PESTO: A Tool for Migrating DOM-based to Visual Web Tests

Andrea Stocco, Maurizio Leotta, Filippo Ricca, Paolo Tonella

Abstract:

Automated testing of web applications reduces the effort needed in manual testing. Old 1st generation tools, based on screen coordinates, produce quite fragile test suites, tightly coupled with the specific screen resolution, window position and size experienced during test case recording. These tools have been replaced by a 2nd generation of tools, which offer easy selection and interaction with the web elements, based on DOM-oriented commands. Recently, a new 3rd generation of tools came up based on visual image recognition, bringing the promise of wider applicability and simplicity. A tester might ask if the migration towards such new technology is worthwhile, since the manual effort to rewrite a test suite might be overwhelming. In this paper, we propose PESTO, a tool facing the problem of the automated migration of 2nd generation test suites to the 3rd generation. PESTO determines automatically the screen position of each web element located on the DOM by a 2nd generation test case. It then calculates a screenshot image centred around the web element so as to ensure unique visual matching. Then, the entire source code of the DOM-based test suite is transformed into a visual test suite, based on such automatically extracted images and using specific visual commands.

Digital Object Identifier (DOI):

`http://dx.doi.org/10.1109/SCAM.2014.36`

PESTO: A Tool for Migrating DOM-based to Visual Web Tests

Andrea Stocco[1], Maurizio Leotta[1], Filippo Ricca[1], Paolo Tonella[2]
[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
[2] Fondazione Bruno Kessler, Trento, Italy
andrea.stocco@dibris.unige.it, maurizio.leotta@unige.it, filippo.ricca@unige.it, tonella@fbk.eu

***Abstract*—Automated testing of web applications reduces the effort needed in manual testing. Old 1st generation tools, based on screen coordinates, produce quite fragile test suites, tightly coupled with the specific screen resolution, window position and size experienced during test case recording. These tools have been replaced by a 2nd generation of tools, which offer easy selection and interaction with the web elements, based on DOM-oriented commands. Recently, a new 3rd generation of tools came up based on visual image recognition, bringing the promise of wider applicability and simplicity. A tester might ask if the migration towards such new technology is worthwhile, since the manual effort to rewrite a test suite might be overwhelming. In this paper, we propose PESTO, a tool facing the problem of the automated migration of 2nd generation test suites to the 3rd generation. PESTO determines automatically the screen position of each web element located on the DOM by a 2nd generation test case. It then calculates a screenshot image centred around the web element so as to ensure unique visual matching. Then, the entire source code of the DOM-based test suite is transformed into a visual test suite, based on such automatically extracted images and using specific visual commands.**

***Keywords*-Web Testing; DOM-based Testing; Visual Testing; GUI Testing; Test Case Maintenance; Test Suite Migration; Test Automation; Selenium WebDriver; Sikuli.**

I. Introduction and Motivation

Web applications have become a core part of everyday lives. People use online services as source of information, means of communication, source of entertainment and venue for commerce. Web applications often ask users to insert and store personal information or sensitive data. *Software testing* plays a crucial role in quality software production, but the specific features of Web-based software makes the evaluation of its correctness quite challenging for all IT experts [8].

Modern web applications are increasingly complex, characterised by rapid evolution, asynchronous interactions and tight integration of different technologies. As business and customers' requirements grow, so does the pressure on software professionals to deliver new stable releases, in reduced time and with high quality. This is why web development teams are increasingly adopting agile techniques and test automation practices.

Automated functional web testing 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 friendly APIs provide commands to control the browser and interact with the web page elements, e.g., click a button or fill a field with text. Test scripts are then completed with assertions, e.g., using xUnit assertions.

There are a number of commercial and open source tools available for assisting the development of automated test suites. Selenium toolkit[1] is one of the most used open source solutions. Its tool Selenium WebDriver offers an object oriented API for accessing the Document Object Model (DOM) of a web page. Hence, it belongs to the so called 2nd generation tools. Selenium WebDriver supports test script development in a high level language, runs such scripts automatically and provides quick, informative feedback to web developers.

While 2nd generation web testing tools are currently widely adopted in the industry [3], the new generation of visual tools [1], [2] offer an interesting alternative. Indeed, there are situations in which 2nd generation tools cannot be used effectively [7]. For example, they do not support easily some UI technologies in which UI elements are not identifiable or accessible from the DOM (e.g., third party ActiveX and Flash components). There are cases in which an entire UI region of the web application consists of multiple functional sub-regions represented by a single UI element (e.g., a UI element representing a city map). The functional sub-parts (e.g., points of interest on the map) cannot be identified by a DOM-based tool.

In such situations, image-based testing tools come to the rescue. Unlike 2nd generation tools, image-based tools are completely independent from the underlying DOM structure of the web pages. Sikuli API[2] is an open source representative of this category and is able to automate and test any graphical user interface using screenshot images. Sikuli API provides image-based GUI automation functionalities to Java programmers.

In this context, a development team might consider whether the migration towards the 3rd generation technology is worthwhile. Actually in the market no single solution supports all the available platforms and GUI objects, therefore, in some contexts, selection of the proper test framework might be difficult. The emergence of new complex visual components in web pages pulls for the adoption of 3rd generation tools, but migrating already existing DOM-based test suites turns out to be an extremely time-consuming, error-prone and boring task.

[1]http://seleniumhq.org/
[2]http://code.google.com/p/sikuli-api/

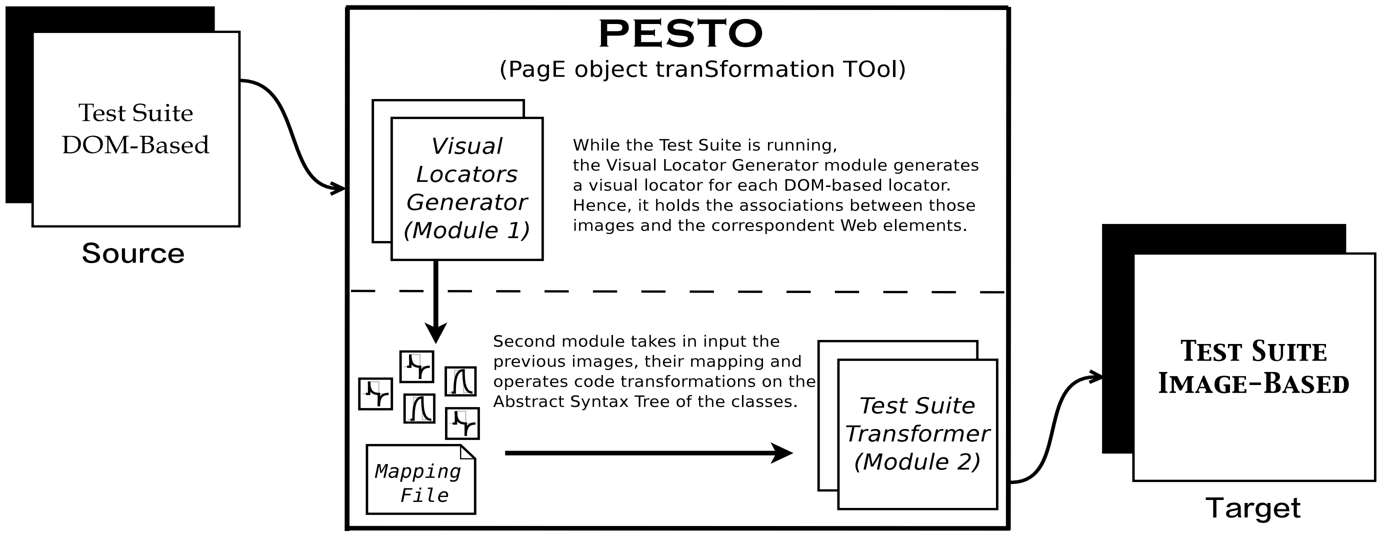

Test Suite DOM-Based
Source
PESTO
(PagE object tranSformation TOol)
Visual Locators Generator (Module 1)
While the Test Suite is running, the Visual Locator Generator module generates a visual locator for each DOM-based locator. Hence, it holds the associations between those images and the correspondent Web elements.
Second module takes in input the previous images, their mapping and operates code transformations on the Abstract Syntax Tree of the classes.
Mapping File
Test Suite Transformer (Module 2)
TEST SUITE IMAGE-BASED
Target

Fig. 1. High Level Architecture of PESTO (PagE object tranSformation TOol)

In this tool demo, we describe PESTO (PagE object tranSformation TOol), a prototype tool able to assist web testers in the difficult task of transforming an existing DOM-based test suite, developed using Selenium WebDriver, to an image-based test suite, built on top of Sikuli API.

This tool demo paper is organised as follows: Section II provides some background on 2nd and 3rd generation web testing tools. Section III describes our tool using a simple test suite as use case, followed by conclusions and future work.

II. Background

Nowadays, tools interact with the web pages elements mostly using one of the following two techniques [7]:

1) *DOM-based localisation:* web page elements are retrieved using the information in the Document Object Model, by means of different mechanisms (e.g., by ID, XPath and LinkText).
2) *Visual localisation*: web page elements are identified using screenshots, representing their visual appearance as rendered by the browser, and image recognition algorithms.

A. The Page Object and the Page Factory Patterns

Page Object is a quite popular web test design pattern, 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 that encapsulates the features of the web page in methods. 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. The use of *Page Object* reduces the coupling between web pages and test cases, promoting reusability, readability and maintainability of the test cases [5], [6].

B. DOM-based and Visual Web Testing

In this paper, we consider Selenium WebDriver[3] as a representative tool for the implementation of DOM-based web test suites (for short, WebDriver). WebDriver is completely open source and provides comprehensive APIs to control the browser. Test cases are written manually in a high level programming language (we used Java), using WebDriver commands and JUnit or TestNG assertions.

Sikuli is an academic and research project at the MIT[4]. It is an open source image based technology to automate testing of graphical user interfaces. Sikuli supports major OS platforms (Windows, Mac and Linux). It ships with an integrated development environment (IDE) for writing visual scripts and an API that can be used from any Java program. The core of Sikuli script consists of a Java robot which delivers keyboard and mouse events to appropriate locations on the screen. Image recognition is provided by the OpenCV (Open Source Computer Vision) C++ engine[5] that is connected to Java via JNI (Java Native Interface).

We believe that Sikuli API is a good target for the migration of WebDriver test suites, since it permits the creation of test cases structurally similar to those written for WebDriver. Moreover, it is open source and we can use the same programming environment with both the chosen testing frameworks: same programming language (Java), IDE (Eclipse), and testing framework (JUnit).

III. Tool Architecture and Use Case

This section describes the design and architecture of PESTO. The tool automatically transforms a DOM-based web test suite, created using WebDriver and adopting the Page Object design

[3]http://seleniumhq.org/projects/webdriver/
[4]http://www.sikuli.org/
[5]http://opencv.org/

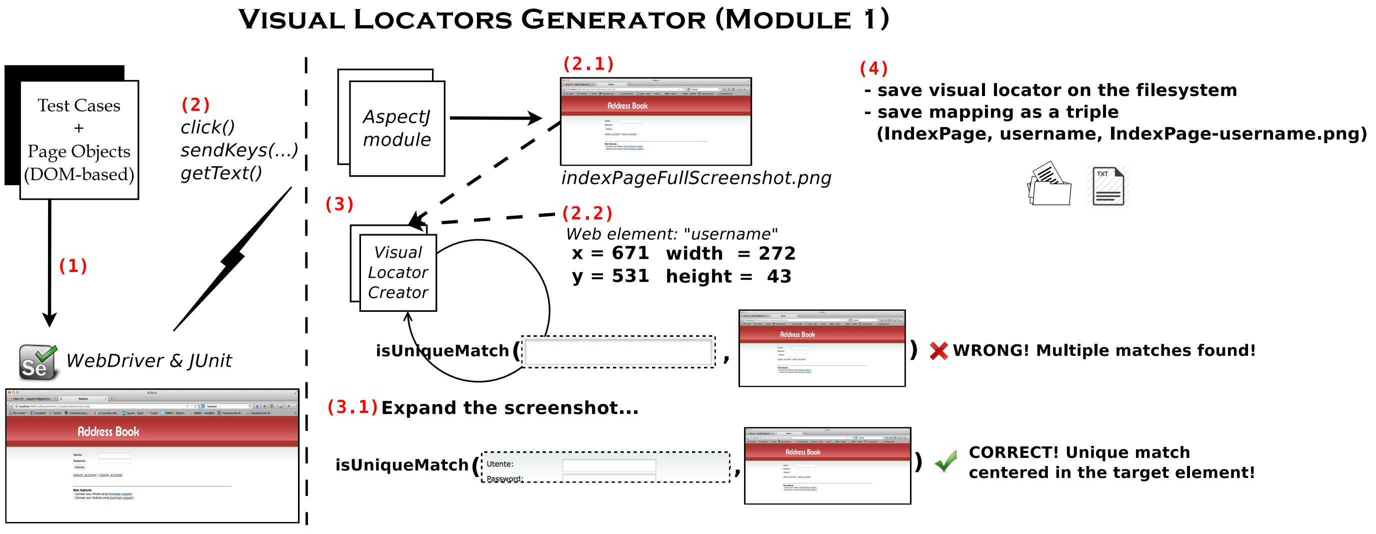

VISUAL LOCATORS GENERATOR (MODULE 1)
Test Cases + Page Objects (DOM-based)
(1)
WebDriver & JUnit
(2) click() sendKeys(...) getText()
AspectJ module
(2.1)
indexPageFullScreenshot.png
(2.2) Web element: "username"
x = 671 width = 272
y = 531 height = 43
(3)
Visual Locator Creator
isUniqueMatch(,)
WRONG! Multiple matches found!
(3.1) Expand the screenshot...
isUniqueMatch(,)
CORRECT! Unique match centered in the target element!
(4)
- save visual locator on the filesystem
- save mapping as a triple
(IndexPage, username, IndexPage-username.png)

Fig. 2. Visual Locators Generator (Module 1)

pattern, into a visual web test suite based on the Sikuli image recognition capabilities.

For the interested reader, a demo video of PESTO and the source code can be downloaded from http://sepl.dibris.unige.it/2014-PESTO.php.

Fig. 1 shows the high level structure of PESTO, which is logically divided into two main modules: the *Visual Locators Generator* (Module 1) and the *Test Suite Transformer* (Module 2). The input of PESTO consists of a DOM-based web test suite, and it produces in output an image-based web test suite.

In the following we describe the modules composing PESTO, considering as a use case a simple test suite of a real web application: *PHP Address Book*[6]. It is a simple Web-based address and phone book, contacts manager, and organiser.

A. *Visual Locators Generator (Module 1)*

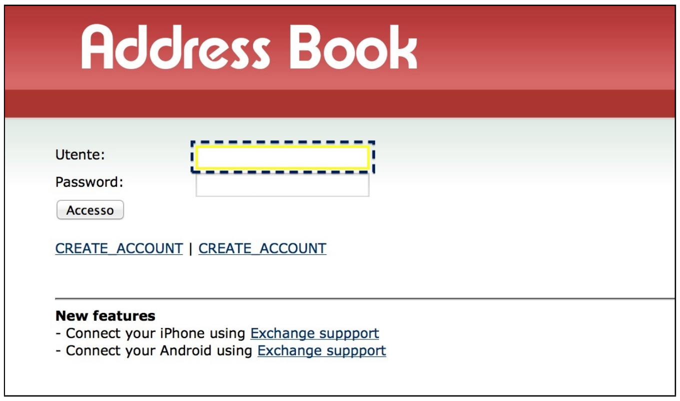
In the first phase, we need to build a catalogue of all the web elements the test cases interact with. In the DOM-based approach, web elements are identified by *locators*, lines of source code containing the specification of how to select a particular target element in the DOM. With the visual approach, a web element is instead identified by an image representing the portion of the web page displaying it. The visual appearance of the rendered elements may change during the application execution and some elements may be not visible until a specific event occurs. For this reason, we need to capture the screenshots of the web elements at runtime, while the test suite is executing. This has been achieved by means of the aspect-oriented programming (AOP) paradigm [4], specifically the AspectJ language[7].

Fig. 2 illustrates the process of the runtime creation of visual locators. Here, we detail each step, following the same enumeration as in the figure.

[6]http://sourceforge.net/projects/php-addressbook/
[7]https://eclipse.org/aspectj/

(1) In Eclipse IDE, JUnit runs the WebDriver test suite, developed following the Page Object design pattern. For example, a test case may open the browser page, type the web application URL and perform some operations.

(2) At the same time, an AspectJ procedure is active and able to intercept all WebDriver calls to the methods `click()`, `sendKeys(...)` and `getText()`, using properly defined join points. Fig. 3 depicts this situation: a test case executing the login function is running and the AspectJ procedure intercepts the WebDriver call to the `sendKeys(...)` method while the username value is being inserted. Correspondingly, a yellow highlight (in b/w printed paper, dashed rectangle) is displayed to the user (see Fig. 3).

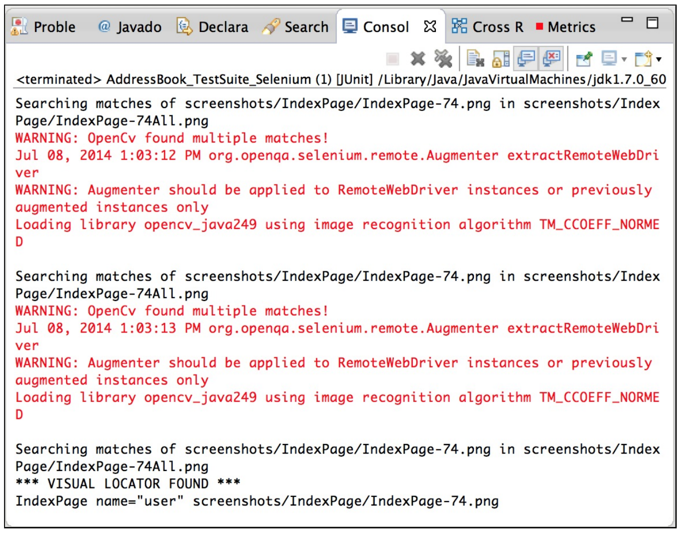


New features
- Connect your iPhone using Exchange suppport
- Connect your Android using Exchange suppport

Fig. 3. AspectJ procedure in action

When a given join point matches, an advice method with an @Before action is run, generating a visual locator by means of the following steps: **(2.1)** the AspectJ procedure calls a WebDriver method that returns a screenshot of the entire web page (e.g., the entire login page shown in Fig. 3) containing the web element of interest (e.g., the username textBox); **(2.2)** the AspectJ procedure calls a WebDriver method gathering important information about the web element of interest: (i) the coordinates of the top left-hand corner of the rendered web element and (ii) its sizes (i.e., width and height).

(3) The Visual Locator Creator procedure calculates a precise rectangle image based on those coordinates and centred on the web element. The auxiliary function `isUniqueMatch` checks whether it is univocal in the screenshot of the web page. Often, a precise crop image cannot be considered a locator, since it is not uniquely selecting the web element of interest.



```
<terminated> AddressBook_TestSuite_Selenium (1) [JUnit] /Library/Java/JavaVirtualMachines/jdk1.7.0_60
Searching matches of screenshots/IndexPage/IndexPage-74.png in screenshots/Index
Page/IndexPage-74All.png
WARNING: OpenCv found multiple matches!
Jul 08, 2014 1:03:12 PM org.openqa.selenium.remote.Augmenter extractRemoteWebDri
ver
WARNING: Augmenter should be applied to RemoteWebDriver instances or previously
augmented instances only
Loading library opencv_java249 using image recognition algorithm TM_CCOEFF_NORME
D

Searching matches of screenshots/IndexPage/IndexPage-74.png in screenshots/Index
Page/IndexPage-74All.png
WARNING: OpenCv found multiple matches!
Jul 08, 2014 1:03:13 PM org.openqa.selenium.remote.Augmenter extractRemoteWebDri
ver
WARNING: Augmenter should be applied to RemoteWebDriver instances or previously
augmented instances only
Loading library opencv_java249 using image recognition algorithm TM_CCOEFF_NORME
D

Searching matches of screenshots/IndexPage/IndexPage-74.png in screenshots/Index
Page/IndexPage-74All.png
*** VISUAL LOCATOR FOUND ***
IndexPage name="user" screenshots/IndexPage/IndexPage-74.png
```

Fig. 4. Log messages from the visual locator creator

Fig. 4 reports some log messages about the visual locator creation and it shows the situation described above. We can notice that the Visual Locator Creator searched for a unique match of the web element screenshots/IndexPage/IndexPage-74.png (the username textBox) in the full screenshot of the Index page, but multiple matches are found (we remember that in the first attempt the crop is precise). Indeed, in Fig. 3 we can see that the web page contains a form with two identical textBoxes, both of the same sizes and appearance, hence multiple matches are found. In this case, Sikuli will not be able to identify such element; **(3.1)** therefore, the Visual Locator Creator expands automatically the size of the rectangle image, scaling it up as many times as needed to produce a unique visual locator. In fact, if we look at the next log messages in Fig. 4, by expanding the rectangle twice, a unique match is found and the process can proceed with the next web element (IndexPage-75.png). The visual locator automatically created for the username textBox is shown in Fig. 5. The normalized correlation coefficient algorithm (*NCC – Fast Normalized Cross-Correlation*) available in OpenCV is used by the Visual Locator Creator for the calculation of the matches.

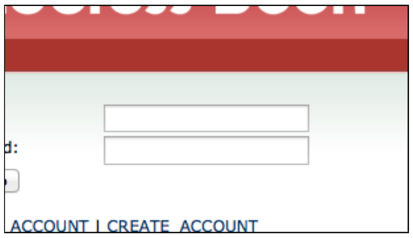


Fig. 5. Example of visual locator generated by PESTO for the username textBox

(4) The screenshot is then saved on the filesystem, together with a textual *mapping file* containing the association between the web element and its visual locator. This information is collected for each web element the test suite interacts with. It represents an important input for the second module of PESTO.

B. *Test Suite Transformer (Module 2)*

Fig. 6 shows the two main steps of Module 2, Test Suite Transformer: **(1)** transformation of the page objects (Page Object Transformer); **(2)** transformation of the test cases (Test Case Transformer).

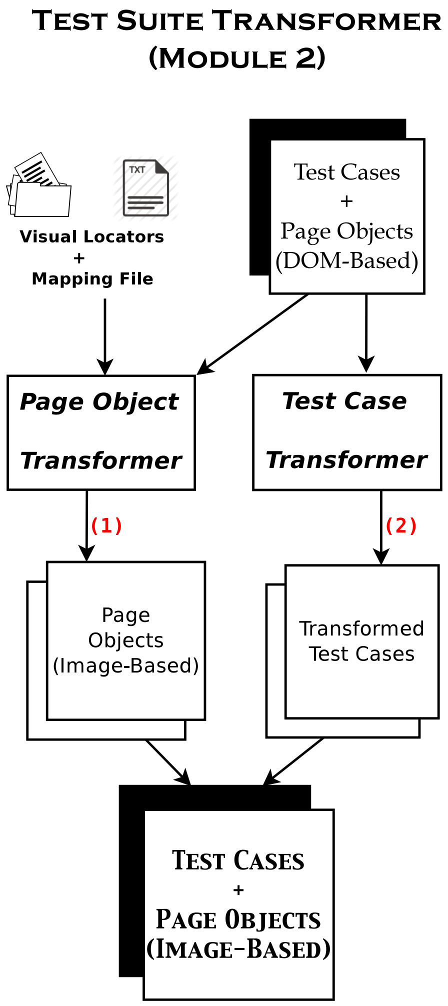
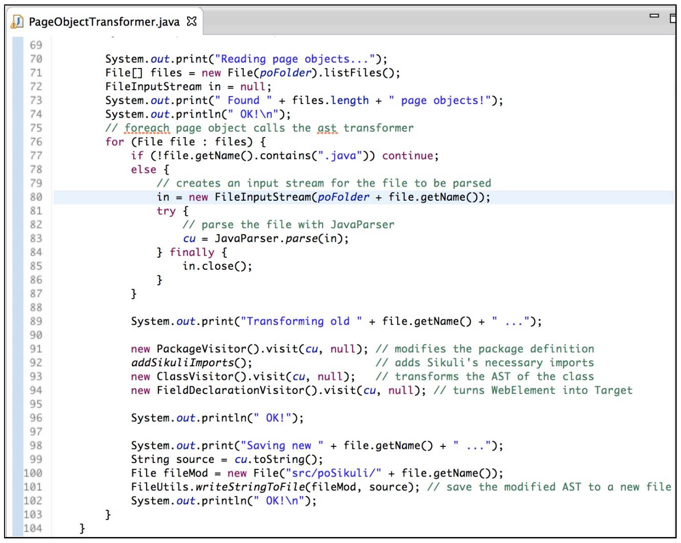

TEST SUITE TRANSFORMER (MODULE 2)
TXT
Visual Locators + Mapping File
Test Cases + Page Objects (DOM-Based)
Page Object Transformer
Test Case Transformer
(1)
(2)
Page Objects (Image-Based)
Transformed Test Cases
TEST CASES + PAGE OBJECTS (IMAGE-BASED)

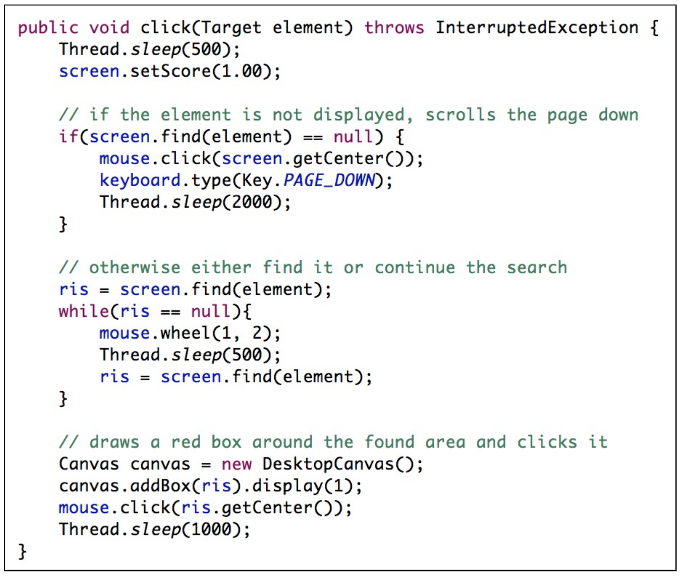
Fig. 6. Test Suite Transformer (Module 2)



```java
        System.out.print("Reading page objects...");
        File[] files = new File(poFolder).listFiles();
        FileInputStream in = null;
        System.out.print(" Found " + files.length + " page objects!");
        System.out.println(" OK!\n");
        // foreach page object calls the ast transformer
        for (File file : files) {
            if (!file.getName().contains(".java")) continue;
            else {
                // creates an input stream for the file to be parsed
                in = new FileInputStream(poFolder + file.getName());
                try {
                    // parse the file with JavaParser
                    cu = JavaParser.parse(in);
                } finally {
                    in.close();
                }
            }

            System.out.print("Transforming old " + file.getName() + " ...");

            new PackageVisitor().visit(cu, null); // modifies the package definition
            addSikuliImports();                   // adds Sikuli's necessary imports
            new ClassVisitor().visit(cu, null);   // transforms the AST of the class
            new FieldDeclarationVisitor().visit(cu, null); // turns WebElement into Target

            System.out.println(" OK!");

            System.out.print("Saving new " + file.getName() + " ...");
            String source = cu.toString();
            File fileMod = new File("src/poSikuli/" + file.getName());
            FileUtils.writeStringToFile(fileMod, source); // save the modified AST to a new file
            System.out.println(" OK!\n");
        }
    }
```

Fig. 7. Fragment of the Page Object Transformer

Fig. 7 shows an excerpt of the Page Object Transformer that produces image-based Page Objects (step **(1)** in Fig. 6). The Java code of each DOM-based Page Object is parsed by the JavaParser `parse` method. Then the package definition is modified: a new poSikuli package is created and the necessary imports to Sikuli API components are added. In this context, specific Sikuli visual methods need to be introduced: methods to *click* or *type text* on a web element need to be reimplemented, because in the visual approach web elements are images and the interaction is purely visual, hence simple DOM-based commands become composite visual operations. Moreover, Sikuli sees and interacts only with the visible portion of the desktop, hence additional scrolling operations, not needed with



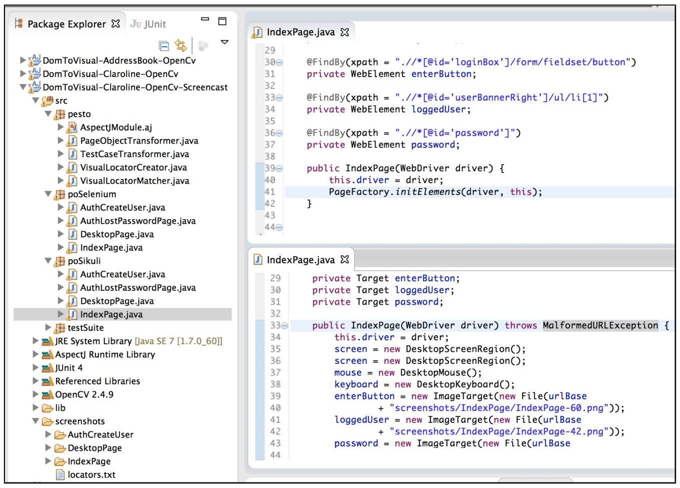
```java
public void click(Target element) throws InterruptedException {
    Thread.sleep(500);
    screen.setScore(1.00);

    // if the element is not displayed, scrolls the page down
    if(screen.find(element) == null) {
        mouse.click(screen.getCenter());
        keyboard.type(Key.PAGE_DOWN);
        Thread.sleep(2000);
    }

    // otherwise either find it or continue the search
    ris = screen.find(element);
    while(ris == null){
        mouse.wheel(1, 2);
        Thread.sleep(500);
        ris = screen.find(element);
    }

    // draws a red box around the found area and clicks it
    Canvas canvas = new DesktopCanvas();
    canvas.addBox(ris).display(1);
    mouse.click(ris.getCenter());
    Thread.sleep(1000);
}
```

Fig. 8. Redefined click method

the DOM-based approach, are required to detect every image before performing any operation on it; otherwise, the test case execution will fail.

Fig. 8 shows the example of the reimplementation of the `click` method. Sikuli first performs a `find` operation to search for the element of interest. If it is not found (i.e., null is returned), it could be because the element is not currently displayed in the browser. Sikuli scrolls the page down and continues its search until it finds the correct match (in Fig. 8, the exception handling code has been removed, for the sake of simplicity).

Finally, WebElement attribute definitions are turned into Target attributes. The inner code of the Page Object class is modified as follows (see Fig. 9): in the constructor, new visual components are added, to provide Sikuli with control over the mouse, the keyboard and the desktop screen region. Target attributes are initialised using the mapping information produced by Module 1: every DOM-based locator is associated with its visual representation.



```java
    @FindBy(xpath = ".//*[@id='loginBox']/form/fieldset/button")
    private WebElement enterButton;

    @FindBy(xpath = ".//*[@id='userBannerRight']/ul/li[1]")
    private WebElement loggedUser;

    @FindBy(xpath = ".//*[@id='password']")
    private WebElement password;

    public IndexPage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
```

```java
    private Target enterButton;
    private Target loggedUser;
    private Target password;

    public IndexPage(WebDriver driver) throws MalformedURLException {
        this.driver = driver;
        screen = new DesktopScreenRegion();
        screen = new DesktopScreenRegion();
        mouse = new DesktopMouse();
        keyboard = new DesktopKeyboard();
        enterButton = new ImageTarget(new File(urlBase
                + "screenshots/IndexPage/IndexPage-60.png"));
        loggedUser = new ImageTarget(new File(urlBase
                + "screenshots/IndexPage/IndexPage-42.png"));
        password = new ImageTarget(new File(urlBase
```

Fig. 9. Fragment of the Input (above) and Output (below) of the Page Object Transformer

The Test Case Transformer modifies the test cases source code in order to use the new visual page objects (step **(2)** in Fig. 6). This is a very simple step, which requires just to change a few import instructions, so as to replace the old Page Objects with the freshly generated ones.

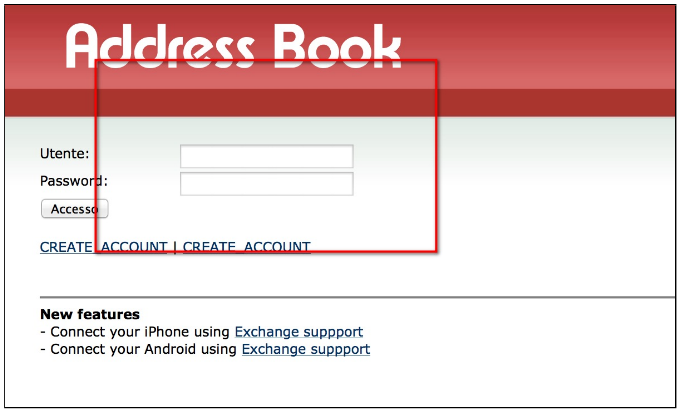
Rendering issue: Explicit vs Implicit Waits. When the execution flow goes from one web page to the next (e.g., after having submitted a form or clicked a link), a little amount of time is required for loading the new web page. If the test case goes ahead without taking this into account, it may not find the next target web element and thus it may return an error. WebDriver provides specific commands to deal with this issue[8] (i.e., Explicit and Implicit Waits). For instance, in the *PHP Address Book* test suite we used implicit waits, allowing WebDriver to poll the DOM for a certain amount of time (e.g.,

[8]http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp

up to 5 seconds) when trying to find a web element, if it is not immediately available. Sikuli provides an implicit wait command that blocks and waits until the target element is found within a given time period. The use of implicit delays slows down the execution of the test suites in cases where the target web element is not shown on the screen and scrolls are required to be able to visualise it. Thus, we preferred to insert explicit delays (e.g., `Thread.sleep(1000)`), whenever these are strictly necessary, as shown in Fig. 8.

C. Test Suite Execution

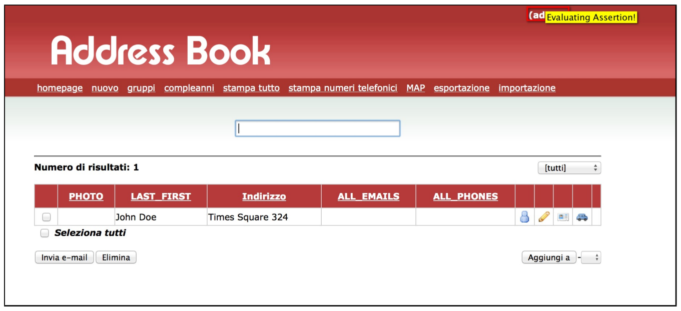
At this point, through a totally automatic process, the software tester has obtained a new, fully working visual test suite, which reimplements WebDriver's features for automating the browser interactions and exploits Sikuli's image recognition capabilities for web element localisation. No manual corrections are typically required. The software tester has just to run the new test suite, in the JUnit environment, and the test cases will deploy the new visual Page Objects instead of the old, DOM-based ones.



New features
- Connect your iPhone using Exchange suppport
- Connect your Android using Exchange suppport

Fig. 10. Run of the test suite, after the transformation

In Fig. 10, we can see that PESTO draws a red rectangle around the portion of the screen where it has found a match for the web element to interact with.



Numero di risultati: 1

	PHOTO	LAST_FIRST	Indirizzo	ALL_EMAILS	ALL_PHONES				
☐		John Doe	Times Square 324						

☐ *Seleziona tutti*

Fig. 11. Visual assertions of PESTO

When a visual assertion is evaluated, a canvas message is shown to the tester (see top-right corner in Fig. 11).

IV. Conclusions and Future Work

We have presented PESTO, a tool able to transform automatically DOM-based web test suites developed using Selenium WebDriver into visual test suites relying on the usage of Sikuli API.

About the experiences gained in developing this tool, we highlight:

- AOP was very useful and effective for the runtime capture of visual locators;
- the OpenCV computer vision library was very effective to automatically determine the number of matches of candidate visual locators;
- the JavaParser class used for parsing and modifying the abstract syntax trees of the test suites was simple to use and fitted for the purpose.

In our future work, we intend to improve PESTO in order to address its current limitations, e.g., the management of: (1) elements with complex visual interactions (e.g., drop-down lists); (2) elements changing their visual appearance during a state change (e.g., checkboxes). We also plan to execute an empirical study to measure the time required by developers to manually transform an existing real-sized DOM-based test suite into a visual one, in comparison with the time required when PESTO is adopted.

References

[1] E. Alegroth, M. Nass, and H. H. Olsson. JAutomate: A tool for system- and acceptance-test automation. In *Proceedings of 6th International Conference on Software Testing, Verification and Validation*, ICST 2013, pages 439–446. IEEE, 2013.

[2] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*, CHI 2010, pages 1535–1544. ACM, 2010.

[3] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of 18th Conference on Computer and Communications Security*, CCS 2011, pages 263–274. ACM, 2011.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, volume 1241 of *Lecture Notes in Computer Science (LNCS)*, pages 220–242. Springer, 1997.

[5] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Proceedings of 6th International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2013, pages 108–113. IEEE, 2013.

[6] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering*, WCRE 2013, pages 272–281. IEEE, 2013.

[7] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In S. Casteleyn, G. Rossi, and M. Winckler, editors, *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *Lecture Notes in Computer Science (LNCS)*, pages 322–340. Springer, 2014.

[8] P. Tonella, F. Ricca, and A. Marchetto. Recent advances in web testing. *Advances in Computers*, 93:1–51, 2014.