Why Creating Web Page Objects Manually If It Can Be Done Automatically?

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*—*Page Object* is a design pattern aimed at making web test scripts more readable, robust and maintainable. Unfortunately, the effort to manually create the page objects needed for a web application may be substantial and existing tools do not help web developers in such task.**

In this paper we present APOGEN, a tool for the automatic generation of page objects for web applications. Our tool automatically derives a testing model by reverse-engineering the target web application and uses a combination of dynamic and static analysis to generate Java page objects for the popular Selenium WebDriver framework. Our preliminary evaluation shows that it is possible to use around 3/4 of the automatic page object methods as they are, while the remaining 1/4 need only minor modifications.

Keywords*—*Web Testing, Testware Evolution, Page Object.

I. INTRODUCTION

Developing large web applications is a challenge for any company. In today's fast moving environment, the effort to adapt a software system running on the web to requirement changes is continuos. This poses serious issues in the maintenance and testing of web systems, demanding for increased levels of automation. For these reasons, test automation tools have become popular in the industry during the last 10 years and across a great variety of testing tasks such as regression, system or GUI testing. Automated tests can be run fast and frequently, making them quite cost-effective for web software with a medium to long expected maintenance and evolution life.

Despite their wide adoption, test automation techniques bring the problem of maintaining test scripts during software evolution – an issue well-known by practitioners. While there are interesting research contributions that try to address the testware evolution problem [1], [3], [6], [10], [12], we are far from a consolidated solution.

Among the practical solutions to cope with the maintenance of test scripts, the adoption of software engineering best practices, such as design patterns, is receiving substantial attention. Particularly famous in web testing is the *Page Object* design pattern, which aims at improving the test suite maintainability by reducing the duplication of code across test cases. A Page Object is a class that represents the web page elements as a series of objects and that encapsulates the functionalities of the web page in methods. The use of the Page Object pattern reduces the coupling between web pages and test cases, promoting reusability, readability and maintainability of the latter. A recent work has empirically shown the benefits associated with the adoption of the Page Object pattern in the maintenance of web test suites in an industrial environment [4].

Implementation of page objects is usually done either: *(i)* manually, or *(ii)* semi-automatically with the support of tools which are still very limited, as described in the paper.

In this paper we consider the problem of the *automatic generation of page objects for web applications*. This problem is challenging and no automatic and effective solution exists. Our approach aims at automatically reverse engineering a testing model, through a combination of dynamic and static analysis of the application under test (AUT). The application is reverse-engineered to expose its internal structure and functionalities, in order to gather useful information, that is used to generate the source code for the page objects. Our approach is implemented in a Java open-source prototype tool, APOGEN (Automatic Page Objects Generator). To our knowledge, there are research contributions using reverse engineering techniques for testing and analysis purposes [2], [7], [9], [11], but none of them specifically address the problem of the automatic representation of a web application into page objects, so as to improve the modularity and reusability of a test suite.

We compared a set of page objects automatically generated by APOGEN with the ones manually created by a human tester for the PHP AddressBook application in the context of a previous work [5]. Preliminary results indicate that our approach is viable, pretty accurate and potentially saving precious time otherwise required for manual page object creation: 75% of the functionalities of the automatic page objects needs no correction, i.e., they are ready for use, while the remaining 25% needs minor modifications.

The paper is organised as follows: Section II provides some background on the Page Object design pattern and the tools available to help developers in the page objects creation. Section III describes our approach and the tool APOGEN. Section IV presents the initial experimental results that evaluate the effectiveness of APOGEN, as well as its limitations and the future work we plan to carry out on the tool. Conclusions are drawn in Section V.

II. BACKGROUND

This section introduces the Page Object design pattern and explains why its adoption in test suites for web applications brings considerable advantages. In addition, we briefly classify the tools available on the market to assist testers in the implementation of the Page Object pattern in the test code, together with their limitations.

Specification vs Implementation. Without the adoption of any design pattern, automated test scripts may result in test code that is difficult to maintain and evolve. One of the main reasons is the duplication of code among the test cases. When the

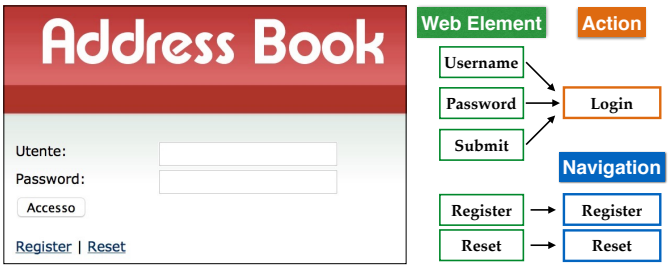

Fig. 1. Login page of PHP AddressBook and associated WebElements and Functionalities (i.e., Actions and Navigations)

same functionality must be necessarily invoked within multiple test cases (e.g., login), this results in some code fragment being scattered across test cases. Such code fragments include implementation details (e.g., sendKeys(username, "admin")) that are therefore duplicated instead of being shared and reused. Indeed, there is an important distinction between the specification *of what* to test versus the implementation *of how* to test it: the lack of a proper abstraction for recurring functionalities makes the two notions collide.

Let us consider the running example in Fig. 1, displaying the index page of the PHP AddressBook web application[1]. A test specification might be: *"When the user enters the correct username and password and clicks the login button, she/he is logged in and can see the personal home page"*. This describes a scenario – a specification of what the test should do. However, the test implementation has to deal with entities like: the username field is named "username", the password field is named "password", the login button is found via the CSS "#loginForm > input::nth-child(3)". If the developer changes the layout of the login page, the specification does not change (users still need to provide credentials and click the login button), whereas the implementation almost certainly needs the tester intervention to correct all test cases affected by the change.

Separating test specification from test implementation makes tests more robust and maintainable. For instance, if the login functionality changes, testers would like to modify only a single, reused code fragment, instead of changing every single test that requires the user to login.

Page Object and Page Factory. The test specification can be separated from its implementation by using the Page Object design pattern. With its introduction, 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. Page Objects serve as an interface of the web application: they represent the GUIs as a series of object-oriented classes that encapsulate the features offered by each page into methods.

For instance, for the web page of Fig. 1 we can identify the *Web Elements*, i.e., the GUI entities on which a user can interact, and the *Actions* associated to them, i.e., the behaviours triggered after that an event has occurred on a web element (e.g., click on the Register link performs a navigation and brings the user to the registration page). In Fig. 2 (a), we can see how these information are represented in a sample Page Object implemented upon the Selenium WebDriver framework[2]: each GUI element is represented as a WebElement class instance, properly named and annotated with a `@FindBy` annotation containing the *locator*, i.e. the specification of how to identify

[1] http://sourceforge.net/projects/php-addressbook/
[2] http://docs.seleniumhq.org/projects/webdriver/

```
public class Index {
  private WebDriver driver;
  @FindBy(xpath = "/html[1]/body[1]/div[1]/div[4]/a[1]")
  private WebElement register;
  @FindBy(xpath = "/html[1]/body[1]/div[1]/div[4]/a[2]")
  private WebElement reset;
  @FindBy(css = "#LoginForm > input:nth-child(2)")
  private WebElement user;
  @FindBy(css = "#LoginForm > input:nth-child(5)")
  private WebElement pass;
  @FindBy(css = "#LoginForm > input:nth-child(7)")
  private WebElement accesso;
  public Index(WebDriver driver) {
    this.driver = driver;
    PageFactory.initElements(driver, this);
  }
  public UserAdd goToRegister() {
    register.click();
    return new Register(driver);
  }
  public EmailPassword goToReset() {
    reset.click();
    return new Reset(driver);
  }
  public void loginForm(String args0, String args1) {
    user.sendKeys(args0);
    pass.sendKeys(args1);
    accesso.click();
  }
}
```

(a)

(b)

```
public class YourPageObjectName {

  private WebDriver driver;

  @FindBy(xpath = "//*[@id="LoginForm"]/input[1]")
  public WebElement username;

  @FindBy(xpath = "//*[@id="LoginForm"]/input[2]")
  public WebElement password;

  @FindBy(xpath = "//*[@type="submit
                  and @value="Accesso"]")
  public WebElement accesso;
}
```

(c)

```
public class Page {

  @FindBy(how=How.XPATH, using= "name(\"username\")")
  public WebElement emailTextBox;

  @FindBy(how=How.XPATH, using= "name(\"password\")")
  public WebElement passwordTexBox;

  @FindBy(how=How.XPATH, using= "name(\"Accesso\")")
  public WebElement loginButton;
}
```

Fig. 2. Comparison between a Page Object generated by APOGEN and those of OHMAP and SWD Page Recorder for the login page in Fig. 1

such web element in the GUI[3]. The class constructor makes use of the Page Factory design pattern, which instantiates the page object and pre-populates its fields based on the annotations. At last, the page object wraps the entire login form in a method providing the login functionality and offers two navigation methods for the Register and Reset links.

Existing Page Object Creation Tools. Currently there exist some open source frameworks to assist the tester during the creation of page objects. These tools mostly wrap the HTML content of the page and offer an aided creation of the associated source code. The most important ones are:

– *OHMAP*[4]: an online website allowing users to copy HTML code portions in a text area. The tool generates a simple Java class containing a WebElement instance for each input field encountered by the internal server-side static analyser. The variable names are taken from HTML attributes and the locators are XPaths similar to the ones generated by FirePath[5], a popular tool for the automatic generation of simple XPath expressions for elements inside web pages.

– *SWD Page Recorder*[6]: allows users to launch a web application and to inspect the GUI with a click&record feature: after every click on the interface, a drop-down menu is shown for the manual insertion of the web element variable name, while a relative XPath locator is produced. Code export is available for several languages (Java, C#, Python, Ruby and Perl).

– *WTF PageObject Utility Chrome Extension*[7]: assists the tester in the creation of the page objects (limited to web elements), by generating locators of kind: id, name, CSS, XPath. The output code is in Python.

Beyond the described tools, there are other open source projects, mostly abandoned or targeting only specific architectures like .NET[8] or Ruby[9]. Despite these tools provide useful features, most of the effort is still put on testers. These tools suffer several limitations, in particular: *(i)* only one page at a time is taken into account, without considering any notion of

[3] http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#locating-elements
[4] http://ohmap.virtuetech.de/
[5] https://addons.mozilla.org/en-US/firefox/addon/firepath/
[6] https://github.com/dzharii/swd-recorder
[7] https://github.com/wiredrive/wtframework/wiki/WTF-PageObject-Utility-Chrome-Extension
[8] https://github.com/patrickherrmann/Bumblebee
[9] https://github.com/cheezy/page-object

dynamism or web application structure; *(ii)* only a subset of web elements that can be used by a test is taken into account; *(iii)* the generated code are basic class skeletons, while the key characteristic of the page objects is to expose the web application functionalities in methods. This is *completely missing* in all the tools we analysed so far. We believe that it is possible to move the automation by far beyond the creation of a class skeleton containing web elements, using the knowledge present in the application itself. In Fig. 2 we report a comparison between the page objects generated by APOGEN (a) and those generated by tools OHMAP (b) and SWD Recorder (c). From the figure it is evident how the approaches implemented in (b) and (c) lack from several features (specifically, web elements for Register and Reset links are missing, as well as methods for any functionality). The page objects generated with APOGEN reflect the graph structure of the AUT and are enriched with the following features: *(i)* WebElement instances for each "clickable" element (i.e., an element on which it is possible to perform an action, e.g., links, buttons, input fields); *(ii)* methods to navigate the aforementioned graph structure; *(iii)* methods to fill and submit the forms. A sample page object of this kind is that in Fig. 2 (a).

Our approach aims to overcome the limitations of the existing frameworks, offering a more complete page object generation tool, so as to reduce substantially the testers' manual development effort.

III. APPROACH

This section describes the design and architecture of APOGEN, the tool implementing our approach. For the interested reader, a demo video and the source code can be found at: http://sepl.dibris.unige.it/2015-APO.php.

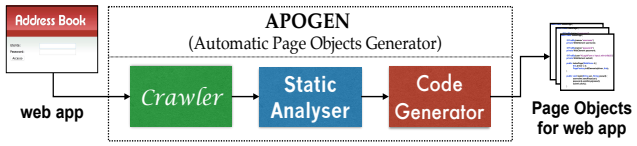

Fig. 3. High Level Architecture of APOGEN

APOGEN consists of three main modules (see Fig. 3): a Crawler, a Static Analyser, and a Code Generator. The input of APOGEN is any web application, together with the login credentials if necessary, while the output is a set of Java files, representing a code abstraction of the web application, organised using the Page Object and Page Factory design patterns, as supported by the Selenium WebDriver framework.

Crawler. In the first step we retrieve a high level representation of the AUT to generate a state-based model of the dynamic DOM (Document Object Model). For this we use a web crawler, i.e., a software that is able to browse a web application and download its pages. In particular, we used CRAWLJAX, a state of the art open source Java tool for automatically crawling and testing a JavaScript-based web application [8]. We chose CRAWLJAX because it automatically creates a state-based graph considering the dynamic DOM states and the event-based transitions between them. We seeded the crawler with proper inputs, such as the URL of the AUT and specific configurations necessary to perform an exploration of the application (in detail, we set no limits on the crawling depth, runtime, and number of states, albeit CRAWLJAX has an internal heuristics to determine whether the crawl is over). Moreover, CRAWLJAX may need the AUT login credentials to access the application and crawl the states accessible to authenticated users only.

Static Analyser. When the crawling is over, CRAWLJAX returns several outputs: the state-based graph of the web app and information about each visited dynamic state, i.e., the URL, the list of "clickable" elements, the DOM, a screenshot image of the web page, the list of links to other states. These information are parsed by the Static Analyser of APOGEN to create the testing model of the web application. In detail, for each state:
1) The URL is parsed and trimmed to get a meaningful class name for the page object class. In case of multiple occurrences (e.g., dynamic pages sharing the same URL, but conceptually in different states) an integer counter is added;
2) The web elements on which the crawler fired an event are inserted as WebElement instances in the page object class. For each of them, a meaningful variable name is retrieved by parsing the textual information and the attributes of the corresponding HTML tags. XPath or CSS locators are used to localise the web elements;
3) The links to other states obtained from the state-based graph are saved in the model;
4) The DOM of the state is saved and analysed to acquire information on forms. In particular, for each form APOGEN collects a series of data to be used for the methods generation: *(i)* a meaningful name is obtained by parsing and trimming the id, name and value attributes of the <form> HTML tag. This will be part of the methods' names produced in the following code generation phase; *(ii)* the list of HTML elements contained into the <form> tag, together with their associated locators, are saved as WebElement instances.

Code Generator. The last step is to transform the model produced by the Static Analyser into comprehensive Page Objects code for the Selenium framework. For each state in the model, the Code Generator module performs the following steps:
1) creates a Java class with the name obtained from the Static Analyser (first step), a standard package name (po) and the necessary Selenium imports.
2) creates a WebElement instance for each web element. For all of them, a @FindBy annotation, specifying the locator, is associated to the WebElement.
3) creates a default constructor with a Selenium WebDriver variable to control the browser. The constructor resorts on the PageFactory pattern to initialise all the web elements.
4) creates a navigational method for each link from the current state towards other page objects. The return type is the target page object.
5) creates a method for each submit button contained in each form. In particular we distinguish two cases: whether the form has *(i)* one submit button, or *(ii)* multiple submit buttons. In the former case, Code Generator creates a method for populating and submitting the form and its components (see page object of Fig. 2 (a)). In the latter case, the form has been used as a container for multiple web elements corresponding to different functionalities and Code Generator creates multiple methods to be later refined manually with the correct specification (an example is in Fig. 4).

IV. PRELIMINARY EVALUATION

This section describes the experimental procedure and the results obtained in a preliminary study we performed for evaluating APOGEN.

Subject Application. In the experiment we used PHP Address Book (ver. 8.2.5) – a PHP/MySQL-based address and phone book, contact manager, and organiser. The application is composed of about 30 kLOC and has been designed to be platform and browser independent. A test suite for the subject application was developed by a junior tester in the context of our previous work [5]. The test suite is written in Java; it follows the Page Object and Page Factory design patterns, and it is used as oracle against which we compare the results of APOGEN. The test suite is composed of 28 test cases and 7 page objects; it accounts for 1472 LOCs (1078 for the test cases and 394 for the page objects).

Research Questions. Our empirical study aims at answering the following research question:

RQ: *What is the percentage of generated methods that are (1) equivalent, (2) to be modified, or (3) missing w.r.t. the ones available in the manual POs?*

We want to understand whether automatically generated methods can be used directly, after minor modifications, or are missing.

Experimental Procedure. First, we ran APOGEN on the subject application. Second, we compared the methods of generated page objects with those of the manual test suite. We excluded from this analysis the getters methods – those retrieving meaningful textual information from the web page and potentially useful when defining the assertions of the test cases – since are not generated by the current, preliminary version of APOGEN. In detail, for each page object of the manual test suite, we manually inspected all methods (getters excluded): *(i)* classifying the kind of functionality as *navigational* or *action*; *(ii)* determining whether the method has a semantically equivalent counterpart in the automatic page objects (we tag such methods as *Equivalent*); *(iii)* determining whether the method has a counterpart in the automatic page objects that needs minor modifications (we tag such methods as *To Modify*); *(iv)* determining any missing methods (we tag such methods as *Missing*).

Experimental Results. Table I shows the data collected to answer RQ. It reports the page object methods used by the manually created test suite (first column), with the indication of the page object where it has been found both in the manual (second column) and in the automatic test suite (third column). Moreover, the table reports (fourth column) whether each method is a navigational method (e.g., a link towards a new page of the application) or an action method (e.g., to login into the application or to create a new address book entry). Finally, the last three columns indicate if each method was tagged as Equivalent, To Modify, or Missing.

Based on these data, we can notice that the test cases of the original test suite covered 16 functionalities of the subject application, for which corresponding methods have been created in four manual page objects. While the page objects of APOGEN are eight, they cover pretty well all the methods of the manual test suite. The different number of page objects is explained by the fact that page object generation is performed from the Crawler output. CRAWLJAX marks a page as a new dynamic state based on an internal heuristics – in short, it performs a DOM comparison after a preprocessing step in which all style, useless and dynamic elements are removed, leaving only the main structure. CRAWLJAX performs a state split only when it gathers evidence that the source and target states are two different entities. For instance, the index page of the subject application contains a login form, as visible in Fig. 1, while the home page is drastically different (not shown on this paper), since it displays the main content of the application. CRAWLJAX splits these two pages into two different states, while the manual tester decided to merge these two states and to incorporate the login method in the `IndexPage` page object.

Methods of the manual page objects	Manual	Automatic	Kind	Eq	TM	M
Navigate to a new Address Book	ABPage	Index1	NAV	✓		
Navigate to the Groups Page	ABPage	Index1	NAV	✓		
Navigate to the Birthdays Page	ABPage	Index1	NAV	✓		
Navigate to the Home Page	ABPage	Index1	NAV	✓		
Navigate to the Print View	ABPage	Index1	NAV	✓		
Navigate to the Print Phones View	ABPage	Index1	NAV	✓		
Create a new Address Book	EditPage	Edit	ACT	✓		
Select and Remove an Address Book	EditPage	Edit1	ACT		✓	
Login into the application	IndexPage	Index	ACT	✓		
Assign a user to a Group	IndexPage	Index1	ACT		✓	
Search into the Address Book	IndexPage	Index	ACT	✓		
Go to a new group	GroupPage	Group	NAV	✓		
Add a new group	GroupPage	Group1	ACT	✓		
Go to edit group	GroupPage	Group	NAV		✓	
Edit a group information	GroupPage	Group3	ACT	✓		
Select and Remove a group	GroupPage	Group2	ACT		✓	
Total	**4**	**8**	–	**12**	**4**	**0**
Coverage			–	**0.75**	**0.25**	**0.00**

TABLE I. Coverage (**Eq** = Equivalent; **TM** = To Modify; **M** = Missing)

In total there are 8 Navigational and 8 Action methods. We can notice how the methods marked as Equivalent are 12 out of 16 (i.e., 75%), while 4 out of 16 (i.e., 25%) need minor modifications. An example of such modification is shown in Fig. 4: only adding a parameter and a statement to the automatically generated method was required to align it with that of the manual implementation (in this case, to specify which web element identifies the correct checkbox from a list, so as to remove the right entry). No methods are missing.

Fig. 4. Element removal from a group requires an additional, initial click on the web element to be removed

Looking at the equivalent methods by method type, we have: 7 out of 8 Navigational methods (i.e., 87.5%) and 5 out of 8 Action methods (i.e., 62.5%). Thus, the first prototype of APOGEN is able to precisely recover almost entirely the navigations between the page objects and a remarkable percentage of the actions; the remaining methods require minimal corrections. To answer RQ, in our case study:

> *75% of the generated methods are equivalent* to those of the manual page objects, while *25% need to be refined* and *none is missing*.

Page Objects Comparison. Table II show how the methods of the manual page objects are covered by the automatic page objects. We can notice that for the ABPage page object, all 6 methods are covered by those of Index1 (in this case APOGEN's strategy mimics exactly that of a human tester). About the others: methods of IndexPage are distributed over Index and Index1, EditPage's over those of Edit and Edit1, while GroupPage's can be retrieved in the 4 automatic page objects Group, Group1, Group2, and Group3.

The results of Table II offer a clue for a possible, simple merging strategy: clustering the Page Objects sharing the same name (e.g., if we merge the methods contained in Edit and Edit1 in only one PO, we obtain a PO similar to IndexPage), to get closer to those that are defined by a human tester. Of course,

Page Object	Index	Index1	Edit	Edit1	Group	Group1	Group2	Group3
ABPage (6)	–	6	–	–	–	–	–	–
IndexPage (3)	1	2	–	–	–	–	–	–
EditPage (2)	–	–	1	1	–	–	–	–
GroupPage (5)	–	–	–	–	2	1	1	1

TABLE II. Page Object Comparison

there must be a balance between big page objects containing the majority of the functionalities and small page objects targeting only a few narrow features.

Estimated Development Effort Reduction. The manual test suite has a total of 1472 LOCs: 1078 LOCs for the test cases and 394 LOCs for the page objects, of which 335 are equivalent to those generated by APOGEN, 8 are to modify, 51 are for getters. By proportion over the LOCs, we infer that the effort reduction due to APOGEN would be about 85% (335:394=x:100) if we consider the development of page objects only, and roughly 23% (335:1472=x:100) for the entire test suite development. The LOCs are correlated with the development time but not directly proportional, hence, this rough estimate gives an approximate idea of the benefits potentially coming from the adoption of APOGEN.

Limitations and Future Work. Several issues are related to crawling the application: APOGEN relies on the performance of CRAWLJAX, which is overall a valid choice, but it is affected by the problems typical of any research tool. In particular, if CRAWLJAX fails at exploring the states space of the application, page objects are not created for those pages the crawler is not able to reach. Another limitation of our prototype comes from the static analysis performed to build the testing model. Although the combination of dynamic and static analysis has revealed to be effective in our case study, it may be not so effective if the DOM of the application has attributes with unintelligible names or has too few attributes on which the page object elements can be built and named (e.g., for naming the web elements variables, attribute names are fundamental). In our future work we intend to face this issue by studying the performance of our tool with more applications. A further idea could be to offer testers the possibility to interact with the tool before triggering the code generation so as to intervene in all the cases in which APOGEN fails to retrieve meaningful names.

In our future work, we will also investigate the aforementioned page objects merging strategy, to see whether it can improve the page objects understandability. Moreover, we intend to augment the page objects with a set of getter methods retrieving meaningful textual information from the web application by the dynamic identification of changing pieces of information in a web page. This would improve the completeness of the page objects and would provide support for the writing of test case assertions.

We plan to address the limitation of the methods that still need to be manually modified by recognising interaction patterns during the dynamic analysis, so that we might be able to automatically add the missing parameters and statements.

Threats to Validity of the Study. One threat to the validity of our study is associated with the approach used to compare manual and automatic page objects. To reduce this threat, we adopted the systematic procedure described in Sec. IV. The chosen application and the test suite considered in this study may have affected the results for the RQ; the percentages, as reported in Table I, may vary when different applications and different test suites are considered. Finally, concerning the generalisation of the results, we selected a real open source web application and a test suite already used in another scientific work, which makes the context realistic, even though further studies with additional applications are necessary to confirm and corroborate the obtained results.

V. CONCLUSIONS

Web test cases are usually decoupled from the implementation details by means of the Page Object design pattern. However, the manual effort to create the needed page objects can be remarkable. We have proposed a novel approach to automatically generate page objects for a web application and we have implemented it in a tool named APOGEN. A preliminary study in which we compared the generated page objects with the ones created manually by a human tester shows that 75% of the page object methods can be directly used by the tester for the development of a test suite, while the remaining 25% need only minor additions.

Although APOGEN is still a research prototype, the approach it implements is highly promising. In our future work, we intend to: *(i)* extend APOGEN with support for the definition of test case assertions, *(ii)* increase its level of automation, targeting all the cases that currently require manual intervention, and *(iii)* expand the empirical evaluation to a larger number of web applications.

REFERENCES

[1] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proc. of the 1st International Workshop on End-to-End Test Script Engineering*, ETSE 2011, pages 24–29. ACM, 2011.

[2] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. Reverse engineering web applications: The WARE approach. *Journal of Software Maintenance and Evolution*, 16(1-2):71–101, Jan. 2004.

[3] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *Proc. of the 31st International Conference on Software Engineering*, ICSE 2009, pages 408–418. IEEE, 2009.

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

[5] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In *Proc. of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *LNCS*, pages 322–340. Springer, 2014.

[6] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust XPath locators. In *Proc. of 25th International Symposium on Software Reliability Engineering Workshops*, ISSREW 2014, pages 449–454. IEEE, 2014.

[7] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proc. of the 1st International Conference on Software Testing, Verification, and Validation*, ICST 2008, pages 121–130. IEEE, 2008.

[8] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[9] C. Sacramento and A. Paiva. Web application model generation through reverse engineering and UI pattern inferring. In *Proc. of the 9th International Conference on the Quality of Information and Communications Technology*, QUATIC 2014. IEEE, 2014.

[10] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, and S. Sathishkumar. Efficient and change-resilient test automation: An industrial case study. In *Proc. of the 35th International Conference on Software Engineering*, ICSE 2013, pages 1002–1011. IEEE, 2013.

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

[12] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proc. of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 304–314. ACM, 2014.