Software Qual J
DOI 10.1007/s11219-016-9331-9

# APOGEN: automatic page object generator for web testing

Andrea Stocco[1] · Maurizio Leotta[1] · Filippo Ricca[1] · Paolo Tonella[2]

**Abstract** Modern web applications are characterized by ultra-rapid development cycles, and web testers tend to pay scant attention to the quality of their automated end-to-end test suites. Indeed, these quickly become hard to maintain, as the application under test evolves. As a result, end-to-end automated test suites are abandoned, despite their great potential for catching regressions. The use of the Page Object pattern has proven to be very effective in end-to-end web testing. Page objects are façade classes abstracting the internals of web pages into high-level business functions that can be invoked by the test cases. By decoupling test code from web page details, web test cases are more readable and maintainable. However, the manual development of such page objects requires substantial coding effort, which is paid off only later, during software evolution. In this paper, we describe a novel approach for the automatic generation of page objects for web applications. Our approach is implemented in the tool Apogen, which automatically derives a testing model by reverse engineering the target web application. It combines clustering and static analysis to identify meaningful page abstractions that are automatically turned into Java page objects for Selenium WebDriver. Our evaluation on an open-source web application shows that our approach is highly promising: Automatically generated page object methods cover most of the application functionalities and result in readable and meaningful code, which can be very useful to support the creation of more maintainable web test suites.

✉ Andrea Stocco
andrea.stocco@dibris.unige.it

Maurizio Leotta
maurizio.leotta@unige.it

Filippo Ricca
filippo.ricca@unige.it

Paolo Tonella
tonella@fbk.eu

[1] DIBRIS - Università degli Studi di Genova, Genoa, Italy

[2] Fondazione Bruno Kessler, Trento, Italy

Published online: 09 August 2016

**Keywords** Web testing · End-to-end testing · Page object pattern · Reverse engineering · Selenium WebDriver · Clustering

## 1 Introduction

Developing large web-based software systems is a challenge for software companies. Modern web applications undergo ultra-rapid development cycles, because business needs demand quick turnaround, pushing new features and bug fixes to production within days, along with quality assurance. Software organizations want to adequately test their software, as quickly as possible, because of time and cost constraints. In this context, manual testing is inadequate, since it is labor-intensive, error-prone, and it does not support the same kind of quality checks that are possible through the use of an automated testing framework.

In test automation, testers develop test scripts representing requirements scenarios, using a high-level programming language (e.g., Java, Python, Ruby). Automated tests can run fast and frequently, making them quite cost-effective for web software with a reasonable life expectancy. In an agile environment, the ability to react fast to ever-changing requirements is essential, because new test cases are developed and added to existing test suites in parallel with the development of the software itself. Automated testing shortens testing cycles and helps teams become more agile. For these reasons, test automation tools have become quite popular in the industry during the last 10 years across a wide range of testing tasks such as regression, system or GUI testing (Binder 1996; Fewster and Graham 1999; Ramler and Wolfmaier 2006; Nguyen et al. 2014; Gao et al. 2015).

Despite their wide adoption, end-to-end test automation tools bring the problem of maintaining test scripts during software evolution—an issue well-known by practitioners. In fact, albeit they make the interaction between tests and applications easier, they do not help testers to write well-architected test suites. This may generate serious issues during maintenance, because the effort to adapt a test suite to a web application in continuous evolution can be burdensome. The consequences range from lack of product quality, customer dissatisfaction and, ultimately, project failure. While there are interesting research contributions that try to address the testware evolution problem (Leotta et al. 2016a, 2014b, 2015; Thummalapenta et al. 2013; Yandrapally et al. 2014; Choudhary et al. 2011; Hammoudi et al. 2016a), we are far from a consolidated solution.

Software engineering best practices, such as design patterns, have proven to be successful in industrial settings. Above all, in web test automation, the “Page Object” design pattern has emerged as the most important pattern for enhancing test maintenance, reducing code duplication and lowering the coupling between test cases and web applications. A *Page Object* is an object-oriented class that serves as an interface toward a web page of the application under test (AUT). Test cases use the methods of the page object class whenever they need to interact with an element of the user interface (GUI). The benefit is that the test scripts do not need to be modified if the application underneath them changes. There is empirical evidence of the benefits associated with the adoption of the *Page Object* pattern in the maintenance of web test suites, both within an industrial environment (Leotta et al. 2013a) and in academia (Leotta et al. 2016a, 2013b). Unfortunately, the burden for the manual implementation of page objects is still on the shoulders of testers. In fact, existing tools are very limited and no automatic and effective research solutions have been proposed yet. We are the first to address the challenging research problem of the *automatic generation of page objects for web testing* by means of our tool Apogen.

The motivation behind our work is twofold. First, Apogen can help the tester save precious testing time in the creation of test suites for web applications, by automating the creation of a considerable amount of code that should be otherwise written manually. Second, it has been empirically shown that low-quality test suites are often abandoned (Christophe et al. 2014). For a tester, it is challenging to cope with the rapid evolution of the web application under test, because even small layout changes may result in several breakages (Leotta et al. 2016a, b; Hammoudi et al. 2016b). A possible solution to mitigate this problem is to have well-architected test suites, based on consolidated design patterns, which make them easy to modify and evolve. This motivates the importance of having page objects within web test suites, as well as our efforts in the development of Apogen.

This article is an extended version of our original short paper about Apogen (Stocco et al. 2015). It includes the following contributions:

- a detailed description of our approach for automatic page object generation;
- the tool Apogen, which implements our approach;
- two totally new modules of Apogen (the Clusterer, and the Cluster Visual Editor) able to improve the usage and effectiveness of the version of Apogen presented in Stocco et al. (2015);
- the automatic creation of page object getter methods, capable of detecting and reporting Document Object Model (DOM) differences observed between web pages within the same cluster;
- a case study illustrating Apogen on a real-world web application.

The paper is organized as follows: Sect. 2 provides background on the *Page Object* design pattern and presents the case study used for the evaluation and the tools available to help developers in the page object creation. Section 3 describes our approach and the tool Apogen, detailing each step on examples taken from the case study. Section 4 presents some initial experimental results to evaluate the effectiveness of Apogen, as well as its limitations, and the future work we plan to carry out to improve our tool. Related works are commented in Sect. 5, while conclusions are drawn in Sect. 6.

## 2 Background

In this section, we concisely present our case study, PetClinic, and we introduce the *Page Object* design pattern, explaining why its adoption in test suites for web applications brings considerable advantages. At last, we briefly classify the tools available on the market to assist testers in the implementation of page objects, together with their limitations.

### 2.1 Case study

In this paper, we use as case study, PetClinic,[1] a veterinary clinic web application allowing veterinarians to manage data about pets and their owners. PetClinic has been developed with the Java Spring Framework[2] and makes use of technologies as JavaBeans, MVC presentation layer and Hibernate. PetClinic consists of 94 files of various type (Java,

[1] https://github.com/spring-projects/spring-petclinic.

[2] http://projects.spring.io/spring-framework/.

XML, JSP, XSD, HTML, CSS, SQL, etc.), for a total of about 12 kLOC (thousands Lines Of Code), of which 6.1 kLOC accounting for Java source files (63 Java classes). Hence, it is a medium-size web system, with features and involving technologies that are quite typical of many similar systems available nowadays on the web. PetClinic supports the following use cases:

- View a list of veterinarians and their specialties
- View information pertaining to a pet owner
- Update the information pertaining to a pet owner
- Add a new pet owner to the system
- View information pertaining to a pet
- Update the information pertaining to a pet
- Add a new pet to the system
- View information pertaining to a pet's visitation history
- Add information pertaining to a visit to the pet's visitation history

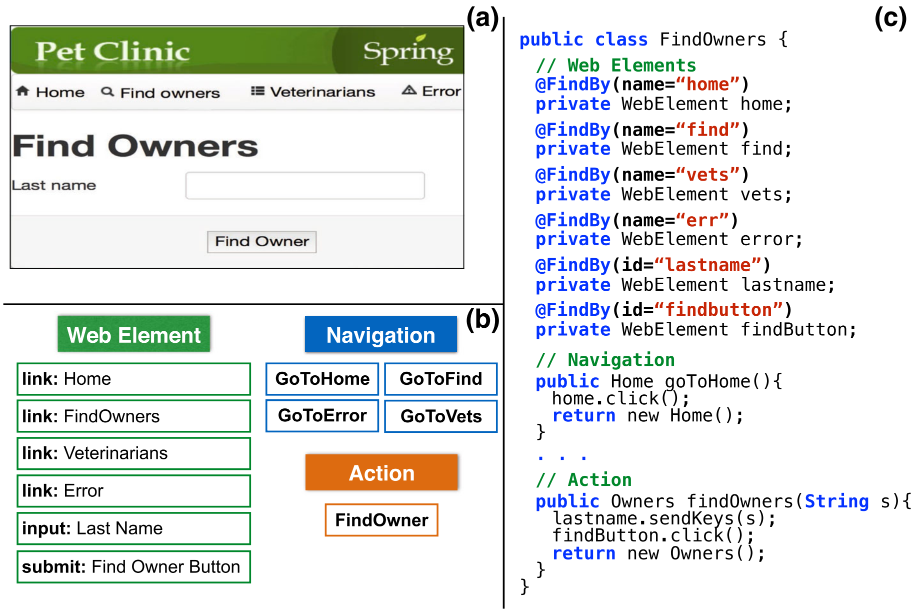
## 2.2 Tension between test specification and implementation

End-to-end functional testing of web applications is a type of black box testing based on the concept of test scenario, a sequence of steps/actions performed on the AUT. One or more test cases can be derived from a test scenario by specifying the actual input data to use in each step, and the expected outcomes. In automated web testing, web testers implement scenarios by writing test code composed by commands, which provide input data and set the values of GUI components, and by assertions to determine whether the program behaves correctly. Unfortunately, resulting test code is usually difficult to maintain and evolve. One of the main issues is the duplication of code among the test cases. When the same functionality must be necessarily invoked within multiple test cases (e.g., user login), a major drawback is the presence of the same code fragment within different test cases. Often, such code fragment only specifies implementation details (e.g., username.sendKeys("admin")), which are duplicated, instead of being shared and reused across test cases. Indeed, while performing any testing activity, it is important to distinguish between *test specification* and *test implementation*, keeping the test logic separated from the implementation details.

Figure 1a displays a page of PetClinic in which users are allowed to search for pets' owners, by specifying the last name. A test specification for the "happy scenario" might be:

> When the user enters an owner's last name and clicks the ⟨*Find Owner* ⟩ button, she can access her pets list

This scenario describes a specification of **what** the test should do. On the other hand, the test implementation includes all the details on **how** to technically perform it, dealing with concrete aspects like: the surname field is named "lastname" and the submit button is found on the GUI by the XPath "//*[@id='search-owner-form']/fieldset/div[2]/button." Thus, the test implementation is often strongly coupled with the underlying HTML page. When a maintenance activity occurs on the AUT, e.g., for functionalities improvement/correction, most of such technical details are impacted and need the tester intervention to correct all test cases invalidated by the changes. On the other hand, the test specification is less prone to change (users still need to insert the last name and click the submit button) and should be preserved (Leotta et al. 2014a).



```java
public class FindOwners {
  // Web Elements
  @FindBy(name="home")
  private WebElement home;
  @FindBy(name="find")
  private WebElement find;
  @FindBy(name="vets")
  private WebElement vets;
  @FindBy(name="err")
  private WebElement error;
  @FindBy(id="lastname")
  private WebElement lastname;
  @FindBy(id="findbutton")
  private WebElement findButton;
  // Navigation
  public Home goToHome(){
    home.click();
    return new Home();
  }
  . . .
  // Action
  public Owners findOwners(String s){
    lastname.sendKeys(s);
    findButton.click();
    return new Owners();
  }
}
```

**Fig. 1** A web page of the case study PetClinic **(a)**, together with its abstract representation in terms of Web Elements and Functionalities (Navigations and Actions) **(b)**, and an associated page object in Java language **(c)**

Separating the test specification from its implementation makes tests more concise and maintainable. However, test specification and implementation are often mixed up in the test code, and the lack of a proper abstraction for recurring functionalities makes the two notions collide. Ideally, if a web page of the AUT changes, testers would like to modify only a single, reused code fragment, instead of changing every single test case impacted by the change.

## 2.3 Page objects to the rescue

The test specification can be separated from its implementation by using the *Page Object* design pattern. Page objects hide the technical details about how the test code interacts with the web page, behind a more readable and business-focused façade. Specifically, page objects act as a programmable interface toward the web application: They represent the GUIs as a series of object-oriented classes that encapsulate the functionalities offered by each page into methods.

For instance, Figure 1a shows a web page of our case study, PetClinic, for which we provide an abstract representation in terms of web elements and functionalities in Fig. 1b. The *Web Elements* are the GUI entities on which a user can interact, whereas the *Functionalities* are behaviors triggered after an event has occurred on a web element (e.g., a click on the Veterinarians link performs a navigation toward the veterinarians page). In Fig. 1c, we can see how this information can be represented in a Java page object, implemented upon the Selenium WebDriver framework:[3] Each GUI element becomes a

[3] http://docs.seleniumhq.org/projects/webdriver/.

WebElement class instance, properly named and annotated with a `@FindBy` annotation containing the *locator*,[4] i.e., the specification of how to identify such web element on the GUI. The page object exposes the web page functionalities as methods. For what concerns *navigations*, in PetClinic there are navigational methods for the four menu bar links (in Fig. 1c, due to space constraints, we report only the navigation toward the Home page). The form for finding the pet's owners is wrapped in a method, which exposes the associated behavior. We call *actions* every data-submitting functionality in a web page. As a rule of thumb, page objects methods should return other page objects (Fowler 2013).

## 2.4 Limitations of the existing page object creation tools

While there are clear advantages in adopting page objects within the test code, their manual development is expensive and it is expected to increase with the application size. Moreover, manual creation of page objects includes many repetitive and boring tasks that could be automated. In fact, usually a tester has to: (a) manually inspect the AUT to gather insights about its functionalities; (b) decide upon which page objects to associate with separate classes; (c) collect the locators for the web elements with which tests shall interact (e.g., using tools as FirePath,[5] a popular tool for the automatic generation of XPath expressions for elements inside web pages).

We carried out a review of the tools available in the market. The existing open-source tools supporting page object creation offer very limited assistance in these tasks. In fact, existing solutions mostly wrap the HTML content of a unique page and offer limited aid to the creation of the source code. Tools that are worth to mention are:

- *OHMAP*:[6] 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. Variable names are taken from HTML attributes, and locators are XPaths similar to the ones generated by FirePath;
- *SWD Page Recorder*:[7] It 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*:[8] It assists the tester in the creation of page objects (providing only access to the 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[9] or Ruby.[10]

As an example, let us consider the Find Owner web page of PetClinic in Fig. 1a. In the corresponding page object, a web tester would expect to find at least: (1) the necessary web

[4] http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#locating-elements.

[5] https://addons.mozilla.org/en-US/firefox/addon/firepath/.

[6] http://ohmap.virtuetech.de/.

[7] https://github.com/dzharii/swd-recorder.

[8] https://github.com/wiredrive/wtframework/wiki/WTF-PageObject-Utility-Chrome-Extension.

[9] https://github.com/patrickherrmann/Bumblebee.

[10] https://github.com/cheezy/page-object

elements declarations (annotated with the respective locators), (2) four navigational methods for the menu bar and (3) one action method performing the search.

Figure 2a shows the code automatically generated by OHMAP. As we can notice, the level of automation of OHMAP is scarce: The tool detected the web elements only within the Find Owner form, and the generated Java code is merely limited to the declarations of such web elements. Further, the choice of a meaningful class name is left to the tester, and no methods for any functionality are present.

Figure 2b shows the code generated by SWD Page Recorder. We recall that the approach is similar to a capture&replay tool such as Selenium IDE, and thus it is semi-automated and does need the tester's intervention. The web elements shown in Fig. 2b are those interacted with while using SWD Page Recorder. Variable names have to be manually inserted, while locators are created by the tool. Also in this case, the generated Java code is limited to web element declarations only, and no methods were automatically recorded/generated.

Figure 2c shows the code generated by WTF PageObject Utility Chrome Extension. Also in this case, the approach is similar to a capture&replay tool such as Selenium IDE, and thus it is semi-automated and does need the tester's intervention. The web elements shown in Fig. 2c are those interacted with while using the plugin. Variables names have to be manually inserted, while locators are created by the tool. In this case, an additional, though trivial, method is present: _validate_page, for evaluating whether the page is in the correct state. This method performs two checks: It evaluates the equality between the web page URL and the output of the call to webdriver.current_url, and the equality between the web page title and the output of the call to webdriver.title. No further methods for any other functionality were automatically recorded/generated.

To summarize, existing tools suffer several, severe limitations and offer low support to the web tester. In particular: (1) only one page at a time is taken into account, without considering any notion of dynamism or web application structure, (2) only a small subset of web elements, among those that can be used in a test, is taken into account, (3) the generated code consists of basic class skeletons, while the key characteristics of page objects would be to expose the web application functionalities in methods. This last, important feature is completely missing in the tools we analyzed so far, with the only exception being WTF PageObject Utility Chrome Extension, which only creates a trivial method to verify whether the current page is in the expected state.

With Apogen we enhanced the level of automation far beyond the creation of a class skeleton containing web elements, exploiting the knowledge present in the application itself. Figure 3 shows the code generated by Apogen for the Find Owner page of PetClinic in Fig. 1a. Let us analyze it in detail.

First, the class provides an automatically generated and meaningful class name (i.e., Find), which is retrieved by parsing the URL of the web page. Second, all the web elements of interest for web testing are present, with automatically generated variable names, which are very understandable and are easy to trace back to the web page GUI. Third, a constructor adopting the Page Factory pattern is automatically created. The constructor aims at initializing the web elements and the WebDriver instance within the class. Fourth, all the navigational methods are automatically generated for all the menu bar links in the page. The return parameter is the target page object, and the method name is, in most cases, very understandable. Finally, there is a method (e.g., search_new_owner_form) exposing the action within the form. The method name is also very understandable. The return parameter is left void, because the target page

(a)

```java
public class YourPageObjectName {
      @FindBy(id = "lastName")
      public WebElement lastName;
      @FindBy(xpath = "//button[text()='Find Owner']")
      public WebElement findOwner;
}
```

(b)

```java
public class Page {
  @FindBy(how=How.XPATH, using= “html/body/div[1]/div/div/ul/li[1]/a)
  public WebElement home;
  @FindBy(how=How.XPATH, using= “html/body/div[1]/div/div/ul/li[2]/a)
  public WebElement find_owners;
  @FindBy(how=How.XPATH, using= “html/body/div[1]/div/div/ul/li[3]/a)
  public WebElement veterinarians;
  @FindBy(how=How.XPATH, using= “html/body/div[1]/div/div/ul/li[4]/a)
  public WebElement error;
  @FindBy(how=How.XPATH, using= “id(\”lastName\”)“)
  public WebElement lastNameTextBox;
  @FindBy(how=How.XPATH, using=“.//*[@id=‘search-owner-form']/fieldset/div[2]/button")
  public WebElement FindOwnerButton;
}
```

(c)

```python
from wtframework.wtf.web.page import PageObject, InvalidPageError

class MyPageObjectPage(PageObject):
  ''' MyPageObjectPage WTFramework PageObject representing a page like:
  http://localhost:9966/petclinic/owners/find.html
  '''
  ### Page Elements Section ###
  _home_a = lambda self:
     self.webdriver.find_element_by_css_selector("html>body>div>div>div>ul>li>a")
  _find_owners_a = lambda self:
     self.webdriver.find_element_by_css_selector("html>body>div>div>div>ul>li:nth-child(2)>a")
  _veterinarians_a = lambda self:
     self.webdriver.find_element_by_css_selector("html>body>div>div>div>ul>li:nth-child(3)>a")
  _error_a = lambda self:
     self.webdriver.find_element_by_css_selector("html>body>div>div>div>ul>li:nth-child(4)>a")
  lastname = lambda self:
     self.webdriver.find_element_by_id("lastName")
  find_owner_button = lambda self:
     self.webdriver.find_element_by_css_selector("html>body>div>form>fieldset>div:nth-child(2)>button")
  ### End Page Elements Section ###

  def _validate_page(self, webdriver):
  ''' Validates we are on the correct page.

  if not 'http://localhost:9966/petclinic/owners/find.html' in webdriver.current_url:
    raise InvalidPageError("This page did not pass MyPageObjectPage page validation.")

  if not 'PetClinic :: a Spring Framework demonstration' in webdriver.title:
    raise InvalidPageError("This page did not pass MyPageObjectPage page validation.")
```

**Fig. 2** Comparison between page objects generated by **a** OHMAP, **b** SWD Page Recorder and **c** WTF PageObject Utility Chrome Extension for the Find Owner page of PetClinic (*top*)

object is unknown, since Apogen misses the next dynamic state. In fact, in this case there could be multiple targets (i.e., multiple dynamic states) depending on the provided input (see Sect. 4.3).

To summarize, the page objects generated full automatically by Apogen reflect the structure of the web pages and are enriched with the following features: (1) 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); (2) methods to navigate the aforementioned graph structure; (3) methods to fill and submit forms. Our approach overcomes the limitations of the existing tools, offering a by far more complete page object generation approach, so as to substantially reduce the testers' manual development effort.

The gap between Apogen and the existing tools is huge, as evident from our example. For this reason, we did not perform an empirical comparison, which would have been of little scientific interest. We instead preferred to evaluate the level of automation of Apogen with respect to human-generated page objects, and we carried out such comparison by manually defining a gold standard, used as a reference for human generation.

```java
public class Find {
  // Web Elements Locators
  @FindBy(xpath = "/HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[1]/UL[1]/LI[1]/A[1]")
  private WebElement a_Home;
  @FindBy(xpath = "/HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[1]/UL[1]/LI[2]/A[1]")
  private WebElement a_Findowners;
  @FindBy(xpath = "/HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[1]/UL[1]/LI[3]/A[1]")
  private WebElement a_Veterinarians;
  @FindBy(xpath = "/HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[1]/UL[1]/LI[4]/A[1]")
  private WebElement a_Error;
  @FindBy(css = "#lastName")
  private WebElement input_lastName;
  @FindBy(css = "#search-owner-form > fieldset > div.form-actions > button")
  private WebElement button_Find_Owner;
  private WebDriver driver;
  /** Page Object for Find (state3) */
  // Constructor
  public Find(WebDriver driver) {
      this.driver = driver;
      PageFactory.initElements(driver, this);
  }
  // Navigation methods
  public Oups goToOups() {
      a_Error.click();
      return new Oups(driver);
  }
  public Owners goToOwners() {
      button_Findowner.click();
      return new Owners(driver);
  }
  public Vets goToVets() {
      a_Veterinarians.click();
      return new Vets(driver);
  }
  public Index goToIndex() {
      a_Home.click();
      return new Index(driver);
  }
  // Action method
  public void search_owner_form(String args0) {
      input_lastName.sendKeys(args0);
      button_Find_Owner.click();
  }
}
```

**Fig. 3** The page object generated by Apogen for the Find Owner page of PetClinic in Fig. 1a

## 3 APOGEN approach

Our approach for the automatic generation of page objects consists of the steps summarized in Fig. 4. First, we infer a model of the AUT by reverse engineering it by means of an event-based crawler. Then, similar web pages are clustered into syntactically and semantically meaningful groups. The event-based model (Graph) and the additional information (e.g., DOMs and clusters) are statically analyzed to generate a state object-based model. At last, this model is transformed into meaningful Java page objects, via model to text transformation. In the following, we detail each step on our case study, PetClinic.

The tool Apogen implements the described approach. It consists of five main modules (see Fig. 4): a Crawler, a Clusterer, a Cluster Visual Editor, a Static Analyzer and a Code Generator. The input of Apogen is any web application, together with the input data necessary for the login and form navigation. The output is a set of Java page objects, organized using the *Page Factory*[11] design pattern, as supported by the Selenium WebDriver framework.

[11] PageFactory is used to initialize the web elements and helps to remove boiler-plate code from the page objects methods. https://code.google.com/p/selenium/wiki/PageFactory.

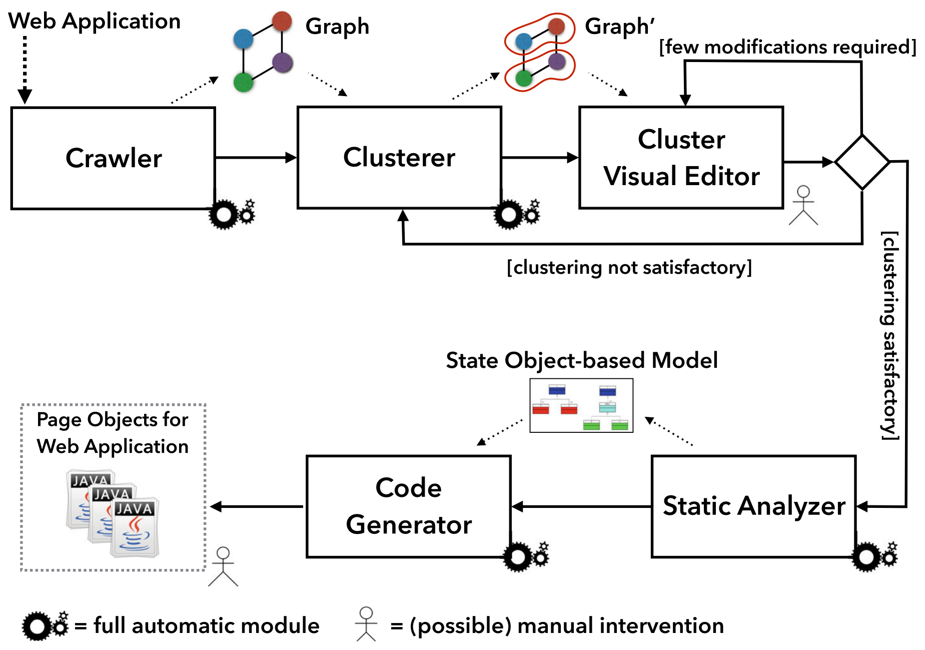

Web Application
Graph
Graph'
[few modifications required]
Crawler
Clusterer
Cluster Visual Editor
[clustering not satisfactory]
[clustering satisfactory]
State Object-based Model
Page Objects for Web Application
JAVA
Code Generator
Static Analyzer
= full automatic module
= (possible) manual intervention

**Fig. 4** High-level overview of Apogen 's approach for web page objects creation

Apogen generates page objects automatically, though manual interventions are possible to refine the result of clustering, or to refine the generated code. We further discuss these aspects in Sects. 3.3 and 4.3, respectively.

For the interested reader, further details about Apogen, the source code and demo videos, are available at: http://sepl.dibris.unige.it/APOGEN.php.

### 3.1 Crawler

The first step of our approach consists of exercising the web application functionalities. We perform this task by means of an automatic tool, called *crawler* (or *spider*), i.e., a software able to automatically browse and navigate a web application, reverse engineering its structure. The goal is to retrieve a high-level representation of the AUT in the form of a state-based model of the dynamic Document Object Model (DOM) states (Tonella et al. 2014).

For the development of this task in Apogen, we chose Crawljax (hereafter referred as the crawler), a state of the art tool for the automatic crawling of interactive web applications (Mesbah et al. 2012a, b). The crawler automatically creates a state-based graph considering the dynamic DOM states and the event-based transitions between them. The crawling results can be visually inspected with the crawler's built-in plugin, *CrawlOverview*. The state graph of PetClinic is shown in Fig. 5.

The crawler is fully customizable for ad hoc exploration. In detail, we provided it with the URL of the AUT, setting no limits on crawling depth, run time and number of states, albeit the crawler has an internal heuristic to determine whether the crawl is over. Login credentials and input data are specified separately, to access the application portions and

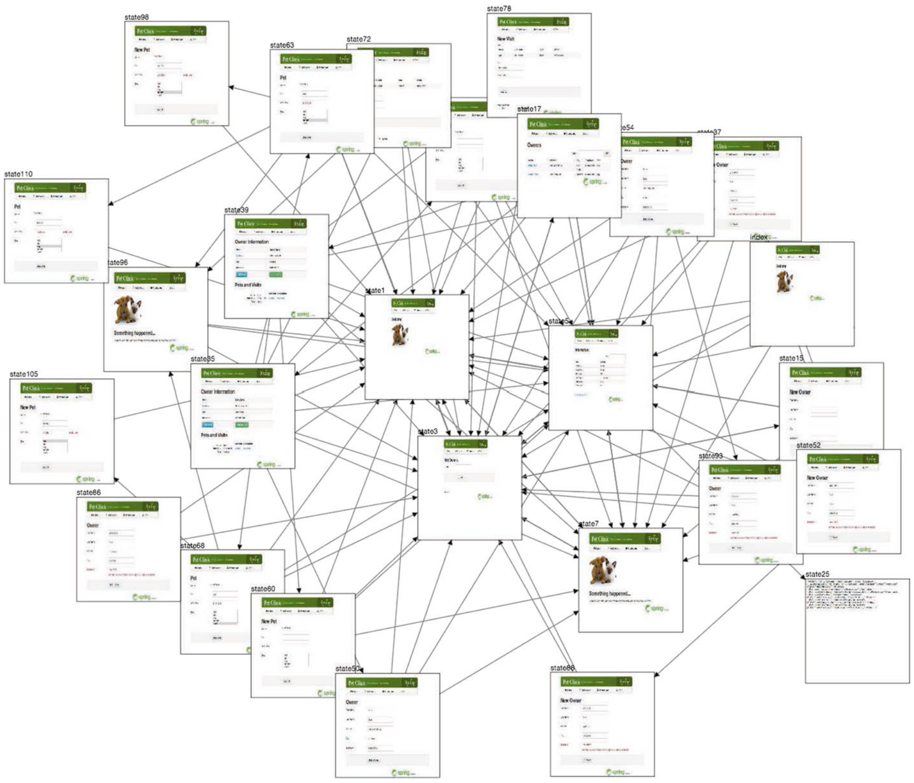


**Fig. 5** State graph representation of PetClinic, based on dynamic DOM states retrieved by Crawljax, the crawler adopted in Apogen

crawl the states accessible only when specific inputs (e.g., user credentials) are provided. When the crawling is over, the crawler outputs a `result.json` file containing the graph of the web app. Moreover, it returns information about each visited dynamic state: (1) URL, (2) list of "clickable" elements on the page, (3) DOM, (4) screenshot image and (5) list of links to other states.

## 3.2 Clusterer

The crawler is able to expose the visible and hidden portions of the application, as well as, the connections among web pages. For a tester, however, the manual inspection of this dense structure can be extremely challenging (see Fig. 5).

The *first issue* comes from the number of retrieved dynamic states that can be huge (in the order of hundreds of web pages). The default state abstraction of the crawler is affected by minor GUI changes leading to the presence of many states (web pages), conceptually clones of each other. Basically, when it visits the same page with different input data, the crawler often creates different dynamic states, even though the page is conceptually the same (Ricca and Tonella 2001; Ricca 2004; Tonella et al. 2014).

The *second issue* is in the number of edges between the states, which makes the visualization of the graph quite tangled.

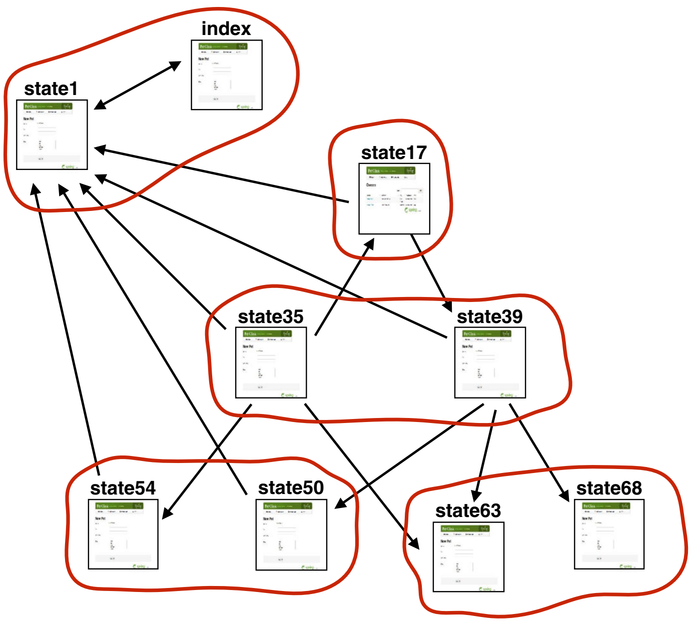


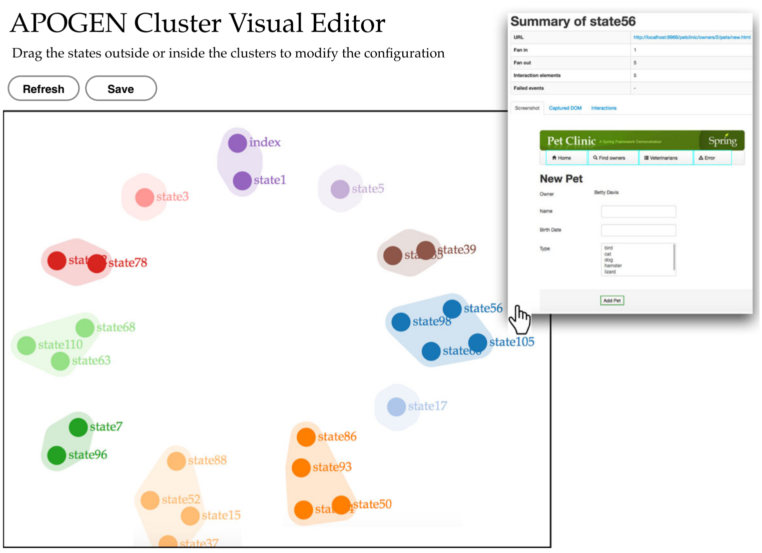
**Fig. 6** Logical cluster-based representation of a portion of the PetClinic dynamic DOM states

In conclusion, when crawling a non-trivial application, the size of the extracted model can be huge, undermining its understandability and reducing the effectiveness of the automated page object creation. In fact, the number of resulting page objects can be excessively high, including many replications and many similar classes that could be instead joined together.

To address both issues, we extended Apogen with an additional module, the Clusterer, which runs a clustering algorithm over the model. The idea is that web pages worthy to be represented within the same page object are grouped in the same cluster (as, for instance, those in Fig. 6). Correspondingly, edges are limited to the cross-cluster connections. We will deepen such aspects in Sect. 3.4.

We opted for two popular clustering algorithms from the machine learning literature: Hierarchical Agglomerative (Kaufman and Rousseeuw 1990) and K-means++ (Arthur and Vassilvitskii 2007). We integrated in Apogen the implementations available from the popular Java machine learning library Weka (Witten et al. 2011).

Clustering algorithms rely on the concept of similarity between web pages. There exist a number of works studying the factors affecting web page similarity (Blanco et al. 2011; Sampath 2012; Tombros and Ali 2005), in which authors observed that structural features are related to semantic properties of the data and provide meaningful means of comparison between web pages. To this aim, the Clusterer considers the following features: Tag Frequency, Word Frequency, URL and Document Object Model (DOM). The Clusterer is able to automatically extract syntactic feature matrixes from the web pages, which are then used by the clustering algorithms to compute the similarities. For instance, in the case of the Tag Frequency (which measures the frequency at which tags occur in a web page), the Clusterer extracts a matrix $TL \times W$, being $TL$ the complete list tags used in the web application, and $W$ the complete set of web pages. A complete list of the possible



Summary of state56

| | |
|---|---|
| URL | http://localhost:9966/petclinic/owners/2/pets/new.html |
| Fan in | 1 |
| Fan out | 5 |
| Interaction elements | 5 |
| Failed events | - |

New Pet

| | |
|---|---|
| Owner | Betty Davis |
| Name | |
| Birth Date | |
| Type | bird<br>cat<br>dog<br>hamster<br>lizard |

**Fig. 7** Cluster visual editor (CVE) of Apogen

*(algorithm, feature)* pairs, their evaluation, together with our in-depth analysis can be found in Stocco et al. (2016).

The tester can specify which algorithm and feature matrix the Clusterer must use, by simply editing a configuration file. Clusterer's default setting is [clustering algorithm="Hierarchical Agglomerative," feature matrix="DOM tree-edit distance"], because this was empirically found to be effective in producing clusters of web pages close to those manually defined by a human tester (Stocco et al. 2016). In case the output clusters are not satisfactory, the tester can experiment with a different combination of algorithms and features.

Technically, the Clusterer takes in input the graph from the file `result.json`, computes the clusters and outputs the result in a separate file `cluster.json`.

## 3.3 Cluster visual editor

Since there is no perfect clustering technique working for all web applications, the result might be somehow imprecise and might need to be manually refined. To this aim, Apogen has been extended to support the tester with an interactive cluster visualization and editor facility, allowing her to inspect and modify the clustering results, as shown in Fig. 7.

The Cluster Visual Editor (CVE) is a web-based tool developed using the Javascript library D3.[12] CVE reads the `cluster.json` file, opens a browser session by means of Selenium WebDriver, and shows the clusters by means of a force-based layout template,[13] where points are attracted by the center of gravity of their own cluster. Each cluster is represented as a colored convex hull containing as many points as the cluster cardinality.

[12] http://d3js.org/.

[13] A customization of http://bl.ocks.org/gmamaladze/9320969.

Each point is labeled with the dynamic state id retrieved by the crawler (e.g., state31), thus giving the tester full traceability with the previous crawling phase.

Looking at Fig. 4, there are three possible outcomes: (1) Clustering is largely unsatisfactory: groups of web pages are formed but strongly in disagreement with the tester's perspective. In this case, the tester can step back and run another clustering algorithm or choose a different number of clusters; (2) clustering is satisfactory: the tester proceeds with the page object generation; (3) clustering is good enough, but few modifications are required to produce better groups of web pages. In this last case, CVE allows the tester to interactively move points to the cluster they should belong to. Indeed, the tester can drag them far from the center of gravity, near to another group. When she drops them, the points, attracted by another center of gravity, snap to the new group. Once the clustering is aligned with the tester's perspective, CVE updates the `clusters.json` file.

## 3.4 Static analyzer

The Static Analyzer (SA) is the fourth component of Apogen, and it is responsible for the creation of the AUT abstraction in terms of state objects. The process is mainly divided into three parts: DOM diff calculation, FSM modification and merged state object creation.

### *3.4.1 Intra-cluster DOM differencing*

In functional web testing, a tester verifies the behavior of the web application through test case assertions on the visible portion of the web page. Hence, in the page objects, it is important to have "getter" methods retrieving textual portions of a web page that can be used by a tester for defining assertions. The automatic creation of getter methods is a challenging task and was a feature totally missing in the preliminary version of Apogen (Stocco et al. 2015).

In this paper, we support the automatic creation of getter methods in the page objects based on the differences between the web pages within the same cluster (namely, *intra-cluster DOM differencing*).

Figure 8 illustrates an overview of the intra-cluster DOM differencing mechanism. For each cluster in the `clusters.json` file, SA identifies the "master" and the "slaves" states. A *master state* is a state that can be regarded as representative of all the others, which we instead tag as *slaves*. The rationale behind this choice is to select a unique web page inside a cluster as the best page object candidate, so as to merge all the slaves inside the master. The master can be identified in various ways. The heuristic implemented in Apogen takes as master the state having the lowest id among the states within that cluster, which means that it was the first dynamic state retrieved by the crawler. Then, SA calculates the differences between the DOM of each slave and the master, to collect the dynamic portions of each web page, on top of which a tester might be interested in creating an assertion. For instance, Fig. 8 shows two dynamic states (*state*35 and *state*39) of the same PetClinic web page, displaying information about the owner. We can see how these two states differ only on some textual content. Correspondingly, a possible assertion might check whether the owner name is the correctly displayed to the user, since the actually displayed name varies from execution to execution.

The detection of the DOM-level differences is based on the differencing engine of XMLUnit,[14] which are also saved in the `clusters.json` file. In order to try to

[14] http://www.xmlunit.org/.

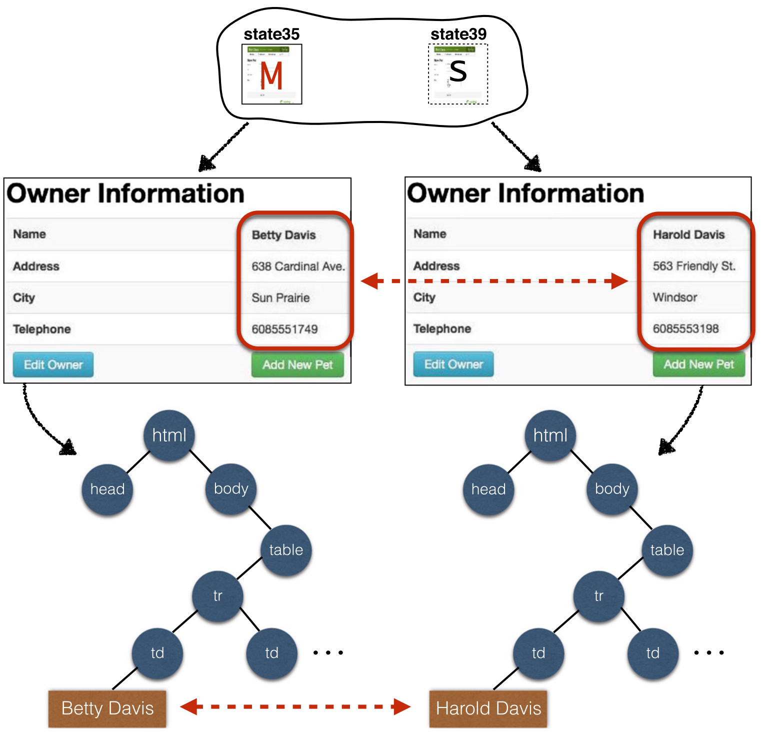


**Fig. 8** Pictorial view of the intra-cluster DOM differencing mechanism: The dynamic textual portions of the web pages are detected so as to support the generation of getter methods for each page object

minimize the number of false positives (i.e., irrelevant differences), the differencing engine ignores case sensitivity, white spaces, attribute value order and whitespaces, retaining only the differences in the textual node elements which were modified or added.

### 3.4.2 *Cluster-based graph modification*

We incorporate the new information about the clusters and the master/slave states into the graph produced by the crawler, by modifying the `result.json` file. Figure 9 shows a high-level overview of a portion of the PetClinic graph before (a) and after (b) such modification.

Figure 9a shows three clusters of five web pages: $C_0 = \{index, state1\}$, $C_1 = \{state17\}$ and $C_2 = \{state35, state39\}$. The master state of each cluster is marked with a capital "M"; the slaves with a lowercase "s." The dashed arrows are the intra-cluster edges (e.g., $index \rightarrow state1$), whereas the thick arrows represent the inter-cluster connections (e.g., $state35 \rightarrow state17$). In Fig. 9b, we can see how the graph connections have been transformed: Intra-cluster edges are removed, because the included web pages form a unique page object, while inter-cluster edges are modified as follows. Let $C_x$ and $C_y$ be two clusters:

1) $edge(M_x \rightarrow M_y)$: When two master states are connected (e.g., $state35$ and $state17$), no modification is triggered on the graph;

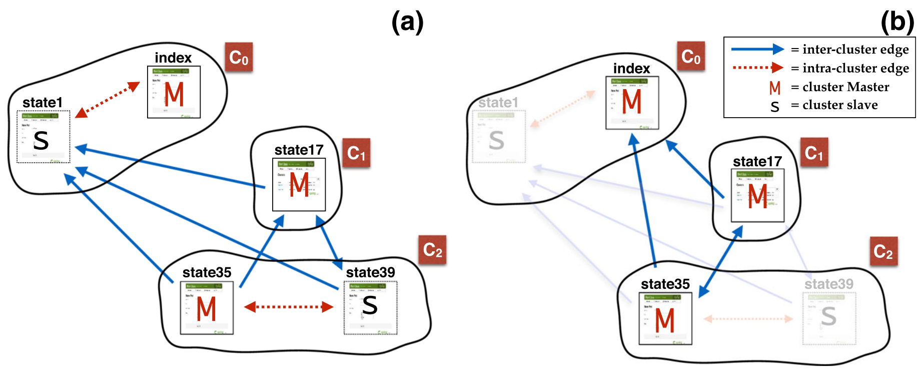


**Fig. 9** An example of graph modification

2) *edge*$(M_x \longrightarrow s_y)$: This is the case of clusters $C_0$ and $C_2$ and of the edge connecting *state*35 and *state*1. In this case, since *state*1 is a slave state, the target of the edge must be modified, so as to connect it to the master of the cluster, which is *index*. Thus, *edge*$(M_x \longrightarrow s_y)$ becomes *edge*$(M_x \longrightarrow M_y)$;
3) *edge*$(s_x \longrightarrow M_y)$: There is an edge between a slave of a cluster and a master of another cluster. This is the case of clusters $C_1$ and $C_2$ with the edge connecting *state*39 and *state*17. In this case, since *state*39 is a slave state, the source of the edge must be modified, so that it departs from the master of the cluster, which is *state*35. Thus, *edge*$(s_x \longrightarrow M_y)$ becomes *edge*$(M_x \longrightarrow M_y)$;
4) *edge*$(s_x \longrightarrow s_y)$: There is an edge between a slave of a cluster toward a slave of another cluster (e.g., *state*39 and *state*1). In this case, we connect the masters of the respective clusters (i.e., *state*35 and *index*), thus modifying *edge*$(s_x \longrightarrow s_y)$ into *edge*$(M_x \longrightarrow M_y)$.

### 3.4.3 *Merged state object creation*

At this point of the process, Apogen has an abstract graph-based representation of the AUT in the `result.json` file, with the clusters and diffs saved in the `clusters.json` file. This information is parsed by the SA to create the state object-based model of the web application. For each dynamic state, a page object abstraction is created. In detail:

1. The URL is parsed and trimmed to get a meaningful class name for the page object abstraction. For instance, *state*3 of PetClinic has URL localhost/petclinic/owners/find.html, from which the Find class name is extracted. In cases of Ajax single page applications, in which many dynamic pages share the same URL, an integer counter is added to the page object name;
2. The web elements on which the crawler fired an event are inserted as WebElement instances in the page object abstraction. For each of them, a meaningful variable name is retrieved by parsing the textual information and the attributes of the corresponding HTML tags. The XPath locators retrieved by the crawler are used to localize the web elements;
3. The transitions toward other states, specified in the `result.json` file, are associated with a functionality of type "Navigational";

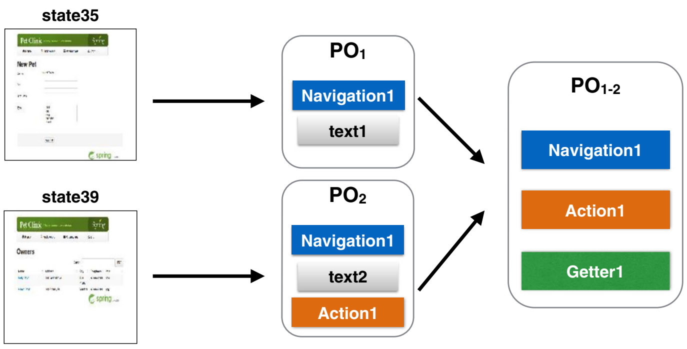


**Fig. 10** An example of page object merge

4. The DOM is analyzed with JavaParser[15] to acquire information about forms, which are associated with a functionality of type "Action." In particular, for each form, Apogen collects a series of data to be used for method generation: (1) A meaningful method name is obtained by parsing and trimming the id, name and value attributes of the FORM tag; (2) the list of HTML elements contained in the FORM tag are saved as WebElement instances, together with their associated CSS locators, as retrieved by JavaParser;
5. the DOM differences contained in the `cluster.json` file are associated with a functionality of type "Getter." It should be noticed that these are present only for the states considered as masters, since the DOM differencing component of Apogen compares the master against each slave.

After generating each page object abstraction in isolation, a merging phase is necessary to reflect the clusters in the `cluster.json` file. Figure 10 shows how the merge of two page object abstractions is performed.

Let us consider cluster $C_2 = \{state35, state39\}$ of Fig. 9a, where $state35$ and $state39$ contain the same navigational web elements, two different textual elements, with $state39$ having an additional action. Without considering the output of clustering, Apogen would generate two page objects PO$_1$ and PO$_2$ for $state35$ and $state39$, with the same navigational method *navigation1* replicated twice in the two page objects.

Although the example reported here is overly simple, when clusters contain tens of pages, for a web tester would be quite difficult to decide which page object to use (PO$_1$ or PO$_2$ in our example). Moreover, manual corrections and adjustments to the automatically generated page objects should be repeated multiple times.

The clustering-aided version of Apogen, instead, generates a merged page object, representing the entire cluster. The navigational method *navigation1* appears only once in such page object. Textual elements that differ across web pages are turned into getter methods. In our example, $state35$ is a structural clone of $state39$ (i.e., their DOMs are structurally equivalent). Since *text1* differs from *text2*, a getter method to retrieve the value of the dynamically variable textual element, namely *getter1*, is generated. Thus, the merged page object PO$_{1-2} = \{navigation1, action1, getter1\}$ exposes all the functionalities of both $state35$ and $state39$ that we believe are relevant for web test creation.

[15] http://javaparser.github.io/javaparser/.

### 3.5 Code generator

In the last step of our approach, the Code Generator transforms the state object-based model produced by the Static Analyzer into working page object code for the Selenium WebDriver framework. Again, we used JavaParser to iteratively create the abstract syntax trees (AST) of the Java page objects. More specifically, for each merged state abstraction (i.e., cluster), the Code Generator:

1. Creates a Java compilation unit (class) with the name obtained from the Static Analyzer, a standard package name (po) and the necessary Selenium WebDriver imports;
2. Creates a WebElement instance for each web element. For all of them, a @FindBy annotation, specifying the locator, is associated with the WebElement;
3. Creates a default constructor with a Selenium WebDriver variable to control the browser. The constructor resorts to the *PageFactory* pattern to initialize all the web elements;
4. Creates a method of type "Navigational" for each transition from the master page of the current cluster toward other clusters. The return type is the master page of the target cluster;
5. Creates one or more methods of type "Action" for each data-submitting form. We distinguish two cases: whether the form has (1) one submit button or (2) multiple submit buttons. In the former case, Code Generator creates a method for populating and submitting the form and its components. All form fields are associated with a variable name and a default action (e.g., submit.click()). In the latter case, the form is assumed to be used as a container for multiple web elements, which correspond to different functionalities. Correspondingly, the Code Generator creates multiple method skeletons. In both cases, the generated methods return *void*, because the type of the return parameter (e.g., Index or Error page object) depends on the input provided to the form (e.g., correct or incorrect credentials, respectively).
6. Creates a method of type "Getter" for each retrieved intra-cluster difference. Such getters return a *String* value.

## 4 APOGEN evaluation on PetClinic

In this section, we evaluate the artifacts produced by Apogen on the case study PetClinic. Our goal is to analyze the effectiveness of Apogen in generating high-quality page objects, with the purpose of understanding the strengths and the weaknesses of the proposed approach.

### 4.1 Procedure

We evaluated the quality of the page objects generated by Apogen on PetClinic by comparing them against a manually created gold standard (GS), according to the following procedure:

1. We manually inspected PetClinic in order to understand which page objects a tester would create, and which functionalities each page object should expose. The first two authors created, in isolation from each other, their own page object gold standards.

Then, they established a discussion to produce a unique gold standard (GS) for the page object of PetClinic. The differences were minimal: Indeed, the number of page objects was exactly the same, as well as their association with the conceptual pages of PetClinic (e.g., Login, Owners and Pets, Veterinarians). A few minor differences were in the definitions of the page object methods.

2. We ran Apogen to create the page objects for PetClinic. As recommended by our approach (see Fig. 4), we inspected the results of the clustering by means of the Cluster Visual Editor. Since clusters were equal to those in the GS, no manual modifications were required, and Apogen could complete the page objects generation in a fully automated fashion.
3. We inspected the page objects automatically generated by Apogen, and we compared each page object against the corresponding one in the GS. In particular, for each page object method: (1) We classified the kind of functionality as *navigational*, *action* or *getter*; (2) we determined whether each GS page object method had a semantically equivalent counterpart in the automatic page object, accepting only minor syntactical differences, as different variable/method names. We tagged such methods as *Equivalent*; (3) we determined the GS methods having a counterpart in the automatic page object in need of minor modifications (i.e., partially correct and requiring a few minor manual modifications, as the addition of a parameter). We tagged such methods as *To Modify*; (4) we determined any missing methods. We tagged such methods as *Missing*. Further, we are interested in determining if the approach implemented in Apogen leads to the generation of extra methods, i.e., methods not contained in the GS. We manually inspected each automatic page object, counting and tagging such methods as *Extra*.

The number of *Equivalent*, *To Modify*, *Missing* and *Extra* methods provides an indication of the possibility to use the code produced by Apogen as is and of the effort needed to manually correct the methods to be modified, or to add/delete the missing/extra methods.

## 4.2 Results

The number of page objects generated by Apogen for PetClinic is 11, equal to the number of page objects in the GS. Moreover, we found a perfect match, i.e., each page object in GS for a conceptual page of PetClinic (e.g., Login, Owners and Pets, Veterinarians) corresponds to one page object generated by Apogen and vice versa. This means that, in the case of PetClinic, Apogen was able to cluster perfectly the various instances of each conceptual page (e.g., the web pages showing the data of different veterinarians were grouped into a single cluster from which the page object Veterinarians.java was generated).

Of course, in the general case there might be mismatches between the automatic clusters and the gold standard. In our paper (Stocco et al. 2016), in which we evaluated the performance of various clustering algorithms on six real-size web applications, we have observed that, in the case of Hierarchical Agglomerative clustering, the realignment is negligible, i.e., the minimum number of web pages that must be moved between clusters to make them equal to the gold standard is in the range [0–9].

It is interesting to notice that the introduction of clustering techniques in Apogen has drastically reduced the amount of duplicated and useless code. Indeed, disabling the Clusterer leads to the generation of 26 page objects (corresponding to a 160 % increment in the number of generated page objects, and therefore of duplicated and useless code). Moreover, in the PetClinic case study, both clustering algorithms available in Apogen (i.e.,

**Table 1** Comparison between automatic and manual page object methods

| Type of method | Equivalent | To modify | Missing | Extra |
|---|---|---|---|---|
| Navigational | 9 | 0 | 0 | 0 |
| Action | 6 | 0 | 3 | 0 |
| Getter | 8 | 0 | 3 | 0 |
| Total | 23 | 0 | 6 | 0 |

Hierarchical Agglomerative and K-Means++) were able to find the same page-to-cluster assignment.

Table 1 shows the number of methods (navigational, action or getter) tagged as Equivalent, To Modify, Missing and Extra, in the page objects generated by Apogen for PetClinic w.r.t. the ones in the GS. The page objects in the GS contain overall 29 methods, while Apogen generated 23 methods. In the case of PetClinic, all the generated methods were found semantically equivalent to the methods reported in the GS, which explains the absence of "To Modify" methods. Looking at the results by type, for what concerns navigational methods, all of them are usable directly as produced by Apogen, while in the case of action and getter methods, only six are missing (three each, respectively). At last, no methods tagged as Extra were generated.

Table 2 provides additional details on the comparison between the page objects in the GS, and those generated by Apogen. We reported the functionality of the page object methods in the GS (Column 1), with the indication of the page object they belong to, either in the GS (Column 2) or in Apogen 's output (Column 3). Moreover, in Column 4, Table 2 reports the method type (NAV, ACT, GET). The last three columns indicate if the method was tagged as Equivalent (Eq), To Modify (TM), or Missing (Mis). Looking at the last column (Mis), we can observe that the six missing methods (equally divided between actions and getters) are concentrated in 3 page objects: Owners, Error, Veterinarians. For instance, in the Error page object, the getter method is missing because all the web pages in this cluster show exactly the same error message and thus the differencing mechanism (see Sect. 3.4.1) is unable to capture any dynamic information to be reported as getters. Further details on the missing methods are provided in Sect. 4.3.

In conclusion, we can say that Apogen has very good performances on the PetClinic case study: 23/29 methods are equivalent, none is to modify, and only 6/29 are missing. No extra methods are produced. In our paper (Stocco et al. 2016), we observed that the generation of additional methods affect only the getter category (an average of 9 extra getters over all the page objects, on six web applications). However, this is not expected to impact so negatively the activity of the tester.

## 4.3 Qualitative analysis

In this section, we analyze the code automatically generated by Apogen. We consider three representative web pages (OwnerInfo, Add Owner, Veterinarians) and their associated automatic page objects (OwnerInfo, New7 and Vets, respectively), with the aim of evaluating the degree of support they provide to web test case development.

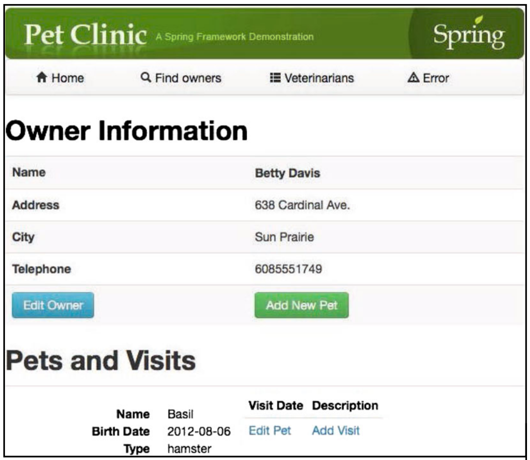
Figure 11 shows a page of PetClinic displaying owner information, together with the page object automatically generated by Apogen (Fig. 11 bottom-left and right). For space reasons, we show only a subset of the web elements, whereas we report the complete list of methods. We can see how the page object constructor makes use of the *Page Factory*

**Table 2** Detailed comparison between automatic and manual page object methods (*Eq* equivalent, *TM* to modify, *M* missing)

| Functionality | Manual GS cluster | Automatic PO | Kind | Eq | TM | Mis |
|---|---|---|---|---|---|---|
| Navigate to the home page | Index | Index | NAV | ✓ | | |
| Navigate to find owners page | Index | Index | NAV | ✓ | | |
| Navigate to error page | Index | Index | NAV | ✓ | | |
| Navigate to veterinarians page | Index | Index | NAV | ✓ | | |
| Add new pet | Add New Pet | New3 | ACT | ✓ | | |
| Get owner name | Add New Pet | New3 | GET | ✓ | | |
| Get invalid input | Add New Pet | New3 | GET | ✓ | | |
| Navigate to the owner page | Owners | Owners | NAV | ✓ | | |
| Search | Owners | Owners | ACT | | | ✓ |
| Export PDF | Owners | Owners | ACT | | | ✓ |
| Get table data | Owners | Owners | GET | | | ✓ |
| Update owner | Update Owner | Edit1 | ACT | ✓ | | |
| Get invalid input | Update Owner | Edit1 | GET | ✓ | | |
| Add owner | Add Owner | New7 | ACT | ✓ | | |
| Get error message | Error | Oups1 | GET | | | ✓ |
| Update pet | Update Pet | Edit5 | ACT | ✓ | | |
| Get owner name | Update Pet | Edit5 | GET | ✓ | | |
| Get invalid date | Update Pet | Edit5 | GET | ✓ | | |
| Add visit | New Visit | New1 | ACT | ✓ | | |
| Get table data | New Visit | New1 | GET | ✓ | | |
| Find owner | Find | Find | ACT | ✓ | | |
| Search veterinarians | Veterinarians | Vets | ACT | | | ✓ |
| Get table data | Veterinarians | Vets | GET | | | ✓ |
| Navigate to the edit owner page | OwnerInfo | OwnerInfo | NAV | ✓ | | |
| Navigate to the add owner page | OwnerInfo | OwnerInfo | NAV | ✓ | | |
| Navigate to the edit pet page | OwnerInfo | OwnerInfo | NAV | ✓ | | |
| Navigate to the add visit page | OwnerInfo | OwnerInfo | NAV | ✓ | | |
| Get owner info | OwnerInfo | OwnerInfo | GET | ✓ | | |
| Get pet info | OwnerInfo | OwnerInfo | GET | ✓ | | |
| Total | – | – | – | 23 | 0 | 6 |

design pattern to instantiate the page object and pre-populate its fields based on the annotations. This pattern is very important to improve the maintainability of the test suites. In fact, when the GUI of the application under test changes, locators might need to be changed. The Page Factory provides a centralized location that allows developers to easily find and change locators (when needed), rather than having them search through the whole set of page objects, since no duplications are present. Further, by correcting one single line of code, the modification is propagated to the entire test suite (Leotta et al. 2016a).

In the body of the page object, we find the navigational methods for: (1) the menu bar (goToIndex, goToFind, goToVets, goToOups1), (2) editing the owner information (goToEdit1), (3) adding/editing a pet (goToNew3, goToEdit5), (4) add a visit (goToNew1). These methods are instances of Equivalent methods. In fact, they replicate exactly what a tester would do while performing a navigation from the current page toward



Owner Information

| | |
|---|---|
| Name | Betty Davis |
| Address | 638 Cardinal Ave. |
| City | Sun Prairie |
| Telephone | 6085551749 |

Pets and Visits

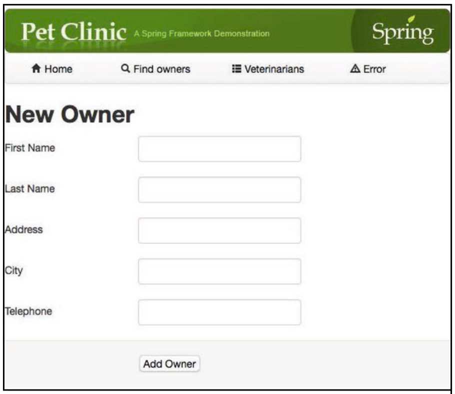
| | | Visit Date | Description |
|---|---|---|---|
| Name | Basil | | |
| Birth Date | 2012-08-06 | Edit Pet | Add Visit |
| Type | hamster | | |

```
public class OwnerInfo {

    // Web Elements
    @FindBy(xpath="…/DIV[1]/DIV[1]/UL[1]/LI[2]/A[1]")
    private WebElement a_Findowners;
    @FindBy(xpath="…/DIV[1]/DIV[1]/UL[1]/LI[1]/A[1]")
    private WebElement a_Home;
    ...
    private WebDriver driver;

    // Constructor
    public OwnerInfo(WebDriver driver){
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
    // Navigation
    public Index goToIndex(){
        a_Home.click();
        return new Index(driver);
    }
    public Vets goToVets(){
        a_Veterinarians.click();
        return new Vets(driver);
    }
    public New1 goToNew1(){
        a_addVisit.click();
        return new New1(driver);
    }
    public Oups1 goToOups1(){
        a_Error.click();
        return new Oups1(driver);
    }
    public Find goToFind(){
        a_Findowners.click();
        return new Find(driver);
    }
    public Edit5 goToEdit5(){
        a_Editpet.click();
        return new Edit5(driver);
    }
    public Edit1 goToEdit1(){
        a_Editowner.click();
        return new Edit1(driver);
    }
    public New3 goToNew3(){
        a_addnewpet.click();
        return new New3(driver);
    }

    // Getters
    /**
        source: state35
        target: state39
        cause: textual change
        before: 638 Cardinal Ave.
        after: 563 Friendly St.
    */
    public String get_dd_Dt_Address() {
        return dd_Dt_Address.getText();
    }
    /**
        source: state35
        target: state39
        cause: textual change
        before: hamster
        after: lizard
    */
    public String get_dd_Dt_Type() {
        return dd_Dt_Type.getText();
    }
    /**
        source: state35
        target: state39
        cause: textual change
        before: 6085551749
        after: 6085553198
    */
    public String get_dd_Dt_Telephone()
        return dd_Dt_Telephone.getText();
    }
    /**
        source: state35
        target: state39
        cause: textual change
        before: Sun Prairie
        after: Windsor
    */
    public String get_dd_Dt_City() {
        return dd_Dt_City.getText();
    }
    /**
        source: state35
        target: state39
        cause: textual change
        before: 2012-08-06
        after: 2010-11-30
    */
    public String get_dd_Dt_Birthdate()
        return dd_Dt_Birthdate.getText();
    }
    /**
        source: state35
        target: state39
        cause: textual change
        before: Betty Davis
        after: Harold Davis
    */
    public String get_b_Name() {
        return b_Name.getText();
    }
    /**
        source: state35
        target: state39
        cause: textual change
        before: Basil
        after: Iggy
    */
    public String get_dd_Dt_Name() {
        return dd_Dt_Name();
    }
}
```

**Fig. 11** A web page of the case study PetClinic, together with its Java page object, including examples of navigational and getter methods

another: click on the menu item/anchor, change the AUT state by instantiating the target page object (e.g., Index in goToIndex), and passing it the WebDriver instance. On Fig. 11 (right), there are the getter methods exposing the dynamic portions of the web page, as retrieved by the DOM diff mechanism described in Sect. 3.4.1. These methods are also tagged as Equivalent, because they capture correctly the salient information useful for assertion definition.

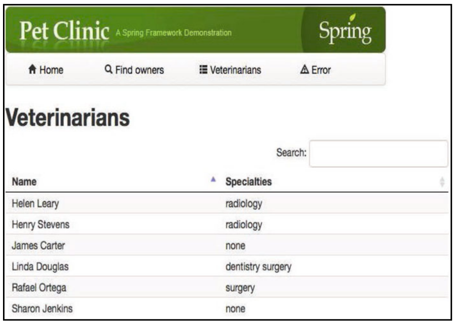
Figure 12 illustrates other two interesting examples. On the top-left part is the page in which the user can insert information about a new owner, whereas on the bottom-left is the corresponding generated automatic page object, in which we can see an instance of action method: add_owner_form. This method allows the tester to fill and submit the data of the owner, by associating each web element used for data insertion with its default action (e.g.,



New Owner

| | |
|---|---|
| First Name | |
| Last Name | |
| Address | |
| City | |
| Telephone | |

Add Owner



Veterinarians

Search:

| Name | Specialties |
|---|---|
| Helen Leary | radiology |
| Henry Stevens | radiology |
| James Carter | none |
| Linda Douglas | dentistry surgery |
| Rafael Ortega | surgery |
| Sharon Jenkins | none |

```
public class New7 {
   // Web Elements
   @FindBy(css = "#firstName")
   private WebElement input_firstName;
   @FindBy(css = "#lastName")
   private WebElement input_lastName;
   ...
   private WebDriver driver;

   // Constructor
   public New7(WebDriver driver){
      this.driver = driver;
      PageFactory.initElements(driver, this);
   }
   // Navigation
   public Index goToIndex(){
      a_Home.click();
      return new Index(driver);
   }
   public Vets goToVets(){
      a_Veterinarians.click();
      return new Vets(driver);
   }
   public Oups1 goToOups1(){
      a_Error.click();
      return new Oups1(driver);
   }
   // Action
   public void add_owner_form(String args0,
   String args1, String args2, String args3,
   String args4){
      input_firstName.sendKeys(args0);
      input_lastName.sendKeys(args1);
      input_address.sendKeys(args2);
      input_city.sendKeys(args3);
      input_telephone.sendKeys(args4);
      button_Add_Owner.click();
   }
}
```

```
public class Vets {
   // Web Elements
   @FindBy(xpath="…/DIV[1]/DIV[2]/DIV[1]/A[1]")
   private WebElement a_Home;
   @FindBy(css = "#lastName")
   private WebElement a_Findowners;
   ...
   private WebDriver driver;

   // Constructor
   public Vets(WebDriver driver){
      this.driver = driver;
      PageFactory.initElements(driver, this);
   }
   // Navigation
   public Index goToIndex(){
      a_Home.click();
      return new Index(driver);
   }
   public Find goToFind(){
      a_Find.click();
      return new Find(driver);
   }
   public Oups1 goToOups1(){
      a_Error.click();
      return new Oups1(driver);
   }
}
```

**Fig. 12** A web page of the running example PetClinic, together with its Java page object, including examples of action and missing methods

sendKeys for the text fields and click for the submit button). The return type is void, because the target page object is unknown since the Static Analyzer misses the next dynamic state. In fact, in this case there could be multiple targets (multiple dynamic states) depending on the provided input. In particular, in case of successful execution of the method, the returned page object should be the Owners page object, whereas if an incorrect value is passed as argument, the page object should not trigger any state change and should manage the error. The tester, well-aware of such two possible scenarios, can better decide if and when to return a page object representing the next dynamic state, depending on the execution context of the test cases.

On the top-right part of Fig. 12 is the page showing the veterinarians table, and a text field to search for a particular entry. We can notice that only the constructor and the navigational methods are present in the automatic page object Vets. The action method to perform the search is Missing (Search Veterinarians in Table 2). This is due to the fact that the Static Analyzer of Apogen builds action methods on top of FORM tags, while the search text field is not contained in any form. Thus, our heuristic fails and the method has to be manually added to the page object. In our future work, we will investigate solutions to face this issue (e.g., by recognizing Javascript methods that reach to the enter key on a text field).

### *4.3.1 Apogen 's naming convention*

The assignment of meaningful names in the page objects code has a fundamental impact on their readability. Concerning methods and fields, Apogen employs a naming rule based on the information contained in the Document Object Model (DOM), which is leveraged to name the variables.

More precisely, the tag of the web element is used as prefix. Then, the value of either attribute *id, name* or *class* is trimmed and utilized to name the variable (e.g, field a_Home in the page object New7 of Fig. 12 (bottom-left) refers to the first anchor Home in the menu bar).

For methods, we distinguish three cases: navigational, action and getter methods. For the navigational methods, the name is composed of the prefix “goTo” and the target page object (e.g., goToIndex). For the actions, Apogen takes the value of either attribute *id, name* or *class* of the HTML form tag to name them. Further, parameters of type *String* are inserted (as many as the number of tags within the form), using the generic name args*n*, with *n* successively incremented from 0. About the getters (see Fig. 11 (right)), the name is composed of the prefix “get_”, the tag, and any meaningful text or label (e.g., get_b_-Name). In case of duplicates (e.g., two table entries in which the tag and the text is exactly identical), a numeric counter is added.

As shown in the example in Sect. 4.3, most of the page object class names are understandable, apart from some cases (e.g., New7), in which the mapping with the web page is less explicit.

In conclusion, when the HTML underneath the web application has meaningful values for the tag attributes analyzed by Apogen (as occurred in the case of PetClinic), the generated page objects are very readable and understandable. In our future work, we intend to enhance the naming strategy to support the cases in which the class names or methods are hard to understand (e.g., page object New7 or method getToOups1 in the page object OwnerInfo).

## 4.4 Page objects to support test case development

In this section, we sketch the benefits of adopting the page objects generated by APOGEN, during the creation of a web test case for PETCLINIC. In Fig. 13 we can see an example of test case for testing the correct behavior of PETCLINIC, when a new owner is added. On the left is the test code without the adoption of any page object, whereas on the right is the same test case, modified to use the automatic page objects generated by APOGEN.

We can notice that the test case on the left is brittle and full of technicalities on how to access to web elements in the underlying DOM structure. This limits the overall readability of the test case and the functionalities are scarcely encapsulated.

The test code on the right, instead, is by far more readable because it corresponds more directly to the use case scenario's steps (e.g., open the index page, go to the find owner page, etc). Moreover, it should be noticed that all the page object-related source code is *automatically generated* and ready to use, limiting to a large extent the tester's burden to produce well-architected test cases, and allowing her to better focus on the test semantics and logics.

This is evident by observing at Fig. 13. In the test case on the left, for inserting the owner info, the test case employs six commands (rows 8–13), whereas in the test case on the right (page object-based), the test calls a unique method add_owner_form (row 9) of page object New7. The test case implementation is cleaner and for a tester becomes easier to call such method with multiple different inputs (or developing more test cases). This enlarges the input coverage and allows to better test the AUT, at a minimum effort.

Creating the required page objects manually would increase substantially the total test case creation time and might be skipped in practical settings, despite its potential future benefits, for lack of time before the release of the application.

```
1  public class TestAddOwnerWithoutPageObjects {

2  @Test
3   public void testAddOwner(){

4   WebDriver driver = new FirefoxDriver();

5   driver.get("http://localhost:9966/petclinic/");

6   driver.findElement(By.css("#find")).click();

7   driver.findElement(By.css("#addOwner")).click();

8   driver.findElement(By.css("#firstname"))
            .sendKeys("Andrea");

9   driver.findElement(By.css("#lastname"))
            .sendKeys("Stocco");

10  driver.findElement(By.css("#address"))
                  .sendKeys("Via Dodecaneso, 35");

11  driver.findElement(By.css("#city"))
                  .sendKeys("Genova");

12  driver.findElement(By.css("#telephone"))
                  .sendKeys("010-1234567");

13  driver.findElement(By.css("#submit")).click();

14  AssertThat(driver.findElement(By.css("#name"))
            .getText(), is("Andrea Stocco"));

15  driver.quit();
   }
  }
```

```
1  public class TestAddOwnerWithPageObjects {

2   @Test
3   public void testAddOwner(){

4    WebDriver driver = new FirefoxDriver();

5    driver.get("http://localhost:9966/petclinic/");

6    Index indexPage = new Index(driver);

7    Find findPage = indexPage.goToFind();

8    New7 addOwnerPage = findPage.goToNew7();

9    addOwnerPage.add_owner_form("Andrea", "Stocco",
     "Via Dodecaneso, 35", "Genova", "010-1234567");

10   OwnerInfo newOwnerPage = new OwnerInfo(driver);

11   AssertThat(newOwnerPage.get_b_Name(),
        is("Andrea Stocco"));

12   driver.quit();
    }
   }
```

**Fig. 13** Two test cases created to test the Add Owner functionality of PETCLINIC: On the left is the test code without the adoption of page objects, whereas on the right is the same test case, using the automatic page objects generated by APOGEN

### 4.5 Summary

Overall, Apogen showed satisfactory performance on PetClinic. The clustering approach was effective in producing clusters of web pages that were identical to those defined manually, thus reducing the manual effort for creating correct web page clusters to zero. Furthermore, the page objects generated by Apogen were very similar to the page objects that a developer would create manually. A challenging issue in the automatic generation of page objects is a way to expose the web page functionalities (not just the web page elements) in methods—a feature which is completely missing in most of the existing tools. About 80 % of the methods generated by Apogen, instead, can be used as is by a tester, breaking down substantially the manual effort for page object creation. Moreover, a big part (73 %) of the dynamic portions of the web pages is correctly captured by our differencing mechanism. The getter methods created on top of such differences correspond to meaningful and useful behavioral abstractions that help to support assertion definition.

Apogen performed equally well when applied to other web applications. Indeed, on six real-size web applications, the obtained results are consistent with those of PetClinic, because the clustering approach provided clusters of web pages close or identical to those manually produced by a human (only 18 differences were noticed, over six web applications) and 75 % of the code generated by Apogen was ready to use as is (Stocco et al. 2016).

### 4.6 Upcoming improvements

Improvements are still possible and in several directions. The Clusterer of Apogen relies on the manual choice of parameter *k* (the number of clusters). We believe it is important to aid the tester in the estimation of how many clusters are reasonable for the given data. The problem of estimating automatically the optimal number of clusters *k* is extremely challenging. In the machine learning literature, there exist a few alternative approaches to estimate the number of clusters in a given dataset. According to these approaches, it is possible to: (1) measure the percentage of variance explained as a function of the number of clusters (Ketchen et al. 2008), (2) use cross-validation, (3) use silhouettes plots (Rousseeuw 1987). Another set of methods is instead based on information criteria, such as the Akaike information criterion (AIC) (Akaike 1981), and the Bayesian information criterion (BIC) (Schwarz 1978). In our future work, we plan to empirically experiment with such methods.

Another improvement area is related to the heuristics used to create action and getter methods. Concerning actions, we will develop techniques to capture the missing functionalities within a web page, e.g., to manage also the cases in which potential actions are not contained in forms (as the case of Search Veterinarians in Sect. 4.3). Concerning getters, when a cluster contains a single page, the differencing mechanism cannot be applied (as was the case of the table of veterinarians, in Table 2). In our future work, we plan to provide a complementary approach for input generation, capable of exposing the variable part of the unique page in singleton clusters. Moreover, not all the differences between web pages necessarily have a textual counterpart, and thus we plan to integrate and experiment with image recognition mechanisms to identify the dynamic page portions (Choudhary et al. 2010).

## 5 Related work

The problem of creating and maintaining automated test suites for web applications has been studied from different viewpoints (Choudhary et al. 2011; Thummalapenta et al. 2013; Yandrapally et al. 2014). There are research contributions using reverse engineering techniques for testing and analysis purposes (Di Lucca et al. 2004; Marchetto et al. 2008; Sacramento and Paiva 2014; Tonella et al. 2014), but none of them specifically addresses the problem of the automatic representation of a web application as page objects, so as to improve the modularity and reusability of web test suites.

Indeed, the automatic creation of page objects for end-to-end web testing is a completely novel research field and, to the best of our knowledge, there are no strictly related works. However, there are papers that deal with applications of clustering techniques to support web testing and engineering.

### 5.1 Page/state objects

*Martin Fowler* described this pattern under the name Window Driver (Fowler 2013). Fowler illustrates basic rules of thumb for page object creation, as that "it should provide an easy programmable interface to the program, hiding the underlying widgetry in the window." Furthermore, Fowler advocates assertions-free page objects, meaning that "although they are commonly used for testing, they should not make assertions themselves. Their responsibility is to provide access to the state of the underlying page, leaving the testers to carry out the assertion logic."

However, the term "page object" has been popularized by the Selenium web testing framework, which has become the generally used name. Selenium's wiki strongly encourages the use of page objects as a best practice and provides advices on how they should be implemented (Selenium Wiki 2013). For instance, Selenium's wiki encourage testers to try and think about the services that they are interacting with rather than the implementation.

Van Deursen (2015a, b) describes a state-based generalization of page objects which can help testers to answer to practical questions as which page objects one should create when testing web applications, or what actions one should include in a page object. From the tester viewpoint, moving a page object to the state level makes the design of test scenarios easier. Besides the mere terminological difference, the work of van Deursen depicts a series of guidelines and best practices that we share and tried to incorporate in the development of Apogen, in consideration of our ultimate goal which is the automatic generation of meaningful page objects.

An empirical study by Leotta et al. (2013b) shows that test suites developed with a programmable approach exhibit higher benefits than those developed with a capture-and-replay method. In addition, maintaining a programmable test suite required less effort, also thanks to the introduction of the Page Object design pattern, able to decouple the test logics from that of the application. This insight is confirmed by a further investigation (Leotta et al. 2013a) on the maintainability benefits when the Page Object pattern is adopted. When the software evolves, the maintenance effort is drastically reduced, which justifies the initial additional effort to implement the test suite using page objects. On the other hand, the manual creation of page objects is a repetitive and time-consuming task. This motivates the present work, focused on the problem of automatically generating page objects to support the creation of high-maintainable web test suites—a completely novel research area.

## 5.2 Web applications clustering

Crescenzi et al. (2005) tackle the problem of automatically discovering the main pages offered by a site, by exploring only a small yet representative portion. They propose a model to describe abstract structural features of HTML pages, the page schema, which is basically a subset of the root-to-link paths in the corresponding DOM tree representation, along with the referenced URLs. Based on this model, they developed an algorithm that accepts the URL of an entry point to a target web site, visits a limited yet representative number of pages, and produces an accurate clustering of pages based on their structure. The output is a model, intended to be used as input for automatic web pages' wrapper generation. On the other hand, in the development of the clustering module of Apogen, we studied several alternative structural similarity measures beyond the DOM, with the aim of supporting the clustering of web pages from the page object perspective.

On a different line, there are several works using clustering for understanding purposes (Tonella et al. 2003a; Lucia et al. 2009; Ricca et al. 2008). Tonella et al. (2003a) propose a clustering method based on the automatic extraction of the keywords to produce cohesive groups of pages. The aim is to support web site understanding, by providing clusters that are displayed as a single node in reverse engineered diagrams. The presence of common keywords also helps to automatically label the recovered clusters. In an extension of the previous work, Ricca et al. (2008) propose a clustering approach based on client-side HTML pages with similar content, which gives good results with content-oriented sites rather than application-oriented ones. A crawler is employed to download the web pages of the target site, and common keywords are used to group pages together. In the experiments on 17 web sites, it is shown that the clusters produced automatically are close to those that a human would produce for a given web site. Differently, Apogen supports clustering over several kinds of web page properties, such as URLs, DOMs, tags and textual content. On the contrary, De Lucia et al. (2009) investigate the effect of using different clustering algorithms in the reverse engineering field to identify pages that are similar either at the structural level or at the content level.

In another work, Tonella et al. (2003b) provide two approaches for web clustering evaluation: the gold standard and a task-oriented approach. In fact, when clustering the entities composing a web application, several alternative options are available (using their structure, their connectivity, or their content), and the problem is how to evaluate the competing clustering techniques. They illustrate two methods for clustering evaluation, the gold standard and the task-oriented approach, analyzing the advantages, disadvantages, guidelines and examples for both of them.

In a previous paper (Stocco et al. 2016), we adopted the first approach and compared the results of various web page clustering algorithms against a gold standard in order to ensure its meaningfulness from the web testing viewpoint. Moreover, we did not limit to content-based metrics and, in fact, structural properties showed to be more effective, especially those based on the HTML structure, as DOM similarity and tag frequency.

## 5.3 Crawling/differencing techniques

In the last years, differencing techniques have been used to detect the so-called cross-browser incompatibilities (XBIs), i.e., inconsistencies and issues in the existing applications when run of different browsers (e.g., Chrome, Opera, Safari). The detection of XBIs requires a lot of manual effort for developers who need to make sure that web applications are compatible with as many browsers as possible.

Choudhary et al. (2010) present a technique based on differential testing to automatically detect cross-browser issues (XBIs) and assist developers in their diagnosis. They compare the behavior of a web application in different web browsers, identifying differences in behavior as potential issues, and reporting them to the developers. Given a page to be analyzed, the comparison is performed by combining a structural analysis of the information in the page's DOM and a visual analysis of the page's appearance, obtained through screen captures. The approach operates on single web pages and focuses on finding XBIs, whereas we perform intra-cluster DOM differencing.

Mesbah and Prasad (2011) pose the problem of cross-browser compatibility testing of modern web applications as a functional consistency check of web application behavior across different browsers. They propose an automatic approach for analyzing the given web application under different browser environments and capturing the behavior as a finite-state machine before formally comparing the generated models for equivalence on a pairwise-basis and exposing any observed discrepancies. Similarly, we adopt crawling and web page differencing, but our approach is constrained to finding textual differences between intra-cluster web pages, on top of which a tester can build meaningful assertions, because usually a tester defines functional assertions on the visible parts of a web page.

Successively, Choudhary et al. (2012) combined and extended the two above-mentioned approaches for XBI detection in the tool CrossCheck. Even though we share some methods, such as the reverse engineering of a web application model with a crawler, and performs DOM differencing between web pages, we use clustering, which is an unsupervised machine learning technique, instead of a classifier, to target a completely different goal, the automatic page object construction to support web test suites development.

## 6 Conclusions

Page objects are used in end-to-end functional web testing to decouple the test case logic from the technical low-level implementation. Although page objects bring undeniable advantages, as decreasing the maintenance effort of an automated test suite, the burden of their manual development limits their wide adoption.

We presented a novel approach for the automatic generation of page objects for web applications, implemented in our tool Apogen, which automatically generates Java page objects for a web application using a combination of reverse engineering, machine learning, web-visualization, HTML static analysis and differencing, and AST creation.

In this paper, we gave a detailed description of our tool, illustrating it by means of a case study application, PetClinic. The results are promising and show that our approach for automatic page object generation is viable. Indeed, Apogen is able to generate meaningful and comprehensive Java page objects that cover most of the PetClinic functionalities. We have also shown that automatic page objects reduce the tester effort for test cases development, because a substantial portion of the test code is automatically generated, and increase the quality (namely, the understandability and maintainability) of the test code, because they decouple its logic from its concrete implementation.

## References

Akaike, H. (1981). Likelihood of a model and information criteria. *Journal of Econometrics*, *16*(1), 3–14.

Arthur, D., & Vassilvitskii, S. (2007). K-means++: The advantages of careful seeding. In *Proceedings of the 18th annual ACM-SIAM symposium on discrete algorithms, SODA 2007* (pp. 1027–1035). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Binder, R. V. (1996). Testing object-oriented software: A survey. *Software Testing, Verification and Reliability*, *6*(3–4), 125–252.

Blanco, L., Dalvi, N., & Machanavajjhala, A. (2011). Highly efficient algorithms for structural clustering of large websites. In *Proceedings of the 20th international conference on world wide web, WWW '11* (pp. 437–446). New York, NY, USA: ACM.

Choudhary, S. R., Prasad, M. R., & Orso, A. (2012). Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proceedings of the 5th IEEE international conference on software testing, verification and validation, ICST 2012* (pp. 171–180). Washington, DC, USA: IEEE Computer Society.

Choudhary, S. R., Versee, H., & Orso, A. (2010). Webdiff: Automated identification of cross-browser issues in web applications. In *Proceedings of the 26th IEEE international conference on software maintenance, ICSM 2010* (pp. 1–10). IEEE Computer Society.

Choudhary, S. R., Zhao, D., Versee, H., & Orso, A. (2011). WATER: Web application test repair. In *Proceedings of the 1st international workshop on end-to-end test script engineering, ETSE 2011* (pp. 24–29). ACM.

Christophe, L., Stevens, R., Roover, C. D., & Meuter, W. D. (2014). Prevalence and maintenance of automated functional tests for web applications. In *Proceedings of 30th international conference on software maintenance and evolution, ICSME*, IEEE.

Crescenzi, V., Merialdo, P., & Missier, P. (2005). Clustering web pages based on their structure. *Data Knowledge Engineering*, *54*(3), 279–299.

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

Fewster, M., & Graham, D. (1999). *Software test automation: Effective use of test execution tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc.

Fowler, M. (2013). PageObject. http://martinfowler.com/bliki/PageObject.html.

Gao, Z., Fang, C., & Memon, A. M. (2015). Pushing the limits on automation in gui regression testing. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)* (pp. 565–575).

Hammoudi, M., Rothermel, G., & Stocco, A. (2016a). WATERFALL: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of 24th ACM SIGSOFT international symposium on the foundations of software engineering, FSE 2016*.

Hammoudi, M., Rothermel, G., & Tonella, P. (2016b). Why do record/replay tests of web applications break? In *Proceedings of 9th international conference on software testing, verification and validation, ICST* page (to appear). IEEE.

Kaufman, L., & Rousseeuw, P. J. (1990). *Finding groups in data: An introduction to cluster analysis. Wiley series in probability and mathematical statistics*. New York: A Wiley-Interscience publication.

Ketchen, D., Boyd, B., & Bergh, D. (2008). Research methodology in strategic management: Past accomplishments and future challenges. *Organizational Research Methods*, *11*(4), 643–658.

Leotta, M., Clerissi, D., Ricca, F., & Spadaro., C. (2013a). 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* (pp. 108–113). IEEE.

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2013b). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th working conference on reverse engineering, WCRE, 2013* (pp. 272–281). IEEE.

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2014a). Visual vs. DOM-based web locators: An empirical study. In *Proceedings of 14th international conference on web engineering (ICWE 2014), volume 8541 of LNCS* (pp. 322–340). Springer.

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2016a). Approaches and tools for automated end-to-end web testing. *Advances in Computers*, *101*, 193–237.

Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2014b). Reducing web test cases aging by means of robust XPath locators. In *Proceedings of 25th international symposium on software reliability engineering workshops (ISSREW 2014)* (pp. 449–454). IEEE.

Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2015). Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th international conference on software testing, verification and validation, ICST 2015* (pp. 1–10). IEEE.

Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2016b). ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, *28*(3), 177–204.

Lucia, A. D., Risi, M., Scanniello, G., & Tortora, G. (2009). An investigation of clustering algorithms in the identification of similar web pages. *Journal of Web Engineering*, *8*(4), 346–370.

Marchetto, A., Tonella, P., & Ricca, F. (2008). State-based testing of Ajax web applications. In *Proceedings of 1st international conference on software testing, verification and validation, ICST 2008* (pp. 121–130). IEEE.

Mesbah, A., & Prasad, M. R. (2011). Automated cross-browser compatibility testing. In *Proceedings of the 33rd international conference on software engineering, ICSE 2011* (pp. 561–570). New York, NY, USA: ACM.

Mesbah, A., van Deursen, A., & Lenselink, S. (2012a). 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.

Mesbah, A., van Deursen, A., & Roest, D. (2012b). Invariant-based automatic testing of modern web applications. *IEEE Transaction on Software Engineering (TSE)*, *38*(1), 35–53.

Nguyen, B. N., Robbins, B., Banerjee, I., & Memon, A. (2014). Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, *21*(1), 65–105.

Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. In *Proceedings of the 1st international workshop on automation of software test, AST 2006* (pp. 85–91). New York, NY, USA: ACM.

Ricca, F. (2004). Analysis, testing and re-structuring of web applications. In *Proceedings of the 20th IEEE international conference on software maintenance* (pp. 474–478).

Ricca, F., Pianta, E., Tonella, P., & Girardi, C. (2008). Improving web site understanding with keyword-based clustering. *Journal of Software Maintenance*, *20*(1), 1–29.

Ricca, F., & Tonella, P. (2001). Analysis and testing of web applications. In *Proceedings of the 23rd international conference on software engineering, ICSE '01* (pp. 25–34). Washington, DC, USA: IEEE Computer Society.

Rousseeuw, P. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, *20*(1), 53–65.

Sacramento, C., & Paiva, A. (2014). Web application model generation through reverse engineering and UI pattern inferring. In *Proceedings of the 9th international conference on the quality of information and communications technology, QUATIC 2014* (pp. 105–115). IEEE.

Sampath, S. (2012). Advances in user-session-based testing of web applications. *Advances in Computers*, *86*, 87–108.

Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics*, *6*(2), 461–464.

Selenium Wiki. (2013). Page objects. https://code.google.com/p/selenium/wiki/pageobjects.

Stocco, A., Leotta, M., Ricca, F., & Tonella, P. (2015). Why creating web page objects manually if it can be done automatically? In *Proceedings of 10th IEEE/ACM international workshop on automation of software test, AST 2015* (pp. 70–74). IEEE.

Stocco, A., Leotta, M., Ricca, F., & Tonella, P. (2016). Clustering-aided page object generation for web testing. In *Proceedings of 16th international conference on web engineering, ICWE 2016, volume 9671 of LNCS* (pp. 132–151). Springer.

Thummalapenta, S., Devaki, P., Sinha, S., Chandra, S., Gnanasundaram, S., Nagaraj, D. D., & Sathishkumar, S. (2013). Efficient and change-resilient test automation: An industrial case study. In *Proceedings of the 35th international conference on software engineering, ICSE 2013* (pp. 1002–1011). IEEE.

Tombros, A., & Ali, Z. (2005). Factors affecting web page similarity. In *Proceedings of the 27th European conference on advances in information retrieval research, ECIR 2005* (pp. 487–501). Berlin: Springer.

Tonella, P., Ricca, F., & Marchetto, A. (2014). Recent advances in web testing. *Advances in Computers*, *93*, 1–51.

Tonella, P., Ricca, F., Pianta, E., & Girardi, C. (2003a). Using keyword extraction for web site clustering. In *Proceedings of the 5th international workshop on web site evolution, WSE 2003* (pp. 41–48).

Tonella, P., Ricca, F., Pianta, E., Girardi, C., Lucca, G. A. D., & Fasolino, A. R. et al. (2003b). Evaluation methods for web application clustering. In *Proceedings of the 5th international workshop on web site evolution, WSE 2003* (pp. 33–40).

van Deursen, A. (2015a). Beyond page objects: Testing web applications with state objects. *Queue*, *13*(6), 20:20–20:37.

van Deursen, A. (2015b). Testing web applications with state objects. *Communications of ACM*, *58*(8), 36–43.

Witten, I . H., Frank, E., & Hall, M . A. (2011). *Data mining: Practical machine learning tools and techniques* (3rd ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yandrapally, R., Thummalapenta, S., Sinha, S., & Chandra, S. (2014). Robust test automation using contextual clues. In *Proceedings of the 25th international symposium on software testing and analysis, ISSTA 2014* (pp. 304–314). ACM.

**Andrea Stocco** is a Ph.D. candidate in Computer Science at the department of Computer Science, Biomedical Engineering, Robotics and Systems Engineering (DIBRIS), University of Genova, Italy. His research interests include Web Applications Testing and Empirical Software Engineering, with particular emphasis on the improvement of the testing techniques quality. His aim is to leverage existing testing tools and ease future testing approaches in order to reach the goal of software engineers testing more and better. He is the recipient of the Best Student Paper Award at the 16th International Conference on Web Engineering (ICWE 2016).

**Maurizio Leotta** is a research fellow at the University of Genova, Italy. He received his Ph.D. degree in Computer Science from the same University, in 2015, with the thesis “Automated Web Testing: Analysis and Maintenance Effort Reduction.” He is author or coauthor of more than 40 research papers published in international journals and conferences/workshops. His current research interests are in Software Engineering, with a particular focus on the following themes: Web Application Testing, Functional Testing Automation, Business Process Modeling, Empirical Software Engineering, Model-Driven Software Engineering. He is the recipient of the Best Student Paper Award at the 16th International Conference on Web Engineering (ICWE 2016).

**Filippo Ricca** is an associate professor at the University of Genova, Italy. He received his Ph.D. degree in Computer Science from the same University, in 2003, with the thesis “Analysis, Testing and Re-structuring of Web Applications.” In 2011, he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: “Analysis and Testing of Web Applications.” He is author or coauthor of more than 100 research papers published in international journals and conferences/workshops. Filippo Ricca was Program Chair of CSMR/WCRE 2014, CSMR 2013, ICPC 2011 and WSE 2008. Among the others, he served in the program committees of the following conferences: ICSM, ICST, SCAM, CSMR, WCRE and ESEM. From 1999 to 2006, he worked with the Software Engineering group at ITC-irst (now FBK-irst), Trento, Italy. During this time, he was part of the team that worked on Reverse engineering, Re-engineering and Software Testing. His current research interests include: Software modeling, Reverse engineering, Empirical studies in Software Engineering, Web applications and Software Testing. The research is mainly conducted through empirical methods such as case studies, controlled experiments and surveys.

**Paolo Tonella** is head of the Software Engineering Research Unit at Fondazione Bruno Kessler (FBK), in Trento, Italy. He received his Ph.D. degree in Software Engineering from the University of Padova, in 1999, with the thesis "Code Analysis in Support to Software Maintenance." In 2011, he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications." He is the author of "Reverse Engineering of Object Oriented Code," Springer, 2005. He participated in several industrial and EU projects on software analysis and testing. His current research interests include code analysis, web and object-oriented testing, search based test case generation.