

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/324116949>

Pesto: Automated migration of DOM-based Web tests towards the visual approach

Article in *Software Testing Verification and Reliability* · March 2018

DOI: 10.1002/stvr.1665

CITATIONS

15

READS

123

4 authors:



Maurizio Leotta

Università degli Studi di Genova

68 PUBLICATIONS 623 CITATIONS

SEE PROFILE



Andrea Stocco

University of Lugano

24 PUBLICATIONS 220 CITATIONS

SEE PROFILE



Filippo Ricca

Università degli Studi di Genova

175 PUBLICATIONS 3,734 CITATIONS

SEE PROFILE



Paolo Tonella

Fondazione Bruno Kessler

292 PUBLICATIONS 7,690 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



ASPIRE [View project](#)



ASPIRE-FP7 [View project](#)

PESTO: Automated Migration of DOM-based Web Tests towards the Visual Approach

Maurizio Leotta^{1*}, Andrea Stocco², Filippo Ricca¹, Paolo Tonella³

¹ *Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Genova, Italy*

² *Department of Electrical and Computer Engineering (ECE), University of British Columbia, Vancouver, BC, Canada*

³ *Fondazione Bruno Kessler (FBK), Trento, Italy*

SUMMARY

Test automation tools are widely adopted for testing complex web applications. Three generations of tools exist: 1st based on screen coordinates, 2nd based on DOM-oriented commands, and 3rd based on visual image recognition. In our previous work, we proposed PESTO, a tool able to migrate 2nd generation Selenium WebDriver test suites towards 3rd generation Sikuli ones. In this work, we extend PESTO to manage web elements having (1) complex visual interactions and (2) multiple visual appearances. PESTO relies on aspect-oriented programming, computer-vision and code-transformations. Our new improved tool has been evaluated on two web test suites developed by an independent tester. Experimental results show that PESTO manages and transforms correctly test suites with web elements having complex visual interactions and multi-state elements. By using PESTO, the migration of existing DOM-based test suites to the visual approach requires a low manual effort, since our approach proved to be very accurate.
Copyright © 2018 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Web Testing; DOM-based Testing; Visual Testing; Test Automation; Selenium WebDriver; Sikuli.

1. INTRODUCTION

Web applications are developed and evolved at a very fast pace. As business and customer requirements grow and evolve, so does the pressure on software professionals to deliver new stable releases, in reduced time and with high quality. Within such ultra-rapid development cycles, testing plays a crucial role in quality assurance. However, the specific features of web software make testing quite challenging [1].

For these reasons end-to-end (E2E) test automation tools have become quite popular in the web scenario during the last decade across a wide range of testing tasks such as regression, system or GUI testing. End-to-end (E2E) functional test automation is based on the creation of test scripts that automate the interaction with a web page and its elements. Test cases can, for instance, automatically fill-in and submit forms and click on hyperlinks. High level languages and programmer-friendly APIs provide commands to control the browser and interact with the web page elements, e.g., to click a button or fill a field with text. Test scripts are then completed with assertions, e.g., using xUnit[†]. E2E

* Correspondence to: Maurizio Leotta, Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Genova, Italy. E-mail: maurizio.leotta@unige.it

[†] <http://www.martinfowler.com/bliki/Xunit.html>

testing is useful because it can execute tests faster than any person and its results can be reproduced automatically. Moreover, E2E testing can be run unattended, saving substantial testing effort and resources [2].

There are a number of commercial and open source tools available for assisting the development of automated E2E test suites. There are three generations of web test automation tools [3, 4, 5], depending on the approach used to identify the web elements displayed on the user interface.

The *1st generation* was based on screen coordinates, resulting in test suites that are quite fragile when the application under test evolves. Minimal changes in the web page layout or screen resolution may in fact invalidate the existing test cases, requiring the developers to fix or recreate them from scratch. The *2nd generation* of tools, called DOM-based or structural [3], overcomes the limitations of the 1st generation by accessing the web page Document Object Model (DOM), a hierarchical data structure where web elements (e.g., anchors, buttons) can be located by accessing their properties (e.g., identifier or text), or by navigating the DOM tree by means of XPath queries. While 2nd generation web testing tools are currently widely adopted in the industry [6], a new generation of visual tools (*3rd generation*) offers an interesting alternative, promising easier and more intuitive test case creation and maintenance. This is witnessed by the abundance of tools based on or supporting visual testing, such as JAutomate [7], Sikuli [8], Ranorex[‡], and EggPlant[§]. Their usefulness is particularly evident when web applications are implemented with complex visual components (e.g., Google Docs or Google Maps), to offer increased user-friendliness and responsiveness. In such cases, DOM-based tools do not suffice because the DOM of the application may not include all user interface controls.

The emergence of new complex visual components in web pages pulls for the adoption of 3rd generation tools, but nowadays the two approaches (i.e., DOM-based and Visual) coexist and it is not clear in what direction the testing community will go (i.e., if an approach will prevail over the other). In this context, a tester or a company might want to understand if the migration towards such new technology is worthwhile, since the manual effort to rewrite a test suite might be overwhelming. Indeed, in case of huge or complex DOM-based test suites, the manual migration could be an extremely time-consuming, error-prone and expensive task.

In this paper, we face this problem and propose an enhancement of our solution to *migrate DOM-based into visual test cases, so as to allow developers to experiment with 3rd generation testing tools at a lower cost compared to manual migration.*

The implementation of such migration is a challenging research problem, because the mapping between DOM-based commands and visual commands is not trivial and is not ensured to be widely applicable with a low error rate. Thus, in this work we aim at studying whether an automatic migration process from the “old” 2nd generation towards the “new” 3rd generation can be established. Among others, a non-trivial research problem we address is the mapping between a DOM locator and its visual counterpart. This is challenging, because such a mapping can be established only dynamically, when the web application is running, and potentially a single DOM-based locator can have multiple and different visual counterparts.

The implementation of our approach, a tool called PESTO, is able to migrate a Selenium WebDriver test suite (2nd generation test suite) into a Sikuli API one (3rd generation test suite). Using PESTO, companies and professionals can evaluate the features of the 3rd generation visual testing tools, without taking the risks and the costs of the manual migration.

PESTO is a component of NEONATE [?], an integrated testing environment that, once completed, will be able to empower the web tester to limit the open problems that in our opinion hinder the web test automation: fragility problem, strong coupling and low cohesion problem and incompleteness problem.

This paper makes the following contributions to the state of the art:

- an approach for the automatic creation of visual locators starting from corresponding DOM-based locators. The approach resorts on the automated visual localization of the web elements involved in any interaction or assertion performed by the original DOM-based test suite;

[‡] <http://www.ranorex.com/> [§] <http://www.testplant.com/eggplant/testing-tools/eggplant-developer/>

- a solution to three challenging problems that must be addressed in order to fully automate the transformation of complex DOM-based test suites to the visual approach and that consists of the generation of visual locators for:
 - web elements having multiple appearances within the same web page (e.g., input fields in a form);
 - web elements that change their graphic appearance in different times/tests (e.g., checkboxes);
 - web elements having a complex visual interaction (e.g., drop-down lists).
- an approach for the automatic transformation of DOM-based test code to the visual technology. Based on the captured images, the original test suites are automatically rewritten as visual test suites. PESTO can be used for generating different visual test suites, tailored for different platform-browser combinations. In such case, the web page rendering may change and, correspondingly, the images used by the visual test cases will also be different.
- an implementation of our approach in an open source tool called PESTO;
- an empirical evaluation to assess the effectiveness and limitations of PESTO. Results indicate that PESTO's accuracy was very high on the two considered case studies and show that the approach is viable, although further empirical evaluation on additional case studies would be required to confirm the general validity of these findings.

This paper is a substantially revised version of our original proposal presented in conference [9] and tool demo [10] papers. The main extensions are as follows:

- we improved PESTO to support a more complete set of Selenium WebDriver commands. In particular, we added the support for: `selectByVisibleText()`, `selectByIndex()`, and `clear()`. The previous version of PESTO [9, 10] supported only `click()`, `sendKeys()`, `getText()`;
- we improved PESTO with multi-state locators, in order to manage the interaction with web elements that change their visual rendering during the execution of the test suite (e.g., a check box that can be checked or unchecked);
- we performed a more thorough evaluation of PESTO, on an extended set of test cases. In our previous work [9], we used four DOM-based test suites composed of 68 test cases and 213 web elements (2556 Java LOCs overall for the DOM-based test suites). In this work, we enhanced the same four test suites and increased by 140% the number of test cases (163) and by 152% the number of web elements (537), resulting in 5626 Java LOCs for the DOM-based test suites. We used the four test suites, with improved GUI coverage, as a *training set* for the improvement process of PESTO. Then, we performed an unbiased, independent evaluation of PESTO, totally absent in our preliminary work [9, 10]. To this aim, we recruited a professional tester to build two DOM-based test suites for two additional web applications (61 test cases, 192 web elements, 2928 Java LOCs for the DOM-based test suites). These two new test suites acted as *validation set* for the assessment of PESTO. By involving an external tester, not aware of the existence of our tool and of the purpose of this work, we have verified that our approach is indeed applicable to DOM-based test suites developed by others in a realistic setting where manual migration to the visual approach may not be affordable;
- we have added a discussion section, about the problems associated with migrating DOM-based test suites to the visual approach, useful to other researchers, as well as to practitioners, interested in further developing our approach or facing similar problems.

The paper is organised as follows: Section 2 provides a background on the three generations of web testing tools with illustrative examples. Section 3 describes the challenges we faced in the automatic transformation from DOM-based to visual test suites. Section 4 describes our approach and tool PESTO. Section 5 describes the improvement process of PESTO based on a training set, consisting of four test suites. Section 6 presents the experimental results obtained on the validation set, consisting of two test suites developed by an external, professional tester. Related works are discussed in Section 7, followed by conclusions and future work in Section 8.

2. DOM-BASED AND VISUAL WEB TESTING

End-to-end testing of web applications is a kind of black box testing based on the concept of test scenario, i.e., a sequence of steps/actions performed on the web application (e.g., insert username, insert password, click the login button, etc.). One or more test cases can be derived from a single test scenario by specifying the actual data to use in each step (e.g., *username*=“John.Doe”) and the expected outcomes (e.g., by means of assertions). There are different approaches to locate the web elements a test case interacts with. The choice depends on a number of factors, such as the technologies employed by the application under test (e.g., Ajax) or the web testing tools used by the testers.

2.1. Web Testing Tools

In this section, we briefly describe the second and third generation[¶] of web testing tools [3, 4, 5]^{||}, depending on the strategy they adopt to localise web elements on a page.

DOM-based: second generation tools (e.g., Selenium WebDriver^{**}) are based on the information contained in the Document Object Model^{††} (DOM), the hierarchical structure underlying a HTML page. DOM-based location strategies are usually in the format *locatorType=location*. The various locator types rely on the attributes present in the DOM (e.g., *id*, *name*) or the textual content (e.g., *links*). When there are no suitable *id* or *name* attributes for the element of interest, one can use navigation expressions, such as XPath^{‡‡}. Smart choice of XPath locators are demonstrated to be effective in making web tests robust to software evolution [12, ?].

Visual: third generation tools (e.g., JAutomate [7], Sikuli [8]) use image recognition techniques to identify the web elements on the displayed web page. Visual web testing tools are promising and emerging in industry [7]. In general, they can be considered as a valid alternative in all cases where DOM-based tools are not effective (for instance, when dealing with highly interactive applications, such as Google Maps).

Let us consider the example in Figure 1, consisting of a web application including a login form shown in the upper right corner of the home page (home.php). This web application requires the authentication of the users, who are requested to enter their credentials, i.e., *username* and *password*. If the credentials are correct, the username (e.g., John.Doe) and a logout button replace the login form in the upper right corner of the home page. Otherwise, for simplicity, the login form is still displayed.

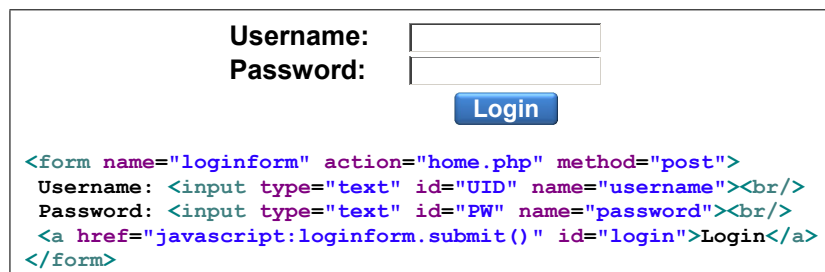


Figure 1. Fragment of home.php – Page and Source

[¶] We do not consider the first generation that is based on screen coordinates and it is nowadays considered obsolete. ^{||} In this categorization, we do not consider tools like TESTAR [11] that have been recently applied in the web context and that perform testing via the GUI, using the operating system's Accessibility API.

^{**} <http://www.seleniumhq.org/projects/webdriver/> ^{††} <https://www.w3.org/DOM/> ^{‡‡} <https://www.w3.org/TR/xpath/>

2.2. The Page Object and Page Factory Patterns

The *Page Object*^{§§} [13] is a popular design pattern used in web testing, which aims at improving the test case maintainability and reducing the duplication of code. A page object is a class that represents the web page elements as a series of objects and encapsulates the features of the web page into methods. Adopting the page object pattern in test script implementation allows testers to follow the *Separation of Concerns* design principle, since the test scenario is decoupled from its implementation. Indeed, all the implementation details are moved into the page objects, a bridge between web pages and test cases, with the latter only containing the test logics. Thus, all the functionalities to interact with or to make assertions about a web page are offered in a single place, the Page Object, and can be easily called and reused within any test case. Usually page objects are initialised by a Page Factory^{¶¶}, a factory class that checks the correct mapping and initialisation of the web elements. Examples of test code adopting the *Page Object* and *Page Factory* patterns are reported in Figures 2 and 3. Such patterns reduces the coupling between web pages and test cases, promoting reusability, readability and maintainability of the test suites [14, 15]. As a part of our ongoing research, we also addressed the automatic generation of page objects with the tool APOGEN [16]. Experimental results indicate that our tool is able to automatically create page objects for non-trivial web applications, with a good level of accuracy [17].

2.3. DOM-based Web Testing

In this work, we selected Selenium WebDriver among the tools for implementing DOM-based web test suites (hereafter referred to as WebDriver). WebDriver is a state-of-the-art tool, widely used for web test automation [18, 19]. WebDriver provides a comprehensive API used to control the browser, thus test cases can be written for instance in the Java programming language and integrated with JUnit or TestNG assertions. The choice of WebDriver as 2nd generation has been a careful choice, motivated by the followings facts: (1) it is a mature and maintained tool available since several years, deeply documented both online^{***} and by specific books [20, 21, 22, 23, 24], (2) it is open-source, (3) it is one of the most widely-adopted open-source solutions for web test automation [18, 19] (the community of users of the Selenium framework organizes since several years the Selenium Conferences^{†††} in Europe, US and India) and it has been used also in several scientific works, e.g., to cite a few [25, 26, 16], (4) it is being standardized by the W3C^{‡‡‡}, and (5) during our previous industrial collaborations, we gained a considerable experience in its usage [14, 27]. At the time of writing, we are not aware of web test automation tools with such characteristics and popularity.

WebDriver is able to locate a web page element using: (1) the values of attributes `id`, `name`, and `class`; (2) the tag name of the element; (3) the text shown in the hyperlink, for anchor elements; (4) CSS and (5) XPath expressions. Not all these locators are applicable to any arbitrary web element; e.g., locator (1) can be used only if the target element has a unique value of attribute `id`, `name`, or `class` in the entire web page; locator (2) can be used if there is only one element with the chosen tag name in the whole page; locator (3) can be used only for links uniquely identified by their text. On the other hand, XPath/CSS expressions can always be used. In fact, as a baseline, the unique path from root to the target element in the DOM tree can always be turned into an XPath/CSS locator that uniquely identifies the element.

For our running example, shown in Figure 1, let us consider a test case corresponding to a successful authentication. The test case submits correct credentials (i.e., `username="John.Doe"` and `password="123456"`) and verifies that, in the home page, the user appears as correctly authenticated (the string "John Doe" must be displayed in the home page, e.g., in the text of a HTML tag having `ID="loggedUser"`).

As a first step, a tester would create the `HomePage.java` page object (see Figure 2), corresponding to the `home.php` web page. The page object `HomePage.java` offers a method to log into the application. The `login()` method takes as input `username` and `password`, inserts them into the corresponding input

^{§§} <https://github.com/SeleniumHQ/selenium/wiki/PageObjects> ¶¶ <https://github.com/SeleniumHQ/selenium/wiki/PageFactory>

^{***} <http://www.seleniumhq.org/projects/webdriver/>

^{†††} <https://twitter.com/seleniumconf>

^{‡‡‡} <https://www.w3.org/TR/webdriver/>

```

public class HomePage {
    private final WebDriver driver;
    @FindBy(id="UID")
    private WebElement username;
    @FindBy(xpath="./form/input[2]")
    private WebElement password;
    @FindBy(linkText="login")
    private WebElement login;
    @FindBy(id="loggedUser")
    private WebElement loggedUser;
    public HomePage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
    public void login(String UID, String PW) {
        username.sendKeys(UID);
        password.sendKeys(PW);
        login.click();
    }
    public Boolean checkLoggedUser(String name) {
        return loggedUser.getText().equals(name);
    }
}

```

Locators Declaration and Initialization

PO Methods

Figure 2. HomePage page object in Selenium WebDriver (2nd generation)

fields and clicks the Login button. Moreover, the page object contains a method that verifies the presence of the authenticated username in the displayed web page. As shown in Figure 2, the web page elements are located using three types of DOM-based locators (i.e., ID, LinkText, XPath).

```

@Test
public void testLogin() {
    WebDriver driver = new FirefoxDriver();
    driver.get("http://localhost:8080/home.php");
    HomePage HP = new HomePage(driver);
    HP.login("John.Doe", "123456");
    assertTrue(HP.checkLoggedUser("John Doe"));
}

```

Figure 3. TestLogin test case in Selenium WebDriver (2nd generation)

The second step consists of developing one or more test cases that make use of the page objects methods. Figure 3 shows a Java implementation of the test case described above for our running example of Figure 1. In the test case, first, a WebDriver of type FirefoxDriver is created to control the Firefox browser; second, WebDriver opens the specified URL and creates an instance of HomePage.java; third, using method login(), the test attempts to login into the application; finally, a test case assertion checks that the authentication has completed successfully.

2.4. Visual Web Testing

In this work we selected Sikuli API (for short, Sikuli) among the 3rd generation (Visual) web testing tools [8]. Sikuli is a visual tool able to automate and test graphical user interfaces using screenshot images. It provides image-based GUI automation functionalities to Java, Python, and Ruby programmers. We chose Sikuli as representative of its category because (1) it is open-source, (2) it is used in the industry (e.g., at Spotify [28]), and (3) it is similar to WebDriver, thus, we can create test cases and page objects similar to the ones produced for WebDriver. Indeed, we can use the same programming environment: same programming language (Java), IDE (Eclipse), and testing framework (JUnit). Sikuli allows testers to write scripts based on images that define the GUI widgets to interact with and the assertions to be checked.

Figure 4 shows an example of the testLogin test case in Sikuli, whereas the related Sikuli page object is given in Figure 5. The test case performs the same conceptual steps as the WebDriver test case, but we can notice some differences. The first operation, `CommonPage.open(...)`, aims at opening the browser at a specified URL. This can be performed in different ways. For instance, the implementation of `CommonPage.open(...)` may execute the command: `Process pr = Runtime.getRuntime().exec("C:\Mozilla Firefox\firefox.exe http://localhost:8080/home.php")`. Such an instruction can be automatically generated starting from the DOM-based test suite. Another purely visual approach would require to identify and click the Firefox icon on the desktop, inserting the URL into the address bar and then clicking on the “go” arrow (since this aspect it is not central in the DOM to visual transformation and can be performed in several ways, we have chosen to encapsulate these operations inside method `open` of class `CommonPage`).

```
@Test
public void testLogin() {
    CommonPage.open("http://localhost:8080/home.php");
    HomePage HP = new HomePage();
    HP.login("John.Doe", "123456");
    assertTrue(HP.checkLoggedUser());
}
```

Figure 4. TestLogin test case in Sikuli API (3rd generation)

```
public class HomePage {
    private String path = "visualLocators/HomePage/";
    private Target username;
    private Target password;
    private Target login;
    private Target loggedUser;

    public HomePage() {
        username = new ImageTarget(new File(path+"username.png"));
        password = new ImageTarget(new File(path+"password.png"));
        login = new ImageTarget(new File(path+"login.png"));
        loggedUser = new ImageTarget(new File(path+"loggedUser.png"));
    }

    public void login(String UID, String PW) {
        type(username, UID);
        type(password, PW);
        click(login);
    }

    public Boolean checkLoggedUser() {
        return check(loggedUser);
    }

    ...
    Auxiliary Visual Methods (e.g., implementation of type, click, check)
    ...
}
```




Figure 5. HomePage page object in Sikuli API (3rd generation)

The next steps are basically the same in Sikuli and WebDriver: the only differences being that in Sikuli driver is not a parameter of the `HomePage` constructor and that the assertion checking method does not need any string parameter (see explanation below). On the contrary, Sikuli’s page object is

quite different from WebDriver's. As shown in Figure 5, all the DOM-based locators are replaced by visual locators (i.e., images). The images must have been previously saved in the file system as files or must be available online. Moreover, some useful WebDriver methods are not natively available in Sikuli (e.g., `click()` and `sendKeys()`). Thus, when using Sikuli, they must be explicitly implemented in the page object class (e.g., see methods `click()`, `type()` and `check()` in Figure 5). More details on the "Auxiliary Visual Methods" are reported in Section 4.2. The source code shown in Figure 2 and Figure 5 is representative of the typical code we have processed/obtained when applying our transformation tool PESTO to real cases.

A major difference between 2nd and 3rd generation tools is the way assertions are written. DOM-based tools allow users to specify fine-grained assertions on the properties of the elements contained in the model of the web page – i.e., the DOM. Visual tools, on the other hand, check that an image representing the visual appearance of an element is present in the web page as rendered at runtime by the browser. Moreover, DOM-based tools can find the target elements regardless of their position in the DOM or their presence on the displayed GUI. Visual tools, instead, do require the target elements to be displayed on the screen (a special case is when the element exists in the current page, but is not actually rendered, unless the visualization is scrolled down).

3. CHALLENGING ASPECTS OF THE TRANSFORMATION FROM DOM TO VISUAL

In order to fully automate the transformation of a complex DOM-based test suite to the visual approach, several challenges must be properly addressed. Among them, the key issues are: (1) generating visual locators for repeated web elements within the same page, (2) localising web elements that change their graphic appearance in different times/tests, and (3) managing web elements having a complex visual interaction. This section provides an overview of these problems, while the solutions implemented in PESTO are discussed in Section 4.

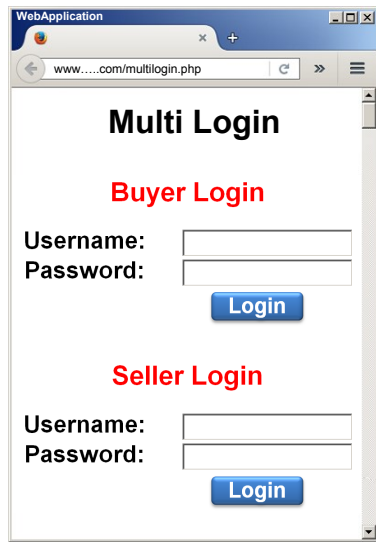
3.1. Generating Visual Locators for Repeated Web Elements

When a web page contains multiple instances of the same kind of web element (e.g., an input box), creating a visual locator is more complex than creating a DOM-based locator. Let us consider a common situation, shown in Figure 6, consisting of multiple forms with multiple input fields (e.g., *Username*, *Password*, etc.), all of which have the same size, thus appearing identical. In such cases, it is not possible to create a visual locator using an image showing only the web element of interest (e.g., the repeated *Username* input field).

There are at least two possible methods for addressing this issue: (1) the visual locator is expanded so as to include some context around the web element of interest, (e.g., a label), in order to create an unambiguous locator, (i.e., an image that matches uniquely the desired portion of web page), as in the example in Figure 6 (right-bottom); this method is the one adopted by PESTO as we will see in Section 4. (2) Another stable web element is identified in the page, close enough to the web element of interest, to be used as an "anchor". In this way, it is then possible to instrument the mouse to move the cursor from the anchor to the desired web element (e.g., using an offset). Both solutions locate the web element of interest by means of another—easier to uniquely locate—web element (e.g., a label).

Let us discuss the pros and cons of these two solutions. The first solution may output *large* visual locators, that include more than one web element (e.g., multiple labels and input fields). This is an advantage for what concerns the locator readability, but a disadvantage for the robustness of the test suite. Indeed, as the application evolves, it is likely that the visual appearance is affected, and visual locators must be re-created. The second solution may produce smaller, hence more robust, locators (i.e., the anchor), but it requires to calculate the distance in pixels between the anchor and the target web element. This brings in all the fragility issues associated with the use of screen coordinates, similarly to the 1st generation tools. Both solutions have problems in case of variations of the relative positions of the elements in the next releases of the application.

Web Page contains Two Login forms



The target web element is the second Username Field

An Image representing the target element is not a locator



4 matches in the web page



so expanding the image



2 matches in the web page



so expanding the image



1 matches in the web page

Visual Locator Found

Figure 6. Visual locator creation for repeated web elements

3.2. *Web Elements Changing their State*

Different localisation approaches (e.g., DOM vs. visual) may require a different number of locators to be created. In a previous experiment [5] we found that the visual approach required 45% more locators w.r.t. the DOM-based one (706 visual vs. 487 DOM-based locators).

One explanation for this increment in the number of locators is the presence of web elements that change their visual appearance depending on the state. For instance, a check box can be checked or unchecked, as in the example shown in Figure 7. In such case a visual locator must be created for each state (checked or unchecked), while with the DOM-based approach only one locator is needed.

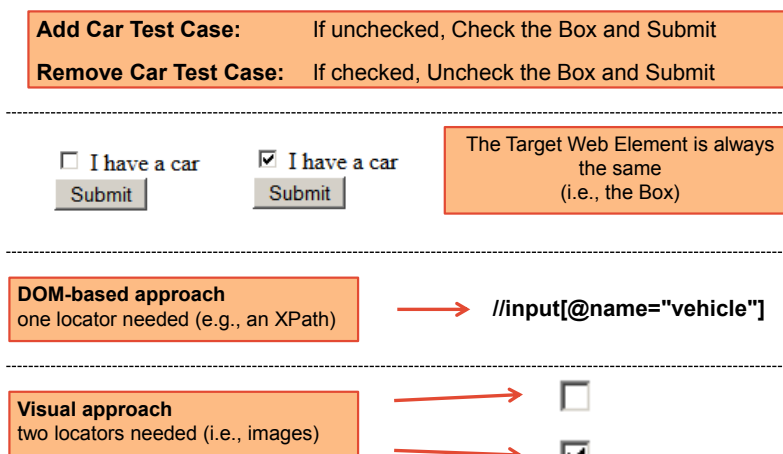


Figure 7. State changes: DOM-based vs. visual localization

3.3. Web Elements with Complex Visual Interaction

There are web elements in which multiple states are associated with complex interactions. An example is the drop-down list, quite common in modern web applications. Figure 8 shows a form in which the user can select the manufacturer of a car. Typically, this is implemented using a drop-down list containing a list of manufacturers. A DOM-based tool, such as Selenium WebDriver, provides commands to select directly a list item from the drop-down list (in the example in Figure 8 only one ID-based locator is sufficient). On the contrary, when adopting the visual approach the interaction becomes more complex. For instance, the visual interaction with a drop-down list may consist of: (1) locating the drop-down list using an image locator; (2) clicking on it; (3) if the required list element is not shown, locating and moving the scrollbar (e.g., by clicking the arrow); (4) locating the required element using another image locator; and, finally, (5) clicking on it. Actually, in this case the visual approach replicates the same steps that a human tester would do, whereas a DOM-based tool like Selenium WebDriver can access the DOM and directly select the value in the drop-down list. Another possible visual interaction with a drop-down list, which takes advantage of the keyboard, consists of: (1) locating the drop-down list using an image locator; (2) clicking on it; and (3) selecting the item by typing its value through the keyboard input interface in the test script.

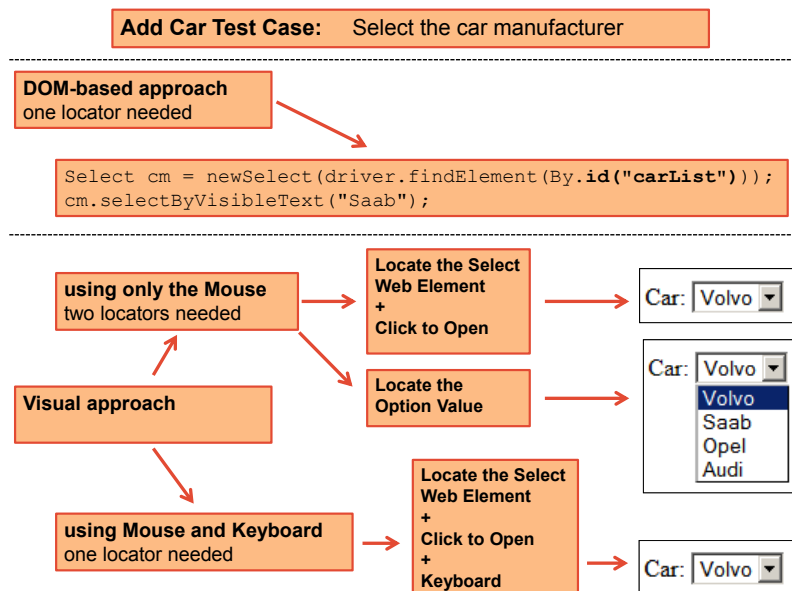


Figure 8. Complex interaction: DOM-based vs. visual localization

4. PESTO: THE PAGE OBJECT TRANSFORMATION TOOL

The aim of PESTO (PagE object tranSformation TOol) is converting a DOM-based web test suite, created using Selenium WebDriver, into a new visual web test suite based on the Sikuli image recognition capabilities. The only important prerequisite for our tool is the use of Page Object and Page Factory design patterns within the initial DOM-based test suite. This has been a careful, though critical, choice, because page objects provide a unique point of localization for web elements declarations and their associated locators. This allows the test specification to be well separated from

its technical implementation (separation of concerns), resulting in cleaner test code and universally recognized as a best practice both from practitioners^{§§§} [13, 29, 30] and scholars [18, 15, 14].

While in general different DOM-based/visual tools could be plugged into our approach at the price of engineering effort^{¶¶¶}, we chose Selenium WebDriver and Sikuli API as state of the art tools for the 2nd and 3rd generation, respectively. The key advantage of choosing the pair WebDriver, Sikuli is that PESTO can keep the structure of the test cases/page objects almost unaltered, focusing on our primary goal, i.e., replacing the approach by which web elements are localized on the web page (i.e., from DOM to Visual). Once the challenges of migration are solved, developing different variants of PESTO (e.g., migration to JAutomate test cases) is mostly a matter of engineering, under the constraint that some relevant functionalities, such as the possibility to retrieve the coordinates of the target web element on the screen are available.

Thus, the techniques and architectural solutions adopted for PESTO are quite general and can be adapted to other DOM to visual web test transformation involving abstractions similar to the ones provided by the page objects. In the actual implementation of PESTO, we chose Java for both the DOM and visual test suites just for convenience. On the other hand, it would be possible to create a variant of PESTO supporting the automatic migration of test suites developed using the other languages supported by both WebDriver and Sikuli, such as Python or Ruby.

Figure 9 shows the high level overview of PESTO, which consists of two main modules:

Visual Locators Generator (Module 1) generates one or more visual locators for each web element used in the DOM-based test suite. The details about this module are reported in Section 4.1.

Test Suite Transformer (Module 2) transforms the source code of the DOM-based test suite to adopt the visual approach. In particular, the majority of the changes are concentrated in the page object code, since page objects are responsible for the direct interaction with web pages (test scripts remain almost unaltered because they rely on the methods offered by the page objects). The details about this module are reported in Section 4.2.

For the interested reader, a demo video of PESTO and its source code can be found at <http://sepl.dibris.unige.it/PESTO.php>.

The next subsections provide a detailed description of the two aforementioned modules composing PESTO, Visual Locators Generator and Test Suite Transformer, and describe the solutions adopted to overcome the challenging aspects of the DOM to visual transformation described in Section 3.

4.1. Visual Locators Generator (Module 1)

The key requirement for automatic visual locator generation is that it must be carried out dynamically, during the execution of the test suite. In fact, visual locators must reflect the appearance of the target web elements as seen at runtime during the execution of the test cases. To capture the images of the target web elements, as rendered by the browser at run time, we need to intercept the commands executed by the test cases. To this aim, for the implementation of Module 1, we adopted the aspect-oriented programming paradigm – in particular, AspectJ¹⁷. Using aspects, PESTO is able to intercept calls to the WebDriver commands and create the corresponding visual locators. Figure 10 summarises the activities carried out by Module 1.

During the execution of the test suite, WebDriver interacts with the web elements of the application under test. For instance, it may click on a link or a button using the `click()` command, fill or clear a text box or a text area using respectively the `sendKeys(...)` and `clear()` commands, select an item in a list, e.g., using the `selectByVisibleText(...)` command, or retrieve the text shown by a web element using `getText()`.

^{§§§} <https://confengine.com/selenium-conf-2014/proposal/310/page-objects-done-right>
^{¶¶¶} <https://confengine.com/selenium-conf-2014/proposal/293/perils-of-page-object-pattern> The chosen tools should provide the commands on which the various transformation steps described in this section are based or rely on (e.g., the ability of saving the screenshot of the page and obtaining coordinates and size of each web element the test cases interact with).
¹⁷ <http://eclipse.org/aspectj/>

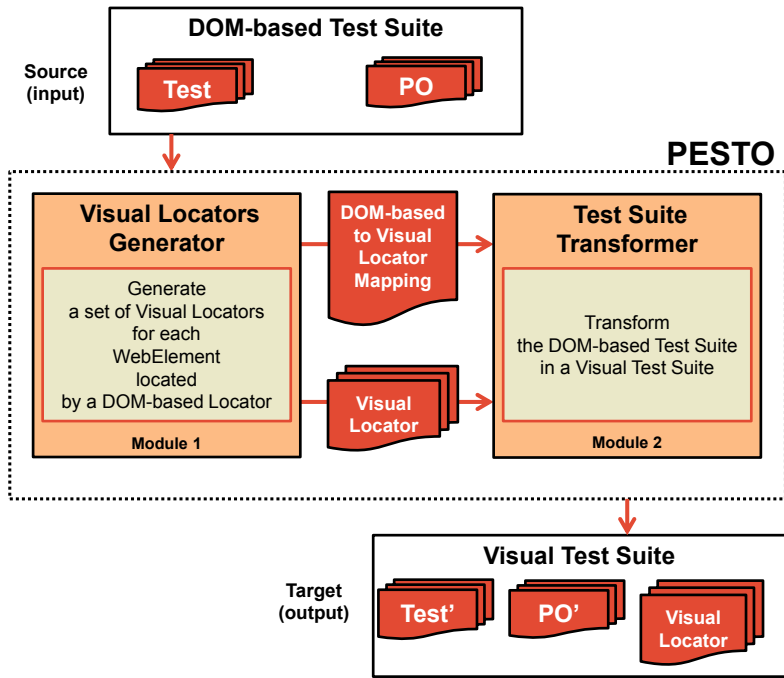


Figure 9. PESTO high level logical architecture

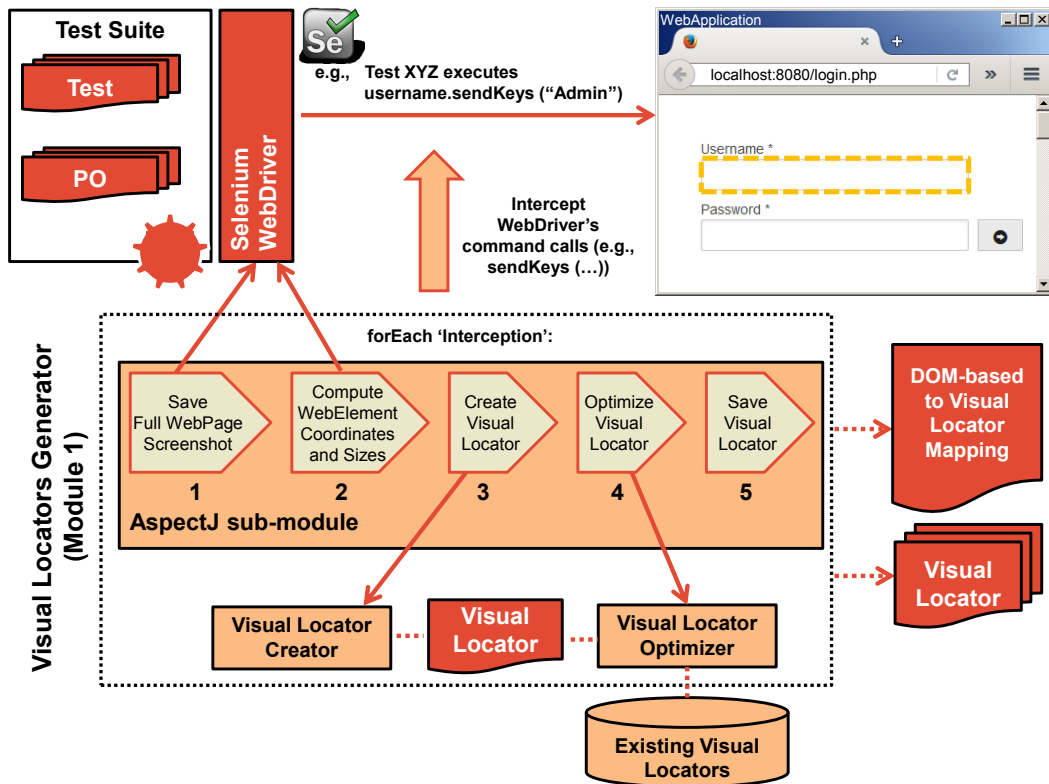


Figure 10. Module 1 of PESTO: Visual Locators Generator

The AspectJ sub-module intercepts such calls (e.g., the call to `sendKeys(...)`, used to fill a text box) before they are carried out, by means of a *before advice*, available in AspectJ, and for each of them it performs steps 1–5, indicated in Figure 10 and detailed below, so as to generate a visual locator for the web element of interest.

Specifically, the AspectJ sub-module performs the following steps:

1. Invoke the WebDriver method `getScreenshotAs(...)`, to create a screenshot of the entire web page (e.g., the login page) containing the web element of interest (e.g., the Username textBox).
2. Invoke two methods available in WebDriver that return the following information about the web element of interest: (i) the coordinates of the top left-hand corner (`getLocation()`) and (ii) its size (i.e., width and height; method `getSize()`).
3. Invoke the Visual Locator Creator, which generates a visual locator for the web element of interest.
4. Invoke the Visual Locator Optimizer, which checks for duplicate locators associated with the same web element.
5. Save the visual locator and keep the mapping between DOM-based locator and visual locator.

For each web element, the saved DOM to visual locator mapping consists of a triple:

(PO Name, Locator Type = Locator Value, Image Path)

for instance:

(HomePage, id="UID", HomePage/username.png)

When the test suite execution is completed, the Visual Locators Generator provides two outputs (see Figure 10): (1) a set of folders, one for each page object, containing the visual locators, and (2) a mapping file that associates the DOM-based locators to their corresponding visual locators.

4.1.1. Visual Locator Creator. As discussed in Section 3 (see Figure 6), often an image representing just the web element, with no surrounding context, cannot be considered a locator. This because such image can not always uniquely locate the target web element. For instance, this happens with forms in which different text boxes have the same size and appearance. In these cases, multiple matches would be found for the perfectly cropped image representing the web element. For this reason, in such cases the Visual Locator Creator expands the size of the rectangle image until a unique locator is found (n expansion steps are indicated in Figure 11). The Visual Locator Creator executes the matching by means of the computer vision library OpenCV¹⁸.

Expansion of the initial locator can be achieved in several ways. The only requirement is that the web element of interest must be kept at the geometric centre of the visual locator, since Sikuli performs actions (e.g., clicks) at the centre of the area that matches the visual locator. We have considered several expansion strategies, such as: (1) one dimension at time (e.g., expand only the width or the height), or (2) both dimensions at the same time. In PESTO we implemented the second approach since it led to the creation of visual locators that are simpler to understand by web testers. More specifically, the algorithm tries to expand the image by the same amount of pixels in both directions until the match is unique. The amount of pixels is a function of the distance of the web element from the closest border of the web page: for instance if the closest border is at 160 pixel distance, the algorithm tries the following four expansion buffers: 40, 80, 120 and 160 pixels. We set the maximum number of expansions to four (4). In our setting and experiments, this value was empirically found to be optimal to strike a balance between the speed of the visual matching algorithm and the quality of the produced locators. However, this value can be fine tuned depending on specific characteristics of the web application under test or the resolution of the screen on which the test cases run (in our setting, screen resolution was 1024x768). Note that in some rare and peculiar cases, a visual locator may not exist at all, e.g., when there are multiple matches of the target element image

¹⁸ <http://opencv.org/>

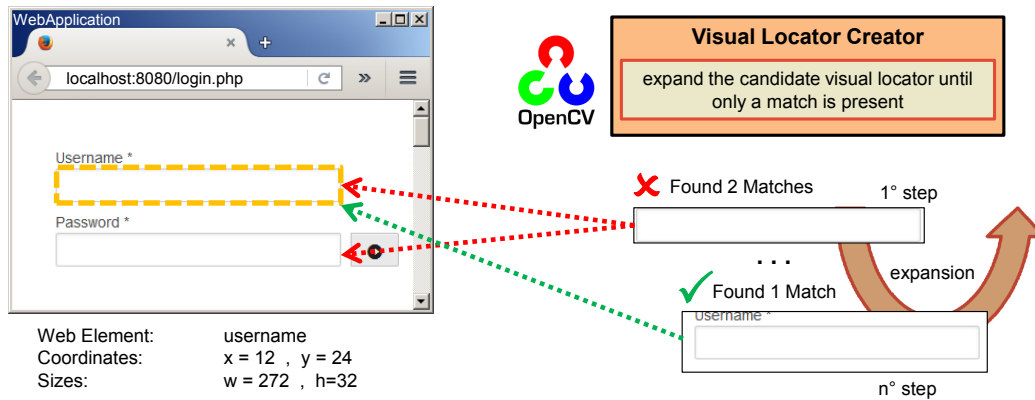


Figure 11. Visual Locator Creator Sub-Module

(i.e., the expansion strategy fails because the web element is too close to the web page borders), or if a match is not found (e.g., the target element is hidden). In such rare cases, the tester may manually repair the visual test case using some different and more complex strategies (e.g., taking another element that can be matched uniquely as a reference point and then moving the mouse cursor by a predefined offset in a proper direction). Note that, to reduce as much as possible the probability of any failure of the expansion strategy, as well as, of false matches during the visual test suite execution, we set the browser to be always displayed full screen.

4.1.2. Visual Locator Optimizer. The Visual Locator Creator generates a correct visual locator (i.e., one able to locate the target web element in the web application under test) only if such web element: (1) is displayed in the web page, and (2) a test case step interacts with it and thus the corresponding command call can be intercepted by AspectJ. During the execution of a test suite, the same web element could occur multiple times (see for instance Figure 12). In such cases, the generated visual locator can be: (a) identical to one generated in a previous interaction (e.g., the username field used to insert the credentials in the application and used in each test cases), (b) different from the previously generated one because of a state change (see, for instance, the 2nd case in Figure 12), (c) different from the previously generated one because of changes in the surrounding elements (see, for instance, the 3rd and 4th cases in Figure 12).

To handle the problem of associating multiple visual locators to a single web element we used the Sikuli MultiStateTarget¹⁹ construct. To keep high the efficiency of the visual test suite, PESTO analyses each created visual locator and compares it with the previously created ones for the target web element in order to recognize and remove the clones (e.g., multiple locators for the same *Username* field). Specifically, the Visual Locator Optimizer takes as input the visual locator generated by the Visual Locator Creator and compares it with visual locators that were previously generated for the web element of interest. If a perfect match is found, the new visual locator is discarded.

4.2. Test Suite Transformer (Module 2)

Module 2 uses the information provided by Module 1 to transform the DOM-based test suite into a visual test suite, as summarized in Figure 13.

The source code transformation has been implemented using the JavaParser²⁰ library. The *Test Cases Transformation* module modifies the source code of test cases in order to use the new visual page objects (PO') instead of the original DOM-based page objects (PO). This step requires to change a few import instructions, so as to replace the old DOM-based POs with the new visual ones. Then,

¹⁹ <https://github.com/sikuli/sikuli-api/blob/master/src/main/java/org/sikuli/api/MultiStateTarget.java>

²⁰ <http://javaparser.github.io/javaparser/>

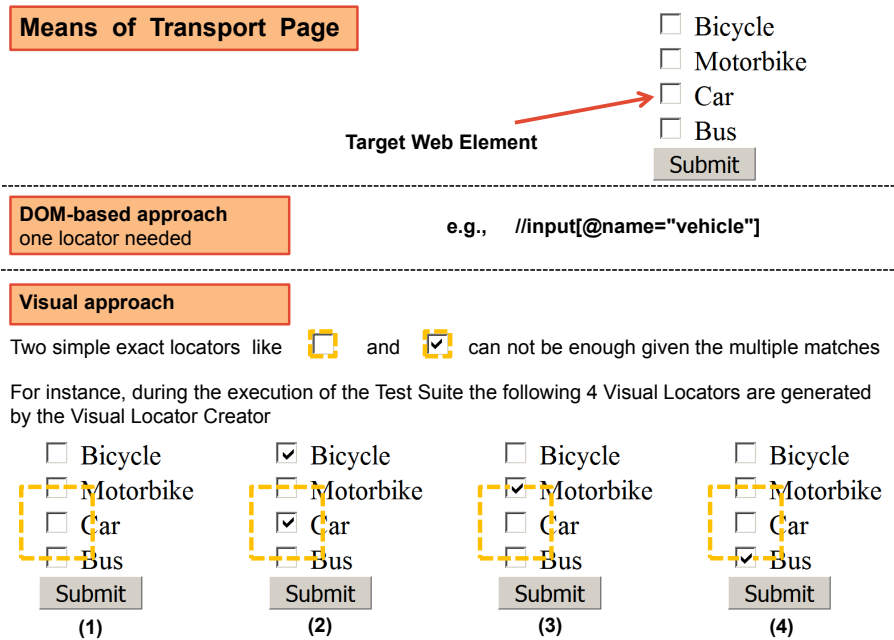


Figure 12. Example of different visual locators for the same web element (yellow dashed lines identify the visual locators)

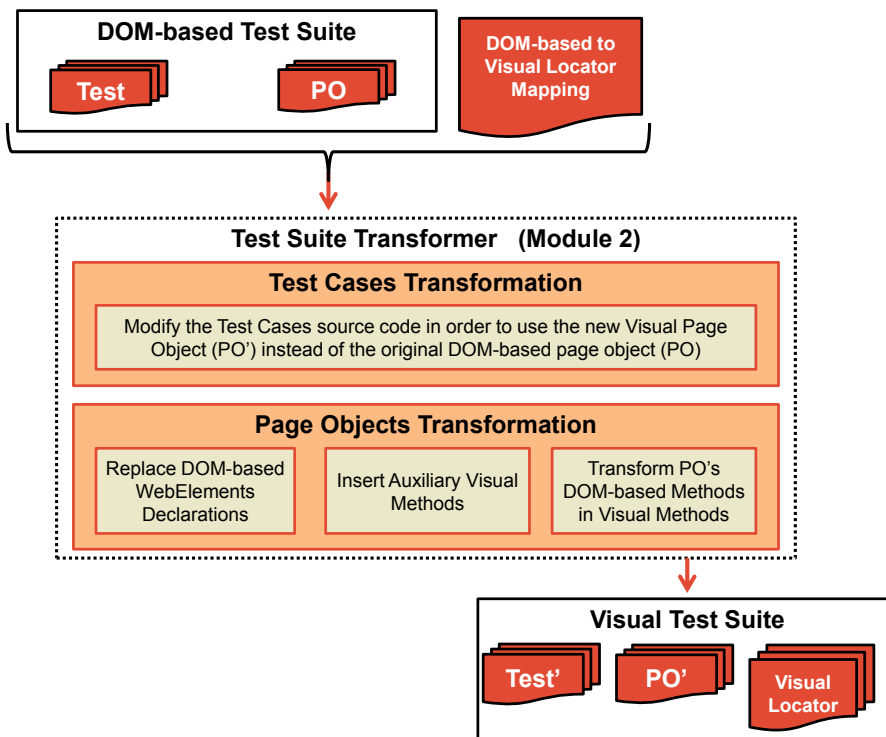


Figure 13. Module 2 of PESTO: Test Suite Transformer

the *Page Objects Transformation* module transforms the DOM-based page objects into new visual page objects, by executing the following three steps:

1) Replace DOM-based *WebElement* Declarations In this step, each declaration of a web element located by a DOM-based locator is removed from the page object and:

- If *only one visual locator* has been created for the web element, the latter is replaced with the declaration of an `ImageTarget` object that contains the path to the image representing the visual locator.

For example, in the case of the Username field shown in Figure 2, PESTO executes the following transformation (the arrow means “is transformed to”):

```
@FindBy(id="UID")
private WebElement username;
```



```
private Target username = new ImageTarget(new File(path+"username.png"));
```

based on the information found in the mapping file:

```
(HomePage, id="UID", HomePage/username.png)
```

The result of this transformation is shown for our running example in Figure 2 and Figure 5 (*Locators Declaration and Initialization* portion).

- If *multiple visual locators* have been created for the web element, the latter is replaced with the declaration of a `MultiStateTarget` object that includes all images representing the visual locators for such an element.

For example, in the case of a `checkBox` associated with multiple visual locators, PESTO executes the following transformation:

```
@FindBy(id="check")
private WebElement checkBox;
```



```
private MultiStateTarget checkBox = new MultiStateTarget();
checkBox.addState(new ImageTarget(new File(path+"checkBox1.png")), "checkBox1");
checkBox.addState(new ImageTarget(new File(path+"checkBox2.png")), "checkBox2");
```

using the following information found in the mapping file:

```
(HomePage, id="UID", HomePage/checkBox1.png)
(HomePage, id="UID", HomePage/checkBox2.png)
```

2) Insert Auxiliary Visual Methods. Sikuli provides only methods to simulate low-level mouse and keyboard operations, through its *mouse* and *keyboard* interfaces. For instance, it is not possible to write directly inside an input field using something similar to the WebDriver command `sendKeys(...)`. The reason is that Sikuli interacts with the complete user interface of the computer. Thus, it is not directly connected to the browser, as in the case of WebDriver. To address this issue, we developed the following auxiliary methods: `locate()`, `click()`, `type()`, `check()`, `selectByVisibleText()`, `selectByIndex()`, `clear()`, which simulate those provided by WebDriver. These auxiliary methods are automatically inserted into the new visual page objects (PO) during the transformation.

locate() (Figure 14). Most basic test execution steps require to locate a web element on the screen. However, it is common to have web pages that are longer than the height of the screen, so the web element of interest might be not visible on the screen. For this reason, the `locate()` method: (1) searches the target element in the visible portion of the page by applying the Sikuli image recognition algorithm and then (2) if the target element is not present, the method automatically scrolls the page down and repeats the search. When a match for the target element is found the corresponding `ScreenRegion` is returned. If the end of the page is reached (i.e., the scroll does not modify the page visualization anymore) and no match is found, the element is not present in the page and thus a null `ScreenRegion` is returned. Figure 14 shows a slightly simplified implementation of this method, where minor details, such as, e.g., the `Thread.sleep(...)` calls — necessary to wait for page loading — are omitted.

```

public ScreenRegion locate(Target element){
    ScreenRegion screen = new DesktopScreenRegion();
    ScreenRegion ris = screen.find(element);
    Mouse mouse = new DesktopMouse();
    while (ris == null){ // web element could be not visible
        mouse.wheel(1,2); // scroll the page down to find it
        ris = screen.find(element);
        if (page.endOfPageReached()) return null;
    }
    return ris;
}

```

Figure 14. Locate auxiliary visual method

click() (Figure 15). Once the web element of interest has been located (i.e., its coordinates are available) it is possible to interact with it. To locate the web element, the above defined method `locate()` is called. Then, a click is triggered at the central point where the match is found. If no match is found, an `EnF` (i.e., `ElementNotFound`) exception is raised.

```

public void click(Target element) throws EnF{
    Mouse mouse = new DesktopMouse();
    ScreenRegion ris = locate(element);
    if (ris == null) throw new EnF();
    mouse.click(ris.getCenter());
}

```

Figure 15. Click auxiliary visual method

type() (Figure 16) allows to type a text inside a web element. In this case it is necessary to: (1) locate the web element; (2) click on it; and, (3) type into it. The first two steps are done by calling the above defined method `click()`. Then, it is sufficient to use the `keyboard` interface of Sikuli.

```

public void type(Target element, String value) throws EnF{
    click(element);
    Keyboard keyboard = new DesktopKeyboard();
    keyboard.type(value);
}

```

Figure 16. Type auxiliary visual method

check() (Figure 17) is used to support the evaluation of assertions. The method verifies if one of the saved images, representing an expected portion of the web application, is present in the page (e.g., a label showing the name of the logged user).

```

public boolean check (Target element){
    ScreenRegion ris = locate(element);
    if (ris == null) return false; else return true;
}

```

Figure 17. Check auxiliary visual method

selectByVisibleText() (Figure 18) is used for selecting an item in a drop-down list using the text shown to the user. The first step requires to get the focus on the drop-down list by (1) locating it and then (2) clicking on it. This is performed by calling the above defined method `click()`. Then, the requested item is selected by resorting on the `keyboard` interface of Sikuli, a mechanism similar to the command available in WebDriver. Given that, during the execution of a DOM-based test case, WebDriver selects directly the item from the drop-down list by text, it is not possible to obtain

the information (multiple images, see complex interactions described in Section 3.3) necessary for creating the visual locators required to interact with the drop-down list. However, by means of the Sikuli keyboard interface it is possible to directly type the string representing the visible text, so as to select the desired item in the drop-down list, with no need of scrolling the drop-down list and matching its elements visually. This simplifies enormously the visual interaction and reduces the execution time of the visual test suite.

```
public void selectByVisibleText (Target element, String value) throws EnF{
    click(element);
    Keyboard keyboard = new DesktopKeyboard();
    keyboard.type(value);
    keyboard.type(Key.ENTER);
}
```

Figure 18. selectByVisibleText auxiliary visual method

selectByIndex() (Figure 19) is used for selecting an item in a drop-down list using the index associated to such item. The index represents the position of the item within a drop-down list (list indexes start at 0). As in the case of `selectByVisibleText(...)`, the `click()` method can be used to get the focus on the drop-down list and then, by means of the `keyboard` interface of Sikuli, we select the requested item by: (1) moving at the beginning of the list (necessary if an item is already selected), by pressing the `PAGE_UP` key, and (2) finding the item at the right position, by generating a sequence of `DOWN` keystrokes.

```
public void selectByIndex (Target element, int value) throws EnF{
    click(element);
    Keyboard keyboard = new DesktopKeyboard();
    keyboard.type(Key.PAGE_UP);
    for(int i=0; i<value; i++) {
        keyboard.type(Key.DOWN);
    }
    keyboard.type(Key.ENTER);
}
```

Figure 19. selectByIndex auxiliary visual method

clear() (Figure 20) is used to erase the text contained in a web element such as a text box. There are several possible implementations of this functionality. PESTO implements the following strategy: (1) locate the web element, (2) perform a triple-click on the web element (i.e., the action of clicking with the mouse button three times quickly without moving the mouse), so as to highlight all the text it contains, and (3) erase the selected text by pressing the `BACKSPACE` key by means of the `keyboard` interface of Sikuli. We did not use the auxiliary method `click()` because each time it is called it executes the image recognition algorithm and thus it is not fast enough to perform a triple-click. Another possible implementation of this method, in case the triple-click is not supported, consists of the following actions: (1) click on the web element of interest and then (2) press the `BACKSPACE` key a high number of times (e.g., 200). We found however this latter method generally slower than the former.

3) Transform DOM-based Methods in the PO into Visual Methods. Each method of the page object is transformed in order to make it compliant with the visual approach. For instance, each call to the `sendKeys` method available in `WebDriver` (e.g., `username.sendKeys("Admin")`) becomes a call to the corresponding auxiliary visual method (`type(username, "Admin")`). It should be noticed that while in the `WebDriver` test suite `username` is a `WebElement` (a specific `WebDriver` type), located using a DOM-based locator, in the Sikuli test suite, `username` is an `ImageTarget` (or a `MultiStateTarget`) representing one or more visual locators. In detail, the transformed method calls are shown in Figure 21.

```

public void clear (Target element) throws EnF{
    Mouse mouse = new DesktopMouse ();
    ScreenRegion ris = locate(element);
    if (ris == null) throw new EnF();
    mouse.click(ris.getCenter());
    mouse.click(ris.getCenter());
    mouse.click(ris.getCenter());
    Keyboard keyboard = new DesktopKeyboard();
    keyboard.type(Key.BACKSPACE);
}

```

Figure 20. Clear auxiliary visual method

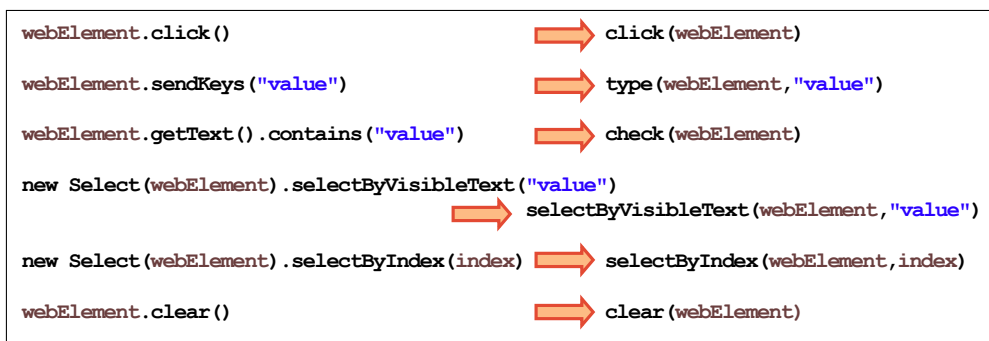


Figure 21. Method calls transformations

The input/output page objects associated with this transformation can be seen in Figure 2 and Figure 5 (*PO Methods* portion), for our running example.

For the test case assertions we adopt a specific transformation template. In particular, PESTO assumes that each assertion contains a call to a page object method that takes as input the expected textual value (the string to match with the text contained in a web element) and returns a boolean value (in practice, we used the JUnit `assertTrue` statement). Thus, the comparison between the expected value and the actual text contained in the web element is executed in the page object (e.g., see the assertion implementation in the WebDriver test case shown in Figure 3 and the corresponding PO method shown in Figure 2). Our transformation produces a visual page object where each textual match in the original assertions becomes an image match. In this way, we maintain a similar semantics as that of `getText(...)`, even if the command execution is not exactly the same, since an image is provided as expected value instead of a string. Indeed, the assertions of the DOM-based test cases check that the value of the expected string is contained in the target web element, while in the case of the visual test cases, assertions check that the target web element is displayed in the web page with the expected visual appearance. This is due to the fact that in the case of the DOM-based approach 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. As we have seen, the generation of the images that replace the strings used in the assertions is performed automatically by Module 1.

Note that, if present, any other DOM-based command not supported by PESTO is simply copied in the generated test suite, which becomes a hybrid²¹ DOM-based / visual test suite. Indeed, DOM-based testing tools can provide commands that have no counterparts in the visual approach (e.g., the management of the cookies) or for which PESTO is not able to create a visual locator (e.g., `accept()` and `selectByValue()`), see Section 6.5).

5. PESTO IMPROVEMENT PROCESS

Starting from the preliminary prototype of PESTO, which was implemented in the context of our previous work [9, 10], we decided to extend PESTO based on the limitations observed during migration of existing DOM-based test suites to the visual approach. Thus, we searched for existing test suites that could be used to guide the development and improvement of PESTO. We call such set of web applications and related test suites the *training set*. Unfortunately, we were not able to find a set of DOM-based test suites suitable that could constitute a valid training set for PESTO, i.e., a set of real size, open source Selenium WebDriver DOM-based test suites for web applications, adopting the Page Object and Factory patterns. Indeed, we found only: (1) test suites that did not adopt the Page Object and Factory patterns, or (2) trivial test suites consisting of a single test case and a single page object interacting with few web elements (i.e., simple code examples for the Page Object and Factory patterns).

In our industrial experience, such test suites do exist [14], but they are rarely made available outside the company. Thus, we decided to construct our own training set, given our expertise in the area of web testing [14, 15, 5].

The training test suites were important to fine tune and improve PESTO, so as to reach a high level of automation in the migration to the visual approach (i.e., the manual effort should be as low as possible) and a high coverage of the employed WebDriver commands (i.e., the residual DOM-based commands should be as few as possible).

The goal of the PESTO improvement process was to reach a high level (ideally, 100%) of automation and coverage in the migration of DOM-based test suites to the visual approach.

5.1. Considered Web Applications

For the training test suites, we chose four open-source web applications from SourceForge.net which are popular (downloaded thousands of times), recent (2008-2014), span different domains, and span from small (12 kLOC) to medium size (352 kLOC). In the following, we report a short description for each selected application.

PHP Password Manager (PPMA)²² is a web based password manager. Each password is encrypted with an individual user password. It is based on the Yii Framework²³. More specifically, we use release 0.2, which is composed overall of 296 kLOC, of which 277 kLOC are PHP code and 9 kLOC are Javascript, while the remaining are mainly XML, CSS, HTML, and SQL code.

Claroline²⁴ is an open source collaborative learning environment allowing teachers or education institutions to create and administer courses through the browser. The system provides group management, forums, document repositories, calendar, chat, assignment areas, links, user profile administration within a single, highly integrated package. Claroline is available in 35 languages and used by hundreds of institutions around the world. More specifically, we use release 1.11.10-1, which is composed overall of 352 kLOC, of which 300 kLOC are PHP code and 41 kLOC are Javascript, while the remaining are mainly HTML, CSS, SQL, and XSD code.

²¹ The visual test suites produced by PESTO always contain also the references to the WebDriver libraries and the corresponding import directives of the original DOM-based test suites. Thus, having DOM-based commands in the visual test suite does not create any problem. ²² <http://sourceforge.net/projects/ppma/> ²³ <http://www.yiiframework.com/>

²⁴ <http://sourceforge.net/projects/claroline/>

PHP Address Book²⁵ is a simple, Web-based address and phone book, contact manager, and organizer. More specifically, we use release 8.2.5, which is composed overall of 33 kLOC, of which 30 kLOC are PHP code and 1 kLOC are Javascript, while the remaining are mainly HTML, SQL, and CSS code.

Meeting Room Booking System (MRBS)²⁶ is a system for multi-site booking of meeting rooms. Rooms are grouped by building/area and shown in a side-by-side view. Although its goal was initially to book rooms, MRBS can also be used to book any resource. More specifically, we use release 1.2.6.1, which is composed overall of 12 kLOC, of which 9 kLOC are PHP code and about 300 LOC are Javascript, while the remaining are mainly HTML, SQL, and CSS code.

All the selected applications, regardless of their size in LOC, include quite complex web pages, as shown by the examples reported in Figure 22. In our experience, the complexity of the web pages of these open source web applications is comparable with the one that can be found in common industrial web applications [14, 27].

5.2. Training Test Suite Development

We created a training test suite for each of the four selected applications: PPMA, Claroline, Address Book, and MRBS. Our experience with WebDriver is comparable at least to that of a Junior Tester, since we participated in several joint academic / industrial projects on web testing (see for instance [14, 27]). The training test suites exhibit the following characteristics: (1) the test cases adopt the *Page Object* and *Page Factory* patterns, (2) the test cases interact with a relevant portion of the web elements rendered on the web pages (e.g., links, buttons, check boxes, drop-down lists), (3) the web elements with multiple visual rendering (e.g., check boxes or text boxes) are stimulated in multiple states (checked/unchecked or filled/empty), and (4) the test cases contain assertions on visible web elements (i.e., visually rendered), because assertions on hidden fields or internal attributes of the DOM cannot be managed by visual testing tools. It is important to highlight that we implemented the training test suites using the more adequate constructs so as to mimic the activity of a real tester.

5.3. PESTO Improvement

After the development of the training test suites, we tried to migrate them to the visual approach by means of the preliminary version of PESTO [9, 10] (see Figure 23).

As expected, PESTO was able to migrate only a subset of the employed WebDriver commands (e.g., the calls to the *Clear* and *Select* commands were not transformed). Hence, the result were *hybrid test suites*, containing both (untransformed) DOM-based commands and (transformed) visual commands. Such hybrid test suites did not run correctly, since the web elements with different visual states were not managed properly. Thus, we started to improve PESTO, following the process depicted in Figure 23. We improved PESTO each time an error was found i.e., a command left untransformed or the generated test suite not behaving as the DOM-based one. This allowed us to fine tune PESTO, remove bugs and reach a high level of automation and coverage in the migration task.

5.4. Summary of the Results on the Training Test Suites

Table I summarizes the most important information about the four DOM-based (WebDriver) training test suites and the corresponding Visual (Sikuli) test suites as generated by PESTO at the end of the improvement process. In particular, each test suite has from 21 to 53 test cases and from 6 to 9 page objects. The size of the four DOM-based training test suites, measured as Java lines of code (LOC), varies from 970 to 2217 LOCs. These test suites would be probably considered as small/medium in the industry, although they are similar in terms of size, number of test cases/page objects, and complexity of the test cases to the ones that we used in a previous industrial case study where the benefits deriving from the introduction of the page object pattern were evaluated [14].

²⁵ <http://sourceforge.net/projects/php-addressbook/> ²⁶ <http://sourceforge.net/projects/mrbs/>

Course title *
e.g. History of Literature

Code *
max. 12 characters, ie.ROM2121

Categories

Linked categories

Unlinked categories

Feel free not to associate courses to any categories.
The categories appearing in grey are invisible categories.

Language *

Lecturer(s)

Email

Course access * Access allowed to anybody (even without login)
 Access allowed only to platform members (user registered to the platform)
 Access allowed only to course members (people on the user list)

Registration settings *

Allowed
 Allowed with validation
 Allowed with enrolment key
 Denied

Breve Descrizione

Descrizione Completa:
(Numero di persone, Interno/Esterno ecc..)

Data:

ora: :

Durata: Tutto il giorno

Stanze: Usa il tasto Ctrl per selezionare piu' di una stanza

Tipo

Tipo ripetizione: Nessuno Giornaliero Settimanale Mensile Annuale Mensile

Ripeti fino al:

Ripeti Girno: (per (n) settimane) domenica lunedì martedì mercoledì giovedì venerdì sabato

Numero di settimane: (per (n) settimane)

Vedi Giorno: dic 31 | gen 01 | gen 02 | gen 03 | gen 04 | gen 05 | [gen 06] | gen 07 | gen 08 | gen 09 | gen 10 | gen 11
Vedi Settimana: dic 06 | dic 13 | dic 20 | dic 27 | [gen 03] | gen 10 | gen 17 | gen 24 | gen 31
Vedi Mese: nov 2015 | dic 2015 | [gen 2016] | feb 2016 | mar 2016 | apr 2016 | mag 2016 | giu 2016 | lug 2016

Figure 22. Examples of web pages containing various kinds of web elements (Claroline and MRBS applications). These images are portions of screenshots saved by PESTO during its execution.

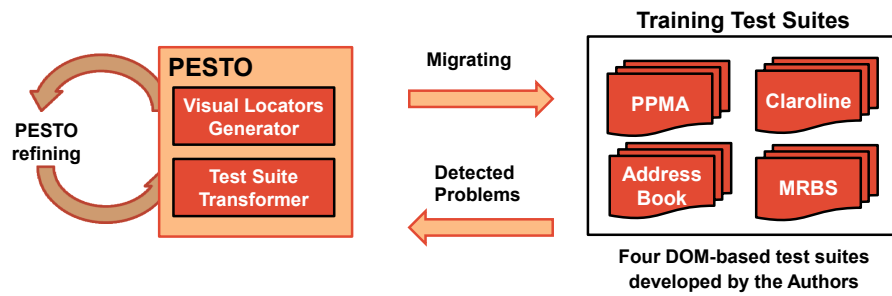


Figure 23. PESTO development process

	Test Cases	Page Objects	WebDriver		Sikuli	
			LOC ^a	DOM-based Locators	LOC ^a	Visual Locators
PPMA	38	9	1305	89	2245	166
Claroline	53	7	2217	267	3296	476
Address Book	21	6	970	83	1571	145
MRBS	24	7	1134	98	1879	157
	136	29	5626	537	8991	944

^a Java LOC - Comment and Blank lines are not considered

Table I. General information on the DOM-based training test suites and the corresponding visual ones generated by PESTO

The number of `WebElement` objects (corresponding to the number of DOM-based locators), used to declare the web elements the test cases interact with, ranges from 83 in the case of Address Book to 267 in the case of Claroline. On the other hand, the generated Visual test suites are larger, varying from 1571 to 3296 LOCs, and the number of Visual locators is almost doubled (944), mostly because the interactions with web elements changing their state require multiple visual locators.

Table II provides additional details on the WebDriver commands used in the four DOM-based training test suites. The three most used command are `click()`, `sendKeys()`, and `getText()`, that were also the three commands supported in the preliminary prototype of PESTO [9, 10]. However, the new supported commands (`selectByVisibleText()`, `selectByIndex()`, and `clear()`) turned out to be fairly common (in total 189 calls out of 942).

	WebDriver Command Calls						
	<code>click()</code>	<code>sendKeys()</code>	<code>getText()</code>	<code>selectByVisibleText()</code>	<code>selectByIndex()</code>	<code>clear()</code>	<code>accept()</code>
PPMA	41	28	30	0	0	7	2
Claroline	273	51	49	46	39	8	4
Address Book	42	60	21	13	13	25	0
MRBS	97	25	29	21	13	4	1
	453	164	129	80	65	44	7

Table II. Interaction commands used in the training test suites

Overall, PESTO was able to manage almost all the command calls used in the four training test suites (935 calls out of 942). In the remaining 7 cases (all `accept()` command calls, used for dealing with *alert* windows), PESTO simply copied the command calls from the original training test suite to

the output test suite, hence creating a hybrid test suite. The resulting test suites could be compiled and executed without requiring any manual intervention. In detail, in the case of Address Book the resulting test suite was completely Visual (since no `accept()` is used for Address Book), while in the other three cases the generated test suites contained also DOM-based command calls. To turn them into fully visual, 3rd generation test suites, the 7 residual command calls that are currently not handled by PESTO must be manually transformed (the reason why the `accept()` call is not supported by PESTO is discussed in Section 6.5; in short, WebDriver does not provide position information for *alert* windows). The migrated test suites contain a total of about 99% of visual command calls and just about 1% untransformed DOM-based command calls.

After having generated the visual test suites with the final version of PESTO, we executed them and no web element localisation, interaction or assertion evaluation errors emerged. Indeed, when we executed the generated visual test suites we had the same results obtained with the original DOM-based training test suites. The correctness of the execution of the test suites has been checked manually. In detail, we executed each generated visual test suite and we visually verified that it behaves as expected. This is an easy task since PESTO highlights each web element the test cases interact with (see an example in the PESTO demo video²⁷).

To summarize, PESTO was able to automatically produce four compilable, executable and fully working test suites starting from the four DOM-based training test suites. The migration process required no human intervention (i.e., it was 100% automatic) and the resulting test suites contained on average about 99% of visual command calls and just about 1% of untransformed DOM-based command calls (all of them `accept()` calls).

6. PESTO EVALUATION

This section describes the results obtained in the experimental evaluation of PESTO. As already discussed in Section 5, finding valid DOM-based test suites suitable for PESTO turned out to be a challenge. For this reason, to evaluate our tool without introducing any author bias we recruited a professional tester (hereafter referred to as *Tester*) who independently developed two test suites covering the most relevant functionalities of two open-source web applications. The recruited Tester has 2 years of professional experience in testing web applications using Selenium WebDriver and the page object / factory patterns. At the time of the test suites development the Tester had no information about the existence of PESTO and the goal of the current work. We limited the working time of Tester to five days corresponding to 40h²⁸.

Our empirical study aims at answering the following research questions.

RQ1 (automation): *What is the amount of manual intervention required to migrate DOM-based test suites?*

– **RQ1.1:** *What are the issues encountered by the Visual Locators Generator (Module 1) of PESTO when creating visual locators for web elements?*

– **RQ1.2:** *What are the compilation and execution issues faced by the Test Suite Transformer (Module 2) of PESTO when generating visual test suites?*

RQ2 (correctness): *How many migrated visual test suites have issues when locating, interacting with, and asserting on the web elements under test?*

RQ3 (multistate): *How often are multiple visual locators, associated with a single web element, required?*

²⁷ The video showing a sample execution of PESTO is available at <http://sepl.dibris.unige.it/PESTO.php> (in particular from min 2:30 where the execution of the generated visual test suite is shown). ²⁸ 8h for selecting the web applications to test, and 32h for developing the corresponding DOM-based test suites.

RQ4 (applicability): *What percentage of the original DOM-based command calls is left untransformed by PESTO, because they are currently not handled by the tool?*

6.1. Web Applications

To remove any selection bias concerning the tested web applications we asked Tester to select two open-source web applications from *SourceForge.net*. It should be noticed that Tester did not have any information about the existence of PESTO.

For the selection, we gave some prerequisites to Tester: (1) excluding trivial web applications: the selected applications should contain different kinds of web elements such as links, text boxes, submission buttons, check boxes, drop-down lists; (2) excluding obsolete web applications: their last release should be recent; and, (3) selecting realistic web applications: they should be well-known and used, having at least several thousands of downloads per year.

Tester, employed a working day (8h) for the selection task, resulting in the following choice of applications:

TikiWiki²⁹ an open source Wiki-based content management system and online office suite. This is a mature web application (its first release dates 2002) and it is under active development by a large international community of over 300 developers and translators. More specifically, it is composed overall of 2517 kLOC, of which 975 kLOC are PHP code and 281 kLOC are Javascript, while the remaining are mainly XML, CSS, HTML, XML, and XSD code.

Nibbleblog³⁰ an engine for creating blogs. It is based on PHP and it is quite simple to install and configure, since it relies on an XML database. It was first released in 2009. More specifically, it is composed overall of 10 kLOC, of which 5 kLOC are PHP code and 2 kLOC are Javascript, while the remaining are mainly CSS, and HTML code.

For both web applications, Tester selected the last major release available on *SourceForge.net* at the time of the experimentation: (1) TikiWiki 14.0 (released on May 2015) and (2) Nibbleblog 4.0.5 (released on September 2015). Both applications, regardless of their size in LOC, include quite complex web pages, as shown by the examples reported in Figure 24. In our experience, the complexity of the web pages of these open source web applications is comparable with the one that can be found in common industrial web applications [14, 27].

6.2. Test Suite Development

We asked Tester to adopt a systematic approach for the creation of the test suites, consisting of three steps: (1) discovering the main functionalities of the target web application from the available documentation or by navigating it; (2) covering the main functionalities with at least one test case, assigning a meaningful name to each test case, so as to keep the mapping between test cases and functionalities; (3) adopting the *Page Object* and *Page Factory* patterns, and (4) defining assertions for web elements that are rendered visually using the `assertTrue` statement as described in Section 4.2.

6.3. Experimental Procedure

For each selected application, the following steps have been executed:

1. **DOM-based Test Suite Development.** The Tester developed a DOM-based test suite. The tester had two working days (16h) available to develop the test suite for each web application (so, in total 32h have been devoted by Tester to this task).
2. **DOM-based Test Suite Transformation.** After the delivery of the test suite we executed PESTO (see Figure 25). The Visual Locators Generator (Module 1) created the visual locators and the mapping file. Then, the Visual Test Suite Transformer (Module 2) generated the visual test suite. We noted down (a) each problem encountered by PESTO and (b) each modification

²⁹ <http://sourceforge.net/projects/tikiwiki/> ³⁰ <http://sourceforge.net/projects/nibbleblog/>

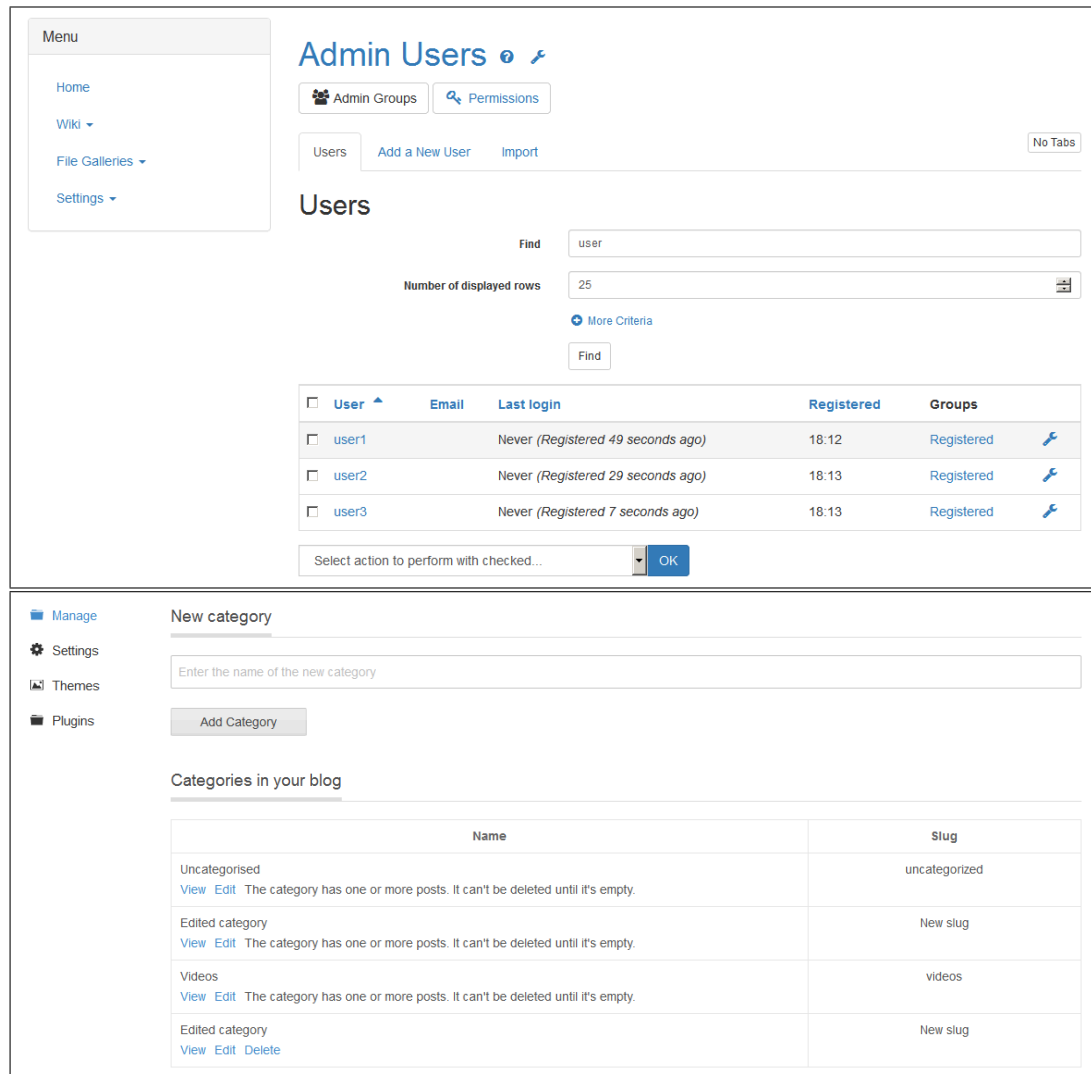


Figure 24. Examples of web pages containing various kinds of web elements (Tiki above and Nibbleblog below). These images are portions of screenshots saved by PESTO during its execution.

required to the original or the generated test suites during the execution of the two modules of PESTO.

3. **Visual Test Suite Evaluation.** The newly generated visual test suite was executed to check for its correctness as done in the case of the training web applications (see Section 5.4).

6.4. Experimental Results

Table III summarizes the most important information about the two DOM-based test suites. They have similar size, both in term of test cases/page objects and LOCs (this is probably due to the 16h time limit given to Tester for developing each test suite). Concerning the size/complexity of the test suites, the same considerations reported for the training test suites hold, see Section 5.4: both web applications and test suites would be considered of small-medium size in the industry.

RQ1 (automation): *What is the amount of manual intervention required to migrate DOM-based test suites?*

To answer RQ1 we first consider the results concerning RQ1.1 and RQ1.2.

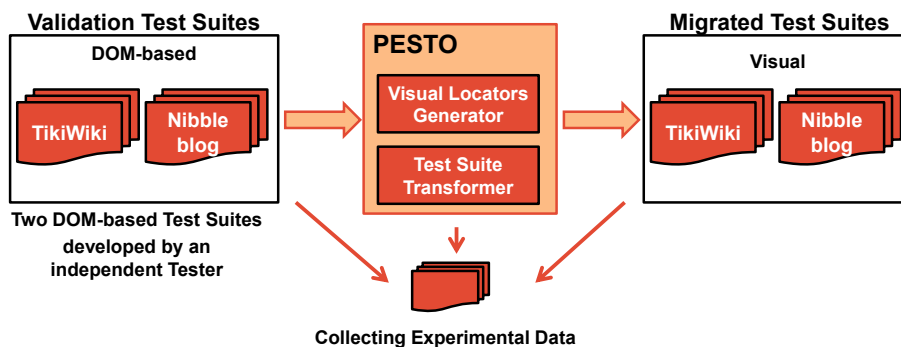


Figure 25. PESTO evaluation

	Test Cases	Page Objects	WebDriver		Sikuli	
			LOC ^a	DOM-based Locators	LOC ^a	Visual Locators
TikiWiki	30	11	1467	103	2835	307
Nibbleblog	31	10	1461	89	2495	142
	61	21	2928	192	5330	449

^a Java LOC - Comment and Blank lines are not considered

Table III. General information on the DOM-based test suites and the corresponding visual ones, generated by PESTO

RQ1.1: *What are the issues encountered by the Visual Locators Generator (Module 1) of PESTO when creating visual locators for web elements?*

Generation of the visual locators (i.e., the result of the execution of Module 1) was successful in the case of the TikiWiki test suite, whereas in the case of Nibbleblog one visual locator out of 142 was not created. The problem concerned the position of such web element on the screen, being very close to the page border. In this corner case, the expansion strategy of the Visual Locators Generator failed and the visual locator was not created. Such web element was used for evaluating an assertion at the end of a test case and thus did not impact on the creation of the visual locators for the other web elements in the test case. We modified the assertion in order to perform a very similar check but using a different web element on the page (i.e., we changed only one DOM-based locator in the test suite) and re-generated the visual locator for such web element by re-executing PESTO. Thus, the visual locators generation phase required to modify only 1 DOM-based locator out of 192 used in the two test suites and produced a total of 449 Visual locators.

RQ1.2: *What are the compilation and execution issues faced by the Test Suite Transformer (Module 2) of PESTO when generating visual test suites?*

The transformation of the DOM-based test suites into the corresponding visual ones (i.e., the execution of Module 2) required only minor manual refactorings performed on the code delivered by Tester before executing the Test Suite Transformer (Module 2). The Tester indeed used a syntactic variant in the declaration and use of the WebDriver Select objects used for interacting with the drop-down list. Specifically, in one case the Tester used:

```
multCatPositionEntry = new Select(categoryPosition);
multCatPositionEntry.selectByIndex(newPosition);
```

while PESTO (Test Suite Transformer) supports the following more compact, though equivalent, syntax:

```
new Select(categoryPosition).selectByIndex(newPosition);
```

Thus, we modified the test suites in order to adopt the syntax supported by PESTO (Test Suite Transformer), by modifying 35 LOCs (i.e., 11 LOCs removed, containing declarations of Select objects, and 24 LOCs changed, so as to perform the selection as supported by PESTO). Then, we re-executed the Test Suite Transformer and obtained a new visual test suite. Eclipse displayed a syntactic error in both visual test suites. In both cases, errors are due to the negation of a boolean return value in a method. We never considered negation of boolean return values in our test suites and thus this problem did not emerge during the improvement of PESTO. We report one of the two errors, concerning the following line of code in the Nibbleblog DOM-based test suite:

```
return !addedCategory.getText().contains(name);
```

that was translated into the following incorrect line of Sikuli Java code:

```
return check(!addedCategory);
```

while the correct Java code should have been:

```
return !check(addedCategory);
```

Thus, we corrected these 2 LOCs and proceeded with the execution of the visual test suites, which at this point completed successfully.

To summarize, for what concerns research question **RQ1**, we can say that PESTO was able to transform the two considered DOM-based test suites to the visual approach at a lower cost compared to a manual migration. Indeed only 38 LOCs required manual modifications (36 in the original DOM-based test suites and 2 in the newly generated visual test suites).

RQ2 (correctness): *How many migrated visual test suites have issues when locating, interacting with, and asserting on the web elements under test?*

The execution of the Nibbleblog visual test suite was completed correctly, i.e., each test case behaved as expected (both commands and assertions) and all the test cases passed successfully. Instead, during the run of the visual test suite for TikiWiki, we obtained an error due to a delay in the transition between two pages. The image recognition algorithm of Sikuli started to search for the target web element during such transition. The problem was solved by introducing a delay of 1s in the test case (i.e., `Thread.sleep(1000);`), before the line which gave rise to the problem.

To summarize, for what concerns research question **RQ2**, we can say that the visual test suites generated by PESTO behaved as the DOM-based ones (both commands and assertions); only in one case, a minor modification was required to manage the web page loading time.

RQ3 (multistate): *How often are multiple visual locators, associated with a single web element, required?*

Table III allows us to understand the importance of associating multiple visual locators to a web element. Indeed, for the 192 web elements (and DOM-based locators), PESTO created 449 visual locators. While a large proportion of the web elements (51%) are associated with only one visual locator, 20% are associated with two (e.g., typically in the case of checkboxes) and 29% with three or more visual locators, like for instance in the case of drop-down lists, which display different values in different test cases. The importance of managing multiple visual locators for the same web element may depend on the complexity of the DOM-based test suite developed for testing the web application. In fact, the more test cases are produced, the higher the probability that a test case may interact with a given web element, displayed in a different visual state in a previous test case.

To summarize, for what concerns research question **RQ3**, we can say that the capability of associating multiple visual locators to the same web element was fundamental for migrating the considered test suites to the visual approach. A large proportion of the web elements (49%) are associated with two or more visual locators.

RQ4 (applicability): *What percentage of the original DOM-based command calls is left un-transformed by PESTO, because they are currently not handled by the tool?*

Table IV reports detailed information about the commands employed by Tester in the two test suites. All the 379 handled command calls used in the two test suites have been transformed automatically into their corresponding visual counterparts. Among the possible choices, Tester chose to interact with the drop-down list always using the `selectByIndex(...)` command and thus no `selectByVisibleText(...)` was used. Moreover `accept()`, the only command that PESTO was not able to transform when executed on our training test suites (see Section 5), was never used by the tester.

To summarize, for what concerns research question **RQ4**, we can say that all the commands used by Tester were transformed to the visual approach in a syntactically correct way and thus the generated test suites were 100% visual (i.e., no command calls are left un-transformed) and compilable. Of course, syntactic correctness alone is not enough and a few simple manual interventions were needed to fix some semantic problems (e.g., the delay manually added to the TikiWiki visual test suite, described when discussing RQ2).

WebDriver Command Calls							
	<code>click()</code>	<code>sendKeys()</code>	<code>getText()</code>	<code>selectByVisibleText()</code>	<code>selectByIndex()</code>	<code>clear()</code>	<code>accept()</code>
TikiWiki	113	38	42	0	6	8	0
Nibbleblog	76	20	46	0	18	12	0
	189	58	88	0	24	20	0

Table IV. Interaction commands used in the test suites

6.5. Discussion

Advantages of automating the generation of visual locators. For migrating a DOM-based test suite to the visual approach, typically hundreds of visual locators are required. The Visual Locators Generator (Module 1) of PESTO creates automatically images representing the web elements of interest and verifies whether: (1) the images work properly as visual locators; (2) there are previously generated images representing exactly the same web element, so as to remove duplicates (this task is performed by the Visual Locator Optimizer sub-module described in Section 4.1.2). These tasks, if performed manually by the tester, can be not trivial. The use of an automated tool such as PESTO can reduce the manual effort.

In our previous work [5], we discovered that the creation of visual locators is one of the reasons that lead to a relevant increment (i.e., $\geq 50\%$) of the time required to develop visual test suites with respect to DOM-based ones³¹. Thus, we can estimate in 6 working days (i.e., 48h) the human effort saved by PESTO, since the DOM-based test suites required a total of 4 working days ($16h \times 2$) for their creation. Actually, some manual work is still required after the transformation is completed, due to limitations of PESTO. In the case of the test suites, we had to modify 38 LOCs. The associated effort can be over-approximated in 2h overall for both the test suites. So, we can compare 10 working days (80h) for the fully manual development of both DOM-based and visual test suites against 4 working days plus 2h, for manually fixing the transformation outcome (34h), when PESTO is used, with a strong manual effort saving. Considering the development of visual test suites alone, the saving is remarkably higher: 48h vs. 2h. Note that these evaluations of the development effort saved by PESTO are hypothetical and thus would require further studies to be confirmed.

³¹ In such previous work we adopted Sikuli as 3rd generation tool; in case of adoption of a 3rd generation tool with recording functionality, able to capture the test suites, and thus also the locators, the time could be lower.

Machine time required for migrating the DOM-based test suites with PESTO. For the two test suites used in our experimental evaluation, we have: (1) executed each test suite, as delivered by Tester, on our machine (a laptop equipped with an Intel Core i5 dual-core processor (2.5GHz) and 8 GB RAM hosting both the test suite and the web server with the web application); this task required 8 minutes and 4 minutes for TikiWiki and Nibbleblog respectively; (2) executed each test suite with PESTO for creating the visual locators (19 minutes and 10 minutes required respectively for TikiWiki and Nibbleblog); and, (3) executed each generated visual test suite to verify its correctness (51 minutes and 26 minutes required respectively for TikiWiki and Nibbleblog). Thus our machine worked without any human intervention for 78 minutes in the case of TikiWiki and 40 minutes in the case of Nibbleblog (around 2h overall). We deem such machine time compatible with the typical web development and testing process.

Execution time of the visual test cases created by PESTO. As already observed in our previous work [5] as well as by Alégroth et al. [4], the execution of the visual test suites is remarkably slower than the execution of the DOM-based test suites. This is due to several factors: (1) execution of the image recognition algorithm, which requires more computational resources (and thus, generally, time) than navigating the DOM; (2) the delays introduced for managing web page loading (e.g., through sleep commands). The latter are due to the time the browser needs to load a web page: before performing actions on the page, the test automation tool must wait for complete page loading into the browser. WebDriver provides specific commands to deal with web page loading. In Sikuli this feature is not available and the tester has to insert an explicit delay (e.g., `Thread.sleep(200)`).

Concerning factor (1), the Visual Locator Optimizer removes the unnecessary image duplicates, thus reducing the computational effort when multiple visual locators have to be tried before a successful one is found. For what concerns factor (2), PESTO adopts a conservative approach. PESTO inserts some small delays (e.g., `Thread.sleep(200)`) in the implementation of the auxiliary visual methods, so as to allow the visual test cases to start the execution of the image recognition algorithm when the new page is likely to be fully loaded. As reported in the results, this workaround worked in almost all cases (see the answer to RQ2 (correctness)). However, the tester might fine tune such delays in PESTO, in order to adapt them to the specific characteristics of the web application under test and its execution environment. Alternatively, it would be possible to adopt dynamic delays. This solution basically consists of frequently (e.g., every 50ms) searching for the web element matching the current visual locator until the target web element is found or a maximum time is spent. In the second case, the `locate` method described in Section 4.2 would continue to scroll the page down and repeat the search. One of the problems of dynamic delays consists of false matches occurring when the rendering of the page has not yet finished and the target web element is temporarily shown in a place different from its final position. We plan to further investigate this solution in our future work.

Visual locators readability. Readability of the automatically generated locator images becomes particularly important when the visual test suite is manually maintained/evolved³². The Visual Locators Generator algorithm of PESTO has been designed to mimic the visual selection strategy of a human tester, i.e., crop a rectangle area around the element of interest, including locator-specific visual features. The authors asked Tester to analyse the visual locators generated by PESTO. Tester found them highly understandable, since it was easy for him to quickly identify the corresponding target elements on the web page (see some examples in Figure 26).

WebDriver commands used in practice. The test suites created by Tester make use only of WebDriver commands that are currently handled by PESTO (it should be noticed that Tester had no previous knowledge of PESTO and its features). In order to better understand how frequent is the usage of the various commands available in WebDriver we analysed (1) an open source test suite³³ created for the *Moodle* web application and (2) a test suite developed three years ago during an industrial work [14].

³² Note that, in case the DOM-based test suite is still valid (i.e., it can be executed on the web application without problems) it is possible maintaining automatically the visual test suite. Indeed, it is sufficient to simply re-run PESTO in order to obtain a new visual test suite aligned with the new release of the web application under test.

³³ <https://github.com/moodlehq/functional-test-suite>

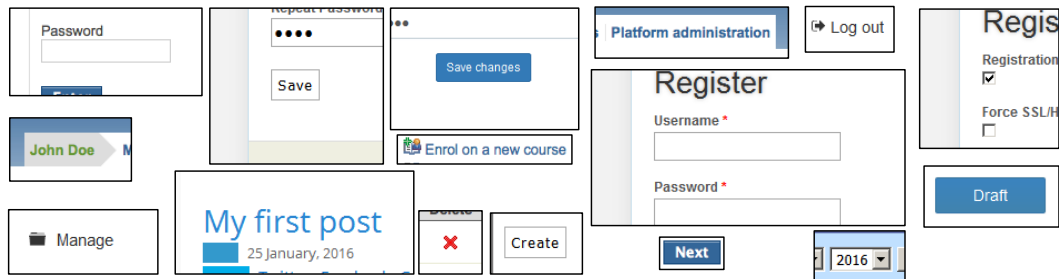


Figure 26. Examples of visual locators automatically generated by PESTO

In the case of the Moodle test suite, we discovered that there are 154 `click()`, 49 `sendKeys()`, 20 `getText()`, 25 selections of an option in a drop-down list (all of the kind `selectByVisibleText()`), 2 `clear()` used to clear text boxes and areas already filled. Unfortunately, the test suite does not adopt the Page Object and Page Factory design patterns, thus we could not use it in our experiment. However, we can measure PESTO's coverage of the WebDriver commands used in this test suite. Overall, PESTO would be able to migrate 248 command calls out of 250 (i.e., around 99%).

In the case of the industrial test suite (the industrial web application is no longer available, thus PESTO could not be executed on it), we repeated the same analysis on the test code, obtaining similar results. We found 75 `click()`, 36 `sendKeys()`, 29 `getText()`, 10 selections of an option in a drop-down list (7 `selectByValue()`, 2 `selectByVisibleText()` and 1 `SelectByIndex()`), 33 `clear()` used to clear text boxes and areas already filled, and 4 `accept()`, used to click the OK button of an alert window. In this case, PESTO would have supported the migration of 176 command calls out of 187 (i.e., around 94%) and only 11 command calls would be not migrated (i.e., 4 `accept()` and 7 `selectByValue()`).

We can conclude that the potential PESTO's coverage of the WebDriver commands used in practice is high, so the tool is expected to be applicable and to provide remarkable benefits in real industrial settings.

PESTO's limitations. From our experimental study we can deduce that the sole command calls that are currently not handled by PESTO are `accept()` and `selectByValue()`. Concerning the former, we were unable to obtain the information (position, size) of the element the test case interacts with in the alert windows from WebDriver. For this reason, PESTO was unable to generate the visual locators for `accept()`. Concerning the latter, `value` is an attribute of the option tag³⁴ representing an item of the list. Such attribute has no visible counterpart on the web page and so it is not possible to migrate the `selectByValue()` command to the visual approach. Indeed, since during the execution of the `selectByValue()` command WebDriver selects directly from the drop-down list the item matching the attribute value, we were not able to obtain the information (multiple images, see complex interactions described in Section 3.3) necessary for creating the visual locators required to interact with the drop-down list. Moreover, since the `value` attribute has not a visible counterpart on the screen or a specific position in the list the corresponding list item cannot be selected with the keyboard as done respectively in the cases of the `selectByVisibleText()` and `SelectByIndex()` commands.

Moreover, as reported in Section 4.1.1, in some rare and peculiar cases a visual locator may not exist at all for the target web element, e.g., when there are multiple matches of the target element image (e.g., the expansion strategy fails because the web element is too close to the web page borders). In our experiments this situation occurred only in one application (Nibbleblog) and only for a single web element (see the answer to RQ1.1). However, there might exist web applications where this problem arises more frequently, resulting in additional manual effort required during test suite migration. Finally, the idea of handling the problem of associating multiple visual locators to a single web element by using the Sikuli MultiStateTarget (see Section 4.1.2) could turn to be imprecise when assertions are checked, because the wrong image could be matched, making the assertion incorrectly pass (i.e., producing a false negative). This problem can be fixed by extending

³⁴ http://www.w3schools.com/tags/tag_option.asp

the signature of the utility method `check(Target element)` to `check(Target element, ImageId id)`, so that the occurrence of a specific screen is checked by the assertion. This problem never occurred in our study. This enhancement of PESTO is part of our future work.

In practice, in our case study such limitations of PESTO resulted in minor manual adjustments to be performed on the output of our tool and this extra overhead is more than compensated by the substantial advantages offered by PESTO when a DOM-based test suite is migrated to the visual approach.

The described implementation of PESTO is focused on a subset of the programming languages, frameworks and patterns that can be used for developing DOM-based web test cases. By implementing and experimenting PESTO we have shown that it is possible to reach a high level of automation in the migration to the visual approach. However, the applicability of the approach to a different set of programming languages or patterns would need to be assessed empirically in a separate, dedicated study, which in turn may require some non negligible engineering effort to adapt PESTO to the new application context. Another applicability limitation of the proposed approach is associated with web applications having strict real-time requirements. The visual test suites generated by PESTO may not be able to perform exactly the same action at the same time of the DOM-based test suites, because of the longer time delays introduced by the visual approach.

6.6. Generalizability of the Results

Concerning the generalisation of results, Tester selected two real open source web applications belonging to different domains and he independently developed two DOM-based test suites for them. He has not received any advice or restriction about which WebDriver commands to use for the development of the test suites. Thus, the context is realistic. Of course, since we only experimented with two objects (two web applications) and one subject (Tester), further studies with other applications and human subjects would be necessary to validate the obtained results. Another threat to the generalizability of our findings is our assumption that the DOM-based test suites adopt the page object design pattern. Additional studies on web applications that do not follow such pattern would be important to extend the validity of our results.

In this study we tried to reduce as much as possible the authors' bias, i.e., 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. For this reason we recruited a professional tester for developing the DOM-based test suites. We did not influence Tester and we provided him just with some high level requirements on the test suites (in particular, the use of page object and factory design patterns). Tester was not aware of the existence of PESTO and of its principles of functioning and algorithms.

7. RELATED WORK

The problem of automatically supporting maintenance of test suites for web applications has been studied from different points of view [31, 32, 33], including: refactoring techniques applied in the industry [34, 35], web re-engineering and testing [36, 37, 38, 39, 40], refactoring of test code [41, 42, 43, 44], GUI testing and refactoring [8, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54]. Nevertheless, none of these works specifically addresses the problem of the automated migration of DOM-based web test suites to the visual approach.

The only work that faced similar challenges with respect to PESTO is the one by Alégroth et al. [4]. More in detail, as part of a two-phase empirical study evaluating and comparing the applicability of automated component-based and Visual GUI testing, Alégroth et al. described the approach they used for migrating existing automated component-based GUI test cases (tool GUITAR [55]) to the Visual approach (tool VGT GUITAR). During the execution of the GUITAR tests, images of the application GUI and of the widgets are saved to get visual locators for the 3rd generation VGT GUITAR test cases based on the usage of Sikuli. There are several differences between our work and the one by Alégroth et al. [4]: (1) the context is different, because PESTO has been conceived for

web application testing while VGT GUITAR for testing desktop systems (indeed it manages Java components such as JButton, JTextField, etc.); as a consequence also (2) the input of the tools is different; PESTO accepts 2nd generation page object based test cases for web applications whereas the tool by Alégroth et al. requires GUITAR automated component-based GUI test cases; finally (3) the output is different, since PESTO has been conceived for generating hybrid 2nd / 3rd generation web test suites when there are interactions (i.e., test case steps) that cannot be migrated to the 3rd generation; on the other hand, VGT GUITAR removes any test case including a test step for which no images can be identified.

Since PESTO is a tool for migrating the test code, while keeping it functionally the same, it can be regarded as a refactoring tool. So, we start our review of the related work with previous papers dealing with refactoring and with its interplay with testing. Then, we consider design patterns that have been proposed specifically for test code. Since the target of PESTO is a visual test suite, we consider the related works on visual web testing, its evaluation and its comparison with the DOM-based approach. Automated program transformations are also a relevant technology for PESTO. Hence, we briefly describe it, in the web context.

7.1. Test Code Refactoring

Martin Fowler was a pioneer of the practice of object-oriented refactoring, defined as “a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior” [56]. Fowler investigated the principles and patterns of good software design, as well as the processes and practices that support it.

Van Deursen et al. [41] give an overview on the bad smells affecting the test code. Such smells are quite specific of test code and differ from those typical of production code, hence they require different refactorings. This work reports a description of 11 test-specific bad smells. Further, 6 useful refactorings are proposed to enhance the quality of test code when such smells are found. In another work [42], the same authors study the interplay between testing and refactoring. Their analysis aims at finding which of the documented refactorings affect the test code, showing the test case implications of each standard refactoring described by Fowler. The majority of Fowler’s refactorings require explicit actions to keep the interface between test and application code compatible. Thus, they propose the notion of test-first refactoring, which modifies the existing test cases so as to obtain an executable starting point for the application code level refactoring.

Chu et al. [57] proposed a plugin tool to guide test case refactoring after having applied well-established pattern-based code refactoring. While refactoring the application, using a series of patterns, their plugin records all the useful steps and information. This allows the plugin to create a mapping relationship between pattern refactoring and test case refactoring. Such mapping is used to transform the test case source code automatically, in accordance with the source code refactoring.

PESTO differs from these works since it aims at refactoring the test code to make it executable by visual web testing tools, without modifying the application code. To the best of our knowledge, PESTO is the first tool supporting developers in such task.

7.2. Design Patterns for Test Code

Gerd Meszaros describes 68 patterns for making test cases easier to write, understand, and maintain [58]. The work of Meszaros aims at making test code robust, repeatable and cost-effective, so as to accelerate the feedback from testing and to improve the code quality.

In web test code, modularisation can be achieved by using the Page Object design pattern. Page objects are facade classes abstracting the internals of web pages into high-level business functions that can be invoked by test cases. By decoupling test code from web page details, web test cases are made more readable and maintainable. The use of the Page Object pattern has proven to be very effective in web testing [15, 14]. Fowler illustrates basic rules for page object creation (e.g., a good page object should provide an easily programmable interface to the program, hiding the underlying widgetry of the GUI [13]). On the other hand, practical aspects, such as which page objects one should create when testing web applications, or what actions one should include in a page object, remain still challenging. *Van Deursen* [18, 19] tried to answer these questions by proposing a

state-based generalization of page objects. Indeed, moving a page object to the state level makes the design of test scenarios easier.

Given the well recognized importance of the page object design pattern for web testing, we designed PESTO under the assumption that the starting DOM-based test suite is written in accordance with such design pattern. As a part of our ongoing research, we also addressed the automatic generation of page objects with the tool APOGEN. A preliminary version of such tool enhanced the navigation graph of the web application to build page objects for each dynamic states [59]. A more refined approach using clustering overcome the limitations of the previous proposal, by providing more succinct though expressive page objects [16]. Experimental results indicate indeed that APOGEN is able to automatically create page objects for non-trivial web applications, with a good level of accuracy [17].

7.3. Visual Testing

The origin of visual testing can be established in the early 90s, but due to hardware limitations and immature image recognition algorithms, it has not become applicable or feasible in practice until recent years. The availability of tools as JAutomate, Sikuli, and EggPlant spread the visual approach widely, because visual testing is easy to use, also by non-technical personnel. The aforementioned tools have different specific features, but they all share the use of image recognition to both interact with and assert the system under test. Research has shown that there are no significant differences between such tools and that the technique is both applicable and viable in the industrial practice [54].

Mahajan et al. [46] introduce a technique for localising presentation failures in web applications. The approach uses computer vision techniques to compare a web page rendered in a browser with its oracle and to identify differences that are likely to expose failures. In another work [45], the same authors describe an approach to discover presentation failures in web applications based on image recognition.

Leotta et al. [5] compare DOM-based and visual web testing on a set of six web applications. DOM-based test suites proved generally more robust and required lower execution time. However, depending on the specific features of the web application under test and its expected evolution, visual test suites might be better, e.g., when the visual appearance is more stable than the DOM structure. This is the case of many modern web applications in which JavaScript continuously modifies the DOM, thus making E2E web testing extremely challenging [60].

More generally, in recent years the research works about visual GUI testing have increased, especially in the context of industrial adoption [52, 61]. *Borjesson and Feldt* [52] evaluated 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. 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. Similar results can be found in the works by *Alégroth et al.* [53, 54].

Chang et al. [8] present an experiment to analyse the long-term reusability of 3rd generation (Sikuli) test cases. They selected two open-source applications (Capybara 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 that as long as a GUI evolves incrementally, a significant number of Sikuli test cases remains reusable.

At the time of writing, it is unclear if the industrial practice will go toward visual or DOM-based web testing. The choice is probably dependent on several, application and process specific, factors. Whatever is the industrial trend, PESTO provides developers with a unique opportunity to experiment the visual testing approach at minimal migration cost.

7.4. Program Transformation

In the web context, *Bruno et al.* [37] proposed the use of test cases as a form of contract between the provider of a web service and its users. They developed a tool to automatically transform JUnit test cases into XML, using a combination of dynamic and static analysis. The obtained test suite is

executed against the web service, to discover potential misbehaviours. Moreover, test cases can be published to serve as a facet of the service description, as well as for regression testing of a service during system evolution.

In the work by Ricca *et al.* [38] transformation rules are applied to a web application with the aim of facing its degradation throughout its evolution. In another work, Ricca *et al.* [39] realized the automatic re-engineering of web applications by means of the DMS Reengineering toolkit and evaluated its applicability to a real world case study.

Concerning the transformation of test cases, Deiß [34] describes TTtwo2three, an automatic tool, developed at Nokia, for the conversion of TTCN-2 test systems to TTCN-3. The tool realizes both semantic and syntactic transformations, but some manual refactoring is needed to ensure that the test cases behave as expected. TTtwo2three has been used to convert two industrial test suites, a Bluetooth Serial Port Profile and a UMTS network element, consisting of about 2,500 test cases.

PESTO shares with these works the program transformation technology, operating on the abstract syntax tree of the test code. However, the goal of program transformation is completely different for PESTO, which aims at automatically producing a visual test suite from a DOM-based one.

8. CONCLUSIONS AND FUTURE WORK

In this work, we proposed and experimented with a novel approach and its implementation, a tool called PESTO, able to *transform DOM-based web test suites developed using Selenium WebDriver into visual test suites relying on the usage of Sikuli API, so as to allow developers to experiment with 3rd generation testing tools at a lower cost compared to a manual migration*. To the best of our knowledge no other solution, both in the academia and in the industry, yet exists to carry out such migration task with the exception of the work by Alégroth *et al.* [4] which is however conceived for the desktop applications context and allows to migrate existing automated component-based GUI test cases (tool GUITAR [55]) to the Visual approach (tool VGT GUITAR). The techniques and the architectural solutions adopted by PESTO and described in this work are quite general and can be adapted to other DOM to visual web test migrations even if the effort to create a variant of PESTO for managing such cases could be not negligible³⁵.

PESTO handles a relevant set of the WebDriver DOM-based commands: `click()`, `sendKeys()`, `getText()`, `selectByVisibleText()`, `selectByIndex()`, and `clear()`. When present, any unhandled DOM-based command (e.g., `selectByValue()`, `accept()`) is simply copied in the generated test suite, resulting in a hybrid (DOM-based / visual) test suite.

We used four test suites as a training set for the improvement of PESTO. Then, the effectiveness of PESTO has been evaluated on two DOM-based validation test suites, developed by an independent, professional Tester for two different open source web applications. PESTO generated the corresponding visual test suites for the DOM-based validation test suites requiring only a small amount of manual work. The visual test suites automatically generated by PESTO were then executed and the test cases exhibited the correct, expected behaviour. In our study, PESTO was able to migrate 100% of the command calls used in the existing DOM-based validation test suites. Moreover, by analysing some other existing DOM-based test suites, we found that PESTO can handle more than 95% of the employed command calls. The visual locators automatically generated by PESTO were checked for readability by the professional tester involved in our experiments and they were judged to be easy to understand.

In our future work, we intend to evaluate PESTO on some industrial case studies in order to understand whether the results obtained on the test suites developed by the recruited Tester hold also with code developed by other professionals. Despite the possible need for customizations (e.g., to adapt the delays used for page loading) or preprocessing steps (e.g., to make the DOM-based

³⁵ Depending on the selected DOM-based and visual tool combination, in case the supporting features required to carry out the transformations were not available (e.g., obtaining the images representing the web elements the DOM-based test case interacts with), it might be even impossible to create a variant of PESTO for such combination of tools.

test suite compliant with the Page Object and Factory patterns), we expect general applicability and major benefits from PESTO and the approach it implements. Moreover, we plan to further investigate different visual locator generation strategies, and to study how they impact the locator comprehensibility (as perceived by human testers) and robustness (i.e., ability to locate the web element of interest even after minor modifications to the web application under test). Currently, PESTO does not generate a visual test suite portable across different platform-browser combinations, since it is based on a single execution of a DOM-based test suite, on a single platform-browser combination. A possible extension of PESTO could take into account the possibility of merging (as a post processing step) two or more visual test suites generated by PESTO in order to obtain a single, platform-browser portable visual test suite. Finally, we plan to investigate whether starting from PESTO it would be possible to derive a more general and tool independent transformational framework. In such framework, different tools/languages combinations could be plugged at both sides of the pipe, to support technologies different from those considered in this work.

REFERENCES

1. Tonella P, Ricca F, Marchetto A. Recent advances in web testing. *Advances in Computers* 2014; **93**:1–51, doi:10.1016/B978-0-12-800162-2.00001-4. URL <http://doi.org/10.1016/B978-0-12-800162-2.00001-4>.
2. Hayes L. *The Automated Testing Handbook*. Software Testing Institute, 2004.
3. Leotta M, Clerissi D, Ricca F, Tonella P. Approaches and tools for automated end-to-end web testing. *Advances in Computers* 2016; **101**:193–237, doi:10.1016/bs.adcom.2015.11.007. URL <http://doi.org/10.1016/bs.adcom.2015.11.007>.
4. Alégroth E, Gao Z, Oliveira R, Memon A. Conceptualization and evaluation of component-based testing unified with visual GUI testing: An empirical study. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015; 1–10, doi:10.1109/ICST.2015.7102584. URL <http://doi.org/10.1109/ICST.2015.7102584>.
5. Leotta M, Clerissi D, Ricca F, Tonella P. Visual vs. DOM-based web locators: An empirical study. *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)*, LNCS, vol. 8541. Springer, 2014; 322–340, doi:10.1007/978-3-319-08245-5_19. URL http://doi.org/10.1007/978-3-319-08245-5_19.
6. Chapman P, Evans D. Automated black-box detection of side-channel vulnerabilities in web applications. *Proceedings of 18th Conference on Computer and Communications Security (CCS 2011)*, ACM, 2011; 263–274, doi:10.1145/2046707.2046737. URL <http://doi.org/10.1145/2046707.2046737>.
7. Alégroth E, Nass M, Olsson HH. JAutomate: A tool for system- and acceptance-test automation. *Proceedings of 6th International Conference on Software Testing, Verification and Validation (ICST 2013)*, IEEE, 2013; 439–446, doi:10.1109/ICST.2013.61. URL <http://doi.org/10.1109/ICST.2013.61>.
8. Chang TH, Yeh T, Miller RC. Gui testing using computer vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2010)*, ACM, 2010; 1535–1544, doi:10.1145/1753326.1753555. URL <http://doi.org/10.1145/1753326.1753555>.
9. Ricca F, Leotta M, Stocco A. Three open problems in the context of E2E web testing and a vision: NEONATE. *Advances in Computers* ; **113**:(in press).
10. Leotta M, Stocco A, Ricca F, Tonella P. Automated generation of visual web tests from DOM-based web tests. *Proceedings of 30th ACM/SIGAPP Symposium on Applied Computing (SAC 2015)*, ACM, 2015; 775–782, doi:10.1145/2695664.2695847. URL <http://doi.org/10.1145/2695664.2695847>.
11. Stocco A, Leotta M, Ricca F, Tonella P. PESTO: A tool for migrating DOM-based to visual web tests. *Proceedings of 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, IEEE, 2014; 65–70, doi:10.1109/SCAM.2014.36. URL <http://doi.org/10.1109/SCAM.2014.36>.
12. Almenar F, Esparcia-Alcázar AI, Martínez M, Rueda U. Automated testing of web applications with testar. *Proceedings of 8th International Symposium on Search Based Software Engineering (SSBSE 2016)*, LNCS, vol. 9962, Sarro F, Deb K (eds.). Springer, 2016; 218–223, doi:10.1007/978-3-319-47106-8_15. URL http://doi.org/10.1007/978-3-319-47106-8_15.
13. Leotta M, Stocco A, Ricca F, Tonella P. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process* 2016; **28**(3):177–204, doi:10.1002/smr.1771. URL <http://doi.org/10.1002/smr.1771>.
14. Leotta M, Stocco A, Ricca F, Tonella P. Using multi-locators to increase the robustness of web test cases. *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, IEEE, 2015; 1–10, doi:10.1109/ICST.2015.7102611. URL <https://doi.org/10.1109/ICST.2015.7102611>.
15. PageObject. <http://martinfowler.com/bliki/PageObject.html>. Accessed: 2016-01-20.
16. Leotta M, Clerissi D, Ricca F, Spadaro C. Improving test suites maintainability with the page object pattern: An industrial case study. *Proceedings of 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013)*, IEEE, 2013; 108–113, doi:10.1109/ICSTW.2013.19. URL <http://doi.org/10.1109/ICSTW.2013.19>.
17. Leotta M, Clerissi D, Ricca F, Tonella P. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, IEEE, 2013; 272–281, doi:10.1109/WCRE.2013.6671302. URL <http://doi.org/10.1109/WCRE.2013.6671302>.

18. Stocco A, Leotta M, Ricca F, Tonella P. APOGEN: Automatic page object generator for web testing. *Software Quality Journal* 2017; **25**(3):1007–1039, doi:10.1007/s11219-016-9331-9. URL <http://doi.org/10.1007/s11219-016-9331-9>.
19. Stocco A, Leotta M, Ricca F, Tonella P. Clustering-aided page object generation for web testing. *Proceedings of 16th International Conference on Web Engineering (ICWE 2016), LNCS*, vol. 9671, Bozzon A, Cudré-Mauroux P, Pautasso C (eds.). Springer, 2016; 132–151, doi:10.1007/978-3-319-38791-8_8. URL http://doi.org/10.1007/978-3-319-38791-8_8.
20. Deursen AV. Testing web applications with state objects. *Communications of the ACM* Jul 2015; **58**(8):36–43, doi:10.1145/2755501. URL <http://doi.org/10.1145/2755501>.
21. Deursen AV. Beyond page objects: Testing web applications with state objects. *ACM Queue* Jun 2015; **13**(6):20:20–20:37, doi:10.1145/2791301.2793039. URL <http://doi.org/10.1145/2791301.2793039>.
22. Collin M. *Mastering Selenium WebDriver*. Packt Publishing, 2015.
23. Avasarala S. *Selenium WebDriver Practical Guide*. Packt Publishing, 2014.
24. Zhan Z. *Selenium WebDriver Recipes in Java: The Problem Solving Guide to Selenium WebDriver in Java*. Web Test Automation Recipes Series, CreateSpace Independent Publishing Platform, 2015.
25. Prasad R. *Learning Selenium Testing Tools - Third Edition*. Community experience distilled, Packt Publishing, 2015.
26. Salunke S. *Selenium Webdriver in Ruby: Learn with Examples*. CreateSpace Independent Publishing Platform, 2014.
27. Harty J. Finding usability bugs with automated tests. *Communications of the ACM* Feb 2011; **54**(2):44–49, doi:10.1145/1897816.1897836. URL <http://doi.acm.org/10.1145/1897816.1897836>.
28. Bruns A, Kornstadt A, Wichmann D. Web application tests with selenium. *IEEE Software* Sept 2009; **26**(5):88–91, doi:10.1109/MS.2009.144.
29. Leotta M, Clerissi D, Ricca F, Spadaro C. Comparing the maintainability of Selenium WebDriver test suites employing different locators: A case study. *Proceedings of 1st International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation (JAMAICA 2013)*, ACM, 2013; 53–58, doi:10.1145/2489280.2489284. URL <http://doi.org/10.1145/2489280.2489284>.
30. Alégroth E. *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. Chalmers University of Technology and Goteborg University, 2015. PhD Thesis. Technical Report No 117D.
31. Collin M. *Mastering Selenium WebDriver*. Packt Publishing, 2015.
32. Avasarala S. *Selenium Webdriver Practical Guide*. Packt Publishing, 2014.
33. Choudhary SR, Zhao D, Versee H, Orso A. Water: Web application test repair. *Proceedings of International Workshop on End-to-End Test Script Engineering (ETSE 2011)*, ACM, 2011; 24–29, doi:10.1145/2002931.2002935. URL <http://doi.org/10.1145/2002931.2002935>.
34. Thummalapenta S, Devaki P, Sinha S, Chandra S, Gnanasundaram S, Nagaraj DD, Sathishkumar S. Efficient and change-resilient test automation: An industrial case study. *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, IEEE, 2013; 1002–1011, doi:10.1109/ICSE.2013.6606650. URL <http://doi.org/10.1109/ICSE.2013.6606650>.
35. Yandrapally R, Thummalapenta S, Sinha S, Chandra S. Robust test automation using contextual clues. *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2014)*, ACM, 2014; 304–314, doi:10.1145/2610384.2610390. URL <http://doi.org/10.1145/2610384.2610390>.
36. Deiß T. Refactoring and converting a TTCN-2 test suite. *International Journal on Software Tools for Technology Transfer* 2008; **10**(4):347–352, doi:10.1007/s10009-008-0079-9. URL <http://doi.org/10.1007/s10009-008-0079-9>.
37. Shin K, Kim S, Park S, Lim D. Automated test case generation for automotive embedded software testing using XMI-based UML model transformations. *Technical Report*, Hanyang University, Daedong Co., Ltd. 2014, doi:10.4271/2014-01-0315. URL <http://doi.org/10.4271/2014-01-0315>.
38. Rossi G, Urbietta M, Ginzburg J, Distanto D, Garrido A. Refactoring to rich internet applications. a model-driven approach. *Proceedings of the Eighth International Conference on Web Engineering (ICWE 2008)*, 2008; 1–12, doi:10.1109/ICWE.2008.41. URL <http://doi.org/10.1109/ICWE.2008.41>.
39. Bruno M, Canfora G, Penta M, Esposito G, Mazza V. Proceedings of the third international conference on service-oriented computing (ICSOC 2005). *Using Test Cases as Contract to Ensure Service Compliance Across Releases*, Springer Berlin Heidelberg, 2005; 87–100, doi:10.1007/11596141_8. URL http://doi.org/10.1007/11596141_8.
40. Ricca F, Tonella P, Baxter ID. Restructuring web applications via transformation rules. *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, IEEE, 2001; 152–162, doi:10.1109/SCAM.2001.972676. URL <http://doi.org/10.1109/SCAM.2001.972676>.
41. Ricca F, Tonella P, Baxter ID. Web application transformations based on rewrite rules. *Information & Software Technology* 2002; **44**(13):811–825, doi:10.1016/S0950-5849(02)00125-8. URL [http://doi.org/10.1016/S0950-5849\(02\)00125-8](http://doi.org/10.1016/S0950-5849(02)00125-8).
42. Ding X, Huang H, Ruan Y, Shaikh A, Peterson B, Zhang X. Splitter: A proxy-based approach for post-migration testing of web applications. *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, ACM, 2010; 97–110, doi:10.1145/1755913.1755925. URL <http://doi.org/10.1145/1755913.1755925>.
43. Deursen AV, Moonen L, Bergh A, Kok G. Refactoring test code. *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP 2001)*, 2001; 92–95, doi:10.1.1.19.5499. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5499>.
44. Deursen AV, Moonen L. The video store revisited - thoughts on refactoring and testing 2002. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.4545>.
45. Guerra E, Fernandes C. Refactoring test code safely. *Proceedings of International Conference on Software Engineering Advances (ICSEA 2007)*, 2007, doi:10.1109/ICSEA.2007.57. URL <http://doi.org/10.1109/ICSEA.2007.57>.
46. Soares G, Gheyri R, Serey D, Massoni T. Making program refactoring safer. *Software, IEEE* July 2010; **27**(4):52–57, doi:10.1109/MS.2010.63. URL <http://doi.org/10.1109/MS.2010.63>.

47. Mahajan S, Halfond WG. Finding HTML presentation failures using image comparison techniques. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, ACM, 2014; 91–96, doi:10.1145/2642937.2642966. URL <http://doi.org/10.1145/2642937.2642966>.
48. Mahajan S, Halfond WGJ. Detection and localization of HTML presentation failures using computer vision-based techniques. *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, 2015, doi:10.1109/ICST.2015.7102586. URL <http://doi.org/10.1109/ICST.2015.7102586>.
49. Nguyen BN, Robbins B, Banerjee I, Memon A. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering* 2013; **21**(1):65–105, doi:10.1007/s10515-013-0128-9. URL <http://doi.org/10.1007/s10515-013-0128-9>.
50. Memon AM, Cohen MB. Automated testing of GUI applications: Models, tools, and controlling flakiness. *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, IEEE, 2013; 1479–1480, doi:10.1109/ICSE.2013.6606750. URL <http://doi.org/10.1109/ICSE.2013.6606750>.
51. Daniel B, Luo Q, Mirzaaghaei M, Dig D, Marinov D, Pezzè M. Automated gui refactoring and test script repair. *Proceedings of the First International Workshop on End-to-End Test Script Engineering (ETSE 2011)*, ACM, 2011; 38–41, doi:10.1145/2002931.2002937. URL <http://doi.org/10.1145/2002931.2002937>.
52. Memon AM. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability* 2007; **17**(3):137–157, doi:10.1002/stvr.364. URL <http://doi.org/10.1002/stvr.364>.
53. Ying M, Miller J. Refactoring legacy AJAX applications to improve the efficiency of the data exchange component. *Journal of Systems and Software* 2013; **86**(1):72–88, doi:10.1016/j.jss.2012.07.019. URL <http://doi.org/10.1016/j.jss.2012.07.019>.
54. Borjesson E, Feldt R. Automated system testing using visual GUI testing tools: A comparative study in industry. *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, 2012; 350–359, doi:10.1109/ICST.2012.115. URL <http://doi.org/10.1109/ICST.2012.115>.
55. Alégroth E, Feldt R. *Continuous Software Engineering*, chap. Industrial Application of Visual GUI Testing: Lessons Learned. Springer, 2014; 127–140, doi:10.1007/978-3-319-11283-1_11. URL http://doi.org/10.1007/978-3-319-11283-1_11.
56. Alégroth E, Feldt R, Ryrholm L. Visual GUI testing in practice: Challenges, problems and limitations. *Empirical Software Engineering* Jun 2015; **20**(3):694–744, doi:10.1007/s10664-013-9293-5. URL <http://doi.org/10.1007/s10664-013-9293-5>.
57. Nguyen BN, Robbins B, Banerjee I, Memon A. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering* 2014; **21**(1):65–105, doi:10.1007/s10515-013-0128-9. URL <http://doi.org/10.1007/s10515-013-0128-9>.
58. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.
59. Chu PH, Hsueh NL, Chen HH, Liu CH. A test case refactoring approach for pattern-based software development. *Software Quality Journal* 2012; **20**(1):43–75, doi:10.1007/s11219-011-9143-x. URL <http://doi.org/10.1007/s11219-011-9143-x>.
60. Meszaros G. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR: Upper Saddle River, NJ, USA, 2006.
61. Stocco A, Leotta M, Ricca F, Tonella P. Why creating web page objects manually if it can be done automatically? *Proceedings of 10th IEEE/ACM International Workshop on Automation of Software Test (AST 2015)*, IEEE, 2015; 70–74, doi:10.1109/AST.2015.26. URL <http://doi.org/10.1109/AST.2015.26>.
62. Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. An empirical study of client-side javascript bugs. *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2013)*, IEEE Computer Society, 2013; 55–64, doi:10.1109/ESEM.2013.18. URL <http://doi.org/10.1109/ESEM.2013.18>.
63. Collins E, Dias-Neto A, de Lucena V. Strategies for agile software testing automation: An industrial experience. *Proceedings of 36th IEEE Annual Computer Software and Applications Conference Workshops (COMPSACW 2012)*, IEEE, 2012; 440–445, doi:10.1109/COMPSACW.2012.84. URL <http://doi.org/10.1109/COMPSACW.2012.84>.