End-User Browser-Side Modification of Web Pages

Oscar Díaz[1], Cristóbal Arellano[1], Iñigo Aldalur[1], Haritz Medina[1], and Sergio Firmenich[2]

[1] University of the Basque Country (UPV/EHU), San Sebastián, Spain
{oscar.diaz,cristobal.arellano,inigo.aldalur}@ehu.es
[2] LIFIA, Universidad Nacional de La Plata and CONICET, Argentina
sergio.firmenich@lifia.info.unlp.edu.ar

Abstract. The increasing volume of content and actions available on the Web, combined with the growing number of mature digital natives, anticipate a growing desire of controlling the Web experience. Akin to the Web2.0 movement, webies' desires do not stop at content authoring but look for controlling how content is arranged in websites. By content, we mainly refer to HTML pages, better said, their runtime representation: DOM trees. The vision is for users to "prune" (removing nodes) or "graft" (adding nodes) existing DOM trees to improve their idiosyncratic and situational Web experience. Hence, Web content is no longer consumed as canned by Web masters. Rather, users can remove content of no interest, or place new content from somewhere else. This vision accounts for a post-production user-driven Web customization (referred to as *"Web Modding"*). Being user driven, appropriate abstractions and tools are needed. The paper introduces a set of abstractions (formalized in terms of a domain-specific language) and an IDE (realized as an add-on from *Google Chrome*) to empower non-programmers to achieve HTML rearrangement. The paper discusses the technical issues and the results of a first validation.

Keywords: Web Modding, Web Widget, End User Programming, Visual Programming, Domain Specific Languages, WebMakeUp.

1 Introduction

Modding is a slang expression that is derived from the verb "modify". Modding refers to the act of modifying hardware, software, or virtually anything else, to perform a function not originally conceived or intended by the designer [19]. The rationales for modding should be sought in the aspiration of users to contextualize to their own situation the artefact at hand. This ambition is not limited to video games, cars or computer hardware. The need also arises for the Web. As an example, consider a TV-guide website (e.g. *www.tvguia.es*). For a given user, favourite channels might be scattered throughout the channel grid, hence, forcing frequent scrolling. In addition, users might move to other websites (e.g. *www.filmaffinity.com*) to get more information about the

B. Benatallah et al. (Eds.): WISE 2014, Part I, LNCS 8786, pp. 293–307, 2014.

scheduled movies. If *tvguia* is recurrently visited, this results in a poor user experience. Traditionally, this is addressed through Web Personalization, i.e. a set of techniques for making websites more responsive to the unique and individual needs of each user [3]. Similar to other software efforts, traditional personalization scenarios prioritize the most demanded requirements while minority requests are put aside. However, as a significant portion of our social and working interactions are migrated to the Web, we can expect an increase in "long-tail" personalization petitions. These idiosyncratic petitions might be difficult to foresee or too residual to be worth the effort. **"Web modding"** moves the power to the users. Web modding (hereafter, just modding) aims at Web content being consumed in ways other than those foregone by Web masters. Rather, users are empowered to rearrange Web content "after manufacture", e.g. removing content of no interest (leading to less cluttered pages while reducing scrolling) or placing new content obtained from somewhere else (reducing moving back and forth between sites so that a single viewing context is provided). The research question is how to achieve this empowerment.

This question admits different answers depending on the target audience. We frame our work along three main requirements: available time (30'), available expertise (no programming experience), and sparking motivation (improving the Web experience). This rules out fine-grained, absorbing programmatic approaches, and demands more declarative and abstract means. This is what Domain-Specific Languages (DSLs) are good for. DSLs are full-fledged languages tailored to specific application domains by using domain-specific terms. Domain abstractions are closer to how users conceive the problem, facilitating engagement, production and promptness. This work's contribution rests on the three pillars of DSLs applied to Web modding, i.e. ascertaining the right concerns (Section 3), finding appropriate DSL constructs to capture those concerns (Section 4), and finally, developing suitable editors that ease the production of DSL expressions (Section 5). The later is realized through *WebMakeup*, a *Google Chrome* extension that turns this browser into an editor for defining Web mods. *WebMakeup* is available at the *Chrome Web Store*: `https:// chrome.google.com/webstore/detail/alnhegodephpjnaghlcemlnpdknhbhjj`. Mods are exported as *Google Chrome* extensions that once installed, will transparently customize the page next time is visited. A first evaluation is provided in Section 6. We start by characterizing Web Modding.

2 Characterizing Web Modding through Related Work

Web Modding sits in between Web Personalization [17] and Web Mashup [20]. As a personalization technique, modding aims at improving the user experience by customizing Web content. There are also important differences. In Web Personalization, the website master (the "who") decides the personalization rules (the "how"), normally at the inception of the website (the "when"), preferentially using a server-centric approach (the "where"). By contrast, modding aims at

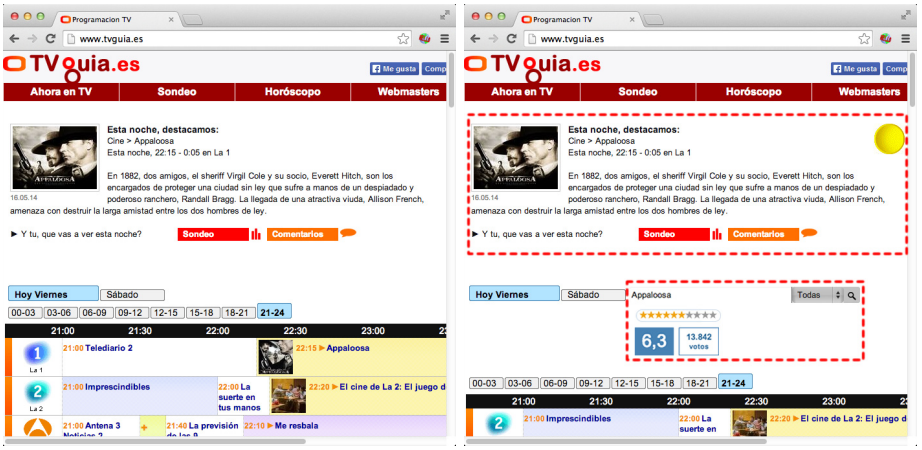

Fig. 1. *www.tvguia.es* before (left) and after (right) being modded: channel *"La 1"* is removed & *filmaffinity* ratings are introduced

empowering end-users (the "who") to rearrange Web content once in operation (the "when") by acting on the DOM tree (runtime realization of HTML pages) (the "how") at the client side (the "where"). Nevertheless, modding also shares similitudes with mashups: both tap into external resources. However, and unlike mashups, modding does not create a bright new website. Rather, it sticks with the modded website. Just like modding a car does not build a new car, modding a website does not create a new website but just operates on the browser side to change its DOM tree.

Web modding pays off for websites frequently visited but unsatisfactory Web experience. As an example, consider *www.tvguia.es*. This website provides the channel grid plus the-movie-of-the-day recommendation (see Figure 1 (left)). A user might just focus on some few channels, hence a thorough channel grip becomes a nuisance. In addition, content from other websites about the recommended movie might be of interest. Figure 1 (right) depicts a modded version: channel *"La 1"* is removed whereas additional content about the recommended movie is obtained from *www.filmaffinity.com*. The fragment extracted from *filmaffinity* is referred to as a *widget*, in this case, the *filmAffinity* widget.

The bottom line is that mod scenarios are characterized as being idiosyncratic, situational, and, potentially, short-lived, aiming not so much at synergistically combining third-party data (as mashups do) but improving the user experience of existing websites. Since these scenarios are very dependent on Web consumption habits and user interests, modding necessarily has to be do-it-yourself (DIY). This implies keeping the modding effort on a scale within the time and the skills of end users. This scale is a main driver in finding a balance between expressiveness (what can be modded) and effort (the cost of developing the mod).

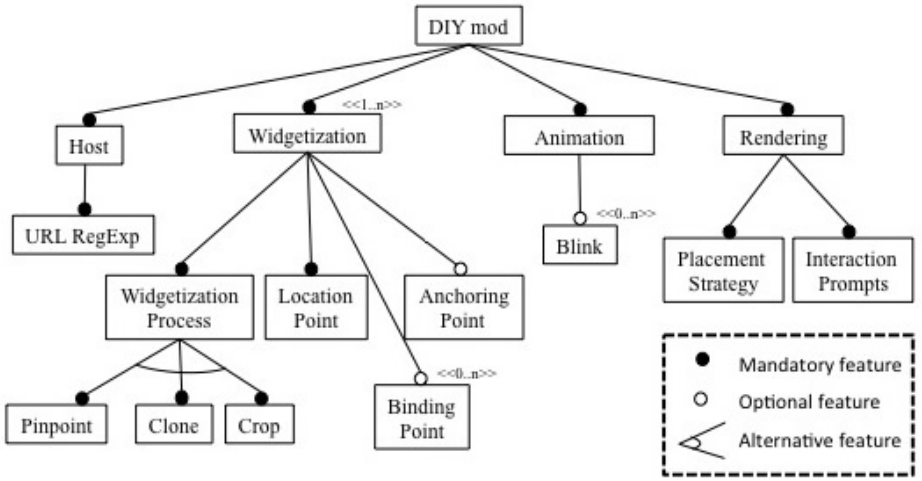

Fig. 2. Feature diagram for DIY Web Modding

Our target is for *Web Modding to be conducted by users with no programming skills in around 30 minutes.*

Implementation wise, modding implies browser-based programming. Modding is already possible for skilful JavaScript programmers but certainly outside the scope of end users [16]. This rules out fine-grained, absorbing programmatic approaches (e.g. *Chickenfoot* [1], *Co-Scripter* [12]), and calls for coarser grained, light-weight component-based standpoints. Unfortunately, most works on Web components (e.g. widgets) favours a programmer perspective, addressing the definition [18], implementation [6,9] and cloning of Web components [13,7]. A higher-level of abstraction is needed. Domain-Specific Languages (DSLs) come to the rescue. DSLs are full-fledged languages tailored to specific application domains by using domain-specific terms [8]. To increase the chances for DSLs to be adopted, three main landmarks stand out: ascertaining the right concerns, finding appropriate constructs to capture those concerns, and finally, developing appropriate editors that intuitively permit users to come up with DSL expressions. Next sections address each of these landmarks for modding.

3 Ascertaining the Right Concerns

Web Modding sits within the field of Web Augmentation [2], i.e. conducting changes upon the runtime representation of HTML pages (i.e. DOM trees) at the time the page is loaded into the browser. Those changes can affect the content, rendering, layout or dynamics of the page. Among the numerous uses of Web Augmentation, modding focuses on performing a function not originally conceived or intended by the host designer [5]. Finally, DIY modding addresses the empowerment of end-users to mod by themselves. As in other areas of

end-user design, more (expressiveness) can be less (usage). Therefore, DIY modding is necessarily going to be less expressive (i.e. more domain-specific) than general modding. We focus on improving the user experience through content rearrangement, i.e. content removal (leading to less cluttered pages) and content cloning, i.e. taking content from somewhere else (providing a single viewing context while cutting down moving back and forth between browser tabs). This sets the domain.

Along DSL good practices [14], concerns raised during DIY modding are captured as a feature diagram [10]. A feature diagram represents a hierarchical decomposition of the main concepts (i.e. features) found in the domain. The diagram also captures whether features are mandatory, alternative or optional. Figure 2 depicts the feature diagram for the domain "DIY modding". Issues include, **hosting** (i.e. setting the ambit of the modding), **widgetization** (i.e. the definition of widgets whose addition and removal shape the modding), **animation** (i.e. defining possible dynamics among the widgets), and finally, the **rendering directives** for the mod. Next paragraphs delve into the details (bold font is used for the features).

3.1 Hosting

A *mod* is a set of changes conducted upon the runtime representation of an HTML page at the time the page is loaded. Therefore modding does not happen in a vacuum but within the setting of an existing website, i.e. the **host**. The host can be characterised by a URL expression or a regular expression (e.g. www.amazon.com/*) so that all pages meeting the expression are subject to the mod. The expressiveness much depends on the target audience. For our purpose, we limit **url regexp** to those ending by "*". More complex expressions are not supported.

3.2 Widgetization

Modding is about customizing HTML content. HTML pages are conceived as DOM documents. The granularity at which HTML customization happens influences complexity. A finer-grained approach will certainly improve expressiveness but at the cost of complexity and learnability. Therefore, we opt for a coarser grained approach: *widgets*. For the purpose of this work, a widget is a coarse-grained DOM node (a.k.a. fragment), which accounts for a meaningful mod unit.

A widget can be defined *from scratch* through HTML and JavaScript. This is not possible for non-programmers. Alternatively, 3rd parties can help. But this also contradicts our setting that is characterized as being idiosyncratic, situational, and, potentially, short-lived, hence, the introduction of 3rd parties does not payoff. We are then forced to explore a different approach: *widget mining*. That is, users do not create widgets on their own but extract them from existing pages at the time the need arises. We then do not talk about widget creation but *widgetization* of existing code. To this end, we support tree variants: pinpoint, crop and clone.

Pinpoint supports inside-the-host widgetization, i.e. the widget is obtained from the host. In this case, extraction points hold the host's URL and a structure-based coordinate, i.e. an XPath expression that pinpoints the DOM node to be turned into a widget (see later). Widget *movie-of-the-day* is a case in point. It singularizes the DOM node that holds the content for the recommended movie. However, outside-the-host widgetization is more complex. A naive approach to extract existing functionality from a web page is just copy&paste. However, since HTML, CSS, and JavaScript are all "context-dependent", moving fragments from their original scope is rarely feasible. This moves us to the other two variants.

Clone is used for outside-the-host widgetization when the fragment to be extracted is "static", i.e. it holds content and style but not functionality (no JS scripts associated). The aim is for the widget to look like the raw content in the original page. Here, widgetization is achieved through cloning. Since style needs to be replicated, cloning is not limited to the selected DOM node but also its ancestors' CSS styles are inherited[1]. Since code is replicated, what if the original is upgraded? How are changes propagated to the replica? To this end, we introduce *refreshTimer,* a parameter that sets the refresh polling time to four possible values: onload (i.e. the widget is calculated every time the host page is loaded), daily, weekly or never.

So far, we assume widgets to be obtained from a single HTML fragment (**singleCloned**). However, the content of interest might be spread across different nodes. An interesting case is that of the Deep Web. Deep Web sources store their content in searchable databases that only produce results dynamically in response to a direct request. Here, the "meaningful functional unit" (i.e. the node to be widgetized) includes two fragments (**complexCloned**): the request fragment and the response fragment. The *filmAffinity* widget illustrates this situation. The "functional unit" includes not only the ranking table (i.e. the output) but also the search entry form to type the movie title. Hence, creating *filmAffinity* implies two extractions: one to collect the ranking table; another to obtain the entry form[2]. Last but not least, so-created widgets are parameterized by the form entries. This permits to fix some form entries (e.g. set "Gone with the wind" as the movie title) or even better, bind the entry to some data which is dynamically extracted from the hosting page at runtime (so called "binding points", see later).

Crop is used for outside-the-host widgetization when the fragment is "dynamic", i.e. it holds scripts. In this scenario, cloning does not work. Functionality is difficult to extract in an automatic way (refer to [13] for the difficulties on extracting JS code). Here, we resort to pixel-based cropping. Using iframes, it is possible to load the source webpage on the background. Next, the desired fragment can be addressed by referencing the height and width w.r.t the cropping start coordinates.

[1] *HTMLClipper* (`http://www.betterprogramming.com/htmlclipper.html`) is used to propagate replication from content to the associated CSS-like directives.

[2] Labelling a newly created *widget* with an existing name, makes the extraction engine glue them together and be offered as a unit (provided they come from the same page).

Once DOM nodes are turned into widgets, they start exhibiting some additional characteristics. Widgets can have parameters and a state (i.e. visible or collapsed). But most importantly, widgets might hold reference points, i.e. directives that refer to some location in terms of Web coordinates. We distinguish tree kind of reference points:

- **Location points**, which indicate from where the widget was obtained. They contain a Web coordinate plus the framing page.
- **Anchoring points**, which refer to the new setting where the widget is to be rendered, i.e. the position (i.e. before or after) w.r.t a given Web coordinate.
- **Binding points**, which denote how widget parameters can be bound to content from the host. It holds the name of the parameter and the host's Web coordinate. As an example, consider *filmAffinity*. This widget needs to be recalculated every time *guiaTV*'s movie-of-the-day changes. To this end, *filmAffinity* holds the *title* parameter. This parameter holds a binding point to the DOM node in *guiaTV* that keeps the title of the recommended movie. At runtime, the movie-of-the-day is recovered, and *filmAffinity* is dynamically computed after the current title.

Previous paragraphs refer to Web coordinates. A Web coordinate is a means to address content within a DOM tree (a.k.a. locators). For considerations about locators refer to [11].

3.3 Animation

Modding is about rearranging content. But this rearrangement does not need to happen in a single shot. Specifically, *widgets* can be in two states: visible or collapsed. When visible, widgets have the capacity to respond to events, such as keystrokes or mouse actions. When collapsed, widgets leave no trace in the screen. A *widget* has an initial state, i.e. the state at the time the hosting page is loaded (e.g. if visible, the widget is rendered as soon as the page is loaded). This state might be amenable to be changed by interacting with other *widgets*. A common approach for describing GUI dynamics is through statecharts [4]. However, statecharts are far too complex for our target audience. A simpler mechanism is needed.

Broadly, state changes can be described as event-condition-action rules. First studies, however, demonstrate that rules were a too fine-grained specification. Needed are higher abstractions that permit to capture recurrent patterns as a single construct. Based on previous evaluations, we noticed a recurrent animation pattern. Let's illustrate it with two widgets: *movieOfTheDay* and *filmAffinity*. Consider the later is to be made visible or collapsed upon mouse in/mouse out *movieOfTheDay*. This can be captured through a pair of rules:

> **ON** mouse-in *movieOfTheDay* **WHEN** *filmAffinity*.state = "collapsed" **DO** *filmAffinity*.state = "visible"
>
> **ON** mouse-out *movieOfTheDay* **WHEN** *filmAffinity*.state = "visible" **DO** *filmAffinity*.state = "collapsed"

We found this pattern so common that decided to introduce a DSL primitive for it: the blink. A ***blink*** accounts for a directed relationship between two widgets W1 and W2. We say *"W1 blinks W2"*, if acting upon W1 (e.g. clicking) causes W2 to change its state (from visible to collapsed or vice versa, depending on the W2 current state). Previous example can now be expressed as *"movieOfTheDay blinks filmAffinity on clicking"*. So far, we limit animation to *blinks*. *Blink* events are limited to *mouse-in* (being *mouse-out* its *blink* counterpart) and *click* (being *click* also its *blink* counterpart).

3.4 Rendering

Inlaying new widgets into an existing DOM structure can make the host's layout be disrupted. Specifically, HTML introduces some attributes to describe the rendering strategies for DOM nodes, namely: the layout strategy (HTML's "display" attribute) which can be arranging the content horizontally (inline) or vertically (block); minimum and maximum size intervals (HTML's attributes *minHeight*, *minWidth*, *maxHeight*, *maxWidth*); and the overflow strategy (HTML's "overflow" attribute) that indicates what to do in case the content exceeds the size intervals (i.e. make container scrollable, show the overflowed content or hide the overflowed content). Widget inlaying might disturb the page layout, causing one-dimension distortion or even worse, two-dimension distortion. We decide this concern to be hardwired within the DSL engine. Better said, the engine supports contingency actions to alleviate this situation (e.g. if container is 80% full then, WA overflow strategy is set to "warn"; if container is 90% full and the widget fits inside then, WA overflow strategy = "resize", etc.).

4 Finding Appropriate Constructs

Previous feature diagram captures main concerns to be solved during DIY modding. Next, these abstractions are realized in a language by looking into variabilities and commonalities in the feature diagram [14]. Variable parts must be specified directly in or be derivable from DSL expressions. In the first case, the variants become DSL constructs. However, some alternatives can be hardwired into the DSL engine as heuristics. Being heuristics, they might fail and hence, they are not as reliable as if provided by the user. The upside is that they simplify the user's life, hence, improving learnability and development. We decided *rendering* to be hardwired into the engine. That is, widget placement is to be assisted by the DSL engine. The rest of features are set by the user through the DSL. This section introduces the DSL metamodel.

Figure 3 provides the metamodel for mod description. A *mod* is a set of changes conducted upon the runtime representation of an HTML page (i.e. the host). These changes are described in terms of widgets. Widgets are characterized by a **locationPoint** (i.e. how to obtain it), an **anchoringPoint** (i.e. where to locate it), and, optionally, distinct **bindingPoints** (i.e. how widget parameters

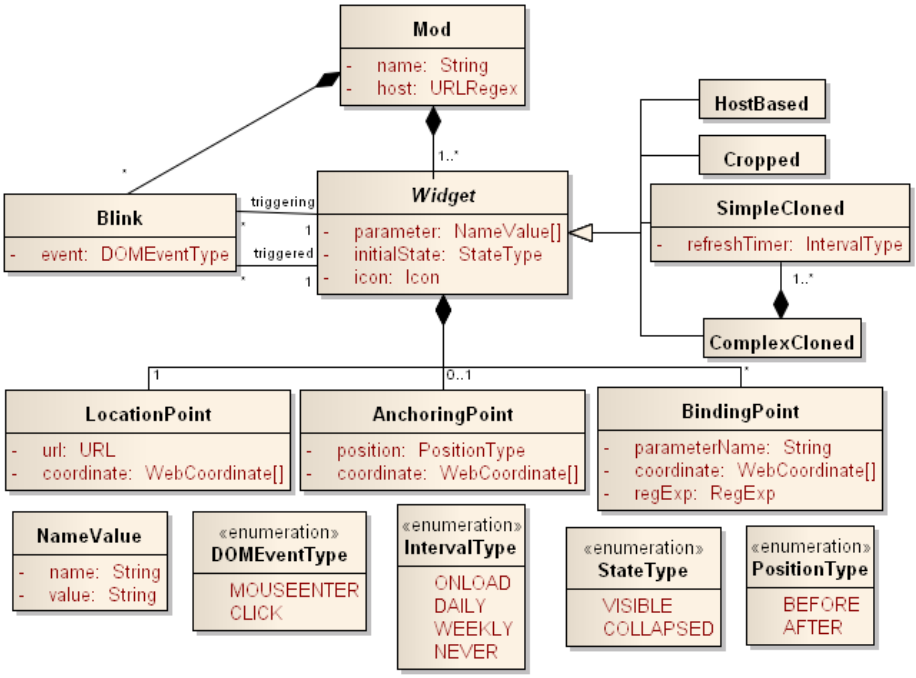

Fig. 3. A DSL for Web Modding: abstract syntax

can be obtained from the host's content). Each widget stands for a rearrangement operation as follows:

- If *locationPoint* exists without *anchoringPoint*, this accounts for content removal (only for host-based widgets).
- If *locationPoint* differs from *anchoringPoint*, this accounts for content displacement (only for host-based widgets).
- Otherwise, the widget captures content addition.

But not all contents need to be added/removed at loading time. *Blinks* permit to hand this decision over to the current user. This makes content rearrangement dependent upon user interactions. For instance, *"movieOfTheDay blinks filmAffinity on clicking"* permits to postpone till runtime the decision of rendering *filmAffinity*. If you click, you get *filmAffinity*. If complementary outside-the-host widgets exists (e.g. *filmIMDB* extracts the ratings from the IMDB website), then this content can be shown either simultaneously (e.g. *"movieOfTheDay blinks filmIMDB on clicking")* or in a cascade way (*"filmAffinity blinks filmIMDB on clicking"*). But not only additions, also removals can be left pending until interaction time: *"movieOfTheDay blinks movieOfTheDay on clicking"* permits current users decide whether they want to delete (i.e. collapse) *movieOfTheDay* by clicking on it. Next, we address how to make mods affordable to end users.

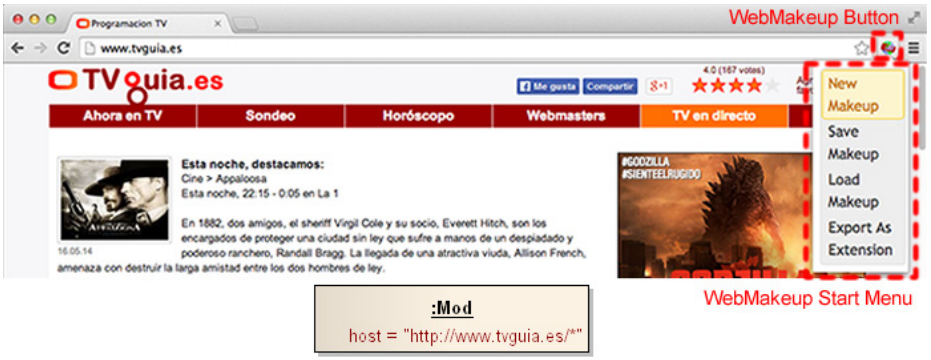

Fig. 4. *WebMakeup*: mod initialization

5 An Editor for DIY Mods

DSL acceptance is heavily influenced by the existence of appropriate editors, more to the point if targeting end users. This section outlines *WebMakeup*, an editor for DIY mods. This editor is available at the *Chrome Web Store*: `https://chrome.google.com/webstore/detail/alnhegodephpjnaghlcemlnpdknhbhjj`. *TVguia* is used as an example. The description goes along the creation of a *mod*, i.e. a model conforming to the metamodel presented in the previous section. A demo video is available at `http://onekin.org/downloads/public/WebMakeup/video.mov`.

Mod creation (Figure 4). *WebMakeup* is a plugin for *Google Chrome* browser. Its installation is reflected by the *WebMakeup* button at the right of the address bar. On clicking this button, a scrollable menu pops up. By clicking *"New makeup"*, the user initializes the mod model (Figure 4 (bottom)). *WebMakeup* turns the current page into the editor canvas: the pointer is turned into a camera, a grid-like structure is interspersed on top of the current DOM tree, and the *piggyBank tab* pops up.

Mod populating (Figure 5). A widget is a DOM node but not all DOM nodes are widgets. We need to singularize the selected DOM node that accounts for a meaningful HTML fragment. Meaningfulness is not inferred by the tool but indicated by the user. To this end, and, as the user moves the cursor around the screen, the DOM node under the current cursor location is highlighted. By clicking, the user singularizes this node as a meaningful HTML fragment, i.e. a widget. A nuisance is the handling of "hidden nodes". These nodes are those that do not have a graphical counterpart and hence, they cannot be pinpointed through the cursor. For instance, a table row (*<tr>*) is graphically hidden if its graphical space is totally taken by its content. If the row does not explicitly have some graphical counterpart (e.g. a border), then all the space is occupied by the row's content so that the cursor will always select the row's content rather than the row element itself. To overcome this problem, we resort to the keyboard. Keys *"w"*, *"s"*, *"a"* and *"d"* help to move up, down, left and right along the DOM tree, respectively, w.r.t to the node being pinpointed by the cursor.

Fig. 5. *WebMakeup*: mod filling up. The *piggyBank* tab is displayed.

No matter the selection mechanism (i.e. cursor vs. keyword), the selected node is surrounded by a decorator. This decorator permits to set the initial widget state by clicking on the "eye" icon (decorators' upper left-hand side corner): visible (open eyes) & collapsed (closed eyes). The example contains two inside-the-host widgets (i.e. *movieOfTheDay* and *TVE1channel*) and outside-the-host widget (i.e. *filmAffinity*). The latter is dragged&dropped from *piggyBank*[3]. Click on this tab to expose the widgets collected from other pages (see it in display in

[3] Outside-the-host widgets can be obtained at any time. To this end, the right-click contextual menu is extended with the *widgetizeIT* item. At any time, select it for a grid-like structure to be interspersed on top of the page you are looking at. As the user moves the cursor around the screen, the DOM node under the current cursor location is highlighted. By clicking, the selected node is turned into a widget and kept in the extension's variable: *piggyBank*.

Fig. 6. *WebMakeup*: defining *blinks*

Figure 5). Placement heuristics will warn or prevent from dropping widgets in certain places. In all cases, *WebMakeup* works out the Web coordinates.

Mod enhancement (Figure 6). At any time during editing, widgets can be:

- Deleted. Widget removal is achieved by clicking upon the X icon on the widget decorator. In the example, we remove *TVE1channel*. Model wise, this is reflected by deleting its anchoring point. An important remark: banners cannot be removed. Though this is a common desire among users, up to 84% of the top 100 websites rely on advertising to generate revenue [15]. Though adverts can be a nuisance, they are the ones that pay the bill. So for the time being, we take the decision of making *WebMakeup* ad-friendly.
- Rearranged. This is conducted through drag&drop once the widget is selected. Model wise, this is reflected as an update on the anchoring point.

– "Blinked". *Blinks* are graphically represented through pipes. Widget decorators have in their right-hand side a yellow circle. This circle denotes a pipe start. Click and drag from this point to expand till reaching another widget. This sets a blink from the triggering widget (the pipe's start) to the triggered widget (the pipe's end). An entry field on top of the pipe serves to indicate the blink's event. Figure 6 illustrates the case *"movieOfTheDay blinks filmAffinity on clicking"*.

Once the edition finishes, the mod can be exported as a Chrome extension. Once the extension is installed, the mod will be automatically enacted next time the host page is loaded. For our running example, the generated extension is available at `http://onekin.org/downloads/public/WebMakeup/extension.zip`.

6 Usability Evaluation

ISO definition of usability (ISO 9241-11, Guidance on Usability (1998)) refers to the extent to which a system (e.g. *WebMakeup*) can be used by specified users (e.g. end users) to achieve specified goals (e.g. content re-arrangement) with effectiveness (e.g. mod completion), efficiency (e.g. 30') and satisfaction in a specified context of use (e.g. browsing sessions). This section provides first insights not only about *WebMakeup* but also about the satisfaction of users on the result of the mod.

Research Method. The study was conducted in a laboratory of the Computer Science Faculty of San Sebastián. Before the participants started, they were informed about the purpose of the study and were given a brief description of it (5 minutes). Then, a *WebMakeup* sample was presented to illustrate the main functionality of the tool. The sample mod adapts a conference website by removing the logo of the conference and adding information about the weather forecast and information about the authors obtained from the DBLP. Next, participants were handed out a sheet with the instructions to create a new mod similar to the one used here as a running example. Participants were asked to write down the time when *"New WebMakeup"* button is clicked and again when they saw the augmentation. Last, participants were directed to a *Google Forms* online questionnaire.

Ten students participated in the study. The majority of participants were male (80.0%). Regarding age, 80.0% were in the 20-29 age range and all participants were below thirty five years old. Concerning the participants' browsing behaviour, 80% accesses to more than 10 websites every day and in the last year participants had installed between 2 and 20 applications/plugins/add-ons, with a mean of 6.5.

An online questionnaire served to gather users' experience. It consisted of four parts, the first one to gather the participants' background, another one to measure the satisfaction, other one to effectiveness and the last one to measure the productivity. In order to evaluate effectiveness, the questionnaire contained the proposed tasks so that participants could indicate if they had performed them, while productivity was measured using the minutes taken in such tasks.

Table 1. Satisfaction results from 1 (completely disagree) to 5 (completely agree)

Item	Mean	St. Dev.
1. I found the tool easy to use	3.4	0.966
2. I have made all the things that I wanted	3.1	0.994
3. I have always known how to do the things	2.1	0.738
4. There was no errors	4.0	0.817
5. It is fast	3.9	0.850
6. I am satisfied with the things I made	4.1	1.197
7. Removing content improves my Web experience	3.2	1.174
8. Adding content in a single view improves my experience	4.5	0.699
9. Demo is interesting to be told to friends	3.9	1.229

Satisfaction was measured using 9 questions, respectively, with a 5-point Likert scale (1=completely disagree, 5=completely agree). Descriptive statistics were used to characterize the sample and to valuate the participants' experience using *WebMakeup*.

Results. All participants but one successfully created the proposed augmentation. Those who successfully ended the sample took between 19 and 35 minutes with a mean of 24.2 minutes to fulfil the task. Table 1 shows scores for the satisfaction survey. As for the tool itself (items 1 to 5), subjects were reasonably happy. A shortcoming detected during the experiment was the lack of facilities to store work-in-progress mods. So far, *WebMakeup* forces to obtain the mod in a single session. Also, two subjects found the *blink* relationship misleading. On the upside, most of users finished under 30'. As for the notion of modding itself, subjects found content deletion and content rearrangement effective means to improve their web experience (items 7 and 8). Interesting enough, providing a single viewing context was found more interesting than content removal. In general, users found the experience rewarding (items 6 and 9).

7 Conclusions

Webies 2.0 no longer take the Web as it is but imagine fancy ways of customizing it for their own purposes. This work presents our vision for DIY modding along three main requirements: available time (30'), available expertise (no programming experience), and spark motivation (improving the Web experience). These requirements ground a coarse-grained, light-weight approach to DIY modding that is so far limited to content rearrangement. A fully-working editor, *WebMakeup*, demonstrates the feasibility of this vision. First evaluation is encouraging about the potentiality of Web Modding to improve the Web experience, and hence, the need for tools that make this vision possible.

Acknowledgments. This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839. Aldalur has a doctoral grant from the Spanish Ministry of Science & Education.

References

1. Bolin, M., Webber, M., Rha, P., Wilson, T., Miller, R.C.: Automation and Customization of Rendered Web Pages. In: UIST 2005, pp. 163–172 (2005)
2. Bouvin, N.O.: Unifying Strategies for Web augmentation. In: HyperText 1999, pp. 91–100 (1999)
3. Cingil, I., Dogac, A., Azgin, A.: A Broader Approach to Personalization. Communications of the ACM 43(8), 136–141 (2000)
4. Daniel, F., Furlan, A.: The interactive API (iAPI). In: Sheng, Q.Z., Kjeldskov, J. (eds.) ICWE 2013 Workshops. LNCS, vol. 8295, pp. 3–15. Springer, Heidelberg (2013)
5. Díaz, O., Arellano, C.: The Augmented Web: Rationales, Opportunities & Challenges on Browser-side Transcoding. ACM Transactions on the Web (2014)
6. Ennals, R., Brewer, E.A., Garofalakis, M.N., Shadle, M., Gandhi, P.: Intel Mash Maker: Join the Web. SIGMOD Record 36, 27–33 (2007)
7. Firmenich, S., Winckler, M., Rossi, G., Gordillo, S.E.: A Crowdsourced Approach for Concern-Sensitive Integration of Information across the Web. Journal of Web Engineering 10(4), 289–315 (2011)
8. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
9. Han, H., Tokuda, T.: A Method for Integration of Web Applications Based on Information Extraction. In: ICWE 2008, pp. 189–195 (2008)
10. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University (1990)
11. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Visual vs. DOM-Based Web Locators: An Empirical Study. In: Casteleyn, S., Rossi, G., Winckler, M. (eds.) ICWE 2014. LNCS, vol. 8541, pp. 322–340. Springer, Heidelberg (2014)
12. Leshed, G., Haber, E.M., Matthews, T., Lau, T.: CoScripter: Automating & Sharing How-To Knowledge in the Enterprise. In: CHI 2008, pp. 1719–1728 (2008)
13. Maras, J., Stula, M., Carlson, J., Crnkovic, I.: Identifying Code of Individual Features in Client-Side Web Applications. IEEE Transactions on Software Engineering 39(12), 1680–1697 (2013)
14. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys 37, 316–344 (2005)
15. PageFair. The Rise of Adblocking (2013), `http://blog.pagefair.com/2013/the-rise-of-adblocking/`
16. Pilgrim, M.: Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox. In: Getting Started, 12. Avoid Common Pitfalls, ch. 1, pp. 33–45. O'Reilly (2005)
17. Rossi, G., Schwabe, D., Guimarães, R.: Designing Personalized Web Applications. In: WWW 2010, pp. 275–284 (2001)
18. W3C. Requirement For Standardizing Widgets (2006), `http://dev.w3.org/2006/waf/widgets-reqs/`
19. Wikipedia. Modding (2014), `https://en.wikipedia.org/wiki/Modding`
20. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding Mashup Development. IEEE Internet Computing 12, 44–52 (2008)