Clustering-Aided Page Object Generation for Web Testing

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

[1] DIBRIS – Università di Genova, Italy
[2] Fondazione Bruno Kessler, Trento, Italy
andrea.stocco@dibris.unige.it, maurizio.leotta@unige.it,
filippo.ricca@unige.it, tonella@fbk.eu

Abstract. To decouple test code from web page details, web testers adopt the *Page Object* design pattern. Page objects are facade classes abstracting the internals of web pages (e.g., form fields) into high-level business functions that can be invoked by test cases (e.g., user authentication). However, writing such page objects requires substantial effort, which is paid off only later, during software evolution. In this paper we propose a clustering-based approach for the identification of meaningful abstractions that are automatically turned into Java page objects. Our clustering approach to page object identification has been integrated into our tool for automated page object generation, APOGEN. Experimental results indicate that the clustering approach provides clusters of web pages close to those manually produced by a human (with, on average, only 3 differences per web application). 75% of the code generated by APOGEN can be used as-is by web testers, breaking down the manual effort for page object creation. Moreover, a large portion (84%) of the page object methods created automatically to support assertion definition corresponds to useful behavioural abstractions.

1 Introduction

Web applications are among the most challenging software systems to test [21]. If, on one side, developing web applications is becoming easier thanks to recent frameworks (e.g., AngularJS[3]), which hide the complexity behind an expressive and readable web programming environment, and allow even newbie programmers to quickly develop highly interactive and complex applications, this comes at a price, because the programmers' inexperience with error-prone languages like Javascript and the combination of new technologies may introduce new kinds of faults, which have unpredictable effects and are hard to detect [14].

End-to-end (E2E) test automation is commonly adopted in such context, often justified by continuous integration and test driven approaches. Test scripts simulate typical end-users' interactions by delivering mouse clicks and keystrokes to the browser at a pace that would be likely infeasible to perform manually. The GUI responses are recorded and validated through assertions to check the web application for functional correctness.

[3] https://angularjs.org/

A disadvantage of test automation is the poor maintainability of the test code throughout the development process. In fact, test scripts are often highly customised and coupled with the technical details of the underlying web pages, which makes them quite difficult to read and maintain when features are added or altered in the web application under test. Web testers try to prevent these issues by using the *Page Object* design pattern, which provides a simplified interface towards the web application. All the technicalities the scripts refer to, such as low-level operations or web elements locators (e.g., an XPath to select an input field [10]), are moved to the page objects. The test code is thus separated from the implementation details, because test scripts interface with page objects methods, not directly with web page elements.

Building page objects for web applications is an activity which is performed manually [19]. Our prototype tool APOGEN [19] is the first solution able to provide a considerable degree of automation, hence reducing the effort for the creation of page objects. However, the initial version of APOGEN suffered two major limitations: (1) in the presence of highly dynamic web pages, it creates a huge number of page objects that should be conceptually regarded as a single page object; (2) it does not support the creation of getter methods in any way. A getter method retrieves textual portions of a web page that can be used to verify the behaviour of the web application (e.g., with assertions) through the results displayed to the user.

In this paper, we overcome such limitations with the following novel contributions, implemented in the new version of the tool APOGEN:

- the automatic detection of cloned and semantically similar web pages, based on clustering, to be associated with the same page object;
- the Clusters Visual Editor (CVE), a web-based interactive clusters visualiser and editor, allowing the tester to inspect and modify the clustering results;
- 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.

We have applied APOGEN to six web applications. We have studied how different clustering algorithms, working on different syntactic features (e.g., DOM), are able to group similar web pages that should be conceptually mapped onto a single page object. Results indicate that: (1) hierarchical clustering provides clusters of web pages close to those manually produced by a human, (2) 75% of the code generated by APOGEN can be used as-is by the tester, reducing the manual effort for page object creation and (3) 84% of the automatically generated getter methods correspond to methods the tester needs when creating test case assertions.

The paper is organised as follows: Section 2 provides some background on the *Page Object* design pattern, our original tool APOGEN and the challenges in the automatic creation of page objects for web applications. Section 3 describes the clustering-aided version of APOGEN and the features we evaluated in our experiment. Section 4 presents the quantitative and qualitative results of the experiment we conducted to evaluate the effectiveness of our approach. Section 5 describes the related work, while conclusions and future work are drawn in Section 6.

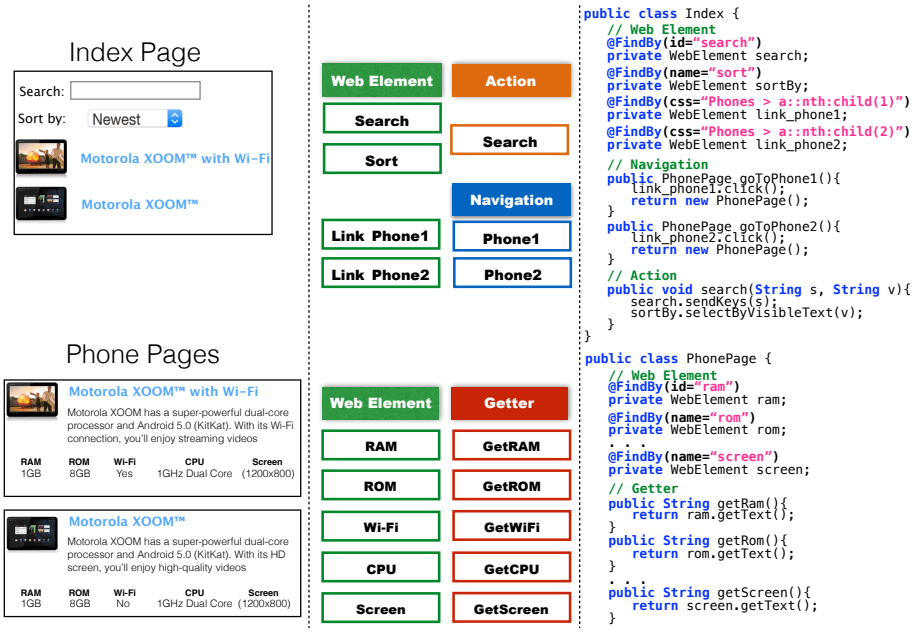


Fig. 1. AngularJS Phonecat web application **(left)**, with its abstract representation in terms of Web Elements and Functionalities (Navigations, Actions, Getters) **(center)**, and associated Java page objects **(right)**

2 Background

In web development, E2E functional testing is a widely adopted practice thanks to the increased popularity of powerful test automation tools, such as Selenium[4]. Automated tests created with these tools operate by instructing a browser to click or type on page elements. Whereas the biggest advantage is an accurate simulation of the user's behaviour, one of the major drawbacks is that such tests tend to be fragile and highly coupled with the web pages. To prevent this, testers adopt a design pattern called the *Page Object*.

A page object is a class that abstracts a web page hiding the technical details about how the test code interacts with the underlying web page behind a more readable and business-focused facade. This brings two main advantages: *(i)* tests are more readable, and *(ii)* the page access logic is centralised in one place, making test suite maintenance easier [9].

Let us consider the running example in Fig. 1, based on the AngularJS Phonecat web application[5], one of the experimental objects considered in this work. On the top-left part is the home page, displaying a list of phones (we limited the figure to two), whereas in the bottom-left part are the web pages obtained after clicking on the links in

[4] http://www.seleniumhq.org/
[5] https://github.com/angular/angular-phonecat/

```
1  public class TestAddOwnerWithoutPageObjects {
2   @Test
3   public void testAddOwner(){
4   WebDriver driver = new FirefoxDriver();
5   driver.get("http://localhost:8080/phonecat/");
6   driver.findElement(By.id("sort")).selectByVisibleText("Newest");
7   driver.findElement(By.id("search")).sendKeys("Motorola");
8   driver.findElement(By.css("Phones > a::nth:child(1)")).click();
9   AssertThat(driver.findElement(By.css("#ram")).getText(), is("1 GB"));
10  AssertThat(driver.findElement(By.css("#rom")).getText(), is("8 GB"));
11  AssertThat(driver.findElement(By.css("#wifi")).getText(), is("Yes"));
12  driver.quit();
   }
  }
```

```
1  public class TestAddOwnerWithPageObjects {
2   @Test
3   public void testAddOwner(){
4   WebDriver driver = new FirefoxDriver();
5   driver.get("http://localhost:8080/phonecat/");
6   Index indexPage = new Index(driver);
7   indexPage.search("Motorola", "Newest");
8   PhonePage phone1 = indexPage.goToPhone1();
9   AssertThat(phone1.getRam(), is("1 GB"));
10  AssertThat(phone1.getRom(), is("8 GB"));
11  AssertThat(phone1.getScreen(), is("Yes"));
12  driver.quit();
   }
  }
```

Fig. 2. 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

the home page. In the central part are the web page abstractions that the page objects should provide, showing the web elements test cases may interact with, and the functionalities the two pages offer (action, navigation, and getter functionalities). In the right part are the page object representations of such pages, written in Java. We can notice how the two web pages displaying phone details for two phones (bottom-centre part) have exactly the same abstract representation: only their textual content varies. Correspondingly, only one Java page object can represent both of them (class *PhonePage*, bottom-right).

In Fig. 2 we can see an example of how page objects improve the readability of the test cases by encapsulating functionalities. The test steps shown on the left (without page objects) are directly coupled with the web page internals, while the steps on the right (with page objects) map directly to human-readable, use case scenario's steps (e.g., open the index page, go to the first phone page, etc).

2.1 First Steps in the Automatic Generation of Page Objects

Our tool APOGEN is the first effective prototype able to generate automatically a set of page objects for a web application. APOGEN consists of three main modules: a Crawler, a Static Analyser, and a Code Generator. The input is a web application, together with the data (e.g., login credentials) required to navigate it, while the output is a set of Java files, representing a code abstraction of the web application, organised using the *Page Object* and *Page Factory*[6] design patterns, as supported by the Selenium WebDriver framework.

The Crawler retrieves a high level representation of the web application and generates a state-based model (graph) in which nodes are dynamic states of the web pages and edges are event-based transitions between nodes. In particular, we use CRAWLJAX [13], a state of the art open source Java tool for automatically crawling highly-dynamic web applications. The Static Analyser of APOGEN takes the Crawler outputs and for each dynamic state builds an abstract state object-based representation. The graph and the DOMs are parsed to collect the necessary information for building comprehensive and readable classes. The class name is generated from the URL, whereas the web elements stimulated by the crawler are inserted as WebElement instances in the page object class.

[6] https://code.google.com/p/selenium/wiki/PageFactory

For each of them, a meaningful variable name and a locator (XPath or CSS) are generated. For what concerns the methods, each transition in the graph is turned into a navigational method between pages, and every data-submitting form is parsed to acquire information about its web elements and the associated functionality. The output of the Static Analyser is an abstract representation of the web pages and their interactions. In the last step, the Code Generator transforms such model into a set of Java page objects tailored for the Selenium WebDriver framework.

2.2 Major Limitations

While experimenting with the initial version of APOGEN, we noticed two major issues that limit its applicability [19]. The *first issue* depends on the default state abstraction of the crawler, which is affected by minor UI changes. Indeed, CRAWLJAX was designed to perform an extensive exploration and when it visits the same page filled with different input data, it often creates different dynamic states, even though the page is conceptually the same. We refer to these duplicate pages as "clones" (e.g., the two phone detail pages of the running example of Fig. 1 bottom-left). As a direct consequence, when crawling a non-trivial application, the size of the extracted model can be huge, with APOGEN generating a high amount of page objects, some of which are conceptually clones of each other. The *second issue* is the lack of assistance in the automatic creation of getter methods, necessary when defining test case assertions.

3 Clustering-Aided Page Object Generation

In order to address the limitations discussed in Section 2.2, we applied clustering as a post-processing technique (after the crawling phase) for a triple aim: (1) grouping pages related to the same functionality, e.g., all the pages concerning user authentication; (2) grouping clone pages, i.e., different versions of the same page, only differing by minor, dynamic details, as the textual content (see, for instance, those in Fig. 1 bottom-left); (3) exploiting the differences between clones to retrieve information useful for getter methods.

We extended APOGEN with an additional module, the Clusterer, which runs a clustering algorithm over the Crawler output. We opted for three popular clustering algorithms from the literature: K-means++ [1], Hierarchical Agglomerative [7], and K-medoids [6]. For the first two we used the implementations available from the , whereas K-medoids was not available, thus we implemented it from scratch. The Clusterer is able to automatically calculate different kinds of syntactic feature matrixes from the web pages (e.g., tag frequency), that are then used by the clustering algorithms to compute the similarities.

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 supports the tester with the Clusters Visual Editor (CVE), an interactive cluster visualisation and editor facility, allowing testers to inspect and modify the clustering results, as shown in Fig. 3.

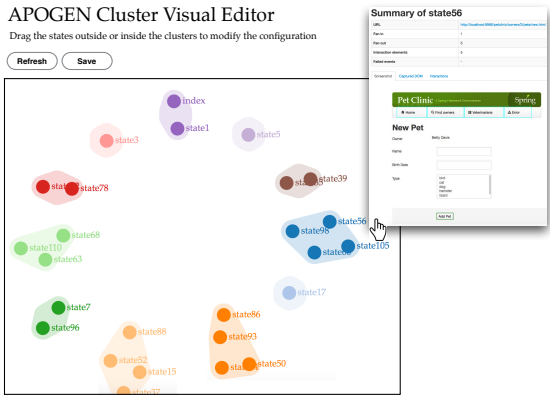


Fig. 3. Clusters Visual Editor (CVE), a web-based tool developed using the D3 library

3.1 Feature Extraction and Matrix Creation

Clustering algorithms rely on the concept of (dis)similarity between web pages. There exist a number of works studying the factors affecting web page similarity [20] in which authors observed that structural features are related with semantic properties of the data and provide meaningful means of comparison between web pages. The Clusterer considers the following features: Tag Frequency, Word Frequency, URL and DOM.

Tag Frequency (TF) measures the frequency at which tags occur in a web page. The general intuition is that such frequency provides an indication of the general layout and structure, and may be effective for detecting structurally similar web pages. TF for a web application W is calculated as follows: (1) extract a Tag List *TL* of the tags from all the pages in W, (2) for each page $p \in W$ and for each tag $t \in TL$; calculate TF(t, p), as the normalised frequency of occurrence of tag t in page p (after min-max normalisation); and (3) create the output matrix $TL \times W$ of the normalised TF values.

Word Frequency. The textual content of a page captures information that may be salient for such web page. We assume that two web pages sharing similar textual content shall have some degree of topical relatedness and thus should be grouped together. The Clusterer can calculate the word frequency in two ways, considering: (1) only words within the tag BODY (**WF1**); (2) only words within the tags TITLE, H1–H6, TABLE, LI–UL–OL (**WF2**). With the former we take into account the full main content of the page, whereas with the latter we follow the intuition that these tags may contain a succinct representation of the page semantic content [20]. WF1 and WF2 for a web application W are calculated as follows: (1) extract the Word List *WL* including the words from all pages in W; (2) remove stop-words[7] from *WL*; (3) for each page $p \in W$ and for each word $w \in WL$, calculate WF1(w, p) and WF2(w, p), as the normalised frequency of occurrence of word w respectively found in the page p within tag BODY or tags TITLE, H1–H6, TABLE, LI–UL–OL (after min-max normalisation); and (4) create two output matrixes $WL \times W$, in our study associated respectively with WF1 and WF2.

[7] Retrieved from http://www.lextek.com/manuals/onix/stopwords1.html

URL (Uniform Resource Locator) may also be a good indicator of similarity between web pages [18]. Two pages sharing a part of the URL are likely to be semantically close. Although this is not always true (e.g., in Ajax single page applications), there are works showing the effectiveness of URLs for structural clustering [2]. Parameters are stripped before computing the Levenshtein distance [11], to reduce their potentially disruptive effects. Given W as the set of web pages of the web application, the output is a matrix $W \times W$ (later indicated as URL-Lev) of values ranging in $[0..1]$, where an entry equal to 0 indicates two totally dissimilar URLs, while 1 indicates a perfectly matching pair of URLs.

DOM (Document Object Model) is a dynamic hierarchical structure representing the user interface elements of a HTML page. We assume that two web pages sharing similarities between their DOMs are likely to represent pages having analogous functionalities and that they should be grouped in the same cluster. The DOM can be treated either (1) as a tree-like structure, or (2) as a string. Given W, the set of web pages of the web application, two distance matrixes $W \times W$ can be calculated, in the first case using the Robust Tree Edit Distance (RTED) algorithm [15], whereas in the second case using the Levenshtein distance between the string representation of the DOM (after word/text removal, to preserve only the structure). In our study, we refer to these two matrixes as DOM-RTED and DOM-Lev, respectively.

Summary. To wrap up, the Clusterer extracts raw features (TF, WF1, WF2) from the web pages, or features representing distance measures (URL-Lev, DOM-RTED, DOM-Lev), to be given in input to the clustering algorithms. It is important to highlight that K-means++ needs to compute the mean feature vector (centroid) from the set of feature vectors in the same cluster. This is not possible in the case of URL-Lev, DOM-RTED, DOM-Lev, since feature vectors represent distance measures.

3.2 Potential Getter Methods Detection

In the Static Analyser of APOGEN we have integrated a differencing engine, based on XMLUnit[8], that takes into account the results of clustering and supports the automatic creation of getter methods based on the DOM differences between web pages within the same cluster (e.g., clones). We believe that such intra-cluster differences point to dynamic portions of web pages, on top of which a tester might be interested in creating an assertion. For instance, in Fig. 1, getter methods are created for the phone details fields that vary across web pages in the same cluster. In order to minimise the number of false positives (i.e., irrelevant differences), the differencing engine ignores case sensitivity, white spaces, attribute value order and white-spaces between values, keeping the focus only on textual node elements which were modified or added.

3.3 From Web Page Clusters to Page Objects

We use hard clustering, i.e., each web page is a member of exactly one cluster, because we want to map each cluster into a page object and each web page must be associated with a unique page object. Let us consider Fig. 4, showing a cluster of web

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

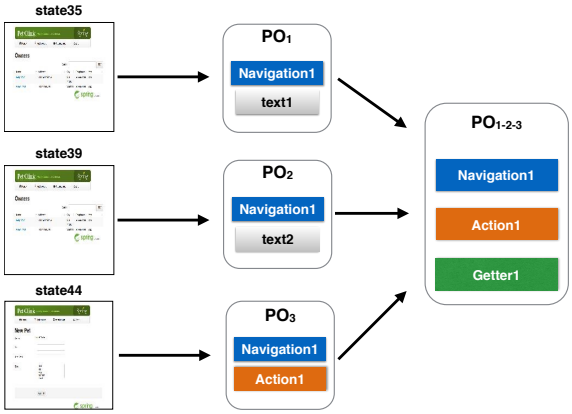


Fig. 4. Pictorial view of APOGEN page object merging strategy, applied to web pages of PetClinic

pages $C = \{state35, state39, state44\}$ from the PetClinic web application, one of the experimental objects considered in our study. $State35$ and $state39$ contain the same navigation web element (e.g., a link that can be clicked) and two different textual elements, while $state44$ contains the same navigation web element and an action (e.g., a text field that can be filled in).

Without considering the results of clustering, APOGEN would generate three page objects PO_1, PO_2, PO_3 for $state35$, $state39$ and $state44$. The same navigation method *navigation1* is replicated three times in all page objects; no getter methods are available for *text1* and *text2*, and the third page object has one method, to perform *action1*. For the web tester it would be quite difficult to decide when to use PO_1, PO_2 or PO_3. Moreover, manual corrections and adjustments to the automatically generated page objects should be repeated three times.

By using clustering, instead, APOGEN generates a single page object, corresponding to the entire cluster. The navigation method *navigation1* appears only once in such page object. The action method *action1* is also included. For what concerns getter methods, only textual elements belonging to structural clones and differing across such clones are turned into getters. In our example, $state35$ is a clone of $state39$ (i.e., their DOMs are structurally equivalent) and *text1* differs from *text2*. Hence, a getter method to retrieve the value of the dynamically changing textual element, namely *getter1*, is generated. The result is a merged page object $PO_{1-2-3} = \{navigation1, action1, getter1\}$, exposing all functionalities of $state35$, $state39$ and $state44$ relevant for web test creation.

4 Empirical Evaluation

This section presents the design, evaluation objects, research questions, metrics, procedure, results, discussion, and threats to validity of the empirical study conducted to evaluate the effectiveness of clustering in grouping similar web pages conceptually associated with the same page object and the quality of the page objects generated by APOGEN. We follow the guidelines by Wohlin *et al.* [24] on designing and reporting

empirical studies in software engineering. Our tool, experimental data and artefacts are available at: http://sepl.dibris.unige.it/APOGEN.php.

4.1 Experimental Objects

We selected six real-world web applications covering different application domains. Their properties are shown in Table 1. *PetClinic* is a veterinary clinic information system which allows veterinarians to manage data about pets and their owners. It has been developed using Java Spring Framework and makes use of technologies as JavaBeans, MVC presentation layer and Hibernate. *AddressBook* is a PHP/MySQL-based address and phone book, contact manager, and organiser. *PPMA* is a web based password manager. *Claroline* is a collaborative learning environment which allows teachers or education institutions to administer courses online. The software provides group management, forums, document repositories, calendar. *Phonecat* is a web-based phone catalog using the AngularJS framework. *FluxBB* is a fast and light PHP forum application.

Id	Name	Source	LOC
WA1	ADDRESSBOOK (8.2.5)	http://sourceforge.net/projects/php-addressbook	30.1K (PHP) 1.1K (JS)
WA2	PHONECAT	https://github.com/angular/angular-phonecat/	0.4K (JS)
WA3	CLAROLINE (1.11.5)	http://sourceforge.net/projects/claroline/	285K (PHP) 36K (JS)
WA4	FLUXBB (1.5.8)	http://fluxbb.org/	21K (PHP)
WA5	PETCLINIC	https://github.com/spring-projects/spring-petclinic	6.1K (JAVA) 432 (JSP)
WA6	PPMA (0.2)	http://sourceforge.net/projects/ppma/	9K (JS) 3.5K (PHP)

Table 1. Experimental Objects

4.2 Research Questions

We conducted our empirical study to address the following research questions:

RQ1 (effectiveness): *What clustering algorithm provides the best result and how do different algorithms compare with each other?*
RQ2 (reduction): *What is the maximum reduction achievable in the number of generated Page Objects when using clustering with* APOGEN*?*
RQ3 (quality): *How successful is the clustering-aided* APOGEN *in generating high quality Page Objects, i.e., Page Objects similar to those a developer would write?*

4.3 Metrics

A human expert has manually defined the Gold Standard for clusters and page objects, i.e., the ideal grouping of web pages into clusters (Clusters Gold Standard, *C-GS*) and the ideal page object classes associated with the clusters (Page Objects Gold Standard, *PO-GS*).

Both Gold Standards require the intervention of a human for their construction. To limit any bias or subjectivity, we asked an external third party (hereafter referred as EXP) to define the Gold Standards. EXP is a programmer with strong professional experience in developing and testing web applications using page objects. EXP has substantial industrial experience and was not involved in the development of APOGEN.

The metric we used to answer RQ1 is the Partition Edit Distance (PED), which in our case measures the minimum number of web pages that must be moved between clusters to make two web page partitions (i.e., the output of clustering and *C-GS*) the same. We chose PED because it provides a direct measure of the tester's manual actions necessary to produce the target clustering (i.e., *C-GS*) starting from the output produced by any of the considered clustering algorithms. In fact, a high value of PED means that many web pages need to be reassigned, whereas a low value of PED means that the clusters are close to *C-GS* (with few moves required). In the following, we introduce the concepts behind PED and how to calculate it. Let us assume to have a set of 6 web pages $W = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ and that we want $k = 4$ separate clusters. Suppose we have the following *C-GS*:

$$gs_0 \rightarrow \{p_1\} \qquad gs_1 \rightarrow \{p_3, p_4\} \qquad gs_2 \rightarrow \{p_2\} \qquad gs_3 \rightarrow \{p_5, p_6\}$$

whereas a hypothetical clustering algorithm C gives:

$$c_0 \rightarrow \{p_1, p_2\} \qquad c_1 \rightarrow \{p_3\} \qquad c_2 \rightarrow \{p_4, p_6\} \qquad c_3 \rightarrow \{p_5\}$$

We first compare each cluster gs_i from *C-GS* with each cluster c_j from C using the Jaccard similarity coefficient:

$$J(c_i, gs_j) = \frac{|c_i \cap gs_j|}{|c_i \cup gs_j|}$$

where 0 indicates no element in common; 1 total agreement. For instance, the Jaccard similarity between c_0 and gs_0 is $J(c_0, gs_0) = |\{p_1\}| \,/\, |\{p_1, p_2\}| = 0.5$.

The Jaccard similarity matrix for all possible pairs $\langle c_i, gs_j \rangle$ is shown in Table 2. Given the similarity matrix between two partitions, PED can be obtained by solving the following linear assignment problem:

Given two partitions C and C-GS, find the partial bijection between the elements of C and C-GS (i.e., partial, unique assignment of elements from C to elements of C-GS) that maximises the total similarity between paired elements.

In our example, a linear assignment algorithm (we used the Hungarian Method [8]) would produce the following best pairs BP (highlighted in bold in Table 2):

$$BP = \{\langle c_0, gs_0 \rangle, \langle c_1, gs_1 \rangle, \langle c_2, gs_3 \rangle, \langle c_3, gs_2 \rangle\}$$

Given BP, the asymmetric set difference cardinality between each pair gives us the number of pages that must be moved to unify the two partitions. Formally, PED is computed as follows:

$$PED(C, \textit{C-GS}) = \sum_{\langle c_i, gs_j \rangle \in BP} |c_i \setminus gs_j| + |\text{unassigned}(C, BP)|$$

Table 2. Jaccard similarity matrix. The best pairs that a linear assignment algorithm would produce are highlighted in bold

C-GS / *C*	gs_0	gs_1	gs_2	gs_3
c_0	**0.50**	0.00	0.50	0.00
c_1	0.00	**0.50**	0.00	0.00
c_2	0.00	0.50	0.00	**0.50**
c_3	0.00	0.00	**0.00**	0.50

If there are unassigned clusters in C, due to the size of C being different from that of *C-GS*, the total number of pages contained in such unassigned clusters are also added in the formula given above. Although the asymmetric set difference operator ($\setminus$) has been used in the formula to compute PED, it can be easily shown that PED is indeed symmetric: $PED(C, \textit{C-GS}) = PED(\textit{C-GS}, C)$. In our example: $PED(C, \textit{C-GS})$=$|c_0 \setminus gs_0|$+$|c_1 \setminus gs_1|$+$|c_2 \setminus gs_3|$+$|c_3 \setminus gs_2| = 1+0+1+1 = 3$. Thus, a tester would need to move 3 web pages from the clusters of C to obtain *C-GS*: p_2 from c_0 to c_2, p_4 from c_2 to c_1 and p_6 from c_2 to c_3. This gives a rough estimate of the effort required for the manual correction of the clustering output.

To answer RQ3, for each page object of *PO-GS*, we manually inspected all methods: *(i)* classifying the kind of functionality as *navigational*, *action* or *getter*; *(ii)* determining whether the method has a semantically equivalent counterpart in the automatic page objects (we tag such methods as *Equivalent*); *(iii)* determining whether the method has a counterpart in the automatic page objects that needs minor modifications (we tag such methods as *To Modify*); *(iv)* determining any missing methods (we tag such methods as *Missing*). Further, we are interested in determining if APOGEN leads to the generation of extra methods, e.g., methods not contained in *PO-GS*. The number of *Equivalent*, *To Modify*, *Missing* and *Extra* methods allows us to estimate the possibility to use the code produced by APOGEN as-is, and the effort needed to manually correct the methods to be modified, or to be added/deleted.

4.4 Experimental Procedure

To answer RQ1, we proceeded as follows:
(i) We ran CRAWLJAX over each web application to infer its model. We fed the crawler with the data necessary to explore each application, such as login credentials. EXP manually inspected the crawling outcomes and created a *C-GS* for each web application.
(ii) Clustering algorithms need the specification of the number of clusters k as input. Such a value can be either provided manually or can be obtained by automated methods, such as the Silhouette method [17]. We have compared the optimal number of clusters, k_{opt} available from the *C-GS*, with the number produced by the Silhouette method and the two are very close to each other in all experimental objects (with median difference 3, maximum difference 5 and minimum 0). Hence, we ran APOGEN on each web application with each *(algorithm, feature)* pair searching for exactly k_{opt} clusters. We compared the clusters obtained from APOGEN with *C-GS*.

Table 3. RQ1: Comparison between automatic clusters and gold standard (PED)

Clustering Algo (Feature)	WA1	WA2	WA3	WA4	WA5	WA6	Tot
Hierarchical (DOM-RTED)	4	0	6	7	0	1	18
Hierarchical (URL-Lev)	1	3	4	8	2	6	24
K-means++ (TF)	2	5	8	8	0	3	26
Hierarchical (DOM-Lev)	4	0	9	5	7	3	28
K-means++ (WF2)	4	4	8	7	6	3	32
K-means++ (WF1)	2	3	9	9	7	3	33
K-medoids (TF)	5	6	11	10	6	4	42
K-medoids (DOM-RTED)	6	5	12	9	6	4	42
K-medoids (WF1)	5	2	11	10	11	3	42
K-medoids (WF2)	5	3	12	10	10	4	44
K-medoids (DOM-Lev)	5	5	14	10	7	5	46
K-medoids (URL-Lev)	5	9	12	10	7	6	49

(iii) We calculated PED for all *(algorithm, feature)* pairs, in order to assess: *(1)* what is the overall best *(algorithm, feature)* pair, and *(2)* how far the best algorithm is from the *C-GS*.

To answer RQ2, we ran APOGEN on each web application twice, both enabling and disabling the Clusterer, and we counted the number of generated page objects.

To answer RQ3, we proceeded as follows. For each web application:

(i) EXP manually created *PO-GS* from the optimal clusters in *C-GS*;

(ii) we inspected and compared *PO-GS* with the page objects automatically generated by APOGEN. In detail, for each page object, we manually classified all methods as *navigational*, *action* or *getter*, and as *Equivalent*, *To Modify*, *Missing*, or *Extra*.

4.5 Results

Table 3 reports the values of PED for the admissible combinations of *algorithm* (Hierarchical, K-means++, K-medoids) and *feature* (TF, WF1, WF2, DOM-RTED, DOM-Lev, URL-Lev).

Globally, the best algorithm is *Hierarchical*, which occupies the first, second and fourth positions of the rank. It scores 18 (DOM-RTED), 24 (URL-Lev), and 28 (DOM-Lev). *K-means++* has variable performance: it is ranked third with a value of 26 (TF) but also fifth with a value of 32 (WF2) and sixth with a value of 33 (WF1). *K-medoids* stabilises in the worse positions of the rank, independently from the input data matrix. Its values range between 42–49.

RQ1 (effectiveness): considering all the applications, Hierarchical clustering resulted to be the optimal choice, being in the first, second and fourth position of the PED rank and undergoing little oscillations in its performance across different web applications. In our experiment, the effort to align its clusters with *C-GS* consists on average of 2–4 page moves per application. K-means++ also proved to be a good choice when used with the data matrix representing the tag frequencies. Indeed, its performance is aligned with that of the Hierarchical algorithm on WA5 (*PetClinic*). To summarise:

Table 4. Reduction of generated page objects when using clustering

Application	No Clustering	Clustering	% Reduction
PETCLINIC	26	10	61.54
ADDRESSBOOK	20	10	50.00
PPMA	16	8	50.00
CLAROLINE	63	15	76.19
PHONECAT	21	2	90.48
FLUXBB	55	13	76.19
Total	**201**	**58**	**67.43**

Hierarchical clustering applied to the DOM tree distance matrix has the best performance, producing clusters of web pages very close to those manually defined in the C-GS.

RQ2 (reduction) Table 4 shows data about the reduction in the number of generated page objects when using clustering in APOGEN. The first column shows the experimental objects, whereas in the second column are the number of page objects generated by APOGEN without considering clustering, which is equal to the number of dynamic states retrieved by the crawler. The third column displays the number of clusters defined in the *C-GS*, which is equal to the number of page objects produced by APOGEN, since k_{opt} was provided as input to the clustering algorithm (the value of k obtained from Silhouette would be only slightly different). To summarise, in our experiment:

When using clustering, the reduction in the number of generated page objects ranges between 50–90% (average 67%).

Beyond the mere quantitative data, the substantial reduction achieved by clustering gives an idea of the reduction in page object maintenance that is expected to occur during software and testware evolution.

RQ3 (quality): Table 5 shows the number of methods (navigational, action or getter) that we tagged as Equivalent (Eq), that need to be modified (TM), missing (Mis) and extra (Extra) w.r.t. *PO-GS*. The first column shows the experimental objects. The second, third and fourth macro-columns show the cardinality of navigational, action and getter methods generated by APOGEN (macro-columns are split into Eq, TM, Mis and Extra). The fifth macro-column shows the amount of methods contained in *PO-GS* (i.e., the key functionalities a web tester would put as methods in the page objects). The sixth macro-column reports the sum over all kinds of methods.

Based on the data, we can notice that on average about 75% of the methods are equivalent, 7% are to modify, and 18% are missing. Looking at results by type, for

Table 5. RQ2: Comparison between automatic and manual page objects

Application	# Navigational				# Action				# Getter				Total (GS)	# Methods			
	Eq	TM	Mis	Extra	Eq	TM	Mis	Extra	Eq	TM	Mis	Extra	Eq+TM+Mis	Eq	TM	Mis	Extra
PETCLINIC	9	0	0	0	6	0	3	0	8	0	3	0	29	23	0	6	0
ADDRESSBOOK	10	0	3	0	4	3	4	0	6	0	1	23	31	20	3	8	23
PPMA	4	0	8	0	4	1	1	0	1	0	2	11	21	9	1	11	11
CLAROLINE	15	0	2	0	7	4	0	0	6	0	3	15	37	28	4	5	15
PHONECAT	1	0	0	0	0	0	1	0	27	0	0	4	29	28	0	1	4
FLUXBB	14	0	0	0	1	4	0	0	6	0	3	0	32	26	4	2	0
Total	**53**	**0**	**13**	**0**	**22**	**12**	**9**	**0**	**59**	**0**	**11**	**53**	**179**	**134**	**12**	**33**	**53**

what concerns navigational methods, most are usable directly, as produced by APOGEN, (about 80%), none is to be modified and 16% are missing. About the actions, we can notice that roughly 51% are equivalent, 28% need to be manually modified and 21% are missing. For the getter methods, which are generated on top of intra-clusters differences (see Section 3), we can notice that 84% are equivalent, none is to be modified and 16% are missing. Concerning the methods tagged as extra, i.e., methods that are not explicitly present in the *C-GS*, we have a total of 53 methods, all falling in the getters category.

> *About 75% of the generated methods are equivalent* to those in the manual gold standard, *7% need to be manually refined*, *18% are missing*.

4.6 Qualitative Analysis

For space reasons, we focus the qualitative analysis on the main page of the FluxBB web application (Fig. 5 top). This example is representative because the page object automatically generated by APOGEN (Fig. 5 bottom) for such page includes all the cases (Equivalent, ToModify, Missing, Extra) described in Section 4.3.

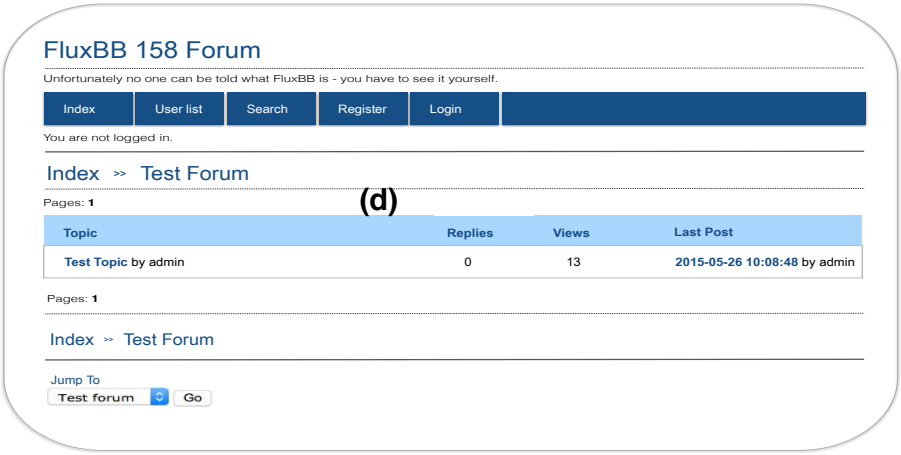
The navigational method goToUserlist() is an example of Equivalent method **(a)**. In fact, it replicates exactly what the tester would do while performing a navigation from the current page toward the user list page: click on the menu item and change the state by instantiating a new proper page object (Userlist) and by passing it the WebDriver instance.

The action method qjump(), instead, is an example of method ToModify **(b)**. First, the name retrieved from the form attributes is not very expressive (the label "Jump To" would have been a better choice for the name, in this case). Second, the return parameter with the target object is missing. The reason is that static analysis misses the next dynamic state. The returned page object should be a TestForum page object, whereas if an incorrect parameter is passed as args0, the page object should manage the error. The second getter method is an example of Extra method **(c)**, because it refers to a web element within the page representing the same information targeted by the first getter (see the two SPAN "Pages: 1" fields in Fig. 5). In this case, the tester may keep only one of the two getters (e.g., the first one), deleting the second. On the other hand, the tester may check for any inconsistency between the two values, so having two separate methods might be regarded as useful. We decided to leave this choice to the tester. In fact, we believe that the generation of extra getter methods does not impact so negatively the readability of the page objects. On the other hand, no clones in the cluster exposed any differences, while some variability might occur, for instance, in the Replies field **(d)**. Thus, we marked such field as a Missing getter.

4.7 Discussion

Hierarchical agglomerative clustering offered stable performance across all web applications, possibly because of the single linkage (min) method, which aggregates clusters when their minimum similarity is the highest among all possible pairs of clusters being aggregated, hence leading to aggregation choices that we think are close to those

Fig. 5. The main page of FluxBB web application (top), and a portion of the page object generated by APOGEN (bottom)




```
public class ViewForum {

   @FindBy(name="userlist")
   private WebElement userlist;

   @FindBy(name="select")
   private WebElement select_id;
   …

   public Userlist goToUserlist(){
      userlist.click();
      return new Userlist(driver);
   }
```

(a)

(b)

```
   public void qjump(String args0){
      select_id.sendKeys(args0);
      input_go.click();
   }
   public String get_span_1(){
      return span_1.getText();
   }
   public String get_span_2(){
      return span_2.getText();
   }
}
```

(c)

made by a human when defining the Gold Standard. Content-based features (WF1 and WF2) seem to capture a significant amount of information related to the semantic content and sometimes improve the effectiveness of clustering, though they are not the best choice, according to our study. The features calculated over the DOM (RTED and Lev) work best with Hierarchical clustering, while they perform quite poorly with K-medoids. Thus, we can conclude that structural properties have the best performance, in particular DOM-RTED, TF and DOM-Lev, especially if coupled with Hierarchical clustering.

Concerning the results for RQ3, we can notice that there were no methods to be modified in the navigational and getter categories. This is mainly due to the code transformation phase, in which the mapping is 1-1 for these kinds of methods (see Section 2.1). On the other hand, 28% of action methods needed a manual refinement, usually to add some complex interaction pattern (e.g., a mandatory click on a checkbox before triggering a form submission). These patterns cannot yet be captured statically by the current version of APOGEN, which is not able to automatically add the missing statements. Although this is an interesting challenge for future work, it represents a minor issue, since the majority of the actions are correctly generated and ready for use by developers.

For similar reasons (static analysis of the DOM and 1-1 model to code transformation), we have a complete absence of Extra action and navigational methods. Concern-

ing the getters, instead, there are 53 extra getter methods. This result is not surprising, since the problem of identifying the getters that are potentially relevant for the construction of assertions is a difficult problem. We implemented a heuristic, which suffers from the problem of false positives. On the other hand, the use of clustering and intra-cluster differencing captured most of the web page dynamic sections, producing a high proportion of the getter methods in the gold standard (84%). As noticed before, the generation of additional getter methods is not expected to impact so negatively the activity of the tester. It should also be noticed that *Phonecat* and *PetClinic* have no extra getters and that there are on average 9 extra getters per web application over all the page objects, an amount which we think is acceptable for web testers.

4.8 Threats to Validity

For what concerns the *external validity* and the generalisation of results, we selected real size web applications spanning different domains, which makes the context realistic, even though studies with other applications are necessary to corroborate our findings.

About the *internal validity*, a possible issue is represented by the manually created gold standards, both for the clusters and the page objects. It should be noticed that we must necessarily rely on a *manual* gold standard for evaluating the output of APOGEN because no automated method can provide us with the ideal clusters and page objects. We minimised this threat by having the gold standards produced by a third subject independent from us and from APOGEN.

For what concerns the *construct validity*, for the evaluation metrics used to answer RQ3 we did not adopt Precision-Recall measures, because they rely on a boolean classification of the output (either correct or incorrect), while in the case of page object methods labeled as To Modify or Extra it is not completely appropriate to deem them as incorrect (neither as correct). We preferred to present the data as they are, split into four categories (Equivalent, To Modify, Missing, Extra), and to discuss them in terms of usability, benefits and expected manual actions required for the refinement of the automatically generated page objects.

5 Related Work

The automated creation of page objects for E2E web testing is a completely new research field, so, to the best of our knowledge, there are no strictly related previous works. However, there are related works that deal with applications of clustering techniques to support web testing and engineering.

State Objects. *Van Deursen* [23] describes a state-based generalisation of page objects. From a testing viewpoint, moving a page object to the state level makes the design of test scenarios easier. Besides the mere terminological difference, the work by van Deursen describes a series of guidelines and good practices (e.g., let each state correspond to a state object) that we share and tried to incorporate in the development of APOGEN, since our ultimate goal is the automatic generation of meaningful page/state objects.

Clustering. *Crescenzi et al.* [5] present an algorithm to cluster web pages, exploiting the structural similarities of the DOMs. In this paper, we studied several structural similarity

measures beyond the DOM, with the aim of supporting the clustering of web pages from a testing perspective. *Tonella et al.* [22] provide two methods for web clustering evaluation, the gold standard and a task oriented approach, together with guidelines and examples for their implementation. In our paper, we compared the results of web page clustering against a gold standard, in order to ensure its meaningfulness from the web testing viewpoint. In another work, *Ricca et al.* [16] utilise keyword-based clustering to improve the comprehension of web applications. In our paper, we did not limit ourselves to content-based metrics. Actually, structural properties (e.g., DOM or TF) showed to be more effective.

Crawling/Differencing. *Choudhary et al.* [4] present a dynamic technique based on differential testing to automatically detect cross-browser issues (XBI) and assist developers in their diagnosis. The approach operates on single web pages and focuses on visual analysis, whereas we perform intra-cluster DOM-differencing. *Mesbah et al.* [12] analyses an entire web application, using dynamic crawling, also for the retrieval of XBIs. 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. *Choudhary et al.* [3] combined and extended the two above-mentioned approaches for XBI detection in the tool CROSSCHECK. Even though this paper shares some methods with us, 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, and we target a completely different goal, automated page object construction.

6 Conclusions and Future Work

We presented a novel approach, based on web page clustering, to automatically generate page objects for web testing. The tool APOGEN, which implements the approach, has been applied to six existing web applications. Experimental results indicate that our clustering approach is effective to group semantically related web pages. Furthermore, the page objects obtained from the output of clustering are very similar to the page objects that a developer would create manually. Indeed, 75% of the code generated by APOGEN can be used as-is by a tester, breaking down the manual effort for page object creation. Moreover, a large part (84%) of the page object methods created to support assertion definition corresponds to meaningful and useful behavioural abstractions.

In our future work, we plan to improve the heuristics used to create the getter methods, which cannot be applied to single page clusters. We will investigate a complementary approach, for input data generation, capable of exposing the variable part of multiple as well as single pages in each cluster. We will also study visual mechanisms, based on image processing, to retrieve dynamic page portions [3].

References

1. D. Arthur and S. Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07,

pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

2. L. Blanco, N. Dalvi, and A. Machanavajjhala. Highly efficient algorithms for structural clustering of large websites. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 437–446, New York, NY, USA, 2011. ACM.
3. S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 171–180, Washington, DC, USA, 2012. IEEE Computer Society.
4. S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *ICSM*, pages 1–10. IEEE Computer Society, 2010.
5. V. Crescenzi, P. Merialdo, and P. Missier. Clustering web pages based on their structure. *Data Knowledge Engineering*, 54(3):279–299, Sept. 2005.
6. L. Kaufman and P. Rousseeuw. Clustering by means of medoids. *Statistical Data Analysis Based on the L1-Norm and Related Methods*, pages North–Holland, 1987.
7. L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Wiley series in probability and mathematical statistics. Wiley, New York, 1990. A Wiley-Interscience publication.
8. H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.
9. M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proc. of 20th Working Conference on Reverse Engineering*, WCRE 2013, pages 272–281. IEEE, 2013.
10. M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust XPath locators. In *Proc. of 25th International Symposium on Software Reliability Engineering Workshops (ISSREW 2014)*, pages 449–454. IEEE, 2014.
11. V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
12. A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 561–570, New York, NY, USA, 2011. ACM.
13. A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.
14. F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side javascript bugs. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE Computer Society, 2013.
15. M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015.
16. F. Ricca, E. Pianta, P. Tonella, and C. Girardi. Improving web site understanding with keyword-based clustering. *Journal of Software Maintenance*, 20(1):1–29, 2008.
17. P. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Compututational and Applied Mathematics*, 20(1):53–65, Nov. 1987.
18. S. Sampath. Advances in user-session-based testing of web applications. *Advances in Computers*, 86:87–108, 2012.
19. A. Stocco, M. Leotta, F. Ricca, and P. Tonella. 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, pages 70–74. IEEE/ACM, 2015.

20. A. Tombros and Z. Ali. Factors affecting web page similarity. In *Proceedings of the 27th European Conference on Advances in Information Retrieval Research*, ECIR'05, pages 487–501, Berlin, Heidelberg, 2005. Springer-Verlag.
21. P. Tonella, F. Ricca, and A. Marchetto. Recent advances in web testing. *Advances in Computers*, 93:1–51, 2014.
22. P. Tonella, F. Ricca, E. Pianta, C. Girardi, G. D. Lucca, A. R. Fasolino, and P. Tramontana. Evaluation methods for web application clustering. *15th IEEE International Symposium on Web Systems Evolution (WSE)*, 0:33, 2003.
23. A. van Deursen. Testing web applications with state objects. *Commun. ACM*, 58(8):36–43, July 2015.
24. C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.