Information and Software Technology 141 (2022) 106735

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Engineering Web Augmentation software: A development method for enabling end-user maintenance☆

Diego Firmenich [a,*], Sergio Firmenich [b,c], Gustavo Rossi [b,c], Manuel Wimmer [d], Irene Garrigós [e], César González-Mora [e]

[a] DIT, Facultad de Ingeniería, Universidad Nacional de la Patagonia San Juan Bosco, Argentina
[b] Lifia, Facultad de Informatica, UNLP, Argentina
[c] CONICET, Argentina
[d] CDL-MINT, Johannes Kepler University Linz, Austria
[e] Universidad de Alicante, Spain

ARTICLE INFO

Keywords:
Web Augmentation
End-user programming
Web adaptation
End-user driven maintenance

ABSTRACT

Nowadays, end-users are able to adapt Web applications when some of their requirements have not been taken into account by developers. One possible way to do adaptations is by using Web Augmentation techniques. Web Augmentation allows end-users to modify the Web sites' user interfaces once these are loaded on the client-side, i.e., in the browser. They achieve these adaptations by developing and/or installing Web browser plugins ("augmenters") that modify the user interface with new functionalities. This particular kind of software artifacts requires special attention regarding maintenance as–in most cases–they depend on third-party resources, such as HTML pages. When these resources are upgraded, unexpected results during the augmentation process may occur. Many communities have arisen around Web Augmentation, and today there are large repositories where developers share their augmenters; end-users may give feedback about existing augmentations and even ask for new ones. Maintenance is a key phase in the augmenters' life-cycle, and currently, this task falls (as usual) on the developers. In this paper, we present a participatory approach for allowing end-users without programming skills to participate in the augmenters' maintenance phase. In order to allow this, we also provide support for the development phase to bootstrap a first version of the augmenter and to reduce the load on developers in both phases, development and maintenance. We present an analysis of more than eight thousand augmenters, which helped us devise the approach. Finally, we present an experiment with 48 participants to validate our approach.

1. Introduction

The "Augmented Web" [1] rests on software artifacts (we call them augmenters in this paper) that are commonly implemented as Web browser extensions. Augmenters are capable of adapting the Web sites' user interfaces (UI). This adaptation is performed when a target Web page is loaded on the client-side. Augmenters can remove or change almost any original UI component as well as add new ones. Adaptations are achieved by manipulating the DOMs (Document Object Models) of existing Web sites; the alteration of the DOM produces the augmentation effect. There exist large augmenter repositories where communities of actual Web site users share this kind of software to adapt their own web experience. At the same time, we are witnessing a trend towards the convergence of end-user programming techniques (EUP) [2] with the final aim of letting users without advanced programming skills specify how to adapt their own interaction experiences when surfacing the Web [1,3–5]. These approaches and accompanying tools clearly show that several kinds of augmenters may be created without programming skills, i.e., by using EUP tools. However, to the best of our knowledge there is not sufficient evidence to affirm that every available augmenter in public repositories may be programmed with existing EUP tools. Nevertheless, it actually serves as clear evidence that by visually manipulating Web pages' UI elements (which are DOM elements) plus some configuration, it is possible to build several kinds of augmenters, or at least, that this visual manipulation and rearrangement of UI elements is enough to prototype how the augmentation artifact must behave [6]. A broad range of both functional and non-functional requirements can be satisfied through the design and development of

☆ Funding information: PICT 2015–2050, TIN2016-78103-C2-2-R and ACIF/2019/044.
* Corresponding author.
E-mail address: dfirmenich@tw.unp.edu.ar (D. Firmenich).

https://doi.org/10.1016/j.infsof.2021.106735
Received 2 September 2020; Received in revised form 9 April 2021; Accepted 15 September 2021
Available online 5 October 2021

augmenters [7], and several of them are complex enough as to require some programming skills (mainly JavaScript programming skills). Moreover, by reviewing public augmenter repositories, it is clear that many augmenters cannot be specified using current EUP approaches. As an example of this, Fig. 1 presents an augmenter [8] that requires the use of an API to perform one augmentation function. The augmenter in Fig. 1 includes a function for adding videos from Youtube in Wikipedia articles. These videos are retrieved by searching in Youtube using the Wikipedia article's title. In this case, the augmenter needs to use the Youtube API, which requires some programming skills.

Currently, in Web Augmentation (WA) communities, those members with programming skills create and share their augmenters in different public repositories from where other users may browse and download them [6]. However, from the moment in which augmenters are published, they may need a non-trivial maintenance process. As we will show, a big part of this maintenance is related to make the augmenter compatible with a new version of a Web site, or even to make it compatible with other view of the same Web page, which may happen because the Web site was designed with progressive enhancement and responsive design in mind [9]. DOM access is therefore the Achilles heel of this kind of software, because when the structure of a particular Web is upgraded (this may happen very often), augmenters may not work because the expected DOM elements do not exist or augmenters may also fail because they manipulate the wrong DOM elements. This gets worse nowadays because volatile functionalities [10] appear very often (Christmas promotions, temporal advertisements, etc.), and consequently, a particular DOM may change just for a couple of days to come back to its previous version when the volatile functionality is removed.

In this paper we show that fixing the augmenter for making it compatible with a new version of the Web page's DOM is a very frequent maintenance task. To make things even worse, in most cases augmenters are maintained by their original developer, and this task may take several days, producing bad usage experiences to those users who are running the broken versions. We believe that it happens given the nonexistence of an engineering method supporting this kind of software. And we also believe that within this process, maintenance plays a very important role and must be driven by end-users, but also we think that this is not possible without considering the support also for developing and testing the augmenter.

Although the Augmented Web is a promising platform for EUP [11], existing EUP approaches tend to be specific to a particular adaptation domain (such as improving accessibility, giving support to a recurrent task, allowing layout changes, etc.). Consequently, existing EUP tools have strong limitations regarding the development of complex and cross-domain augmenters. However, there are some studies showing that end-users may manipulate (annotate, rearrange, hide, remove, integrate, etc.) existing UI elements by means of visual tools [7]. Based on this kind of visual tools, our underlying idea is to encourage developers to decouple the augmenter's source code associated with the intrinsic augmentation logic from the code that requires retrieving a particular DOM element. This decoupling allows to dynamically changing the references to DOM elements without changing the augmenter source code. To support this idea, we provide a development framework to create augmenters. We show that by using a mockup-driven development tool, such as MockPlug [6], the framework may be instantiated, easing the development of augmenters. But more importantly, using this visual manipulation of UI elements, we propose an approach that enables EUP for the augmented Web focused on the maintenance stage in our method, i.e., mainly during the corrective maintenance related to Web pages' upgrades.

Our contribution in this paper is three-fold. We first present an analysis of more than 8.500 augmenters, to understand how they were updated and maintained concerning their DOM dependencies. Second, based on our findings, we designed a framework for Web augmentation development that let developers define the access to the Web sites' DOMs without hard-coding these dependencies in the source code. DOM references are instead specified as UI augmentation widgets, creating an API for accessing the corresponding DOM elements references through Web services. Our approach takes advantage of the existent evidence about end-users capabilities regarding visual programming tools, to let them collaborate on the augmenter maintenance. Using our approach, when a Web site is upgraded, end-users may add new references to DOM elements with which the augmenter will work; this is done by visual selection of the target elements. Third, we have conducted an experiment with 48 participants. It demonstrates that end-users without skills for programming augmenters are capable of maintaining augmenters that have stopped working because of a Web site upgrade. Additionally, our experiment shows that collaborative maintenance reduces the time gap between the Web site's upgrade and finishing the corrective maintenance.

The remainder of this work is organized as follow. In Section 2, we introduce the background for this work, i.e., the ideas underlying the Augmented Web, the existing communities, and how augmenters are currently developed. Section 3 presents a study we have made with more than 8.500 augmentation scripts, whose results support our aims. In Section 4, we introduce our Web augmentation engineering approach. It starts with an overview of our approach and then introduces the basic underlying architecture for the augmenters' development and maintenance processes. In Section 5, we describe the visual programming tool for developers, for testing the augmenters, and the tools for augmenter's users, which are aimed to end-users without programming skills. An evaluation of our approach is presented in Section 6 based on an experiment. Section 7 discusses related work, while in Section 8 we finally present the conclusions and outline future work.

2. Background: Web augmentation basics

In this section, we briefly introduce the basic concepts of Web augmentation and describe the existing software communities developing augmenters.

As we explained briefly in the introduction, Web Augmentation is a technique to create software artifacts that aim to adapt existing, third-party Web applications. Augmentation is achieved by manipulating their public part, i.e., what is available through HTTP requests, before it is shown to the user. From the architectural point of view, this manipulation may occur on a proxy server (several systems based in transcoding improve some Web pages aspects, such as accessibility [12]), but the most popular incarnation of Web augmentation is through client-side DOM manipulation. Augmenters are usually implemented in some type of Web browser extensions. These extensions are allowed to know when a particular content is loaded and they can manipulate that content. According to existing literature [1], extensions may be classified based on the augmentation level: single site, domain-based cluster, and concept-based cluster. The basic idea is that an extension may work for a single Web page or for multiple Web pages. For the first case, an example is the addition of download buttons for Youtube videos in their respective Web pages. This kind of extension may also integrate other Web contents, e.g., augmenting Wikipedia articles with images retrieved from Google Images using the Wikipedia article's title. An augmentation for multiple Web pages may be more generic, e.g., adding behavior to every anchor in order to allow end-users to preview the target Web page, or to create an image carousel with all HTML image elements in the current Web site.

Beyond those Web Augmentation approaches from academia, different user communities have emerged in the last years around Web Augmentation tools. Among these communities, we have selected the two most representative in term of artifacts, developers. and end-users:

- **Userscripts** is one of the biggest communities which focuses on a specific kind of software called userscript. A userscript defines a JavaScript program that the Web browser executes in the context

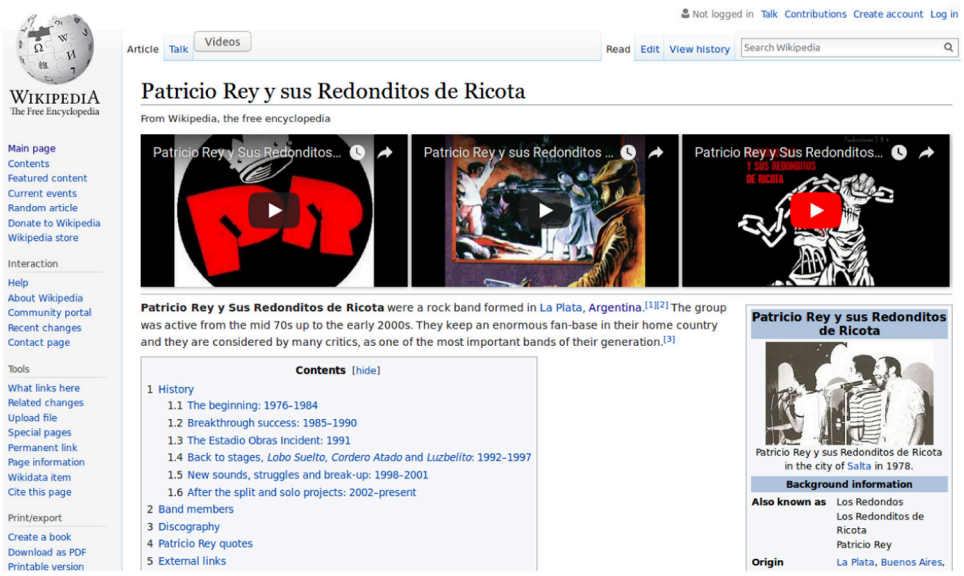

Fig. 1. Wikipedia augmenter for adding related Youtube videos.

of the current (loaded) DOM. With this kind of artifacts, Web Augmentation may be used for adding or modifying both content and functionality of Web applications. The first popular plug-in of this kind was GreaseMonkey, a Firefox plug-in working as a JavaScript engine. It allows users to execute external JavaScript scripts (userscripts) when a particular Web page is loaded. However, nowadays there are equivalent engines compatible with other popular Web browsers (Safari, Chrome, etc.); most of them run the same type of userscripts. A userscript defines, besides the JavaScript logic for manipulating Web content, a set of metadata. This metadata indicates, for instance, which Web pages are going to be manipulated. There are several userscripts repositories that offer, altogether, several thousand userscripts developed by those community members with programming skills. Some of the scripts were downloaded more than one million times. The most well-known community for userscripts is greasyfork.org. Userscripts are reusable among Web browsers, because they depend just on the browser plug-in, for instance, TamperMonkey, which works in the same way in every Web Browser.

- **Userstyles**, in contrast to userscripts, are written in CSS. Although this technology is very powerful for performing layout adaptations and also improves the look and feel, their expressiveness in comparison with userscripts is lower. For instance, web content integration is hardly made with userstyles. However, this is a large community sharing several thousands of artifacts which shows how relevant the adaptation of third-party Web sites is for users.

3. Maintainability of Web augmentation software: The case of userscripts

In the introduction of this paper, we have stated that the dependency of DOM references is one of the main problems of Web augmentation artifacts. This dependency overloads developers' work due to the required corrective maintenance and it hampers good user experience due to the fact that augmenters are failing or simply do not work. Here we share the point of view of other authors regarding this aspect [1]. However, in order to provide readers with concrete figures regarding corrective maintenance activities in these communities, we have analyzed more than 8.500 userscripts available in an open and representative community. As we will show, the results strongly support our view on and understanding about this kind of software.

Our analysis focuses on userscripts, because of two reasons. On the one hand, all userscripts work at single or multiple Web page levels–since they are always executed when a Web page is loaded. On the other hand, in comparison with userstyles, userscripts cover most kinds of augmentations requirements [7], i.e., change, delete and addition of content, presentation but also functionality. Thus, the way in which userscripts depend on DOM elements is very comprehensive.

In the following, Section 3.1 presents the details of the analysis and its results. In Section 3.2, we summarize our findings. The data obtained from the experiment is published at FigShare[1] and at GitHub.[2]

3.1. UserScripts analysis

We performed our study on greasyfork.org, one of the most updated and used userscript repositories. In order to carry out this study, we implemented a script that performs web scraping, a technique to automatically extract information from a website [13]. In this case, the web scraping script goes through the list of userscripts and their authors, obtaining the source code of every userscript version and their release dates, users' comments and feedback, description, user developer, number of installations, etc. All this information was parsed and used to generate a userscript database for our analysis, one in October 2015 and another one in July 2020 to compare the evolution and to check if the problem still remains. At the start of the userscript collection period, the number of userscripts in greasyfork.org was about 7.500, whereas in 2020 the number has increased up to more than 25.720.[3]

As Fig. 2 shows, the amount of scripts created by a developer has increased from 2015 until 2020 due to the popularity growth of this repository of scripts (GreasyFork). Although most of the developers have created just one userscript (63% in 2015 and 41% in 2020), there is also a huge amount of users that created more than one script. The authors that created only 1 script is lower and those who uploaded 2 or more scripts is higher in 2020 than in 2015, with an average of almost 5 scripts uploaded per developer in 2020. Therefore, the comparison

[1] For experiment's data see "GreasyFork Data" at https://figshare.com/projects/Engineering_Web_Augmentation_Software/37385.

[2] https://github.com/cgmora12/GreasyFork-Web-Scraping.

[3] The up-to-date total amount of scripts in GreasyFork is available here: https://greasyfork.org/en/scripts.json?meta=1.

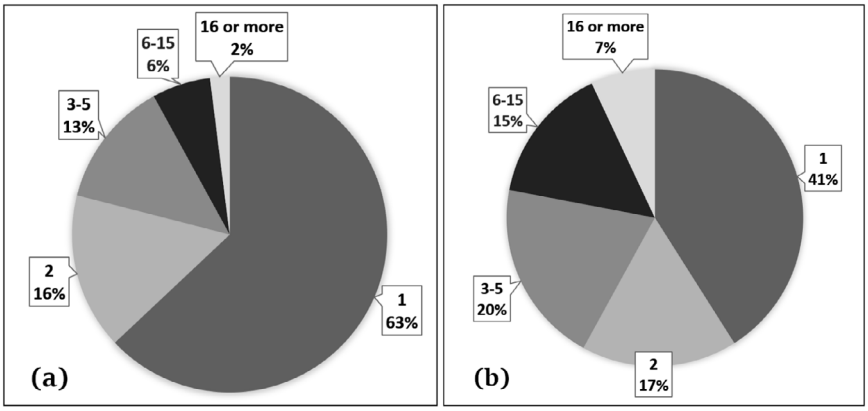

Fig. 2. Amount of userscripts uploaded to GreasyFork per developer: (a) in 2015 and (b) in 2020.

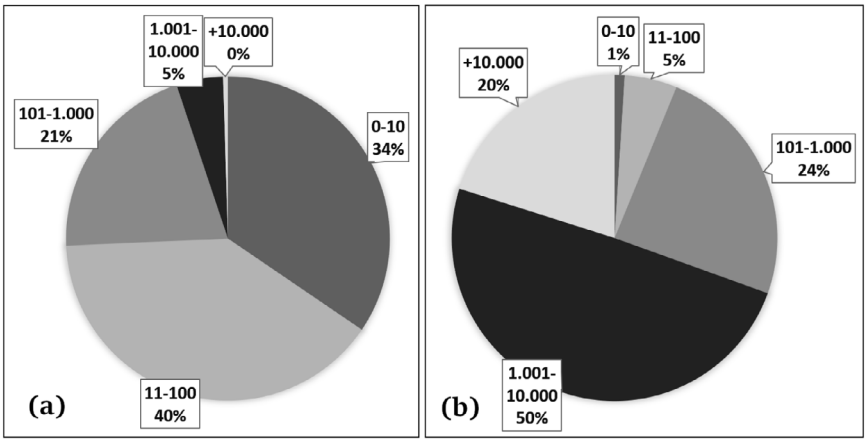

Fig. 3. Installations per userscript: (a) in 2015 and (b) in 2020.

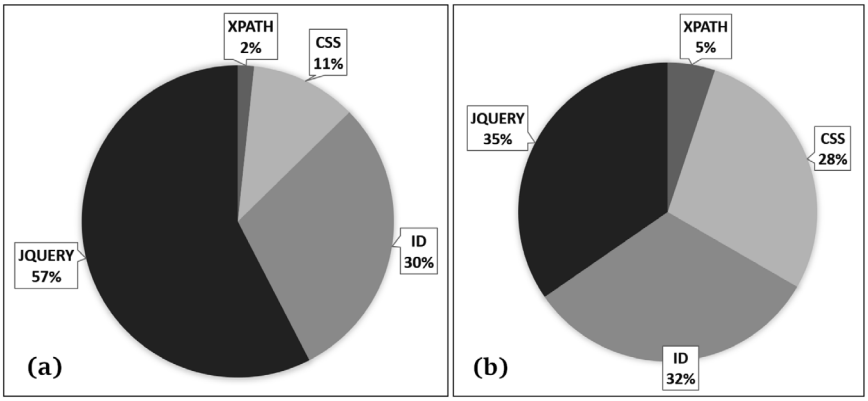

Fig. 4. Type of hard-coded DOM element references found in userscripts: (a) for Sample A (automatically analyzed) and (b) for Sample B (manually analyzed).

between 2015 and 2020 points out that the amount of userscripts uploaded to GreasyFork per developer has significantly increased.

Moreover, the number of installations of a particular userscript is an indicator of the community's interest in it. As Fig. 3 illustrates, in 2015, 34% of the userscripts only reach up to 10 installations, but the rest of the artifacts seem to have several dozens of users as minimum. However, the installations per script have increased a lot from 2015 to 2020. In 2020, the majority of userscripts (70%) reach more than 1.000 installations.

3.1.1. DOM references

We have also taken into account the ways in which developers reference DOM elements. We have analyzed the source code of a set of userscripts in their last version — initially in 2015 and subsequently in 2020.

In 2015, we created a code analyzer to automatically detect DOM references. With this code analyzer we discovered that 71.5% of the 7.538 artifacts analyzed had references to DOM elements in their source code, which we tagged as "hard-coded DOM reference". This hard-coded DOM references are based on DOM attributes as ID, and they are performed using different techniques such as jQuery, CSS selectors, or XPath expressions. The 71.5% of artifacts with references to DOM

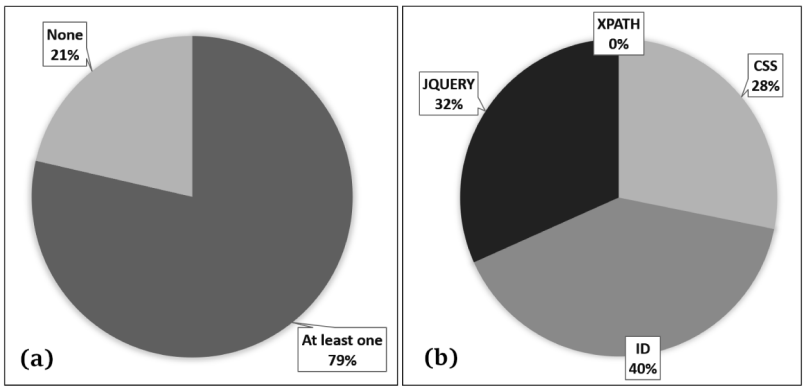

Fig. 5. DOM element references in userscripts for Sample C automatically analyzed in 2020: (a) distribution of userscripts with and without detected DOM references and (b) distribution DOM reference kinds.

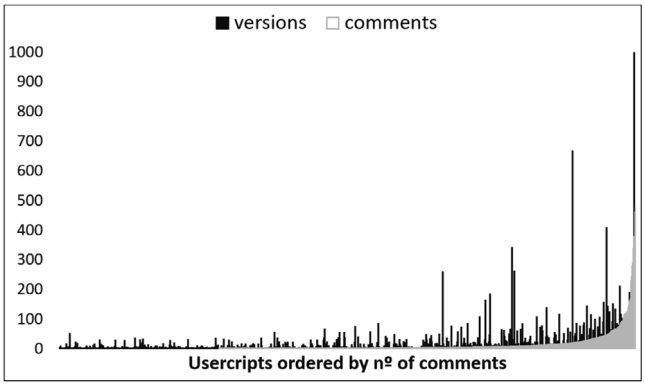

Fig. 6. Relation between userscript upgrades and feedback for Sample C.

represent 5.386 userscripts in absolute numbers, to which we will refer as Sample A in the rest of this section. A total of 68.925 hard-coded DOM references were found in Sample A. Fig. 4(a) depicts the kind of reference, with a vast majority of jQuery references to DOM (57%), followed by references to the ID (30%), references made with CSS selectors (11%) and references by XPath expressions (2%). However, there were a 28.5% of the artifacts (a total of 2.152) for which our code analyzer did not detect references. From these 2.152 artifacts with no detected DOM references we have selected 200 randomly to check if they actually have these kind of references by performing a manual inspection of their source code. These 200 userscripts formed the Sample B and we obtained that the 39% of them indeed included hard-coded DOM references. The kind of references from Sample B is summarized by Fig. 4(b), with still a majority of jQuery references (35%), followed by references to ID (32%), references made with CSS selectors (28%) and references by XPath expressions (5%). Therefore, we can conclude that in 2015 even more than the 71.5% of userscripts include hard-coded DOM references (as demonstrated with our automatic and manual analysis).

Finally, a set of 1.000 artifacts analyzed in 2020 are considered as Sample C. As we can see, there is still a 79% of scripts that include hard-coded DOM references as it is illustrated by Fig. 5(a). Again, the DOM references are based on DOM attributes as ID (40%), using jQuery (32%) and CSS selectors (28%), including also XPath expressions with slightly less than 1% of total references as shown by Fig. 5(b).

3.1.2. Upgrades

In addition to this information about DOM element references, our study was focused on userscripts maintenance. With this in mind, we compared the amount of comments for a userscript made by other users with the number of upgrades for this same userscript. In this sense, Fig. 6 shows the amount of comments and the number of upgrades made for each userscript of Sample C. In particular, the x-axis corresponds to the scripts analyzed (ordered by the amount of feedback obtained) and the y-axis corresponds (i) to the number of comments and (ii) to the number of updates. As can be seen, there is a relation between the amount of userscripts' versions and comments for the userscript, showing a strong maintenance activity as the feedback for the userscript increases. By performing the Pearson Correlation test we can state that there is a moderate relation as the correlation coefficient is 0.48 and the correlation coefficient is statistically significant because the P-value is lower than 0.05 (1,80E−58). Moreover, an interesting finding is that most of the userscripts with comments (i.e., feedback from the end-users) have been upgraded, showing that end-user involvement not only happened but it may have consequences on the artifact's maintenance activities. A total of 722 of 1.000 scripts in 2020 have been commented by users and all of them have been upgraded. In 2015, 1.181 userscripts have been commented by end-users, and 81.2% of them have been upgraded.

In addition, Fig. 7(a) shows the number of days elapsed between two consecutive script upgrades in Sample A. Focusing just on upgrades that are between one day and one year old in relation with their immediately previous version, the mean is 27.17 days. Among upgrades that are at most 10 days old, the mean is 3.15 days.

Regarding Sample C, we analyzed when did the last upgrade occur, showing the results in Fig. 7(b). We obtained that the vast majority of userscripts analyzed were upgraded recently regarding their publication date: 62% of the scripts have been upgraded in the last year, with 35% of them upgraded within the last 2 months.

Finally, we analyzed the source code of every upgrade regarding how hard-coded DOM references changed. In this sense, we discovered that 51% of the artifacts in Sample A changed their references at least once in the last year (Fig. 8(a)). Comparing the 68.925 references found in their previous versions with the total of 903.342 references that we found analyzing the source code of every userscript version demonstrates that instead of remaining static, DOM references have been updated at least once in half of the analyzed artifacts. Fig. 8(b) shows the same analysis results for Sample B, which we made manually. In this case, we see that from the 39% of the userscripts that contain at least one hard-coded DOM reference, 46% of them have suffered from reference upgrades. Moreover, in 2020 more than 62% of scripts from Sample C, whose 79% have hard-coded DOM references, were upgraded at least once in the past 12 months.

3.2. Discussion

From our analysis, we can conclude that almost 80% of the current artifacts (cf. Fig. 5(a)) have at least one hard-coded DOM reference.

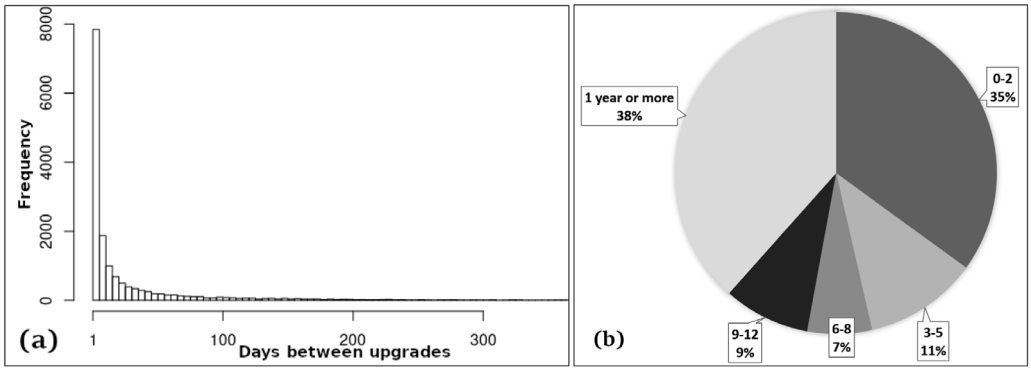

Fig. 7. Userscript upgrades: (a) days between upgrades for Sample A and (b) last userscript upgrade in months.

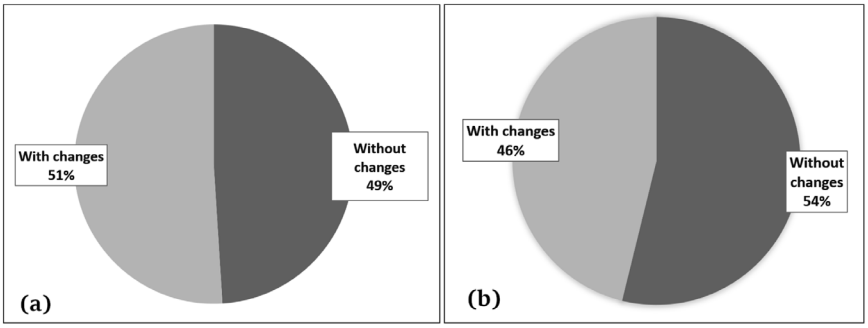

Fig. 8. Userscript upgrades affecting DOM references: (a) for Sample A (automatically analyzed) and (b) for Sample B (manually analyzed).

For these artefacts, we found out that the 62% of them have been upgraded recently (in the previous year, cf. Fig. 7(b)), and also in the samples from 2015 more than 50% have been upgraded at least once for updating DOM references (cf. Fig. 8(a)). With these figures in mind plus the fact that this weakness is frequently described in the literature [1,14,15], we strongly believe that hard-coded DOM references are an obstacle for a broader success of the Augmented Web. In spite of the current efforts for improving the robustness of DOM references for avoiding this kind of problem, as we will discuss in the related work section, it is still required to maintain the source code of augmenters. Moreover, the study shows that part of the end-user community cooperates with augmenter developers by giving feedback to them.

The following section presents an approach for decoupling DOM references during the augmenter development process. Then, once in production, these references may be maintained collaboratively by end-users. Since the proposed DOM reference system may be used in any underlying architecture for building plugins, we stop mentioning userscripts specifically, but we refer to them more generally as augmenters from now on.

4. Engineering Web augmenters: A development and maintenance approach

Considering our findings related to DOM references maintenance presented in the previous section, we propose an augmenter development and maintenance approach supporting stakeholders (developers and end-users) during the entire process.

Our aim is to support the community with tools which support them to achieve the kind of tasks which are appropriate for each role. From our point of view, and also following the collaboration between developers and end-users in existing communities, the natural division of these tasks is that developers–as they currently do–create augmenters, while end-users maintain augmenters when these can no longer reach the desired DOM elements. This general division of labor has two consequences for our approach. On the one hand, our approach aims to be used during the augmenter life-cycle, considering requirement definition, development and maintenance stages, without the need of programming. On the other hand, it aims to make it possible to create augmenters (or part of the augmenters) with the lowest abstraction level, i.e., by programming JavaScript. In this way, we may guarantee that expressiveness is not spoiled, and complex requirements (that cannot be satisfied using pure end-user programming and visual programming tools) may be tackled, but still conserving the possibility of being maintained by end-users when the augmenter does not work due to Web pages' upgrades. These two sides of the same coin must be supported by an underlying architecture that lets developers to specify the UI manipulation without depending directly in DOM references, in order to allow either the change or update during run time. The userscripts survey we discussed in the previous section has guided us to the following premises regarding the underlying architecture guidelines:

- **Separation between logic maintenance and DOM matching maintenance**: when an augmenter is modified for including new functionality, we think that developers are who must be responsible of this maintenance, because probably this new functionality requires programming JavaScript. However, we have explained that corrective maintenance required after a Web page's upgrade may be done by end-users. Nevertheless, we must support the integration of both maintenance kinds.
- **No hard-coded DOM references**: in order to reach a sustainable life-cycle, augmenters source code must be as independent as possible from Web pages' DOMs. In order to achieve this goal, we propose to replace common DOM references by a set of objects, called UIWidget, DOMElementOfInterest (DEOI), DOMElementOfInterestReference (DEOIReference) and DOMReference, which are defined as follows:
  - **UIWidget**: is a composite unit of user interface elements that will be treated as a whole widget. An augmenter must

be developed using an event-based approach, considering the events (load, mouseover, click, etc.) over these UIWidgets. In this way, the augmenter must be specified in terms of these UIWidgets and its behavior. However, as the reader must know, the user interaction events actually happen over real DOM elements, which makes indispensable that a UIWidget has knowledge of these DOM elements to set event listeners and manage the augmentation effect. Therefore, instead of coding a hard-coded DOM reference in the source code, a UIWidget depends on a DEOI object, as explained below.

  - **DEOI**: A DEOI is a proxy object that represents a DOM element. When a DEOI is asked for retrieving the actual DOM elements, it uses a dynamic method for referencing current DOM elements. In fact, a DEOI may have several references instead of just one. But these references are not common DOM references (such as xPath, CSS selector, etc.) but a wrapper of them called DEOIReference.
  - **DEOIReference**: a DEOIReference is a complete snapshot from the context in which a real DOM element was found. It includes information about the user's device, the Web browser version, date, etc., plus the concrete information to retrieve the real DOM element, which is wrapped in another object called DOMReference.
  - **DOMReference**: a DOMReference maintains the information required to obtain a DOM element given an URL.

Altogether, these objects hide DOM references from the augmenter's source code, giving the flexibility to have more than just one valid reference. Different DEOIReferences may be defined for the same augmenter to be executed in different contexts. If some DEOIReference fails, the DEOIElement can recover by selecting another DEOIReference from its collection.

- **Transparency for developers**: the main idea is that developing augmenters focuses on specifying the augmentation logic rather than how to retrieve DOM elements, their events listeners, etc. In this sense, we believe that developers must be assisted with visual tools to create their augmenters without "inspecting" the Web page. With this in mind, we use MockPlug, a high fidelity prototyping tool that let developers create UIWidgets and its behavior by visually manipulating UI elements. When the developer selects a DOM element to be part of a UIWidget, our tool creates all the necessary objects (i.e., DEOI, DEOIReference and DOMReference).
- **Collaborative corrective maintenance**: the kind of task we propose for fixing augmenters that became old given an upgrade of their target Web sites is very similar to a content annotation task. In the literature, several kind of approaches based on Web content annotation (for instance for improving accessibility [12]) are based on collaboration. This means that an annotation made by a particular user is available for all the community. In this sense, we propose that if a user fixes or modifies an augmenter by adding a new DOMReference, this is also synchronized for everyone.
- **Context-awareness and durability of DOMReferences**: different users may run the same augmenter in different environments, environments that may determine how the original UI is rendered and also how the augmented UI is rendered. In this way, if there are several DOMReferences defined for a specific DEOI, when the augmenter is installed on the end-user devices, it is self-configured to use a DOMReference compatible with the user's environment. However, the approach must guarantee that the durability of previous DOMReference, because they would be important for other end-users.
- **Testable DEOI**: Defining a new DOMReference for a particular DEOI is a risky task if the retrieved DOM element is not the correct one from the point of view of the UIWidget goals. In this way, the approach must support the testing of these retrieved DOM elements. In order to achieve this testability, we propose that developers may specify different conditions that DOM elements obtained by DEOIs must satisfy to be considered an acceptable target.

We have designed and implemented an architecture that considers all these premises. This architecture is composed by a framework for managing dynamic DOM references, called DEOIProxy Framework, a back-end application to allow collaboration and synchronization of these references, plus some tools for supporting specifically each role (developers and end-users).

The rest of Section 4 is organized as follows. First, and based on the core definitions presented above, Section 4.1 shows our approach in a nutshell, focusing on the stakeholders responsibilities and the two main phases defined for the augmenters life-cycle: development and maintenance. Section 4.2. discusses the details of the DEOIProxy framework. Finally, Section 4.3 explains deeply the differences between our approach and the classical ad-hoc programming of complex augmenters.

4.1. The approach in a nutshell

We propose an approach that uses visual programming tools for developing and maintaining augmenters as outlined in Fig. 9 and described in the following two subsections. The first subsection is focusing on the three steps for the script developer (upper part of Fig. 9), while the second subsection is focusing on the three steps for the end-user maintenance (lower part of Fig. 9).

4.1.1. Augmenter development

In a first step, the augmenter developer uses a visual programming tool (which is integrated into the Web browser) to specify every part of the augmentation layer that impacts in the UI perceived by the user (cf. Step 1 in Fig. 9). This tool, called MockPlug, is inspired by typical mockup tools and it allows developers to export the initial augmenter source code. Similarly to existing EUD approaches for building augmenters [16], the idea is to convert Web pages into a sort of canvas where HTML elements may be manipulated visually. With this tool, developers may generate preliminary versions of augmenters. Developers may drag and drop new UIWidgets and also convert existing parts from the Web page into UIWidgets. When the developer finishes the definition of the augmenter with our tool, two steps are required as illustrated in Fig. 9 by Steps 2 and 3.

In Step 2, the DEOI references (together with all the augmenter's metadata) are stored in the repository. Once in the repository the augmenter definition may be revised by the community, generating several versions of the same augmenter that are discussed by the community. To support collaboration during development, the approach provides an evolution tree for this augmenter, showing in color and sizes the votes and opinions of the community members. As this step is more focused on the requirement specification, we do not provide further details in this paper and refer the interested readers to our previous work, in which we have focused on collaborative aspects for the requirements specification stage [6,7].

In Step 3, the developers must define a set of cases that will help to test if the DOM elements retrieved by the augmenter at run-time are the correct ones. In order to do it, we define the concept of Web Augmentation (WA) Unit Test, which are essential to reduce the errors during the maintenance (see Section 4.3 for more details). Finally, the developer may generate the augmenter source code with all the augmenter behavior plus the test cases, which are delivered with the augmenter source code, and subsequently, they may be run at the

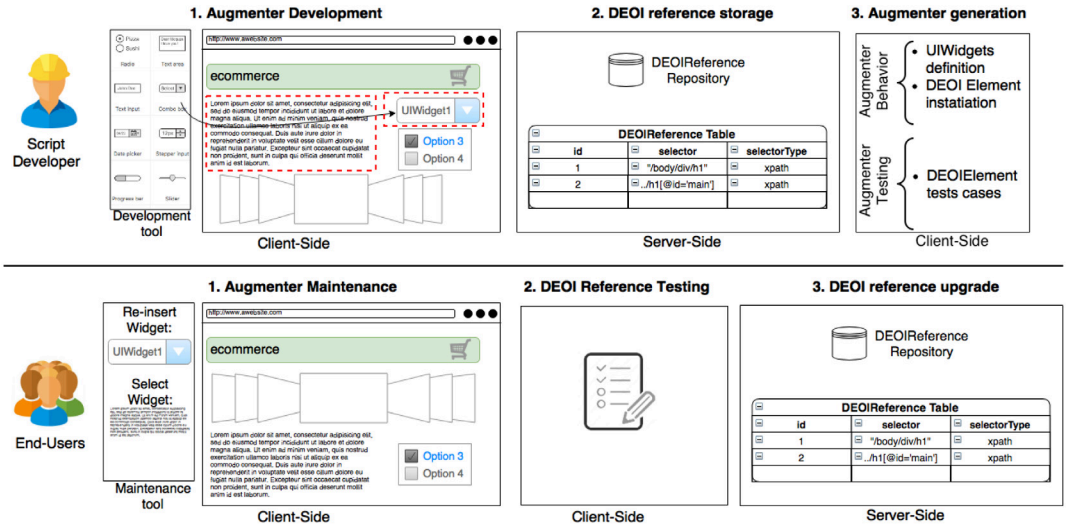

Fig. 9. Overview of the approach.

client-side, in the end-user's devices. The generated source code is not dependent on DOM references, but for very complex augmenters developers may program JavaScript code once the augmenter's first version is generated.

4.1.2. Augmenter maintenance

From our point of view, and based on our userscripts analysis, the most common maintenance task is related to Web pages' upgrades. At this point, WA unit tests play a very important role, because these tests are run automatically in the end-users' browsers. When a WA unit test fails, end-users are advised to start the maintenance task. To empower end-users to carry out this task without depending on the original augmenter's developer, we provide them with a EUP tool for redefining those broken DEOI references (cf. Step 1 in Fig. 9). This tool asks users to select or re-insert the UIWidgets that could not be woven. When end-users fix these issues, the test cases defined by developers are run to check if the new DOM element is compatible with the UIWidget (cf. Step 2 in Fig. 9). If the augmenter works appropriately with the new DEOI references, the end-user maintainer may synchronize the new DEOI reference with the repository (cf. Step 3 in Fig. 9), so the next end-user that executes the augmenter would not perceive any problem. It is important to note that this maintenance tool is also deployed together with the augmenter, in this way everything may be done inside the Web browser.

4.2. The DEOIProxy framework

Our approach for developing augmentation software is composed by two main components. One runs at client-side, performing the augmentation in end-user devices; and the other one runs at server-side, working as a repository of augmenters and DEOIReferences that allows the collaborative maintenance. On client-side, the DEOIProxy framework provides the behavior for adding and retrieving DEOIReferences from the repository. The client-side counterpart may also include Web Augmentation (WA) tests, for checking whether the current DEOIReferences used by the augmenter are broken or not. In the following, we first explain several aspects of the framework and subsequently we briefly explain how the WA tests are defined and executed.

A simplified overview on the design of the framework is presented in Fig. 10. The DEOIProxy Framework has three main hot spots, which allow developers to create UIWidgets (cf. green classes in Fig. 10). UIWidget is an abstract class that manages all the dependencies with the DOM. In order to do that, a subclass for each kind of UIWidget is available. At least, each UIWidget knows a DEOIElement, which indicates how the UIWidget must be woven in the target Web page. This DOM dependency is then decoupled and no hard-coded references are used for this task. Also the abstract UIWidget class provides all the behavior required to manage event listeners and to retrieve concrete DOM elements, defining a minimal behavioral interface that the concrete UIWidgets must implement. For instance, the method "getConcreteValue" must be redefined to specify which data to extract from the retrieved DOM element. Another example would be that an augmenter may require to obtain the textual content of the retrieved DOM element, and then, this behavior is defined in that method.

All the dependencies with DOM elements are managed through the DEOIElement object, which is a proxy that returns a concrete DOM element but also provides behavior to add new DEOIReferences. Although a DEOI may have a set of DEOIReferences associated, when the augmenter is run at the client-side, and for the sake of performance, the last DEOIReference used successfully is specifically implemented through the "stableReference" association. The DEOIElement also has the behavior for upgrading references from the central repository. The DEOIReference element is an abstraction that wraps a DOM references.

Currently, we support three kinds of UIWidgets. We believe that by these three kinds we cover most common uses of augmenters presented in the literature. We shortly characterize each kind of UIWidget in the following.

- **ExistingWidget**
  - Use: when the augmentation requires manipulating an existing DOM element, this is the kind of UIWidget to extend. This kind of UIWidget knows another DEOI that actually represents the original DOM Element.
  - Specific aspects: in the case of ExistingWidget, the inherited relation "insertionReference" is used also because those original DOM elements converted into UIWidget may be rearranged in the Web page layout, requiring then an insertion reference.
- **DomainBasedWidget**
  - Use: as it is indicated in the literature [1], sometimes, the augmentation requires to define domain concepts, which occurs mainly when an important behavior of the augmentation requires to manipulate more complex data structures presented in the Web page, and also this manipulation is the same for a set of sibling elements. In this case, this subclass of ExistingWidget works as a kind of annotation that allows

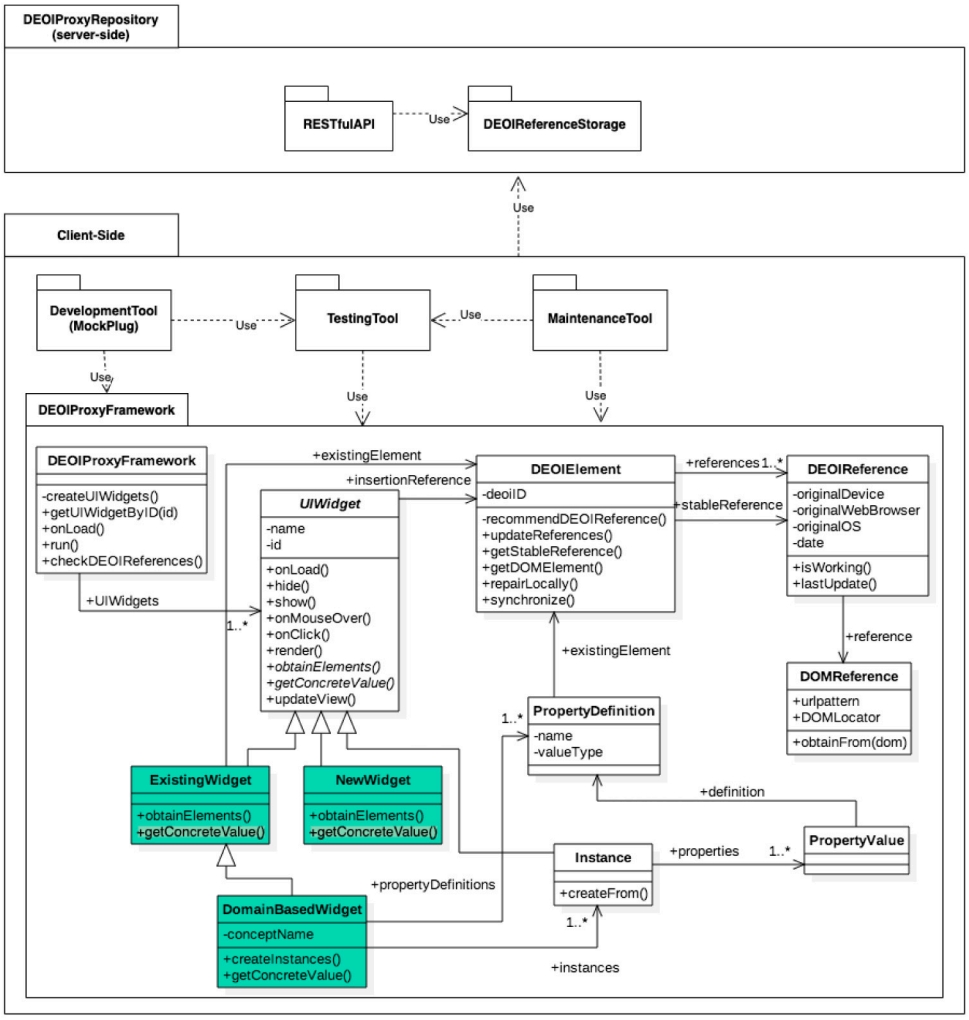

Fig. 10. Design of the DEOIProxy framework (default multiplicity is 1..1).

developers to define the domain concept and their properties. This implies to specify how the value properties must be obtained by parsing the Web page DOM.

- Specific aspects: this kind of UIWidget implements the method "createInstance". This method applies the annotation template and consequently one or several instances of the domain concept can be created. It is important to note that all the behavior defined in the DomainBasedWidget subclass is attached to every Instance created, which is made transparently from developers' point of view.

- **NewWidget**
  - Use: it is very common that the augmentation requires adding new UI elements. In this case, the developer must define a subclass of NewWidget.
  - Specific aspects: in the case of NewWidgets, the "obtainElement" method does not retrieve any existing DOM element; it must create and return the new DOM element to be woven in the Web page. In this case, the "insertionReference" association is used to maintain a reference to the DEOI element used as an anchor for the weaving process.

Summarizing, with our approach, developed augmenter software includes:

1. The DEOIProxy framework.
2. The augmenter source code, including all the definition and instantiation of UIWidgets that had been manipulated with the visual tool. Also, any other aspect defined with the visual tool is present in the source code, e.g., event listeners (onclick, onload, etc.). UIWidgets have a unique ID.
3. The test cases (either automatically generated or defined by the developer).
4. The EUP tool for maintaining DEOI references is integrated in the augmenter. In this way, end-users do not need to install additional tools for fixing their preferred augmenters.
5. Additionally, when the artifact is created, the tool stores in a repository the set of UIWidgets and their DEOI objects, with the current references. This repository exposes existing UIWidgets and DEOIs through a RESTful API.

Test cases are programmed by convention names (following the principle of convention over configuration). In this sense, if a programmer extended the ExistingWidget class, creating a UIWidget called "TitleWidget", then the WA test framework would check for the class "TitleWidgetTest". With these components, when an end-user executes the augmenter for the first time, each UIWidget requests the DEOI references to the repository using the RESTful API. Besides this first time, when the execution of the augmenter results in a problem, the DEOIProxy framework requests to the repository for new DEOI references. If there are new references, it adds them on the client-side and run tests (specified by the original augmenter's developer) to guarantee that new references are correct regarding the obtained DOM element.

If there are no new references available, end-users are offered to fix the augmenter with the maintenance tool (as presented in more detail in Section 5).

4.3. Comparison between DEOIProxy references and traditional DOM references

The key of our approach is to support the development of augmenters without requiring hard-coded DOM references. In this manner, the corrective maintenance (related with DOM references compatibility) may be performed by end-users without requiring the intervention of the developer. To achieve this, we have implemented a framework, called DEOIProxy, which allows developers to program the augmenter logic without depending directly on DOM elements. Instead, our framework uses DEOI elements that act as proxies to the concrete DOM elements. In addition to how the references are managed, our approach supports a higher abstraction level with UIWidgets. A UIWidget may represent an original DOM element or a new DOM element required and created by the augmentation. Every UIWidget is composed by, and related with, DEOIs elements in different ways depending on the particular kind of UIWidget. When the target Web page is upgraded, new DOM references may be added to the DEOI. Consequently, the augmentation logic (implemented in the UIWidgets) does not change and the augmentation continues working properly.

Table 1 illustrates this idea in the different phases of the augmenter's life-cycle, i.e., during development, installation and usage /maintenance (cf. first column of the table). In the second column, the table shows how traditional development of userscripts manages DOM references, and in the third column, the table summarizes our approach. We summarize in the following the main differences based on the more detailed table.

- **Development**: in the traditional way, the developer needs to detect a DOM selector by inspecting the Web page's DOM. In contrast, with our approach, when the developer defines an UIWidget (which is done by selecting DOM elements), our framework creates an instance of DEOIElements and attaches to it a valid DOM reference (wrapped into a DEOI Reference element) and stores it into the repository.
- **Installation**: when end-users install the augmenter, in the traditional approach nothing special happens concerning the DOM references. With our approach, the framework creates the tables for DEOIElements and DEOIReferences locally (i.e., the Web browser) using the local storage.
- **Usage and maintenance**: finally, when the augmenter is running, the references are hard-coded in the augmenter's source code using the traditional approach. In our approach, the DOM references are retrieved dynamically when the augmenter is executed. This step is technically the main difference to allow end-users maintenance. As it is shown in the last row in Table 1 (maintenance), if the Web page's DOM changes, with the traditional approach it is mandatory to modify the augmenter source code, meanwhile with the DEOIProxy framework a new DEOIReference is created and added to the local storage (and also to the main repository) and the augmenter may work as it is.

As we will show in the next section, we propose the use of visual development tools which allow developers to define all UIWidgets (and their related DEOIElement and DEOIReferences) just by pointing and selecting elements from any existing Web page.

5. Tool support for augmenter development, testing and maintenance

In this section, we present the tool support for the presented augmenter approach of the previous section.

5.1. Augmenter development

As we mentioned before, we support the early phases of the augmenter development by means of a model-driven development (MDD) tool [17]. The tool is deployed as a Web browser extension, and offers to the developer a set of tools to define the UIWidgets and their underlying DEOIReferences. It also configures a diverse set of features for them. Fig. 11 shows this tool, where the developer uses a pallet of UIWidgets for augmenting the Web page. When a UIWidget is added, it is woven in the DOM in reference to a particular DOM element. However, at the moment of the addition, the UIWidget is not related directly to this DOM element, but it is related through a DEOIElement, which is created to decouple the UIWidget insertion from the current version of the DOM. Also, at this moment, the new DEOIElement is stored in the repository.

As an example, imagine a developer who is creating the augmenter shown in Fig. 1 that includes a function for integrating Youtube videos in any Wikipedia article using its title for retrieving the videos. To develop this augmenter, the developer has to add an UIWidget button for loading the videos on demand. In order to do that, the developer may add this button by drag and drop it from the widgets panel, as it is shown in Fig. 11. The reader may note that the new widget is actually a new DOM element, with the inherent behavior of this kind of element (in this case a button) plus the behavior that is added by the tool through the menu that is shown when the mouse is over the widget. Different things may be done, such as editing intrinsic HTML element attributes, adding behavior, moving the widget to another position, adding a comment, etc.

Besides of adding new UIWidgets, the developer has the option to collect any existing DOM element and convert it into a UIWidget. As Fig. 12 shows at the right, the same panel has the "Select" button that once clicked, allows the developer to point and click with the mouse over the current Web site. When the developer selects a DOM element by clicking on it, this element is converted into a UIWidget, and the same menu appears floating over it, which is appreciated at the right hand side of Fig. 12. It is important to mention that, also in this case, the corresponding DEOI element is the one in charge of retrieving the DOM element.

Following with our example, in this case the developer collects the Wikipedia article, which is from where the text will be extracted to retrieve videos from Youtube.

Once the UIWidgets are defined, the developer may specify some behavior through the UIWidget configuration tool, which is shown at the left in Fig. 13. Here the developer may give to the UIWidget a significant name, add annotations for different kind of behavior (such as requiring XMLHttpRequest). With this annotation, the source code generator may create little snippets of code. The developer may even write some code for specific events that can happen with the UIWigdet (when it is loaded, when the mouse is over, when the user clicks on it, etc.).

Finally, the developer may generate code with the same tool, as Fig. 13 shows at the right. To do it, developer must select a particular code generator (new generators may be programmed). In the case of the example, by selecting "CM FW" (CrowdMock FrameWork) a full stack augmenter is generated regarding our approach, which includes:

- The DEOI framework library
- The creation of UIWidgets and their DEOI elements, which involves also any specification done (basic properties like the name, the code related to the defined behavior, etc.).
- The specified test cases and predefined hot spot extensions for programming.
- The EUP tool for enabling collaborative maintenance of DEOI references.

In this work, since our augmenter analysis was made over repositories of userscripts, we use a code generator that includes all these elements in the form of a userscript.

Table 1
Traditional hard-coded DOM reference vs. DEOIReference.

Step/task	Traditional approach	DEOIProxy framework
Developing	**Developers**: (1) Inspect the DOM (2) Select a target DOM element (3) Obtain a DOM reference (4) Write the DOM reference in the augmenter's source code (5) Code any manipulation with JavaScript for the DOM elements	**Developers must**: (1) Use Mockplug to select visually those DOM elements to be converted into UIWidgets (2) Use Mockplug to define UIWidget behavior (3) Code any further logic that could not be expressed using Mockplug (4) Select default test cases & define specialized test cases **DEOIProxy framework**: (1) Creates DEOIElement, DEOIReferences and DOMReferences for the DOM elements composing the intervenient UIWidget
Sharing	**Developers**: (1) Updload the augmenter file to the repository	**Developers**: (1) Generate the augmenter source code with Mockplug (2) Upload the augmenter file to the repository **DEOIProxy framework**: (1) Synchronizes DEOIElement, DEOIReferences and DOMReferences with the repository
Install & deploy	**Users**: (1) Download the augmenter source code (2) Install the augmenter	**Users**: (1) Download the augmenter source code (2) Install the augmenter **DEOIProxy framework**: (1) Obtains DEOIElements, DEOIReferences and DOMReferences from repository and store them locally (2) Runs test cases to validate retrieved DOMReferences
Testing	**Users and Developers**: (1) Detect if the augmenter is not working well (2) Use the Web browser's console to see errors	**DEOIProxy framework**: (1) Runs test cases to validate retrieved DOMReferences (2) Notifies users and the repository that augmenter's DOMReferences are not working
Maintenance	**Developers**: (1) Debug and fix DOM references (2) Upload the new version **Users**: (1) Reinstall the augmenter	**DEOIProxy framework**: (1) Checks references and upload local storage (2) Runs test cases for new DOMReferences (3) Fixes automatically or open visual maintenance tool for manual fix **Users**: (1) Use this tool to correct DOM references **DEOIProxy framework**: (4) Runs test cases to check compatibility (5) Synchronize new DOMReferences

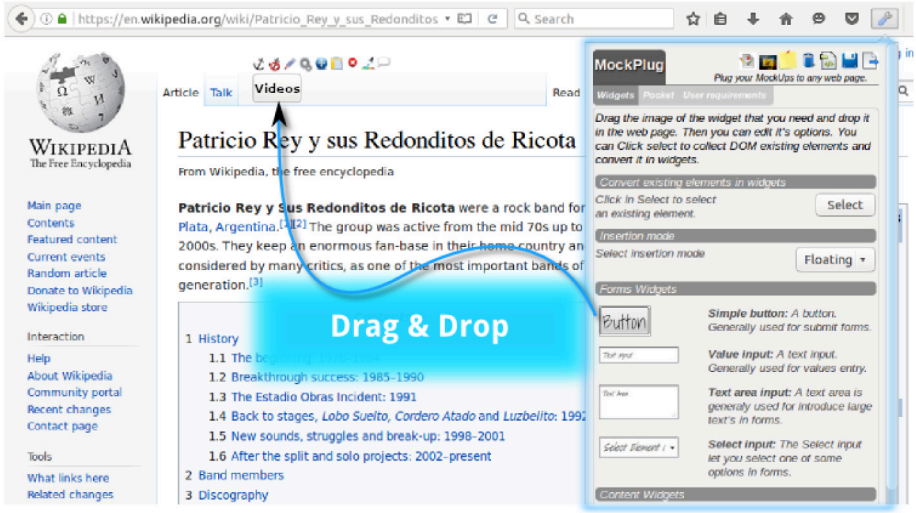

Fig. 11. Mockplug in action: adding an additional button.

5.2. Testing DEOI references

Our approach also allows developers to define test cases in order to detect when an augmenter is not reaching the desired DOM elements (cf. Fig. 14). An augmenter may fail because a DOM selector does not retrieve any element but also when it retrieves wrong DOM elements, i.e., other elements that are not of interest for the augmentation. To tackle this problem, our approach allows developers to define DEOIReference test cases for their UIWidgets. Predefined test cases may be generated when the developer generates the augmenter source code, considering: kind of element, value, structure, and collection.

When the predefined test cases are not enough, developers may program new test cases by editing the generated source code. This is made by creating a subclass of the WATestCase using the name of the UIWidget plus the token "Test".

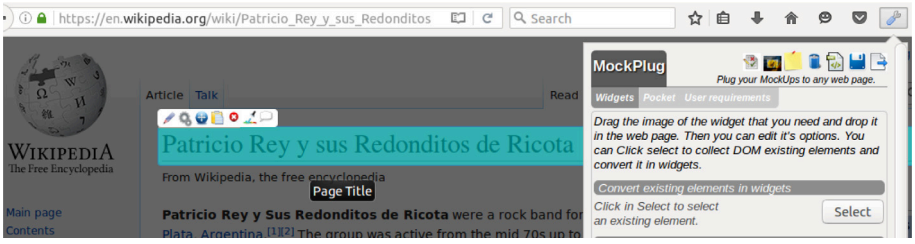

Fig. 12. Mockplug in action: Converting an existing DOM element into a UIWidget.

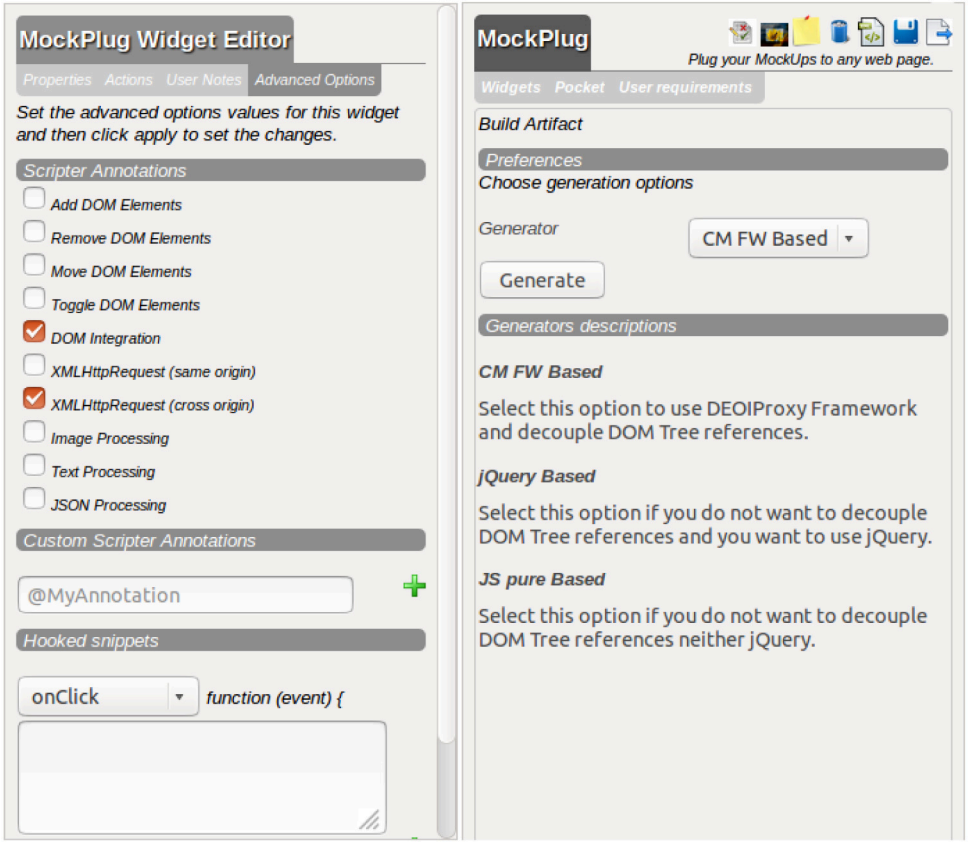

Fig. 13. Mockplug in action: advanced options for the UIWidget edition at the left and code generation options at the right.

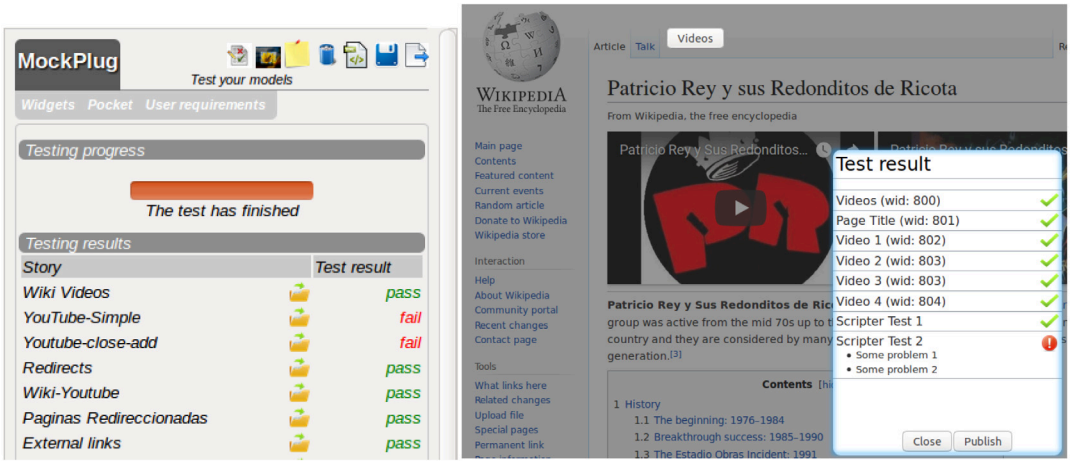

Fig. 14. Testing tool in action: developers' view at the left and end-users' view at the right.

5.3. Augmenter maintenance

Once the augmenter is completed and published, end-users may install it in their Web browsers. The first time an end-user installs an augmenter developed with our approach, the augmenter obtains the DEOIReferences from the repository and stores these on client-side, so the augmenter may work independently. When an augmenter fails because it cannot retrieve the corresponding DOM element, an advice is shown to the user.

If the users click on it, the EUD tool for DEOI references maintenance is opened, which is shown in Fig. 15. When an augmenter fails, the DEOIProxy framework looks for DEOIReferences upgrades in the repository. If there are any upgrade available, then the references are updated on client-side and tested. If there are no references to upgrade, this tool alerts to the users of which are the UIWidgets that are not working, or even those that could not be inserted because the DEOIReference is broken. These UIWidgets are shown as a little snapshot that would help users to visually recognize the element. Together with these, there are the corresponding buttons that will allow users to select the new DOM element. When the user tries to fix a DEOIReference, the test cases are run as a first control. If these test cases pass, they are reported to repository in order to repair the augmenter also for other users.

In our example, the broken references are those related to the UIWidget button (the broken references is the one used for inserting the button in relation to an original DOM element), and the reference related to UIWidget title (i.e., the original DOM element containing the article's title). The DEOI maintenance tool presents to the end-user those UIWidgets that could not be managed in the last execution of the augmenter. In the case of the "Videos" button, the tool asks to the user to drag and drop it to the corresponding position, and when it is done, the DEOIReferences are added. In the case of the "Title" UIWidget, the tool presents to the user the "selection" option (see sight icon), which will allow the end-user to select the DOM element again. In both cases, when DEOIReferences are created, the defined test cases are run, and if they pass, the user may upload the new DEOI references to the repository, so the rest of the users of this augmenter will receive them in their Web browsers.

6. Evaluation

The advantages of developing software through code generation have been shown in the past, even in the case of augmentation software [7,18]. Therefore, in this paper, we aim to evaluate our approach regarding the end-user and collaborative maintenance of DEOI references by comparing it against the traditional maintenance based on programming a new version of the augmenter, where the programmer has to provide code changes in order to repair the reference and then he/she has to redeploy the artifact.

The evaluation was set up as a controlled experiment where participants have used augmenters defined with both approaches (the traditional and with the DEOIProxy framework), and measuring the amount of times that an augmenter was executed with correct and broken references with each approach in a defined time lapse. In this type of experiment an observer tests a hypothesis looking for changes caused by alterations in a variable. An independent variable (in this case, the underlying development technique) is the main factor of which we want to measure the effect caused in the dependent variable (in this case, the execution results).

The experiment description is organized mainly according to the template proposed by Jedlitschka et al. [19].

6.1. Experiment planning

6.1.1. Goal

The goal of this experiment described according to the Goal /Question/Metric (GQM) method [20] is:

- Analyze *the presented approach and the traditional approach for maintaining augmentation artifacts*
- for the purpose of *evaluating the DEOIProxy approach*
- with respect to *maintenance effectiveness*
- from the point of view *of end-users*
- with respect to *restoring the correct functionality of augmentation artifacts when referenced elements change on the target Web site.*

6.1.2. Participants

A group of 48 people has participated in the evaluation: 8 of them were female and 40 male. All the participants were starting their careers in the Engineering Faculty at the Universidad Nacional de la Patagonia San Juan Bosco (UNPSJB). It is very important to mention that, at the moment of the evaluation, these students were starting their bachelors degree. Thus, they did not have skills on Web technologies or Web augmentation. These participants generated a sample with 611 cases.

6.1.3. Materials

To perform the experiment, it was necessary to use the following set of techniques, artifacts, and tools:

- **Techniques**: to maintain augmenters we used two ways, (*i*) the traditional one (i.e., source code modifications by the augmenter's developer) and (*ii*) our approach presented in this paper.
- **Augmenters**: we used two augmentations, one for Wikipedia and a second one for Youtube. We choose these two Web sites given that they are visited by participants daily in their normal use of the Web. For each augmentation, we prepared two versions: one using hard-coded DOM references by jQuery and another one using the DEOIProxy framework.
- **Programmed failures**: since it was a controlled experiment, to guarantee replicability, we have developed four small scripts that change the underlying HTML pages of Wikipedia and Youtube in order to simulate the upgrade of these Web sites and force a malfunction of augmenters.
- **Extension**: we developed a Firefox extension for being used by the participants. This extension asked personal information (name, email) at the beginning of the experiment and reminded participants to visit the target Web sites (Wikipedia and Youtube) if they did not do it in their normal use of the Web. The same extension was used for executing the programmed failure scripts.

6.1.4. Tasks

Participants were asked to perform the following tasks:

- Install the extension for running the experiment and the augmenters.
- Subscribe to the experiment.
- Visit the Web sites under study on their computers.
- Those using the proposed approach were also asked to use the DEOI maintenance tool when augmenters were broken.

6.1.5. Hypothesis and variables

The experiment has only one goal: *compare the DEOIProxy maintenance effectiveness, involving actively end-users for DEOI references maintenance, against traditional techniques, that just rely on their artifacts authors capacity to repair them when it is needed.* Therefore, we defined the following two hypotheses:

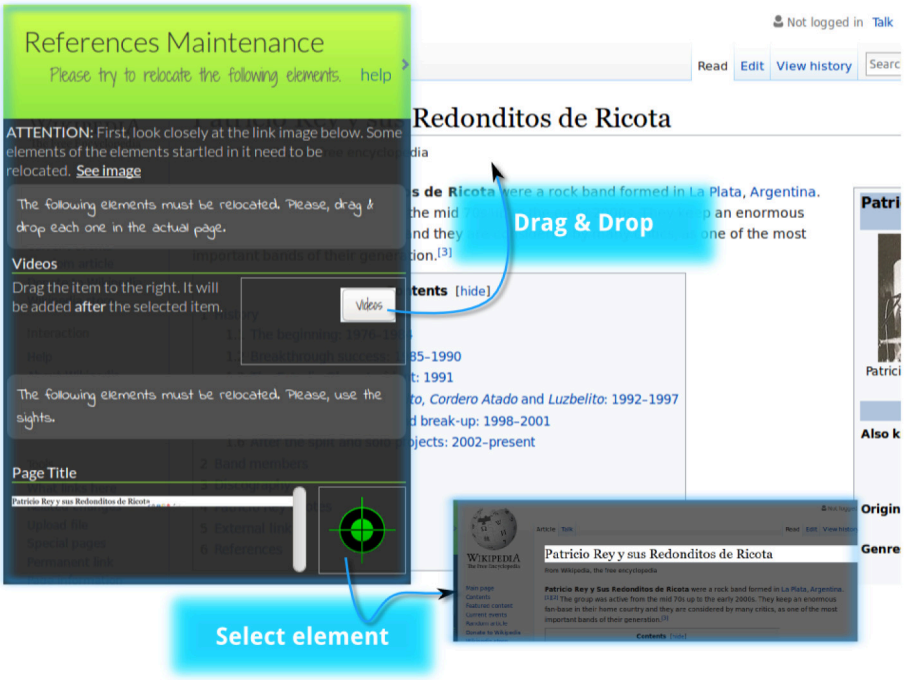

Fig. 15. EUP tool in action: Maintaining DEOI references.

- H0: There are no differences in the amount of failed executions of augmenters with the amount of successful executions whether the augmenters are maintained using the DEOIProxy framework or the traditional maintenance (by augmenter developer).
- Ha: There are differences in the amount of failed and successful executions when artifacts are maintained using DEOIProxy framework or using traditional maintenance (by the augmenters' developers).

With this in mind, we have defined as independent variable the underlying development and maintenance technique (i.e., using DEOIProxy framework and the traditional way to do it). The dependent variable is the execution result (failed, successful) in relation to the way in which augmenters are maintained. Finally, the augmenters themselves and the alterations in the Web pages' DOM to produce failures were defined as control variables.

6.1.6. Experiment design

The experiment had a simple between-subject design, where the participants formed four groups according to the two maintenance techniques and the two augmented websites to be used. The websites were YouTube and Wikipedia as these were considered very popular and well known. For traditional augmentation groups, only the users who created the augmenter artifacts could repair them in case of failures, and for the DEOIProxy-based, the subjects could participate in maintaining the DEOI references. In the two groups of traditional artifacts, there was the collaboration of an advanced user who pretended to be the developer of the artifact and had the responsibility to repair them when the failures occurred.

6.1.7. Procedure

Participants were invited to an initial meeting for explaining the overall goal of experiment. In this meeting, they installed the extension in their computers. When participants were subscribed to the experiment, one augmenter was assigned to them. In this way, given that 48 participants were involved, each augmenter was used by 12 participants. Participants were involved in the experiment by 10 days. These 10 days were planned, as Fig. 16 shows, as follow:

- Phase 1, days 1–2: The first two days, they used the assigned augmenters.
- Phase 2, days 3–6: In days 3 and 4 we insert the first programmed failure (by altering the underlying DOM in order to break the DOM references). After these two days, in days 5 and 6 we stop running the first programmed failure scripts, so the DOM came back to the original state.
- Phase 3, days 7–10: we did the same as in Phase 2, but introducing a second alteration in the DOM, forcing a new maintenance stage but different to the previous one.

Phases 2 and 3 were designed to simulate that those augmenters traditionally maintained are fixed by their developers in two days. That is the reason why these phases are of 4 days, 2 days for simulating the time lapse in which the augmenter is broken, and another two days to fix the augmenter. The same mechanism was used for the case of augmenters developed with the DEOIProxy framework. However, if any end-user fixes the augmenter during the first two days of one of these phases, the new DEOI references are added to the collection of DEOI references for that specific DEOI element, instead of replacing the existing one. Then, we expected that after turning off the programmed failure script of the phase, the augmenter still will work with the previously defined DEOI references. In this way, we were sure that augmenters (without taking into account if these were developed traditionally or using the DEOIProxy framework) were broken two times.

6.1.8. Analysis procedure

To test the hypotheses introduced before, we used the Chi-square test because the variables are qualitative, dichotomous, and nominal. Chi-square (also known as the Pearson Chi-square test) is a non-parametric statistical, distribution-free, hypothesis test that should be used when sample sizes of the study groups are unequal or the level of measurement of all the variables is nominal [21,22].

6.2. Execution and experiment tooling

The main tools to support the experiment were the following two: (*i*) an extension for the participants and (*ii*) a back-end component implementing a Restful API.

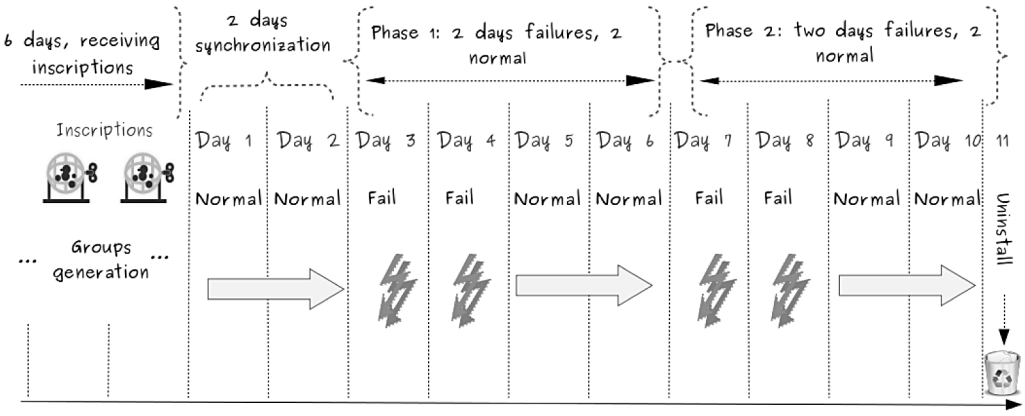

Fig. 16. Experiment planning.

Table 2
Execution details for each augmenter.

Augmenter	Youtube		Wikipedia	
	Traditional	DEOIProxy	Traditional	DEOIProxy
Successful executions	85	107	106	172
Failed executions	120	2	81	4

Table 3
Total of successful and failed execution per approach.

Executions	Traditional	DEOIProxy	Total
Failed	201	6	207
Successful	191	279	470
Total	392	285	677

Fig. 17 shows the sequence diagram of the experiment extension that enabled participants to complete the required tasks described in Section 6.1.4. Transparently, each time the target website was visited, the experiment extension would inject the programmed failures according to the experiment planning. These fails were JavaScript artifacts whose purpose was to change the original DOM references.

For the group of participants working with the DEOIProxy Framework, the JavaScript libraries corresponding to the DEOIProxy framework were injected. Finally, the framework had to execute the corresponding augmenter and prior to its execution, the corresponding reference tests were triggered. When these were not satisfactory, the participant will have the opportunity to provide new references through the maintenance tool described in Section 5.3.

In the case of the participants of the traditional artifacts, the extension was limited to inject the fails and then to inject the augmenters in the traditional version, which only differed from ordinary augmenters, in that after executing, they had to invoke a callback provided in order to record the success or failure of its execution. The artifacts, failures, and everything corresponding to the planning and execution of the experiment, such as cases of successful and failed executions in any of the groups, were stored in the back-end through an application developed and managed with the Django web framework.

6.3. Results

All the participants visited at least once the augmented Web sites. Youtube was visited 205 times using the traditional augmenter and 109 times using the augmenter developed with our framework. Wikipedia was visited 187 times using the traditional augmenter and 176 times using the augmenter developed with the framework. A total of 677 samples were obtained. Table 2 shows the results of executions for each augmenter and if these were successful or failed.

Fig. 18 shows the execution of augmenters developed with our approach day by day during the experiment. On the left hand side, we show the results for the Youtube augmenter, and the results for the Wikipedia augmenter are shown at the right hand side. Similarly, Fig. 19 shows the same detail for the augmenters developed traditionally. In both figures, we can observe the relation between successful and failed executions. As the reader may note, using our approach, once the programmed failure was launched, the broken references were fixed the same day, and consequently, a small amount of executions failed, because end-users could fix the problem. In contrast, for the traditional approach, for each programmed failure there were failed executions even after the developer fixed the augmenter, this is because in the execution of userscripts, the new version of the script is executed after restarting the Web browser. Finally, Table 3 illustrates the executions (failed and successful) for each kind of augmenter.

Regarding the contribution of end-users to maintain the augmenters using our approach, we can appreciate that for the Youtube augmenter, the first end-user that detected the problem also fixed the DEOI references using our tool. This implied that the users selected a DOM element compatible with the required one for augmenting and shared their solutions with the rest of end-users, who did not realize that there was a problem. In the case of Wikipedia, four failed executions happened, two for each programmed failure. Particularly, during the first phase, two end-users visited the Web site in the moment that the augmenter required maintenance. However, we could check that both end-users proposed solutions for the problem. During the execution of the Wikipedia augmenter in the second phase were a programmed failure was introduced, two end-users ran the broken augmenter, but only one of them contributed a solution.

6.4. Analysis

We analyzed the obtained samples during the phases 2 and 3, and we got the results shown in Table 3. We used the Chi-square test [21], which establishes that the null hypothesis must be rejected when the result is above to the established critical values in the distribution tables for the desired confidence level [23,24]. The tables indicate a critical value of $X2 = 6{,}635$ for a confidence level $p = 0{,}01$ and a degree of freedom $df = 1$.

For our contingency table (Table 3), the statistic X2 is equal to 185 and being this greater than the critical value (6,635), the null hypothesis is rejected. We can affirm with a confidence level of 99% that the traditional approach and the DEOIProxy approach generate different amount of failed executions.

If we analyze Fig. 19, we can easily appreciate that using the traditional approach, the amount of failures is related to the time lapse in which the augmenter was broken, as it was expected. In contrast, with our approach, the end-users had the opportunity to discover new references, fix the augmenter, and share the solution with other end-users, without any need of upgrading the augmenters' source code.

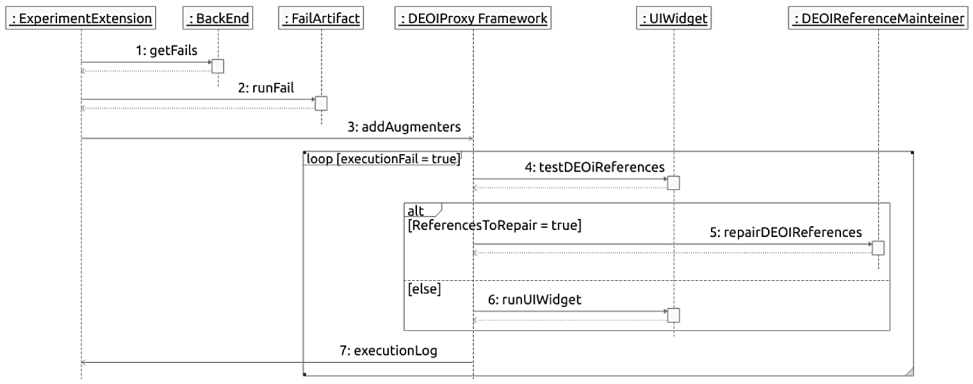

Fig. 17. Sequence diagram of experiment extension on DEOIProxy framework artifacts.

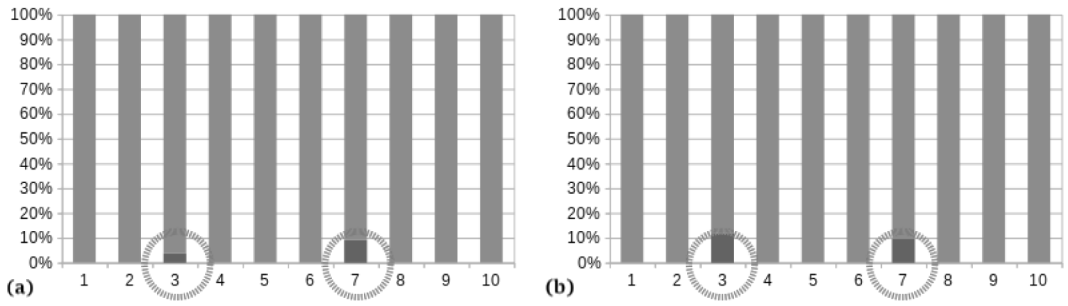

Fig. 18. Augmenter executions using DEOIProxy framework: (a) Executions in Youtube. (b) Executions in Wikipedia.

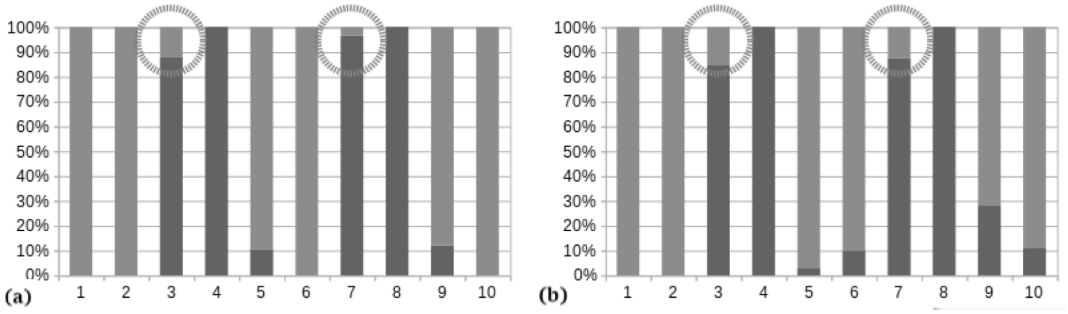

Fig. 19. Augmenter executions using traditional augmenters: (a) Executions in Youtube. (b) Executions in Wikipedia.

6.5. Threats to validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results. We categorize them as internal, external, construct, and conclusion validity threats [25].

Internal validity. We used the Chi-square test—as being non-parametric—which does not require specific distribution in the samples used. On the other hand, this test requires nominal variables, as they were obtained with the experiment, given that end-users could not interfere in the sampling process because the results were obtained automatically.

External validity. The augmenters used for the experiment are conceptually simple, but they contain the two most common ways of interacting with original DOM, i.e., (*i*) adding new elements created with JavaScript into the existing DOM and (*ii*) manipulating existing DOM elements. This was carefully defined for the two augmentations of both websites Wikipedia and Youtube.

As was described in the experiment's methodology, all the subjects were students from the engineering faculty who just started their career, with the same short training in the Web augmentation used in the experiments. All of them stated not having background on web technologies, i.e., they could not develop an augmenter using traditional client-side technologies. Thus, the results can be generalized for such participants.

Construct validity. In relation to the participants that had the opportunity of selecting new DEOI references for the corresponding approach, they had no knowledge about when the programmed failures were executed. Besides that, they had no experience with the approach neither tools. Also it is important to consider that these participants did not performed programming tasks during the experiment, they just had to select Web pages elements by means of mouse pointing and drag&drop, two mouse interactions deeply studied in the past [26,27].

An important issue is the diversity on the participants. Regarding participants, we consider that heterogeneity was warranted because they were recruited by an open call and assigned to augmenters randomly.

Although it is well known that volunteers are often motivated to participate, which can lead to biases, real users of this type of software artifact are aware of their execution and, being that these are used mainly for customization purposes, real users are naturally involved in its correct operation.

We have also avoided any bias by selection of the technique, because we warranted that the programmed failure scripts were executed during the same time lapse in both cases, for traditional and DEOIProxy augmenters.

Finally, we have to mention that the time lapse of 2 days used for simulating the time that the augmenter developer took for fixing the source code had been defined empirically by analyzing some maintenance evolution of augmenters, i.e., considering some real cases and counting that the mean for upgrades less than 10 days old is 3.15 days, as we point out in Section 3.

Conclusion validity. We have presented the results about the effectiveness and efficiency of the presented approach in comparison with the traditional approach. We have used the appropriate tests considering the analytical strategy. Given the results, the only way that traditional maintenance could reach the figures achieved with our approach, is that the developer is the first user of the augmenter who executed it after the Web site upgrade and the maintenance time (i.e., discover the error, fixing and upload the new version of the script), and it should happen in a very short time for avoiding that another end-user of the augmenter visits the Web site using the broken augmenter.

In relation to the hypothesis test used that requires a sample size of at least 30 [21], the participants generated a sample of 611 cases, so we consider that this experiment is statistically robust. Also its important to mention that we fulfill all the pre-conditions for using the Chi-Square test [21].

7. Related work

Our approach considers both the development and maintenance of augmenters. In this regard, we split the related work section; focusing on development first (Section 7.1), and in second step (Section 7.2) additional works related to the resilience of DOM locators plus approaches to make augmenters more resilient to Web page's DOM changes.

7.1. Augmenter development

Regarding the development process, there are different approaches. Probably, the first very popular approach to create augmenters was by programming userscripts using JavaScript. Currently, another format is by programming independent Web browser extensions, but in this case it is required to have even more technical knowledge. This development approach is very powerful from the point of view of what can be made with the augmentation layer, and how DOM references management is made in a very artisan way, as we show in previous section.

From academia, different approaches have emerged to facilitate the creation of augmenters. These approaches may combine the work of developers and end-users, or just put the end-users to work on the augmenter development directly. For instance, CSWR [28] is based on a framework for improving accessibility by adapting Web content on client-side. This approach allows developers (volunteers with JavaScript programming skills) to create generic artifacts which later may be instantiated for a specific Web site using a visual tool, a task that does not require programming skill. Another approach, called AccessMonkey, allows the creation of userscripts by annotating first the Web page [29]. In this case, the approach rests on volunteers that, without programming skills, may create userscripts for accessibility.

Programming-by-example is also a frequently used method. For instance, Procedures [30] defines a DSL that allows developers to automate inter-application tasks, allowing information migration among Web applications and the use of this information for triggering adaptations. Other approaches are based on DSLs as well. A similar approach is Koala [5]. Sticklet [31] proposes a DSL, which is more oriented to reduce the technical background required to program an augmenter.

Finally, several approaches are based on End-User Development (EUD) [2], and as it is pointed in the literature, the augmented Web is a very powerful ambient for EUD. A very common idea in this line is to convert the Web page into an editable canvas, and what vary from one approach to another one, is the construction over this idea. For instance, Marmite [4] is a mash-up oriented tool that allows end-users without programming skills to define contents integration's at the same time that integrates these mash-ups in the context of a particular Web site. Other approach called WebMakeUp [31] allows end-users to redefine the layout, edit text, and even perform some simple Web site integration's parameterized with DOM element values.

Although we believe that this kind of approaches are very powerful and help to put the augmented Web nearer to end-users, it is also true that EUP tools are not expressive enough to define a huge variety of augmenters.

It is worth to note that our approach for developing also takes advantage of the possibility to convert Web pages into a sort of canvas. However, we provide a high fidelity mockup tool that supports users to add elements to the DOM in order to define the main aspects of the manipulation, integration, and behavior of contents and UI controls. As we explained before, the tool offers the possibility of generating the source code based on our framework, that will enable end-user maintenance.

7.2. Reference maintenance

Behind the discussed approaches related specifically with Web augmentation, resides the problem of DOM locator resilience. On the one hand, a locator is defined as a mechanism for uniquely identifying an element on the Web Content once it is parsed and the DOM is created [32]. In this way, the locator resilience is a locator's attribute that determines how fragile the locator is in presence of DOM changes. There are works that propose algorithms for the construction of robust locators [33]. A common idea to improve the resilience is to manage redundancy, in order to have different ways to retrieve the same element, which is one of the strategies considered in our approach.

However, augmenters, and other kinds of software using locators, may fall in a complex maintenance stage that require to make compatible the locator with the new versions of the Web sites. As it is mentioned in existing literature [1], Web sites' DOM element dependencies were studied in near fields, such as Web testing, Web Annotation and Web scraping. In these fields, algorithms have been proposed which aim to reconstruct lost references or automatically adapt them to continue working [34,35].

Specifically for the augmented Web, two important works related to augmenter resilience are ModdingInterface [32] and ScriptingInterface [36]. ModdingInterface proposes a conceptual layer that defines concepts (basically a kind of annotation) and how these concepts match DOM elements of a particular Web site. In this approach, augmenters import a ModdingInterfaces, and by this, they do not have to use hard-coded DOM references. However, ModdingInterfaces are defined with Semantic Web technologies plus xPath expressions, and consequently, they cannot be easily maintained by end-users. Another disadvantage is that all these indirect references are defined in the same file, so references cannot be maintained separately. ScriptingInterface is a similar approach but does not require hosting these interfaces in a public Web server. The same authors have recently made a very interesting description of this problem [14]. In that work, these authors put special attention in regenerative locators, which beyond the structured data related to the target DOM element, also maintain contingency data that allows to recalculate the structured data when the underlying DOM structures change. They applied this idea for the context of Web augmentation, and they achieved a locator strategy that is capable to regenerate broken locators in 70% of cases when augmenters were malfunctioning.

Although our approach may incorporate these ideas in order to reduce the intervention of end-users, it is clear that there is not the silver bullet for locators. When this fragility may impact the confidence and adoption of particular software products, it is important to empower end-users with the possibility of collaboratively fix augmenters, reducing the dependencies to augmenter developers.

8. Conclusions and future work

The Augmented Web is very powerful and an interesting way to allow end-users to customize the way in which they surface the Web. However, it comes with particular challenges, given that both scenarios of production and testing are very diverse because of the big crowd of end-users and Web sites. This is even exacerbated because the adapted content on the client-side is not always the same for every user, given the features of modern Web applications, which depend on platforms and sessions. With this in mind, each augmentation session constitutes a production ambient in itself, and to allow the generation of public and automatized reports when the augmenter fails is determining. We strongly believe that a key point in the adoption of augmentation technologies is the correct working of augmenters. When augmenters fail because the target Web site has been upgraded, it is necessary to notice it, test if the augmenter is compatible with the current version of the DOM and, if it is not compatible, allow the crowd of users of the augmenter to fix the references in order to continue using it correctly as soon as possible. If this process is not supported by a methodological approach accompanied with specific tools, end-users may be frustrated and uninstall the augmenter before it is maintained by its developer.

In this paper, we proposed a method for developing augmenters without hard-coded references to Web sites' DOM and for generating the augmenter source code that includes a visual tool for end-users to maintain broken DEOI references. Our approach has been evaluated in comparison with traditional maintenance of augmenters. Considering a two days time lapse, the results show the convenience of our approach and the feasibility of maintaining augmenters by end-users.

Even though hard-coding data is a bad smell [37], as we also described for augmenters before, hard-coding references to augmented websites is currently an Achilles heel for augmenters. Our approach proposes to breathe life into these references by treating them as dedicated objects rather than data clumps. We strongly believe that our approach, where end-users are allowed to share and maintain collectively these DEOI references, facilitates the evolution of the augmenters which we have shown in our evaluation.

Adaptive development and maintenance have been highlighted in the context of user-centric software artefacts [38]. According with our study, userscripts are actually user-centric software artefacts because their developers are usually actual users of them, that share these artefacts in the community, reaching more users who contribute with the evolution of the augmenters by commenting on these repositories.

The kind of study presented in this paper is related to existing ones in mash-up communities, specifically for Yahoo! Pipes [39]. In that work, Stolee et al. discovered, that in these communities, configurable artifacts are desired. Also they conclude "here is a need for better end-user programmer support in several stages of the software life-cycle, including development, maintenance, search, and program understanding", which we have demonstrated that it is also important in the case of userscripts.

In the future, we plan to improve the support for adaptive or perfective maintenance of augmenters in order to support backward compatibility in those cases in which developers have written source code after generating the augmenter with our tool. Finally, also the tool supporting end-users for maintaining DEOI references may be improved in several ways to enhance usability, something that should be measured as well in the future.

CRediT authorship contribution statement

Diego Firmenich: Conceptualization, Methodology, Software, Data curation, Investigation, Data analysis, Writing – original draft. **Sergio Firmenich:** Conceptualization, Methodology, Writing – original draft. **Gustavo Rossi:** Writing – review & editing, Investigation, Supervision. **Manuel Wimmer:** Writing – review & editing. **Irene Garrigós:** Writing – review & editing. **César González-Mora:** Data analysis, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] O. Díaz, C. Arellano, The augmented web: Rationales, opportunities, and challenges on browser-side transcoding, Trans. Web 9 (2) (2015) 1–30.

[2] A.J. Ko, B. Myers, M.B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, The state of the art in end-user software engineering, ACM Comput. Surv. 43 (3) (2011) 1–44.

[3] G. Leshed, E.M. Haber, T. Matthews, T. Lau, C. Ave, H. Rd, S. Jose, CoScripter : Automating & sharing how-to knowledge in the enterprise, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2008, pp. 1719–1728.

[4] J. Wong, J. Hong, Making mashups with marmite: Towards end-user programming for the web, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2007, pp. 1435–1444.

[5] G. Little, T.A. Lau, A. Cypher, J. Lin, E.M. Haber, E. Kandogan, Koala: capture, share, automate, personalize business processes on the web, in: M.B. Rosson, D.J. Gilmore (Eds.), Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2007, pp. 943–946.

[6] D. Firmenich, S. Firmenich, J.M. Rivero, L. Antonelli, A platform for web augmentation requirements specification, in: Proceedings of the 14th International Conference on Web Engineering (ICWE), in: LNCS, vol. 8541, Springer, 2014, pp. 1–20.

[7] D. Firmenich, S. Firmenich, J.M. Rivero, L. Antonelli, G. Rossi, CrowdMock: an approach for defining and evolving web augmentation requirements, Requir. Eng. 23 (1) (2018) 33–61.

[8] C. González-Mora, I. Garrigós, S. Casteleyn, S. Firmenich, A web augmentation framework for accessibility based on voice interaction, in: M. Bieliková, T. Mikkonen, C. Pautasso (Eds.), Proceedings of the 20th International Conference on Web Engineering (ICWE), in: LNCS, vol. 12128, Springer, 2020, pp. 547–550.

[9] E. Marcotte, Responsive Web Design, Editions Eyrolles, 2011.

[10] M. Urbieta, G. Rossi, D. Distante, W. Schwinger, Managing volatile requirements in web applications, in: Proceedings of IEEE International Symposium on Web Systems Evolution (WSE), 2013, pp. 77–82.

[11] I. Aldalur, M. Winckler, O. Díaz, P. Palanque, Web augmentation as a promising technology for end user development, in: New Perspectives in End-User Development, Springer, 2017, pp. 433–459.

[12] C. Asakawa, H. Takagi, K. Fukuda, Transcoding, in: Y. Yesilada, S. Harper (Eds.), Web Accessibility - A Foundation for Research, second ed., in: Human-Computer Interaction Series, Springer, 2019, pp. 569–602.

[13] T. Karthikeyan, S. Karthik, D. Ranjith, V. Vinoth Kumar, J.M. Balajee, Personalized content extraction and text classification using effective web scraping techniques, Int. J. Web Portals (IJWP) 11 (2) (2019) 41–52.

[14] I. Aldalur, O. Díaz, Addressing web locator fragility: a case for browser extensions, in: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), 2017, pp. 45–50.

[15] E. Ferrara, P. De Meo, G. Fiumara, R. Baumgartner, Web data extraction, applications and techniques: A survey, Knowl.-Based Syst. 70 (2014) 301–323.

[16] O. Díaz, I. Aldalur, C. Arellano, H. Medina, S. Firmenich, Web mashups with WebMakeup, in: Rapid Mashup Development Tools - First International Rapid Mashup Challenge, 2015, pp. 82–97.

[17] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, second ed., in: Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2017.

[18] S. Firmenich, I. Garrigós, M. Wimmer, (De-)Composing web augmenters, in: Proceedings of the 14th International Conference on Web Engineering (ICWE), 2014, pp. 359–369.

[19] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting experiments in software engineering, in: Guide to Advanced Empirical Software Engineering, Springer, 2008, pp. 201–228.

[20] V.R. Basili, G. Caldiera, H.D. Rombach, The goal question metric approach, Encycl. Softw. Eng. 2 (1994) 528–532.

[21] M.L. Mchugh, The Chi-square test of independence Lessons in biostatistics, Biochem. Med. 23 (2) (2013) 143–149.

[22] S. Siegel, N.J. Castellan, Nonparametric Statistics for the Behavioral Sciences, second ed., McGraw–Hill, Inc., 1988.

[23] E.B. Wilson, M.M. Hilferty, The distribution of chi-square, Proc. Natl. Acad. Sci. USA 17 (12) (1931) 684.

[24] N. Pandis, The chi-square test, Am. J. Orthod. Dentofac. Orthop. 150 (5) (2016) 898–899.

[25] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.

[26] K.M. Inkpen, Drag-and-drop versus point-and-click mouse interaction styles for children, ACM Trans. Comput.-Hum. Interact. 8 (1) (2001) 1–33.
[27] I.S. MacKenzie, A. Sellen, W.A. Buxton, A comparison of input devices in element pointing and dragging tasks, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1991, pp. 161–166.
[28] A. Garrido, S. Firmenich, G. Rossi, J. Grigera, N. Medina-Medina, I. Harari, Personalized web accessibility using client-side refactoring, IEEE Internet Comput. 17 (4) (2013) 58–66.
[29] J.P. Bigham, R.E. Ladner, Accessmonkey: a collaborative scripting framework for web users and developers, in: Proceedings of the 2007 International Cross-Disciplinary Conference on Web Accessibility (W4A), 2007, pp. 25–34.
[30] S. Firmenich, G. Rossi, M. Winckler, A domain specific language for orchestrating user tasks whilst navigation web sites, in: Proceedings of the 13th International Conference on Web Engineering (ICWE), 2013, pp. 224–232.
[31] C. Arellano, O. Díaz, M. Azanza, A language for end-user web augmentation: Caring for consumers and producers alike, ACM Trans. Web 7 (2) (2013) 9:1–9:51.
[32] J. Iturrioz, O. Diaz, C. Arellano, Layman tuning of websites: facing change resilience, in: Proceeding of the 17th International Conference on the World Wide Web (WWW), 2008, pp. 1127–1128.
[33] M. Leotta, A. Stocco, F. Ricca, P. Tonella, Robula+: an algorithm for generating robust XPath locators for web testing, J. Softw.: Evol. Process 28 (3) (2016) 177–204.
[34] X. Meng, D. Hu, C. Li, Schema-guided wrapper maintenance for web-data extraction, in: Proceedings of the 5th ACM CIKM International Workshop on Web Information and Data Management (WIDM), 2003, pp. 1–8.
[35] E. Ferrara, R. Baumgartner, Design of automatically adaptable web wrappers, in: J. Filipe, A.L.N. Fred (Eds.), Proceedings of the 3rd International Conference on Agents and Artificial Intelligence (ICAART), 2011, pp. 211–217.
[36] O. Díaz, C. Arellano, J. Iturrioz, Interfaces for scripting: Making greasemonkey scripts resilient to website upgrades, in: Proceedings of the 10th International Conference on Web Engineering (ICWE), 2010, pp. 233–247.
[37] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.
[38] C. Pahl, Adaptive development and maintenance of user-centric software systems, Inf. Softw. Technol. 46 (14) (2004) 973–986.
[39] K.T. Stolee, S.G. Elbaum, A. Sarma, Discovering how end-user programmers and their communities use public repositories: A study on yahoo! pipes, Inf. Softw. Technol. 55 (7) (2013) 1289–1303.