Web Mashups with WebMakeup

Oscar Díaz[1], Iñigo Aldalur[1], Cristóbal Arellano[1], Haritz Medina[1], 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. 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. The rationales for modding should be sought in the aspiration of users to contextualize to their own situation the artifact at hand. Websites are not exception. *WebMakeup* targets mod scenarios where web pages are turned into canvases users can tune to account for their situational, idiosyncratic, and potentially, short-lived needs. By clicking, users turn DOM nodes into widgets. Widgets can next be rearranged, deleted, updated or stored for later reuse in other pages. In addition, widgets can be involved in "blink" patterns where interactions with a widget might affect the related widgets. This empowers users to tune not only *what* but also *when* content is to show up in an AJAX-like way. *WebMakeup* is publicly available as a Chrome extension.

1 Context and Goals

A mashup has been defined as "a composite application developed starting from reusable data, application logic, and/or user interfaces typically, but not mandatorily, sourced from the Web" [1]. It has been observed that mashups tend to be limited in their scope, addressing what is being referred to as *the long tail* of the software market whose limited demands and/or benefits make mashups fall outside mainstream applications [1]. This observation rises the question of *who* develops mashups, i.e., the profile of those addressing *the long tail*. Hence, it is relevant to start by first characterizing this audience. Differences between mashup tools frequently rest on the different user profiles being targeted. In other words, tool success very much depends on the accuracy to which these profiles are pinpointed.

Our mashup scenario is characterized as being situational, idiosyncratic and, potentially, short-lived. These aspects challenge traditional software development, and shift the focus from professional programmers to hobby programmers or even, laymen. This changes the rules of the game. Available time, available skills or motivation greatly differ depending on the target developers. For professional programmers, development takes place in a working setting where time and skills are assumed, and motivation is turned into duty. This setting

changes when development is handed over to laymen. It might well be part of work or not. Some support might be available but most of the time, development is conducted on layman's own account. Basically, we characterize our target audience (i.e. the mashup developer) along three features:

1. available expertise: no programming experience. Our target audience should not need to known HTML, APIs, JavaScript or other programming environment in which mashup are realized.
2. available time: 30'. The expectation is for the mashup to be developed in around 30'
3. sparking motivation: improving the Web Experience.

Broadly, our approach can be characterized as follows. First, and unlike traditional mashup approaches, we do not aim at creating a brand new application (the mashup) but customizing an exising one. Second, we do not consider any kind of data source but HTML pages. The term "modding" is used to refer to the possibility of users to tune HTML content and interactions to fit their own patterns. The ultimate goal is improving the User Experience (UX). This is achieved through *modding* mashups (here after referred to as "mods"). This vision accounts for a post-production (i.e. once the modded website is in operation), user-driven Web customization. This paper describes *WebMakeup*, a Chrome plug-in for mod development. Specifically, we focus on the mashup side of *WebMakeup*, i.e. how *WebMakeup* allows for copying HTML fragment from the Web to be later pasted into the modded website. A more complete account of *WebMakeup's* functionality can be found at [3]. This paper focuses on the case study at the Mashup Contest held at the International Conference on Web Engineering (ICWE) in 2015.

2 A mod scenario

Consider a layman browsing *The New York Times* website (NYT). What can make him mod this website? Better said, how strong should this mod desire be for the user getting down to work and develop a mod? Although motivations vary, a common source of discomfort is when other websites need to be visited. This might involve opening new tabs, and moving back and forth between different tabs. This makes the user loose focus and break the reading flow. Consider three scenarios when reading the NYT (see Figure 1):

1. the user is a frequent traveller between Amsterdam Central Station and Rotterdam Central Station. Periodically, the user checks when the next train leaves. NYT is often read while waiting at the train station. Checking next train, involves googling in a new tab,
2. the user is a broker. He needs to keep an eye on share prices even when reading the newspaper,
3. the user likes to check how headlines are covered by media other than the NYT (e.g. NBC news).

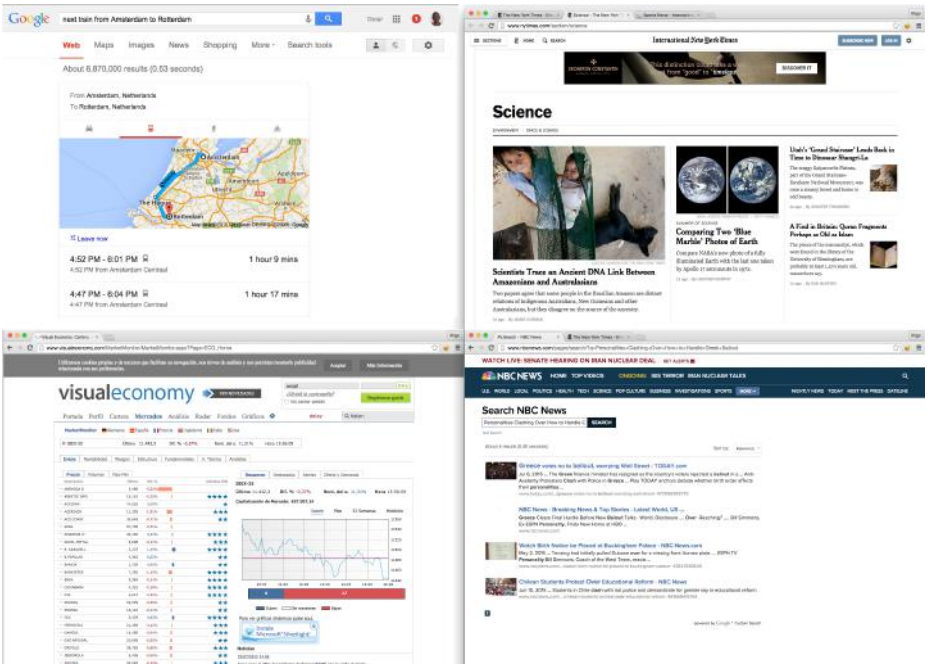

Fig. 1. Websites accessed while reading the NYT (in clockwise order): Google Search, Science section of the NYT, Visual Economy, and search facility at the NBC website.

4. the user is interested in two sections of the NYT: Science and Sports. He doesn't always check them fully but like to have a glance to the headlines in these sections

These scenarios involve a tab shifting from the NYT website to other websites. Despite its simplicity, the few clicks involve might well break the reading flow. This is not a main discomfort except if conducted in a regular basis. If you are a frequent train traveler, working as a broker, curious about NBC news coverage, or interested in Science and Sports, tab shifting might be a main discomfort in your UX when accessing the NYT website. Modding might help by moving scattered Web content to the website when the main task is conducted, in this case the NYT website. Next section addresses how this NYT scenario can be tackled by *WebMakeup*.

3 A session with *WebMakeup*

WebMakeup is both an editor and an engine for Web modding. As an editor, it offers a GUI for obtaining mods. As an engine, it interprets mods, and modifies the target page accordingly. *WebMakeup* is available at the *Chrome Web Store*: `https://chrome.google.com/webstore/detail/alnhegodephpjnaghlcemlnpdknhbhjj`.

Fig. 2. The *WebMakeup* scrollable menu.

Usability studies were conducted and reported at [3]. This section describes the creation of a mod for supporting the NYT scenario.

The process starts by the user focusing on the website causing the disconfort. If disconfort is due to visual clutter, then he can start by removing some content. If disconfort is due to disperse content, then he can start by singling out this content, and somehow making it appear at the host website. Finally, he should decide whether all content should be readily available or rather, become visible provided some user interaction occurs. More specifically, this notion of content *that is singled out to be operated upon* is captured in terms of a "widget". For our purposes, a widget is basically an HTML fragment that is being singling out and equipped with some operations and additional meta-properties. Therefore, modding is achieved in terms of widgets, specifically, through four main interventions: widget creation, widget mining, widget handling and widget animation. Next subsections present each intervention with the help of the running example.

3.1 Widget Creation

WebMakeup is a plugin for *Google Chrome*. Its installation is reflected by the *WebMakeup* button at the right of the address bar. On clicking this button, a scrollable menu pops up (see Figure 2). Clicking on *"New"* causes the following effects:

1. the current page is turned into an editor canvas where the pointer is turned into a camera,
2. a grid-like structure is interspersed on top of the current DOM tree, and
3. two tabs pops up: the *piggyBank* tab and the *patterns* tab.

By mousing over the page, the underlying DOM nodes are highlighted. By clicking, the user singles this node out as a meaningful HTML fragment, i.e. a widget. A limitation is the handling of "hidden nodes", i.e. DOM nodes 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

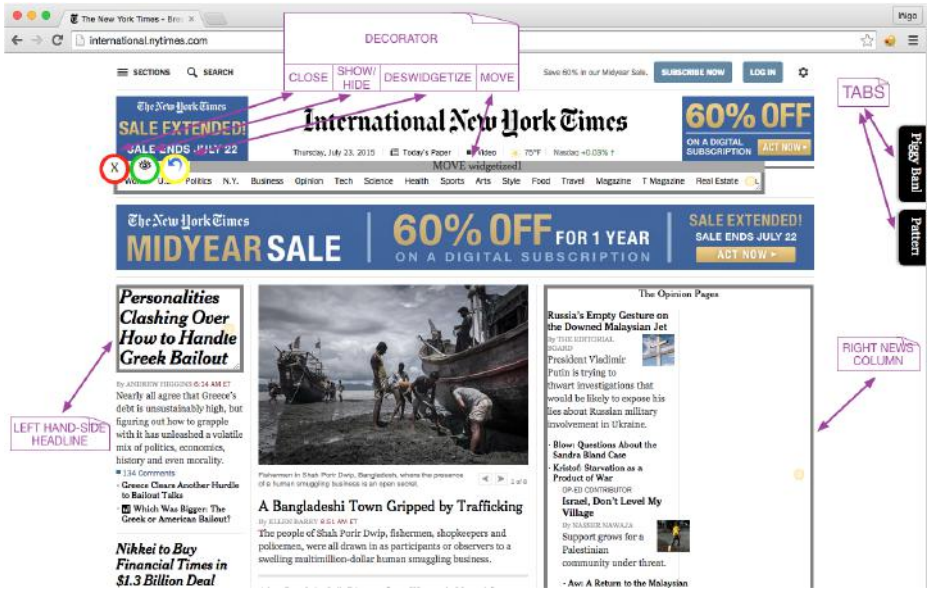

Fig. 3. DOM nodes are turned into widgets. A decorator permits to operate upon the widget: remove, visibility-state modification and un-widgetization (i.e. turning back to a mere DOM node).

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.

No matter the mechanism (i.e. cursor vs. keyword), the selected node is surrounded by a decorator. In other words, the HTML fragment is turned into a widget, and hence, amenable to be manipulated. Figure 3 depicts the DOM nodes from the NYT website once three DOM nodes are turned into widgets, namely, *linkBar*, *headline*, and *rightColumn*. Broadly, widgets are "those page chunks" to be operated upon in order to be deleted, re-allocated or changed in some of its content. But before moving to widget handling, it is important to note that widgets are not limited to those of the modded page (e.g. the NYT page) but they can be obtained from any place in the Web sphere. This moves us to widget mining.

3.2 Widget Mining

For our purposes, Web mashuping involves putting together otherwise scattered Web content. The basic aim: avoiding tab switching and, in some case, copy&paste operations between websites. In the NYT example, we aim at

offering train information, stock exchange data or headlines for other newspapers, all without leaving the NYT page. In this example, Google (for the train information), Visual Economy (for the share prices) or NBC (for the headlines) act as the information providers. This information is supported in terms of HTML pages in their respective websites. Therefore, the process goes along a similar pattern as the one described in the previous section, i.e. HTML fragments are turned into widgets. Nevertheless, some subtle differences exist.

Widgets can be mined any time while browsing, not just when creating the mod. To this end, the right-click contextual menu is extended with the *mineIT* item (see Figure 4). When you come across with a content of interest, select it, and a grid-like structure will be interspersed on top of the current page. Due to mouse hovering, the DOM node under the current cursor location is highlighted. Once the desired node is highlighted, click *mineIT* to be prompted to name the just-created widget. So, mined widgets are kept in the *PiggyBank*, a clipboard-like facility that is later reachable through the *PiggyBank* tab (see later).

Worth noting, a mined widget might stand not just for a single node but a set of nodes can be agglutinated upon the same widget as long as all come from the very same page. Just provide the same widget name, and the highlighted node will be merged with the existing widget's structure. The NBC widget is a case in point (see Figure 4). It aggregates the search bar and the node standing for the first answer. When inlayed, this widget will allow to obtain the first answer without leaving the NYT page.

Besides *NBC*, the running example extracts four widgets (see Figure 5): namely, *sportHL* from `http://www.nytimes.com/pages/sports/international/index.html`, *scienceHL* from `http://www.nytimes.com/section/science`, *stockMarket* from `http://www.visualeconomy.com/`, and *trainData* from googling "next train from Amsterdam to Rotterdam Central Station". These widgets will be available in the *PiggyBank* clipboard.

From a users perspective, all widgets are obtained in the same way, i.e. through the contextual menu. However, their internal representation might greatly differ based on the underlaying HTML code. Though the technical details are outside the scope of this work [3], readers can gain some insights by looking at the previous examples:

- *sportHL* and *scienceHL*, capture static and always-visible content,
- *trainData* holds also static content but some parts are initially hidden (e.g. trip details) and become visible after some user interaction,
- *stockMarket* holds dynamic data. Share prices are continuously being updated, i.e. frequent server requests are needed to keep the content in sync.

These examples serve to get an insight into the complexities of widgetization. It is rarely the case that just cloning the DOM node will do. More often, CSS and associated JS scripts should also be considered.

3.3 Widget Handling

Previous subsections illustrate how widget can be obtained from the modded page itself or mined from somewhere else. Mined widgets are kept in the

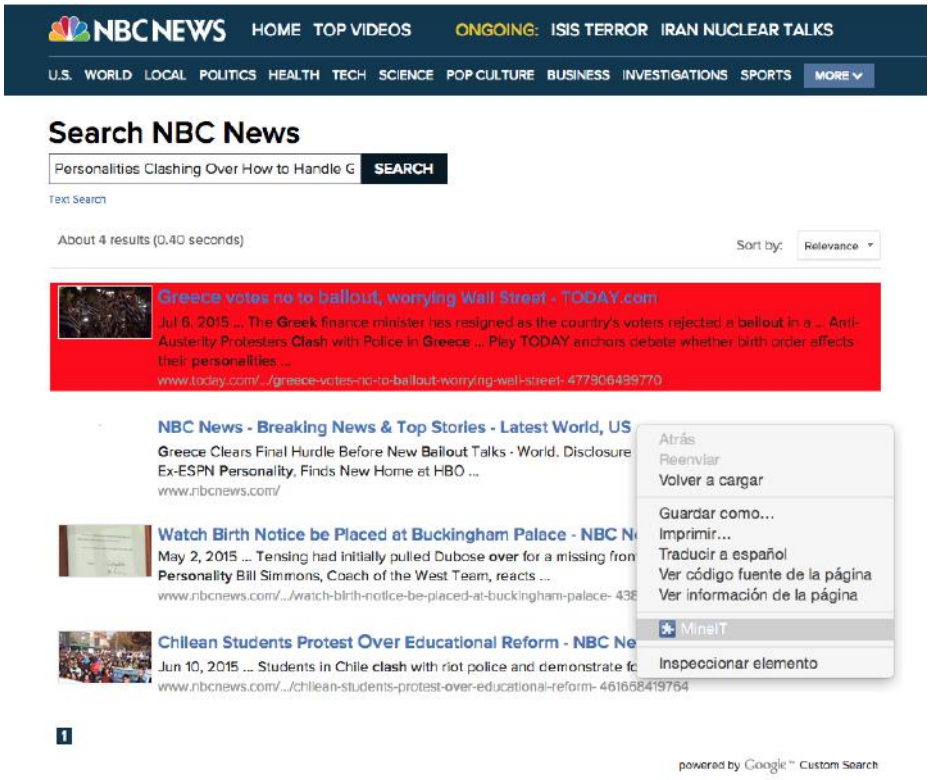

Fig. 4. Widget mining from *http://www.nbcnews.com/*: right click, select *MineIT*, highlight the desired node, and press enter. A popup will request the widget name (e.g. *NBC*). From them on, the widget is kept in the *PiggyBank*.

PiggyBank (available through the namesake tab), and moved to the canvas (i.e. the current page) through drag and drop. Widget placement is automatically handled by the engine through some built-in heuristics. Once on the canvas, all widgets behave the same, i.e.

1. widgets can be deleted or moved around by interacting through the widget decorator,
2. widgets have an initial state, either visible or collapsed, reflected in the decorator through the opened-eye icon or closed-eye icon, respectively (see Figure 3). At runtime, this state can be changed through user interactions (see Subsection 3.4) so that widgets move from visible to collapsed, or vice versa,
3. widgets can be parameterized. Parameters are automatically derived based on the underlying HTML fragment. This includes labels, entry form parameters or the refresh polling frequency (for mined widgets). Double click upon the widget to see its parameters.

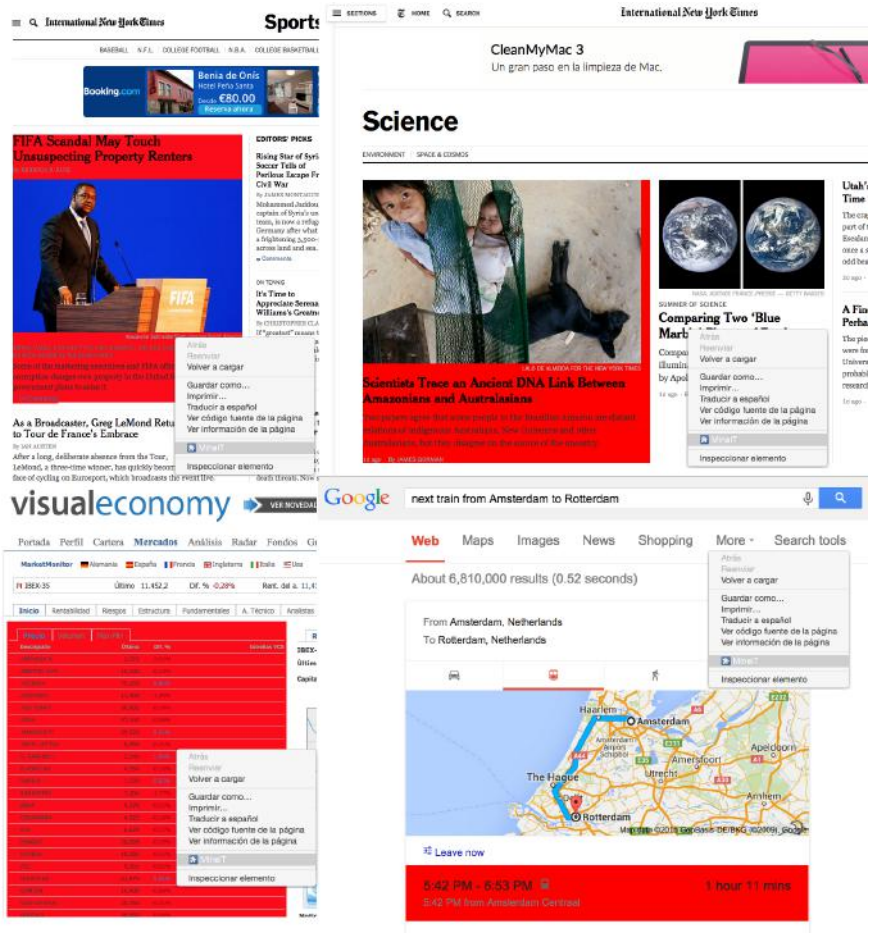

Fig. 5. Mining widgets for the sample scenario (in clockwise order): *sportHL*, *scienceHL*, *stockMarket* and *trainData*. The widget's node counterpart is highlighted.

Figure 6 shows the parameters after double clicking the *linkBar* widget. Basically, labels and hrefs are made available so that the user can now change any of them. In this case, we change the first link from pointing to the World News to the ICWE program. Parameter assignment can be by value or by reference. By value refers to the user manually providing the value as in the previous example. By reference involves the system automatically retrieving the value by applying an XPath upon the modded page at runtime. XPaths are derived from user interaction upon the host page at parameterization time. Uses do not need to know XPath. The NBC widget is a case in point. This widget's parameters include the searching text. If you type a value, the widget will always look for this value. By contrast, a reference to some content of the NYT page can be set. While the parameter list is visible, go to the canvas, copy the right hand-side

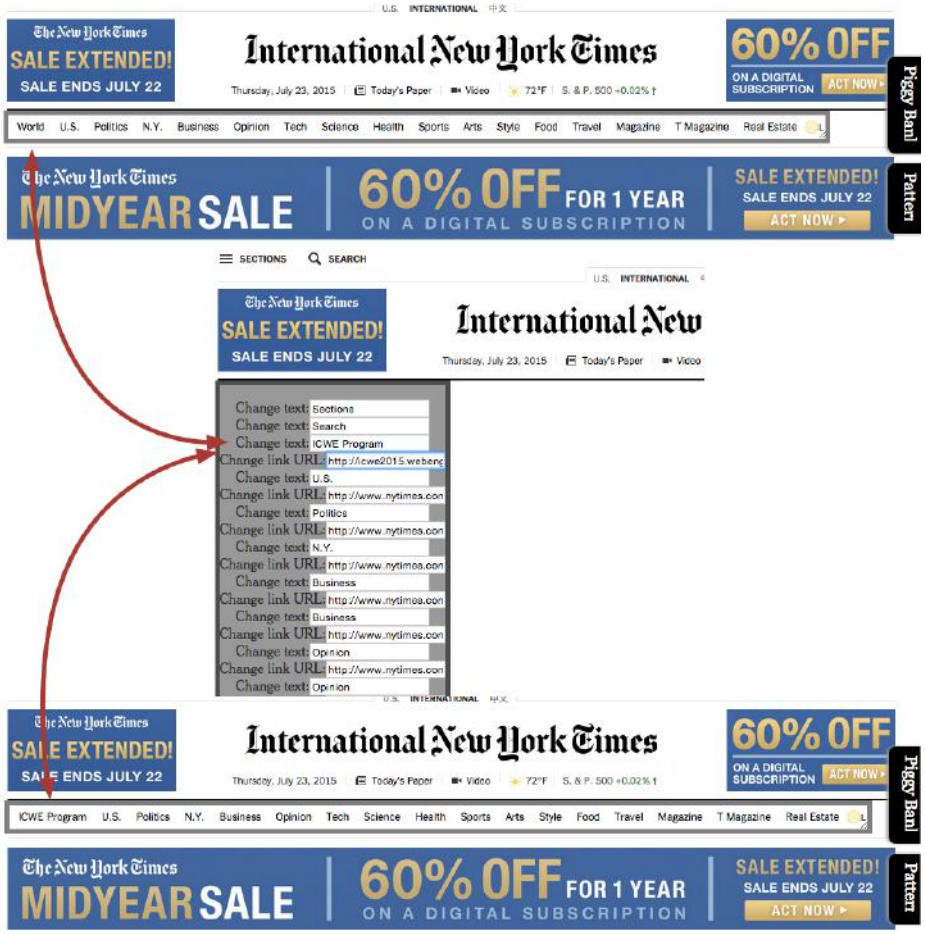

Fig. 6. Changing *linkBar's* parameters. First hyperlink's label is changed from *World* to *ICWE Program* while its URL now points to the ICWE website.

headline, and next, paste it as the value of the searching parameter. Internally the engine associates this parameter to the headline's XPath expression. At runtime, the engine enacts the XPath expression and assigns the result to the NBC's searching parameter "Personalities Clashing Over How to Handle Greek Bailout". In this way, the NBC widget will search for the current headline and not for the headline at the time the mod was created.

3.4 Widget Animation

Modding happens in an existing page which will probably have most of its space taken. Indeed, our running example handles seven widgets, namely:

– from the modded page: *linkBar*, *headline, rightColumn*

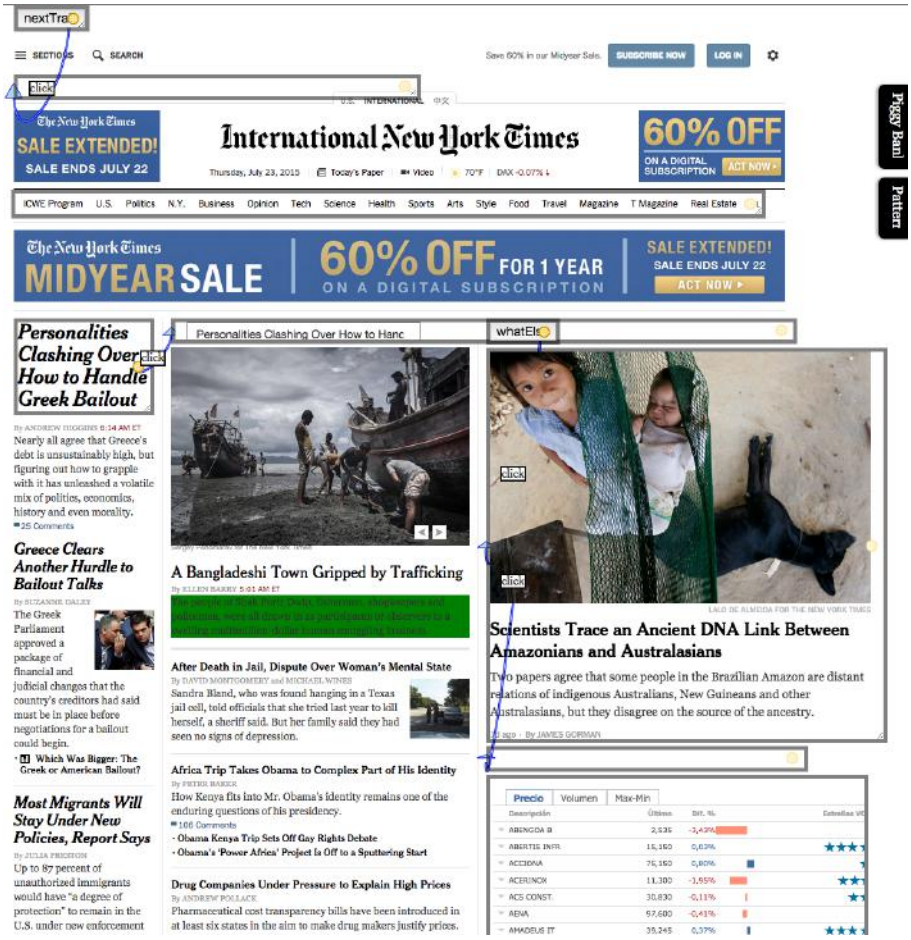

Fig. 7. Setting blinks between widgets. Widget below will be visible after clicking on the widget above.

– from the websphere: *trainData*, *stockMarket*, *NBC*, *SportHL*, *ScienceHL*

Displaying all these widgets simultaneously will lead to an even more cluttered NYT page, impacting the UX. Hence, it is common to turn some nodes into widgets with the only purpose of deleting them, and making room for new content. This is the case of *rightColumn*. This widget is removed to leave room for *stockMarket*. But, this might not be enough. We should also consider which widgets should be readily visible (i.e. at loading time), and which should be visible on demand, i.e. subject to a previous user interaction upon another widget. The latter is referred to as widget animation.

Widgets can be in two states: visible or collapsed. At design time, users decide the initial state. At runtime, this state can be changed through "blinks". *Blink*

relationships can be set between widgets so that interactions upon a widget can impact another widget's state. *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. The default triggering event is *click*, though users can select other DOM events. Figure 7 depicts such a pipe from *headline* to *NBC*. *NBC'* initial state is collapsed. This blink instructs that clicking *headline* will change NBC state. Let's see the rest of the animation (see Figure 7):

Specifically, buttons can be introduced to make widgets available on demand (i.e. through button interaction).

Let's see a possible animation strategy for our sample case (see Figure 7):

- *headline and stockMarket* are always visible (i.e. they are never involved as triggered widgets in a *blink*),
- *NBC* is initially collapsed. It becomes visible when clicking on *headline,*
- *trainData* is initially collapsed. We introduce the *nextTrain* button to make it available on demand. To this end, *PiggyBank* always holds three handy widgets (i.e. *link*, *image* and *button*) which can be cloned and parameterized as any other widget,
- *sportHL* is initially visible but collapsed when clicking on the *whatElse* button,
- *scienceHL* is initially collapsed but becomes visible when clicking on the *whatElse* button.

The later introduces a disjunction-blink pattern whereby two widgets are shown in alternation on clicking upon a common widget. By letting users play with the tool, we noticed other recurrent composition of blinks:

- **click2erase**. This pattern involves only one widget. It accounts for a single blink. For instance, consider *"stockMarket blinks stockMarket on clicking". stockMarket* will be available till the user click on it. On clicking, *stockMarket* is gone for the current session.
- **click2alternate**. This pattern involves two widgets which are shown alternatively. It accounts for two blinks: *"scienceHL blinks sportHL.state=visible on clicking" & "sportHL blinks scienceHL.state=collapse on clicking".* Initially only *sportHL* is visible. Click on it, and *sportHL* is substituted by *scienceHL*. Click again, and *sportHL* shows up again.
- **conjunction**. These patterns involve three widgets or more: the triggering widgets, and two triggered widgets that are shown simultaneously. It accounts for two blinks: *"whatElse blinks sportHL on clicking" & " "whatElse blinks scienceHL on clicking".* On clicking, both *sportHL* and *scienceHL* pops up.
- **disjunction**. These patterns involve three widgets: the triggering widgets, and two triggered widgets that are shown in alternation. It accounts for two blinks: *"whatElse blinks sportHL.state=visible on clicking" & "*

whatElse blinks scienceHL.state=collapse on clicking". Clicking successively on *whatElse* shows *sportHL* and *scienceHL* in alternation.
- **incremental**. This pattern involves "n" widgets which are gradually presented as the user clicks. It accounts for "n-1" blinks. The first blink involves the triggering widget (e.g. *"headline blinks sportHL on clicking")* while subsequent blinks subordinate the rendering of a widget to click in its widget predecessor (e.g. *"sportHL blinks scienceHL on clicking")*. Therefore, widget order matters.
- **domino**. It leverages the previous pattern so that clicking on the last widget collapses all its predecesors except the triggering widget (i.e *"headline")*.

These patterns are available through the namesake tab. Pattern definition is achieved using a similar approach to PowerPoint's SmartArts (see Figure 8). Keeping the *ALT* key pressed down, select the involved widgets. As widgets are being selected, the widget region is shadowed, highlighting the order of the widget at hand. Once all the participating widgets are picked out, and keeping the *ALT* key pressed down, choose the desired behavior in the *pattern* tab. *WebMakeup* will automatically generate the blinks that jointly account for the pattern at hand.

4 The mod lifecycle

Though previous subsections present the different operations in sequence, the user is free to intermingle those operations as they come to mind. Indeed, we envisage mod development to be characterized as being in "perpetual beta" in the sense of the mod being able to be easily modified at any time. Ease deployment of partial mods allows users to get a glimpse of the development so far. To this end, the *WebMakeup* menu offers the *"Deploy"* option (see Figure 2). On clicking, the page is reload but with the mod enacted. Now, the user can get a real feeling on the result so far. For instance, Figure 9 depicts the NYT website with the sample mod. By interacting with the different widget regions, the user can check out the mod's animation. Previous figure depicts the outcome after clicking *nextTrain* and *headline*. Finally, important and export facilities are available for mod sharing through the namesake options in the *WebMakeup* menu. Export generates a *.mkp* file. This file can then be imported, or even easier, dragged and dropped into the browser for the consumer to enjoy the mod.

5 Level of maturity & Discussion

WebMakeup is available at the *Chrome Web Store*: `https://chrome.google.com/webstore/detail/alnhegodephpjnaghlcemlnpdknhbhjj`. It is then available for public download. The case study described in this paper was conducted with this plug-in. More complex cases still present main chanllenges. Mining widgets from content resulting from AJAX interactions is still difficult. Implementation details can be found at [3]. Technically, WebMakeup exhibit some limitations that were highlighted during the Mashup contest:

Fig. 8. Setting patterns. Keeping the *ALT* key pressed down, select first the widgets, and next, the *blink* pattern.

– upgrades on the NYT website can break the mod apart. Since widget placement and data binding are based on the page structure, changes to this structure can make the mod stop working. True. Notice however that re-building the mod from scratch will take around 30', and that after all, the layout of the NYT website does not change so often. However, the risk is there.
– mod reuse might be limited to users exhibiting the same browser settings. By browser settings, we refer to those client-side aspects that might impact the page structure. First, extensions. Mods might not be the only extensions deployed at the user's browser installation. Thousands of extensions are available at browsers' Web stores that might co-exist and interact with mods. A common case is that of ad blockers. These popular extensions prevent

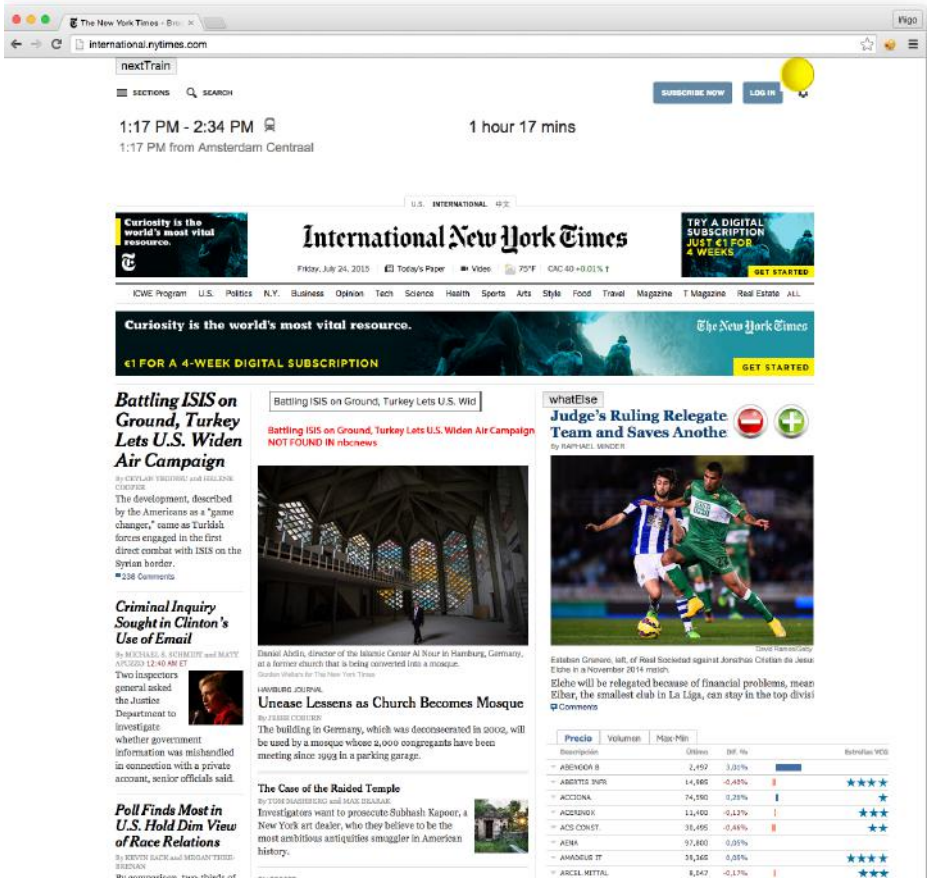

Fig. 9. The mod at work. Screenshot once *nextTrain* and *headline* have been clicked.

adverts from showing up. In so doing, they change the page structure, and hence, they might impact the mod outcome.

– incremental development of mods might be penalized by rich, heavy Web pages. The point is that mod enactment takes place once the page is fully downloaded. That is, widgets start showing up once all the server content is being loaded. No way to click *nextTrain* till all the content is available. During the contest, this was a cause of distress since it took several seconds for the NYT page to be fully loaded, hence hindering the quick feedback that *WebMakeup* aims at.

– installability (i.e., the quality of requiring minimum installation burden) is regarded as a main advantage of *WebMakeup*. Being an extension itself, *WebMakeup* can be easily downloaded from *Chrome Web Store*: `https://chrome.google.com/webstore/detail/alnhegodephpjnaghlcemlnpdknhbhjj`.

Mashup Feature Checklist		Mashup Tool Feature Checklist	
Mashup Type	UI mashup	**Targeted End-User**	Non Programmers
Component Types	UI components	**Automation Degree**	Semi-automation
Runtime Location	Client-side only	**Liveness Level**	Level 3[3]
Integration Logic	UI-based integration	**Interaction Technique**	WYSIWYG
Instantiation Lifecycle	Short-living	**Online User Community**	Private but sharable

Table 1. Characterizing *WebMakeup* as a mashup tool.

This makes the *WebMakeup* icon to show up in the browser bar. This is all needed to start modding your favorite websites.

6 Related Work

The first question is whether modding should be considered a mashup technique. The answer is unclear. It might be so in spirit but not in architecture. That is, mods aim at improving the UX, and one way to achieve this is through mashuping, here understood as side-by-side integration of Web content. However, from an architectural perspective, mods are not self-contained Web applications but browser extensions (a.k.a. plugs-in) to be frequently achieved at the back of the website and by users who might not have server access. From this perspective, modding falls within the area of Web Augmentation [2]. Table 1 sets *WebMakeup* within the feature checklist put forward by the Contest organizers:

- mashup components (i.e. the artifact to be reused and that is accessible either locally or remotely) are limited to HTML fragments which are extracted from websites and included in the modded website.
- mashup logic (i.e. the internal logic of operation of a mashup) includes aspects such as widget location within the modded page, data flow between the modded page and the hosted widgets, or widget animation.

Specifically, *WebMakeup* pivots around the notion of "widget". There already exist W3C standards for UI Reuse like Widgets [**?**] and Web Components [**?**]. W3C Widgets are "full-fledged client-side applications that are authored using Web standards such as HTML and packaged for distribution". Web Components allow "Web application authors to define widgets with a level of visual richness and interactivity not possible with CSS alone, and ease of composition and reuse". Reusing such components is possible in our context. However, we decided not to integrate them due to its immaturity and the low number of such components that already exist on the Web.

Another possibility for widget creation is to create them based on a fragment selected by the user. This process comprises two steps: the selection of the area to be widgetized and the extraction of such area. For the selection step, it would be useful any guidance. As introduced earlier, a widget is meaningful piece of information support as a DOM element. It is trivial to allow users the selection of any DOM element. However this is not the same for filter this selection to

such elements that are meaningful as a unit. In the accessibility area, there are some works that face the problem of page segmentation. This page segmentation is used to slice a webpage in meaningful units that are later consumed by impaired users [**?**,**?**]. This algorithms can be used in our context to guide to end-users while selecting a DOM element. For mirroring the fragment as closely as possible, it would be needed to extract the content, style and functionality of the original webpage. This is far from trivial. Whereas there are multiple libraries to extract content and style automatically [**?**,**?**], as far as we know, there is no automatic mechanism to extract the functionality. There are some works that relates user interactions with the JavaScript code that handles them [**?**,**?**], in order to help programmers during the maintenance tasks. Departing from such point, it could be possible to extract such code and execute in the augmented web in an automatic way. However, again, this is far from trivial. A possible way could be the programatic generation of all possible interactions, the extraction and dependency resolution of the executed code and its injection in the augmented page.

Acknowledgments.

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

References

1. Daniel, F., and Matera, M. *Mashups - Concepts, Models and Architectures*. Data-Centric Systems and Applications. Springer, 2014.
2. Díaz, O., and Arellano, C. The augmented web: Rationales, opportunities, and challenges on browser-side transcoding. *TWEB 9*, 2 (2015), 8.
3. Díaz, O., Arellano, C., Aldalur, I., Medina, H., and Firmenich, S. End-user browser-side modification of web pages. In *Web Information Systems Engineering - WISE 2014 - 15th International Conference, Thessaloniki, Greece, October 12-14, 2014, Proceedings, Part I* (2014), pp. 293–307.