchecker/blob/develop/documentation/Who-Is-MrChecker/Test-Framework-Modules/Selenium-Test-Module-Selenium-Best-Practices.asciidoc. Accessed: 2022-02-23.

[S22] J. Unadkat. Best practices for selenium test automation, 2021. URL https://www.browserstack.com/guide/best-practices-in-selenium-automation. Accessed: 2022-02-23.

[S23] E. Nogueira. Lean test automation architecture using java and selenium webdriver, 2021. URL https://github.com/eliasnogueira/selenium-java-lean-test-architecture. Accessed: 2022-02-23.

[S24] M. Gibbs. Aspect oriented programming: Definition & concepts, 2022. URL https://study.com/academy/lesson/aspect-oriented-programming-definition-concepts.html.