Visual vs. DOM-based Web Locators: An Empirical Study

Maurizio Leotta, Diego Clerissi, Filippo Ricca, Paolo Tonella

Abstract:

Automation in Web testing has been successfully supported by DOM-based tools that allow testers to program the interactions of their test cases with the Web application under test. More recently a new generation of visual tools has been proposed where a test case interacts with the Web application by recognising the images of the widgets that can be actioned upon and by asserting the expected visual appearance of the result.

In this paper, we first discuss the inherent robustness of the locators created by following the visual and DOM-based approaches and we then compare empirically a visual and a DOM-based tool, taking into account both the cost for initial test suite development from scratch and the cost for test suite maintenance during code evolution. Since visual tools are known to be computationally demanding, we also measure the test suite execution time.

Results indicate that DOM-based locators are generally more robust than visual ones and that DOM-based test cases can be developed from scratch and evolved at lower cost. Moreover, DOM-based test cases require a lower execution time. However, depending on the specific features of the Web application under test and its expected evolution, in some cases visual locators might be the best choice (e.g., when the visual appearance is more stable than the structure).

Digital Object Identifier (DOI):

http://dx.doi.org/10.1007/978-3-319-08245-5_19

Visual vs. DOM-based Web Locators: An Empirical Study

Maurizio Leotta[1], Diego Clerissi[1], Filippo Ricca[1], Paolo Tonella[2]

[1] DIBRIS, Università di Genova, Italy
[2] Fondazione Bruno Kessler, Trento, Italy
maurizio.leotta@unige.it, diego.clerissi@gmail.com,
filippo.ricca@unige.it, tonella@fbk.eu

Abstract. Automation in Web testing has been successfully supported by DOM-based tools that allow testers to program the interactions of their test cases with the Web application under test. More recently a new generation of visual tools has been proposed where a test case interacts with the Web application by recognising the images of the widgets that can be actioned upon and by asserting the expected visual appearance of the result.
In this paper, we first discuss the inherent robustness of the locators created by following the visual and DOM-based approaches and we then compare empirically a visual and a DOM-based tool, taking into account both the cost for initial test suite development from scratch and the cost for test suite maintenance during code evolution. Since visual tools are known to be computationally demanding, we also measure the test suite execution time.
Results indicate that DOM-based locators are generally more robust than visual ones and that DOM-based test cases can be developed from scratch and evolved at lower cost. Moreover, DOM-based test cases require a lower execution time. However, depending on the specific features of the Web application under test and its expected evolution, in some cases visual locators might be the best choice (e.g., when the visual appearance is more stable than the structure).

1 Introduction

The importance of test automation in Web engineering comes from the widespread use of Web applications (Web apps) and the associated demand for code quality. Test automation is considered crucial for delivering the quality levels expected by users [14], since it can save a lot of time in testing and it helps developers to release Web apps with fewer defects [1]. The main advantage of test automation comes from fast, unattended execution of a set of tests after some changes have been made to a Web app.

Several approaches can be employed to automate functional Web testing. They can be classified using two main criteria: the first concerns how test cases are developed, while, the second concerns how test cases localize the Web elements (i.e., GUI components) to interact with, that is what kind of *locators* (i.e., objects that select the target web elements) are used. Concerning the first criterion, it is possible to use the capture-replay or the programmable approach. Concerning the second criterion, there are three main approaches, Visual (where image recognition techniques are used to *locate* GUI

components and a locator consists of an image), DOM-based (where Web page elements are *located* using the information contained in the Document Object Model and a locator is, for instance, an XPath expression) and Coordinates-based (where screen coordinates of the Web page elements are used to interact with the Web app under test). This categorization will be deeply analysed in the next section.

For developers and project managers it is not easy to select the most suitable automated functional web testing approach for their needs among the existing ones. For this reason, we are carrying out a long term research project aimed at empirically investigating the strengths and weaknesses of the various approaches (see also our previous work [9]).

In this work we evaluate and compare the visual and DOM-based approaches considering: the robustness of *locators*, the initial test suite development effort, the test suite evolution cost, and the test suite execution time. Our empirical assessment of the robustness of locators is quite general and tool independent, while the developers' effort for initial test suite development and the effort for test suite evolution were measured with reference to specific implementations of the two approaches. We instantiated such analysis for two specific tools, Sikuli API and Selenium WebDriver, both adopting the programmable approach but differing in the way they localize the Web elements to interact with during the execution of the test cases. Indeed, Sikuli API adopts the visual approach, thus using images representing portions of the Web pages, while Selenium WebDriver employs the DOM-based approach, thus relying on the HTML structure. We selected six open source Web apps and for each tool, we first developed a test suite per application and then we evolved them to a subsequent version. Moreover, since visual tools are known to be computational demanding, we also measured and compared the test suite execution time.

The paper is organized as follows: Sect. 2 gives some background on test case development using the visual and the programmable approaches, including examples for the two specific tools used in this work. In the same section, we describe test case repair activities. Sect. 3 describes our empirical study, reports the obtained results and discusses the pros and cons of the two considered approaches. We then present the related works (Sect. 4), followed by conclusions and future work (Sect. 5).

2 Background

There are several approaches for functional Web testing [13] and the choice among them depends on a number of factors, including the technology used by the Web app and the tools (if any) used for Web testing. Broadly speaking, there are two main criteria to classify the approaches to functional Web testing that are related to: (1) test case construction; and, (2) Web page element localisation.

For what concerns the first criterion, we can find two main approaches:

1) *Capture-Replay (C&R) Web Testing*: this approach consists of recording the actions performed by the tester on the Web app GUI and generating a script that provides such actions for automated, unattended re-execution.
2) *Programmable Web Testing*: this approach aims at unifying Web testing with traditional testing, where test cases are themselves software artefacts that developers write, with the help of specific testing frameworks. For Web apps, this means that the

framework supports programming of the interaction with a Web page and its elements, so that test cases can, for instance, automatically fill-in and submit forms or click on hyperlinks.

An automated functional test case interacts with several Web page elements such as links, buttons, and input fields, and different methods can be employed to locate them. Thus, concerning the second criterion, we can find three different cases[1]:

1) *Coordinate-based localisation:* first generation tools just record the screen coordinates of the Web page elements and then use this information to locate the elements during test case replay. This approach is nowadays considered obsolete, because it produces test cases that are extremely fragile.
2) *DOM-based localisation:* second generation tools locate the Web page elements using the information contained in the Document Object Model. For example, the tools Selenium IDE and WebDriver employ this approach and offer several different ways to locate the elements composing a Web page (e.g., ID, XPath and LinkText).
3) *Visual localisation:* third generation tools have emerged recently. They make use of image recognition techniques to identify and control GUI components. The tool Sikuli API belongs to this category.

In our previous work [9], we compared the capture-replay approach and the programmable approach using two 2nd generation tools: Selenium IDE and Selenium WebDriver. In this work, we fixed the test case definition method (i.e., programmable) and changed the locator type, with the aim of comparing the visual approach and the DOM-based approach. Empirical results refer to two specific programmable tools: Sikuli API and Selenium WebDriver.

Let us consider a running example, consisting of a typical login web page (login.asp). The login page requires the users to enter their credentials, i.e., *username* and *password* (see Fig. 1). After having inserted the credentials and clicked on "Login", the application shows the home page (homepage.asp). If credentials are correct, the username (contained in an HTML tag with the attribute `ID="uname"`) and the logout button are reported in the upper right corner of the home page (e.g., *John.Doe*). Otherwise, *Guest User* and login button are shown. For the sake of simplicity, the application does not report any error message in case of invalid credentials or unrecognised users.

2.1 Programmable Web Testing

Programmable Web testing is based on manual creation of a test script. Web test scripts can be written using ad-hoc languages and frameworks or general purpose programming languages (such as Java and Ruby) with the aid of specific libraries able to play the role of the browser. Usually, these libraries extend the programming language with user friendly APIs, providing commands to, e.g., click a button, fill a field and submit a form.

[1] http://jautomate.com/2013/08/22/730/

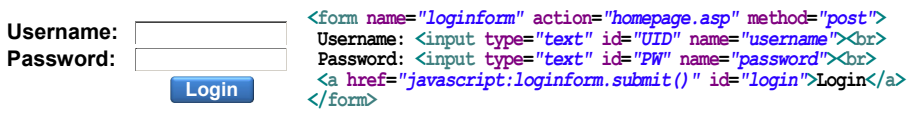

Fig. 1. login.asp – Page and Source

```
public class LoginPage {
 private final WebDriver driver;
 public LoginPage(WebDriver driver) {this.driver=driver;}
 public HomePage login(String UID, String PW) {
  driver.findElement(By.id("UID")).sendKeys(UID);
  driver.findElement(By.xpath("./input[2]")).sendKeys(PW);
  driver.findElement(By.linkText("Login")).click();
  return new HomePage(driver);
 }
}
```

```
public class HomePage {
 private final WebDriver driver;
 public HomePage(WebDriver driver)
   {this.driver = driver;}
 public String getUsername() {
  return
  driver.findElement(By.id("uname")).getText;
 }
}
```

Fig. 2. LoginPage and HomePage page objects in Selenium WebDriver

Test scripts are completed with assertions (e.g., JUnit assertions if the language chosen is Java).

A best practice often used when developing programmable test cases is the *page object* pattern. This pattern is used to model the Web pages involved in the test process as objects, employing the same programming language used to write the test cases. In this way, the functionalities offered by a Web page become methods exposed by the corresponding page object, which can be easily called within any test case. Thus, all the details and mechanics of the Web page are encapsulated inside the page object. Adopting the *page object* pattern allows the test developer to work at a higher level of abstraction and it is used to reduce the coupling between Web pages and test cases, and the amount of duplicate code. For these reasons, the adoption of the *page object* pattern is expected to improve the test suite maintainability and evolvability [8].

DOM-based Programmable Test Case Creation: The tool for Web app testing belonging to the DOM-based/Programmable category that we used in this work is Selenium WebDriver release 2.25.0 (in the following shortly referred to as WebDriver - http://seleniumhq.org/projects/webdriver/). WebDriver is a tool for automating Web app testing that provides a comprehensive programming interface used to control the browser. WebDriver test cases are implemented manually in a programming language (in our case Java) integrating WebDriver commands with JUnit or TestNG assertions. We chose WebDriver as the representative of this category, because: (1) it is a quite mature tool, (2) it is open-source, (3) it is one of the most widely-used open-source solutions for Web test automation (even in the industry), (4) during a previous industrial collaboration [8], we have gained a considerable experience on its usage.

As an example, we here use a simple WebDriver test case (Fig. 3 left): a successful authentication test case. It submits a valid login, using correct credentials (i.e., *username=John.Doe* and *password=123456*) and verifies that in the home page the user appears as correctly authenticated ("John.Doe" must be displayed in the top-right corner of the home page).

The first step is to create two page objects (LoginPage.java and HomePage.java) corresponding to the Web pages login.asp and homepage.asp respectively (see Fig. 2). The page object LoginPage.java offers a method to log into the application. This method takes username and password as inputs, inserts them in the corresponding input fields, clicks the Login button and returns a page object of kind HomePage.java, because after clicking the Login button the application moves to the page homepage.asp. HomePage.java contains a method that returns the username authenticated in the application or *"Guest"* when no user is authenticated. As shown in Fig. 2, the Web page elements are located

```
public void testLogin() { // WebDriver
 WebDriver driver = new FirefoxDriver();
 // we start from the 'login.asp' page
 driver.get("http://www.....com/login.asp");
 LoginPage LP = new LoginPage(driver);
 HomePage HP = LP.login("John.Doe","123456");
 // we are in the 'homepage.asp' page
 assertEquals("John.Doe", HP.getUsername());
}
```

```
public void testLogin(){ // Sikuli

 // we start from the 'login.asp' page
 CommonPage.open("http://www.....com/login.asp");
 LoginPage LP = new LoginPage();
 HomePage HP = LP.login("John.Doe", "123456");
 // we are in the 'homepage.asp' page
 assertTrue(HP.isUsernamePresent(new URL("JohnDoe.png")));
}
```

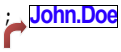

Fig. 3. TestLogin test case in Selenium WebDriver (left) and in Sikuli API (right)

by searching for values in the DOM (using ID and LinkText locators) or navigating it (using XPath locators). While WebDriver offers several alternative ways to locate the Web elements in a Web page, the most effective one, according to WebDriver developers (http://docs.seleniumhq.org/docs/03_webdriver.jsp), is searching the elements by their ID values. Hence, whenever possible, we used this method. The second step is to develop the test case making use of the page objects (see Fig. 3 left). In the test method, first, a WebDriver of type FirefoxDriver is created, so that the test case can control a Firefox browser as a real user does; second, the WebDriver (i.e., the browser) opens the specified URL and creates a page object that instantiates LoginPage.java (modelling the login.asp page); third, using method login(...) offered by the page object, a new page object (HP) representing the page homepage.asp is created; finally, the test case assertion is checked.

Visual Programmable Test Case Creation: The Web app testing tool, belonging to the Visual/Programmable category, that we used in this work is Sikuli API release 1.0.2 (in the following shortly referred to as Sikuli - http://code.google.com/p/sikuli-api/). Sikuli is a visual technology able to automate and test graphical user interfaces using screenshot images. It provides image-based GUI automation functionalities to Java programmers. We chose Sikuli as the representative of this category mainly because: (1) it is open-source and (2) it is similar to WebDriver, thus, we can create test cases and page objects similarly to the ones produced for WebDriver. In this way, using Sikuli, we are able to make the comparison between visual and DOM-based programmable tools fair and focused as much as possible on the differences of the two approaches. In fact, in this way we can use the same programming environment: programming language (Java), IDE (Eclipse), and testing framework (JUnit). Sikuli allows software testers to write scripts based on images that define the GUI widgets to be tested and the assertions to be checked. This is substantially different from the way in which WebDriver performs page element localisation and assertion checking.

As an example, the Sikuli version of the `testLogin` test case is shown in Fig. 3 (right) while the related page objects are given in Fig. 4. The test case developed in Sikuli performs the same conceptual steps[2] as the WebDriver test case, as apparent from Fig. 3 (left) and Fig. 3 (right). The page objects (shown in Fig. 4) are instead quite different. To locate a Web page element, an instruction (based on the Sikuli Java API) is used which searches for the portion of Web page that looks like the image saved for the test suite

[2] Actually, in Sikuli there is no command to open Firefox at a specified URL as in WebDriver. We have encapsulated this functionality in a method, `CommonPage.open(...)`, that clicks the Firefox icon on the desktop, inserts the URL into the address bar and then clicks on the "go" arrow.

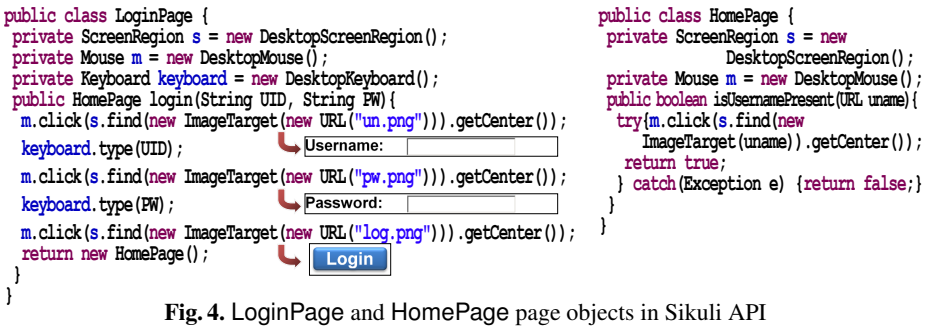

Fig. 4. LoginPage and HomePage page objects in Sikuli API

(e.g., in a png file). Thus, in Sikuli, locators are always images. In addition, some other minor differences can be noticed in the test case implementation. For instance, in the case of WebDriver it is possible to assert that an element must contain a certain text (see the last line in Fig. 3 (left)), while in Sikuli it is necessary to assert that the page shows a portion equal to an image where the desired text is displayed (see the last line in Fig. 3 (right)).

2.2 Test Case Evolution

When a Web app evolves to accommodate requirement changes, bug fixes, or functionality extensions, test cases may become broken. For example, test cases may be unable to locate some links, input fields and submission buttons and software testers will have to repair them. This is a tedious and expensive task since it has to be performed manually (automatic evolution of test suites is far from being consolidated [11]).

Depending on the kind of maintenance task that has been performed on the target Web app, a software tester has to execute a series of test case repairment activities that can be categorised, for the sake of simplicity, in two types: *logical* and *structural*.

Logical Changes involve the modification of the Web app functionality. To repair the test cases, the tester has to modify the broken test cases and the corresponding page objects and in some cases, new page objects have to be created. An example of a change request that needs a logical repairment activity is enforcing the security by means of stronger authentication and thus adding a new Web page, containing an additional question, after the login.asp page of Fig. 1. In this case, the tester has to create a new page object for the Web page providing the additional authentication question. Moreover, she has to repair both the `testLogin` test cases shown in Fig. 3, adding a new Java command that calls the method offered by the new page object.

Structural Changes involve the modification of the Web page layout/structure only. For example, in the Web page of Fig. 1 the string of the login button may be changed to Submit. Usually, the impact of a structural change is smaller than a logical change. To repair the test cases, often, it is sufficient to modify one or more localisation lines, i.e., lines containing locators. In the example, the tester has to modify the LoginPage.java page object (see Fig. 2 and 4) by: (1) repairing the line:

```
driver.findElement(By.linkText("Login")).click()
```

in the case of Selenium WebDriver; or, (2) changing the image that represents the Login button in the case of Sikuli.

3 Empirical Study

This section reports the design, objects, research questions, metrics, procedure, results, discussion and threats to validity of the empirical study conducted to compare visual vs. DOM-based Web testing.

3.1 Study Design

The primary *goal* of this study is to investigate the difference in terms of robustness (if any) that can be achieved by adopting visual and DOM-based locators with the purpose of understanding the strengths and the weaknesses of the two approaches. Then, after having selected two tools that respectively belong to the two considered categories, as secondary *goal* we investigated the cost/benefit trade-off of visual vs. DOM-based test cases for Web apps. In this case, the *cost/benefit focus* regards the effort required for the creation of the initial test suites from scratch, as well as the effort required for their evolution across successive releases of the software. The results of this study are interpreted according to two *perspectives*: (1) *project managers*, interested in understanding which approach could lead to potentially more robust test cases, and in data about the costs and the returns of the investment associated with both the approaches; (2) *researchers*, interested in empirical data about the impact of different approaches on Web testing. The *context* of the study is defined as follows: two *human subjects* have been involved, a PhD student (the first author of this paper) and a junior developer (the second author, a master student with 1-year industrial experience as software tester); the *software objects* are six open source Web apps. The two human subjects participating in the study are referred below using the term "developers".

3.2 Web Applications

We have selected and downloaded six open-source Web apps from *SourceForge.net*. We have included only applications that: (1) are quite recent, so that they can work without problems on the latest versions of Apache, PHP and MySQL, technologies we are familiar with (actually, since Sikuli and WebDriver implement a black-box approach, the server side technologies do not affect the results of the study); (2) are well-known and used (some of them have been downloaded more than one hundred thousand times last year); (3) have at least two major releases (we have excluded minor releases because with small differences between versions the majority of the test cases are expected to work without problems); (4) belong to different application domains; and, (5) are non-RIA – Rich Internet Applications (to make the comparison fair, since RIAs can be handled better by the visual approach, see Sect. 3.7).

Table 1 reports some information about the selected applications. We can see that all of them are quite recent (ranging from 2008 to 2013). On the contrary, they are considerably different in terms of number of source files (ranging from 46 to 840) and number of lines of code (ranging from 4 kLOC to 285 kLOC, considering only the lines of code contained in the PHP source files, comments and blank lines excluded). The difference in lines of code between 1st and 2nd release (columns 7 and 11) gives a rough idea of how different the two chosen releases are.

Table 1. OBJECTS: WEB APPLICATIONS FROM *SourceForge.net*

	Description	Web Site	1st Release				2nd Release			
			Vers.	Date	File[a]	kLOC[b]	Vers.	Date	File[a]	kLOC[b]
MantisBT	bug tracking system	sourceforge.net/projects/mantisbt/	1.1.8	2009	492	90	1.2.0	2010	733	115
PPMA[c]	password manager	sourceforge.net/projects/ppma/	0.2	2011	93	4	0.3.5.1	2013	108	5
Claroline	learning environment	sourceforge.net/projects/claroline/	1.10.7	2011	840	277	1.11.5	2013	835	285
Address Book	address/phone book	sourceforge.net/projects/php-addressbook/	4.0	2009	46	4	8.2.5	2012	239	30
MRBS	meeting rooms manager	sourceforge.net/projects/mrbs/	1.2.6.1	2008	63	9	1.4.9	2012	128	27
Collabtive	collaboration software	sourceforge.net/projects/collabtive/	0.65	2010	148	68	1.0	2013	151	73

[a] Only PHP source files were considered [b] PHP LOC - Comment and Blank lines are not considered

3.3 Research Questions and Metrics

Our empirical study aims at answering the following research questions, split between considerably tool-independent (A) and significantly tool-dependent (B) questions:

RQ A.1: *Do Visual and DOM-based test suites require the same number of locators?*
The goal is to quantify and compare the number of locators required when adopting the two different approaches. This would give developers and project managers a rough idea of the inherent effort required to build the test suites by following the two approaches. Moreover, the total number of locators could influence also the maintenance effort, since the more the locators are, the more the potential locators to repair could be. The metrics used to answer the research question is the number of created locators.

RQ A.2: *What is the robustness of visual vs. DOM-based locators?*
The goal is to quantify and compare the robustness of the visual and the DOM-based locators. This would give developers and project managers an idea of the inherent robustness of the locators created by following the two approaches. The metrics used to answer this research question is the number of broken locators.

RQ B.1: *What is the initial development effort for the creation of visual vs. DOM-based test suites?*
The goal is to quantify and compare the development cost of visual and DOM-based tests. This would give developers and project managers an idea of the initial investment (tester training excluded) to be made if visual test suites are adopted, as compared to DOM-based test suites. The metrics used to answer this research question is the time (measured in minutes) the two developers spent in developing visual test suites vs. DOM-based test suites.

RQ B.2: *What is the effort involved in the evolution of visual vs. DOM-based test suites when a new release of the software is produced?*
This research question involves a software evolution scenario. For the next major release of each Web app under test, the two developers evolved the test suites so as to make them applicable to the new software release. The test case evolution effort for visual and for DOM-based test suites was measured as the time (measured in minutes) spent to update the test suites, until they were all working on the new release.

RQ B.3: *What is the execution time required by visual vs. DOM-based test suites?*
Image processing algorithms are known to be quite computation-intensive [2] and execution time is often reported as one of the weaknesses of visual testing. We want to quantitatively measure the execution time difference (in seconds) between visual and DOM-based test execution tools.

It should be noticed that the findings for research questions A.x are mostly influenced by the approaches adopted, independently of the tools that implement them, since the

number of (DOM-based/visual) locators and the number of broken locators depend mostly on the test cases (and on the tested Web app), not on the tools. On the other hand, the metrics for research questions B.x (effort and execution time) are influenced by the specific tools adopted in the empirical evaluation.

3.4 Experimental Procedure

The experiment has been performed as follows:

– Six open-source Web apps have been selected from *SourceForge.net* as explained in Section 3.2.

– For each selected application, two equivalent test suites (written for Sikuli and WebDriver) have been built by the two developers, working in pair-programming and adopting a systematic approach consisting of three steps: (1) the main functionalities of the target Web app are identified from the available documentation; (2) each discovered functionality is covered with at least one test case (developers have assigned a meaningful name to each test case, so as to keep the mapping between test cases and functionalities); (3) each test case is implemented with Sikuli and WebDriver. For both approaches, we considered the following best practices: (1) we used the page object pattern and (2) we preferred, for WebDriver, the ID locators when possible (i.e., when HTML tags are provided with IDs), otherwise Name, LinkText, CSS and XPath locators were used following this order. Overall, for the WebDriver case we created: 82 ID, 99 Name, 65 LinkText, 64 CSS and 177 XPath locators. For each test suite, we measured the number of produced locators (to answer RQ A.1) and the development effort for the implementation as clock time (to answer RQ B.1). Each pair of test suites (i.e., visual and DOM-based) are equivalent because the included test cases test exactly the same functionalities, using the same sequences of actions (e.g., locating the same web page elements) and the same input data. The WebDriver test suites had been created by the same two developers about a year ago during a previous case study [9], while the Sikuli test suites have been created more recently, for the present study, which potentially gives a slight advantage to Sikuli (see *Threats to Validity* section).

– Each test suite has been executed against the second release of the Web app (see Table 1). First, we recorded the failed test cases (we highlight that no real bugs have been detected in the considered applications; all the failures are due to broken locators and minimally to modifications to the test cases logic) and then, in a second phase, we repaired them. We measured the number of broken locators (to answer RQ A.2) and the repair effort as clock time (to answer RQ B.2). Finally, to answer RQ B.3 we executed 10 times (to average over any random fluctuation of the execution time) each of the 12 repaired test suites (both WebDriver and Sikuli test cases) and recorded the execution times. We executed the test suites on a machine hosting an Intel Core i5 dual-core processor (2.5 GHz) and 8 GB RAM, with no other computation or communication load, in order to avoid CPU or memory saturation. To avoid as much as possible network delays we installed the web server hosting the applications on a machine belonging to the same LAN.

– The results obtained for each test suite have been compared to answer our research questions. On the results, we conducted both a quantitative analysis and a qualitative analysis, completed with a final discussion where we report our lessons learnt. The test suites source code can be found at: http://softeng.disi.unige.it/2014-Visual-DOM.php

3.5 Quantitative Results

This section reports the quantitative results of the empirical study, while the reasons and implications of the results are further analysed in Section 3.6.

RQ A.1. Table 2 shows the data to answer the A.x research questions by reporting, for each application, the number of visual and DOM-based broken/total locators. The number of locators required to build the test suites varies from 81 to 158 when adopting the visual approach and from 42 to 126 when adopting the DOM-based one. For all the six applications the number of required locators is higher when adopting the visual approach. Considering the data in aggregate form, we created 45% more Visual locators than DOM locators (706 visual vs. 487 DOM-based locators). To summarise, with respect to the research question RQ A.1, the visual approach has always required to create a higher number of locators.

RQ A.2. From the data shown in Table 2, we can see that in only two test suites out of six visual locators result more robust than the DOM-based ones (i.e., Mantis and Collabtive), while in the remaining four cases the DOM-based locators are more robust. Overall, 312 visual locators out of 706 result broken, while only 162 DOM-based locators out of 487 have been repaired (i.e., 93% more broken locators in the case of the visual approach). To summarise, with respect to RQ A.2, the result is not clear-cut. Generally, DOM-based locators are more robust but in certain cases (i.e., depending on the kind of modifications among the considered releases), visual locators proved to be the most robust (e.g., in Collabtive only 4 broken visual locators vs. 36 DOM-based ones).

RQ B.1. Table 3 reports some data about the developed test suites. For each application, it reports: the number of test cases and page objects in the test suites built for the newer release of each application (c. 1- 2); the time required for the initial development of the test suites (c. 3- 4); the percentage difference between the initial development time required by WebDriver vs. Sikuli (c. 5); the p-value of the Wilcoxon paired test used to assess whether the development time difference is statistically significant (c. 6) and finally the test suites size (measured in Lines Of Code (LOC), comment and blank lines have not been not considered), split between page objects and test cases, for the newer release of each application (c. 7-12). The development of the Sikuli test suites required from 229 to 498 minutes, while the WebDriver suites required from 98 to 383 minutes. In all the six cases, the development of the WebDriver test suites required less time than the Sikuli test suites (from 22% to 57%). This is related with the lower number of locators required when adopting the DOM-based approach (see RQ A.1). According to the Wilcoxon paired test (see c. 6 of Table 3), the difference in test suite development time between Sikuli and WebDriver is statistically significant (at $\alpha = 0.05$) for all test suites. For what concerns the size of the test suites (Table 3, c. 9, *Total*), we can notice that in all the cases, the majority of the code is devoted to the test case logics, while only

Table 2. VISUAL VS. DOM-BASED LOCATORS ROBUSTNESS

	Visual		DOM-Based	
	Broken Locators	Total Locators	Broken Locators	Total Locators
MantisBT	15	127	29	106
PPMA	78	81	24	42
Claroline	56	158	30	126
Address Book	103	122	14	54
MRBS	56	83	29	51
Collabtive	4	135	36	108

Table 3. TEST SUITES DEVELOPMENT

	Number of		Time (Minutes)				Code (Java LOC)					
	Test Cases	Page Objects	Sikuli API	Web Driver		p-value	Sikuli API			WebDriver		
							Test	PO	Total	Test	PO	Total
MantisBT	41	30	498	383	-23%	**< 0.01**	1645	1291	**2936**	1577	1054	**2631**
PPMA	23	6	229	98	-57%	**< 0.01**	958	589	**1547**	867	346	**1213**
Claroline	40	22	381	239	-37%	**< 0.01**	1613	1267	**2880**	1564	1043	**2607**
Address Book	28	7	283	153	-46%	**< 0.01**	1080	686	**1766**	1078	394	**1472**
MRBS	24	8	266	133	-50%	**< 0.01**	1051	601	**1652**	949	372	**1321**
Collabtive	40	8	493	383	-22%	**< 0.01**	1585	961	**2546**	1565	650	**2215**

a small part is devoted to the implementation of the page objects. Moreover, the number of LOCs composing the test cases is very similar for both Sikuli and WebDriver, while it is always smaller in WebDriver for what concerns the page objects (often, in the Sikuli page objects, two lines are needed to locate and perform an action on a web element while in WebDriver just one is sufficient, see for example Fig. 2 and 4). To summarise, with respect to the research question RQ B.1 we can say that for all the six considered applications, the effort involved in the development of the Sikuli test suites is higher than the one required by WebDriver.

RQ B.2. Table 4 shows data about the test suites repairing process. In detail, the table reports, for each application, the time required to repair the test suites (Sikuli and WebDriver), and the number of repaired test cases over the total number of test cases. The WebDriver repair time is compared to the Sikuli repair time by computing the percentage difference between the two and by running the Wilcoxon paired test, to check for statistical significance of the difference. Sikuli test suites required from 7 to 126 minutes to be repaired, while WebDriver test suites required from 46 to 95 minutes. The results are associated with the robustness of the two kinds of locators (RQ A.2) employed by the two tools and thus follow the same trend: in four cases out of six, repairing of the WebDriver test suites required less time (from 33% to 57% less) than Sikuli. In one case (i.e., MantisBT), the WebDriver test suite required slightly more time (25% more) to be repaired than the corresponding Sikuli test suite. In another case (i.e., Collabtive), WebDriver required a huge amount of time for test suite repairment with respect to the time required by Sikuli (about 10x more time with WebDriver where, as seen before, we have about 10x more locators to repair). According to the Wilcoxon paired test, the difference in test suite evolution time between Sikuli and WebDriver is statistically significant (at $\alpha = 0.05$) for all test suites (sometimes in opposite directions) except for Claroline (see Table 4). Note that, the maintenance effort is almost entirely due to repair the broken locators (i.e., structural changes) and minimally to modifications to the test cases logic (i.e., logical changes). Indeed, during maintenance, we have approximately modified only the 1% of the LOCs composing the test suites in order to address logical

Table 4. TEST SUITES MAINTENANCE

	Sikuli API		WebDriver			
	Time	Test	Time			Test
	Minutes	Repaired	Minutes		p-value	Repaired
MantisBT	76	37 / 41	95	+ 25%	**0.04**	32 / 41
PPMA	112	20 / 23	55	- 51%	**< 0.01**	17 / 23
Claroline	71	21 / 40	46	- 35%	0.30	20 / 40
Address Book	126	28 / 28	54	- 57%	**< 0.01**	28 / 28
MRBS	108	21 / 24	72	- 33%	**0.02**	23 / 24
Collabtive	7	4 / 40	79	+ 1029%	**< 0.01**	23 / 40

Table 5. TEST SUITES EXECUTION

	Number of Test Cases	Sikuli API			WebDriver				
		Mean	σ		Mean			σ	
		Seconds	Absolute	Relative	Seconds		p-value	Absolute	Relative
MantisBT	41	2774	60	2,2%	1567	- 43%	**< 0.01**	70	4,5%
PPMA	23	1654	12	0,7%	924	- 44%	**< 0.01**	35	3,8%
Claroline	40	2414	34	1,4%	1679	- 30%	**< 0.01**	99	5,9%
Address Book	28	1591	19	1,2%	977	- 39%	**< 0.01**	106	10,9%
MRBS	24	1595	19	1,2%	837	- 48%	**< 0.01**	54	6,5%
Collabtive	40	2542	72	2,8%	1741	- 31%	**< 0.01**	59	3,4%

changes of the Web apps. Very often the modifications were exactly the same for both the approaches (i.e., Visual and DOM-based). To summarise, with respect to RQ B.2, the result is not clear-cut. For four out of six considered applications, the effort involved in the evolution of the Sikuli test suites, when a new release of the software is produced, is higher than with WebDriver, but in two cases the opposite is true.

RQ B.3. Table 5 shows data about the time required to execute the test suites. For both tools we report: the mean execution time, computed on 10 replications of each execution; the standard deviation (absolute and relative); the difference in percentage between the time required by the Sikuli test suites and the WebDriver test suites; and, the p value reported by the Wilcoxon paired test, used to compare Sikuli vs. WebDriver's execution times. Execution times range from 1591s to 2774s for Sikuli and from 837s to 1741s for WebDriver. In all the cases, the WebDriver test suites required less time to complete their execution (from -30% to -48%). According to the Wilcoxon paired test, the difference is statistically significant (at $\alpha = 0.05$) for all test suites. To summarise, with respect to the research question RQ B.3 the time required to execute the Sikuli test suites is higher than the execution time of WebDriver for all the six considered applications.

3.6 Qualitative Results

In this section, we discuss on the factors behind the results presented in the previous section, focusing more on the ones that are related to the two approaches and, for space reasons, less on the factors related to the specific tools used:

Web Elements Changing their State. When a Web element changes its state (e.g., a check box is checked or unchecked, or an input field is emptied or filled), a visual locator must be created for each state, while with the DOM-based approach only one locator is required. This occurred in all the six Sikuli test suites and it is one of the reasons why, in all of them, we have more locators than in the WebDriver test suites (see RQ A.1 and Table 2). As a consequence, more effort both during the development (RQ B.1) and maintenance (RQ B.2) is required in the case of Sikuli test suites (more than one locator had to be created and later repaired for each Web element, RQ A.2). For instance, in MRBS, when we tested the update functionality for the information associated with a room reservation, we had to create two locators for the same check box (corresponding to the slot: Monday from 9:00 to 10:00) to verify that the new state has been saved (e.g., from booked, *checked*, to available, *unchecked*). Similarly, in Collabtive, we had to verify the changes in the check boxes used to update the permissions assigned to the system users.

Changes behind the Scene. Sometimes it could happen that the HTML code is modified without any perceptible impact on how the Web app appears. An extreme

example is changing the layout of a Web app from the "deprecated" table-based structure to a div-based structure, without affecting its visual aspect in any respect. In this case, the vast majority of the DOM-based locators (in particular the navigational ones, e.g., XPath) used by DOM-based tools may be broken. On the contrary, this change is almost insignificant for visual test tools. A similar problem occurs when auto-generated ID locators are used (e.g., *id1, id2, id3, ... , idN*) by DOM-based locators. In fact, these tend to change across different releases, while leaving completely unaffected the visual appearance of the Web page (hence, no maintenance is required on the visual test suites). For example, the addition of a new link in a Web page might result in a change of all IDs of the elements following the new link [8]. Such "changes behind the scene" occurred in our empirical study and explain why, in the case of Collabtive, the Sikuli test suite has required by far a lower maintenance effort (see RQ B.2 and Table 4). In detail, across the two considered releases, a minor change has been applied to almost all the HTML pages of Collabtive: an unused div tag has been removed. This little change impacted quite strongly several of the XPath locators (XPath locators were used because IDs were not present) in the WebDriver test suite (see RQ A.2). The majority of the 36 locators (all of them are XPaths) was broken and had to be repaired (an example of repairment is from .../div[2]/... to .../div[1]/...). No change was necessary on the Sikuli visual test suite for this structural change. Overall, in Sikuli, we had only few locators broken. For this reason, there is a large difference in the maintenance effort between the two test suites. A similar change across releases occurred also in MantisBT, although it had a lower impact in this application.

Repeated Web Elements. When in a Web page there are multiple instances of the same kind of Web element (e.g., an input box), creating a visual locator requires more time than creating a DOM-based one. Let us consider a common situation, consisting of a form with multiple, repeated input fields to fill (e.g., multiple lines, each with *Name*, *Surname*, etc.), all of which have the same size, thus appearing identical. In such cases, it is not possible to create a visual locator using only an image of the Web element of our interest (e.g., the repeated *Name* input field), but we have to: (i) include also some context around (e.g., a label as shown in Fig. 4) in order to create an unambiguous locator (i.e., an image that matches only one specific portion of the Web page) or, when this is not easily feasible, (ii) locate directly a unique Web element close to the input field of interest and then move the mouse of a certain amount of pixels, in order to reach the input field. Both solutions locate the target Web element by means of another, easier to locate, element (e.g., a label). This is not straightforward and natural for the test developer (i.e., it requires more effort and time). Actually, both solutions are not quite convenient. Solution (i) requires to create large image locators, including more than one Web element (e.g., the label and the corresponding input field). On the other hand, even if it allows to create a small locator image for only one Web element (e.g., the label), Solution (ii) requires to calculate a distance in pixels (similarly to 1st generation tools), not so simple to determine. Both solutions have problems in case of variation of the relative positions of the elements in the next releases of the application. Thus, this factor has a negative effect on both the development and maintenance of Sikuli test suites. Repeated Web elements occurred in all test suites. For instance, in Claroline, a form contains a set of radio buttons used to select the course type to create. In Sikuli, localisation of these

buttons requires either Solution (i) or (ii). Similarly, in AddressBook/MantisBT, when a new entry/user is inserted, a list of input fields, all with the same appearance, has to be filled. In these cases, we created the Sikuli locators as shown in Fig. 4. Recently, JAutomate (http://jautomate.com/), a commercial GUI test automation tool, provided a different solution to this problem by mixing visual locators and position indexes. When a visual locator selects more than one element, it is possible to use an index to select the desired element among the retrieved ones.

Elements with Complex Interaction. Complex Web elements, such as drop-down lists and multilevel drop-down menus, are quite common in modern Web apps. For instance, let us consider a registration form that asks for the nationality of the submitter. Typically, this is implemented using a drop-down list containing a list of countries. A DOM-based tool like WebDriver can provide a command to select directly an element from a drop-down list (only one locator is required). On the contrary, when adopting the visual approach the task is much more complex. Once could, for instance: (1) locate the drop-down list (more precisely the arrow that shows the menu) using an image locator; (2) click on it; (3) if the required list element is not shown, locate and move the scrollbar (e.g., by clicking the arrow); (4) locate the required element using another image locator; and, finally, (5) click on it. All these steps together require more LOCs (in the page objects, see RQ B.1) and locators. Actually, in this case the visual approach performs exactly the same steps that a human tester would do.

Execution Time. The execution time required by the Sikuli tool is always higher than the one required by WebDriver (see RQ B.3 and Table 5). This was expected, since executing an image recognition algorithm requires more computational resources (and thus, generally, more time) than navigating the DOM. However, surprisingly, the difference in percentage between the two approaches is not high, being only 30-48%. It is not very much considering that: (1) Sikuli is a quite experimental tool, (2) it is not focused on Web app testing and, (3) the needed manual management of the pages loading delay (through *sleep* commands) we applied is not optimal[3]. For what concerns the latter point, according to our estimates, the overhead due to the Web page loading delay is not a major penalty for Sikuli (only 20-40 seconds per test suite) as compared to the total processing time. Indeed, we carefully tuned the delays in order to find the smallest required. The standard deviation (see Table 5) is always greater in the case of WebDriver given that, sometimes, it unexpectedly and randomly stops for short periods during test suites execution (e.g., 2-3s between two test cases).

Lesson Learnt: In the following, we report some lessons learnt during the use of the two experimented approaches and tools:

Data-driven Test Cases. Often in the industrial practice [8], to improve the coverage reached by a test suite, test cases are re-executed multiple times using different values. This is very well supported by a programmable testing approach. However, benefits depend on the specific programmable approach that is adopted (e.g., visual vs. DOM-based). For instance, in WebDriver it is possible to use data from various sources, such

[3] A browser needs time to open a Web page. Thus, before starting to perform actions on the page the test automation tool has to wait. WebDriver provides specific commands to deal with this problem (i.e., waiting for the web page loading). In Sikuli this is not available and testers have to insert an explicit delay (e.g., `Thread.sleep(200)`).

as CSV files or databases, or even to generate them at runtime. In Sikuli it is necessary to have images of the target Web elements, so even if we can use various data sources (e.g., to fill input fields), when assertions are evaluated, images are still needed to represent the expected data (see Fig. 3). For this reason, in the visual approach it is not possible to create complete data-driven test cases (i.e., for both input and assertions). In fact, while it is indeed possible to parameterise the usage of image locators in the assertions, it is not possible to generate them from data. This happens because using a DOM-based tool there is a clear separation between the locator for a Web element (e.g., an ID value) and the content of that Web element (e.g. the displayed string), so that we can reuse the same locator with different contents (e.g., test assertion values). On the contrary, using a visual tool, the locator for a Web element and the displayed content are the same thing, thus if the content changes, the locator must be also modified. Moreover, it is important to highlight that, if necessary, parameterising the creation of DOM-based locators is usually an easy task (e.g., .//*[@id='list']/tr[**x**]/td[1] with **x**=1..n), while it is infeasible in the visual approach. In our case study, we experienced this limitation of the visual approach since we had, in each test suite, at least one test case that performs multiple, repeated operations that change only in the data values being manipulated, such as: insert/remove multiple different users, projects, addresses, or groups (depending on the considered application). In such cases we used: (1) a single parameterized locator in WebDriver, and (2) several different image locators in Sikuli (e.g., for evaluating the assertions), with the effect that, in the second case, the number of locators required is higher.

Test Case Comprehensibility. The locators used by the two approaches have often a different degree of comprehensibility. For instance, by comparing Fig. 2 with Fig. 4, it is clear that the visual locator pw.png (password) is much easier to understand than the corresponding XPath locator. In fact, the visual approach works in a manner that is closer to humans than the DOM-based approach. In our case study, we experienced this fact several times. For instance, during test suites maintenance, understanding why a locator is broken is generally easier and faster with Sikuli than with WebDriver.

Test Suites Portability. If a Sikuli test suite is executed on a different machine where the screen resolution or the font properties are different, Sikuli test cases may not work properly. We experienced this problem two times while executing the Sikuli test suites on two different computers: in one case because the default font size was different, resulting in broken image locators, and in another case because the screen resolution was lower than expected, thus more mouse scroll operations were required.

3.7 Threats to Validity

The main threats to validity that affect this study are: Construct (authors' bias), Internal and External validity threats.

Authors' Bias threat concerns the involvement of the authors in manual activities conducted during the empirical study and the influence of the authors' expectations about the empirical study on such activities. In our case, two of the authors developed the test suites and evolved them to match the next major release of each application under test. Since none of the authors was involved in the development of any of the tools assessed in the empirical study, the authors' expectations were in no particular direction for what

concerns the performance of the tools. Hence, we think that the authors' involvement in some manual activities does not introduce any specific bias.

Internal Validity threats concern confounding factors that may affect a dependent variable (number of locators, number of broken locators, development, repair, and execution time of the test suites). One such factor is associated with the approach used to produce the test cases (i.e., the chosen functional coverage criterion). Moreover, the variability involved in the selection of the input data and of the locators could have played a role. To mitigate this threat, we have adopted a systematic approach and applied all known good-practices in the construction of programmable test suites. Concerning RQ B.1, learning effects may have occurred between the construction of the test suites for WebDriver and Sikuli. However, this is quite unlikely given the long time (several months) elapsed between the development of WebDriver and Sikuli test suites and the kind of locators (DOM-based vs. visual), which is quite different. Moreover, given the high level of similarity of the test code (in practice, only locators are different), learning would favour Sikuli, which eventually showed lower performance than WebDriver, so if any learning occurred, we expect that without learning the results would be just amplified, but still in the same direction.

External Validity threats are related to the generalisation of results. The selected applications are real open source Web apps belonging to different domains. This makes the context quite realistic, even though further studies with other applications are necessary to confirm or confute the obtained results. In particular, our findings could not hold for RIAs providing sophisticated user interactions, like, for instance, Google Maps or Google Docs. In fact, using a visual approach it is possible to create test cases that are very difficult (if not impossible) to realise with the DOM-based approach. For instance, it is possible to verify that in Google Docs, after clicking the "center" button, a portion of text becomes centred in the page, which is in practice impossible using just the DOM. The results about number and robustness of locators used by the visual and DOM-based approaches (RQ A.1 and RQ A.2) are not tied to any particular tool, thus we expect they hold whatever tool is chosen in the two categories. On the other hand, the same is not completely true for RQ B.1 and RQ B.2, where the results about the development and maintenance effort are also influenced by the chosen tools, and different results could be obtained with other Web testing frameworks/tools. The problem of the generalisation of the results concerns also RQ B.3 where, for instance, employing a different image recognition algorithm could lead to different execution times.

4 Related Works

We focus our related work discussion considering studies about test suite development and evolution using visual tools; we also consider automatic repairment of test cases.

Several works show that the visual testing automation approach has been recently adopted by the industry [2, 6] and governmental institutions [3]. Borjesson and Feldt in [2], evaluate two visual GUI testing tools (Sikuli and a commercial tool) on a real-world, safety-critical industrial software system with the goal of assessing their usability and applicability in an industrial setting. Results show that visual GUI testing tools are applicable to automate acceptance tests for industrial systems with both cost and potentially quality gains over state-of-practice manual testing. Differently from us, they

compared two tools both employing the visual approach and did not focus specifically on Web app testing. Moreover, our goal (comparing visual vs. DOM-based locators) is completely different from theirs.

Collins *et al.* [6], present three testing automation strategies applied in three different industrial projects adopting the Scrum agile methodology. The functional GUI test automation tools used in these three projects were respectively: Sikuli, Selenium RC and IDE, and Fitnesse. Capocchi *et al.* [3], propose an approach, based on the DEVSimPy environment and employing both Selenium and Sikuli, aimed at facilitating and speeding up the testing of GUI software. They validated this approach on a real application dealing with medical software.

Chang *et al.* [4] present a small experiment to analyse the long-term reusability of Sikuli test cases. They selected two open-source applications (Capivara and jEdit) and built a test suite for each application (10 test cases for Capivara and 13 test cases for jEdit). Using some subsequent releases of the two selected applications, they evaluated how many test cases turned out to be broken in each release. The lesson drawn from this experiment is: as long as a GUI evolves incrementally a significant number of Sikuli test cases can still be reusable. Differently from us, the authors employed only a visual tool (Sikuli) without executing a direct comparison with other tools.

It is well-known that maintaining automated test cases is expensive and time consuming (costs are more significant for automated than for manual testing [15]), and that often test cases are discarded by software developers due to huge maintenance costs. For this reason, several researchers proposed techniques and tools for automatically repairing test cases. For instance, Mirzaaghaei *et al.* [12] presents TestCareAssistant (TcA), a tool that combines data-flow analysis and program differencing to automatically repair test compilation errors caused by changes in the declaration of method parameters. Other tools for automatically repairing GUI test cases or reducing their maintenance effort have been presented in the literature [16, 7, 10]. Choudhary *et al.* [5] extended these proposals to Web apps, presenting a technique able to automatically suggest repairs for Web app test cases.

5 Conclusions and Future Work

We have conducted an empirical study to compare the robustness of visual vs. a DOM-based locators. For six subject applications, two equivalent test suites have been developed respectively in WebDriver and Sikuli. In addition to the robustness variable, we have also investigated: the initial test suite development effort, the test suite evolution cost, and the test suite execution time. Results indicate that DOM-based locators are generally more robust than visual ones and that DOM-based test cases can be developed from scratch at lower cost and most of the times they can be evolved at lower cost. However, on specific Web apps (MantisBT and Collabtive) visual locators were easier to repair, because the visual appearance of those applications remained stable across releases, while their structure changed a lot. DOM-based test cases required a lower execution time (due to the computational demands of image recognition algorithms used by the visual approach), although the difference was not that dramatic. Overall, the choice between DOM-based and visual locators is application-specific and depends quite strongly on the expected structural and visual evolution of the application. Other

factors may also affect the testers' decision, such as the availability/unavailability of visual locators for Web elements that are important during testing and the presence of advanced, RIA functionalities which cannot be tested using DOM-based locators. Moreover, visual test cases are definitely easier to understand, which, depending on the skills of the involved testers, might also play a role in the decision.

In our future work we intend to conduct further studies to corroborate our findings. We plan to complete the empirical assessment of the Web testing approaches by considering also tools that implement capture-replay with visual Web element localisation (e.g., JAutomate). Finally, we plan to evaluate tools that combine the two approaches, such as SikuliFirefoxDriver (http://code.google.com/p/sikuli-api/wiki/SikuliWebDriver), that extends WebDriver by adding the Sikuli image search capability, combining in this way the respective strengths.

References

1. S. Berner, R. Weber, and R. Keller. Observations and lessons learned from automated testing. In *Proc. of ICSE 2005*, pages 571–579. IEEE, 2005.
2. E. Borjesson and R. Feldt. Automated system testing using visual GUI testing tools: A comparative study in industry. In *Proc. of ICST 2012*, pages 350–359, 2012.
3. L. Capocchi, J.-F. Santucci, and T. Ville. Software test automation using DEVSimPy environment. In *Proc. of SIGSIM-PADS 2013*, pages 343–348. ACM, 2013.
4. T.-H. Chang, T. Yeh, and R. C. Miller. Gui testing using computer vision. In *Proc. of CHI 2010*, pages 1535–1544. ACM, 2010.
5. S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proc. of ETSE 2011*, pages 24–29. ACM, 2011.
6. E. Collins, A. Dias-Neto, and V. de Lucena. Strategies for agile software testing automation: An industrial experience. In *Proc. of COMPSACW 2012*, pages 440–445. IEEE, 2012.
7. M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *Proc. of ICSE 2009*, pages 408–418. IEEE, 2009.
8. M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Proc. of 6th Int. Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2013, pages 108–113. IEEE, 2013.
9. M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proc. of 20th Working Conference on Reverse Engineering*, WCRE 2013, pages 272–281. IEEE, 2013.
10. A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *TOSEM*, 18(2):4:1–4:36, Nov. 2008.
11. M. Mirzaaghaei. Automatic test suite evolution. In *Proc. of ESEC/FSE 2011*, pages 396–399. ACM, 2011.
12. M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *Proc. of ICSM 2010*, pages 1–5. IEEE, 2010.
13. F. Ricca and P. Tonella. Testing processes of web applications. *Ann. Softw. Eng.*, 14(1-4):93–114, Dec. 2002.
14. F. Ricca and P. Tonella. Detecting anomaly and failure in web applications. *IEEE MultiMedia*, 13(2):44–51, 2006.
15. M. Skoglund and P. Runeson. A case study on regression test suite maintenance in system evolution. In *Proc. of ICSM 2004*, pages 438–442. IEEE, 2004.
16. Q. Xie, M. Grechanik, and C. Fu. Rest: A tool for reducing effort in script-based testing. In *Proc. of ICSM 2008*, pages 468–469. IEEE, 2008.