Layman Tuning of Websites: Facing Change Resilience

Oscar Díaz
oscar.diaz@ehu.es

Cristóbal Arellano
carellano001@ikasle.ehu.es

Jon Iturrioz
jon.iturrioz@ehu.es

ONEKIN Research Group
University of the Basque Country
San Sebastián, Spain

ABSTRACT

Client scripting permits end users to customize content, layout or style of their favourite websites. But current scripting suffers from a tight coupling with the website. If the page changes, all the scripting can fall apart. The problem is that websites are reckoned to evolve frequently, and this can jeopardize all the scripting efforts. To avoid this situation, this work enriches websites with a *"modding interface"* in an attempt to decouple layman's script from website upgrades. From the website viewpoint, this interface ensures safe scripting, i.e. scripts that do not break the page. From a scripter perspective, this interface limits tuning but increases change resilience. The approach tries to find a balance between openness (scripter free inspection) and modularity (scripter isolation from website design decisions) that permits scripting to scale up as a mature software practice. The approach is realized for *Greasemonkey* scripts.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Reuse Models

General Terms

Standardization, Design

Keywords

Personalization, Web 2.0, Semantic Web

1. INTRODUCTION

Traditional adaptive techniques permit to adjust websites to the user profile with none (a.k.a. adaptive) or minimum (a.k.a adaptable) user intervention. But, these techniques do not preclude the need for a do-it-yourself (DIY) approach where users themselves can locally tune websites for their own purposes. To denote this scenario, the term *"modding*" is borrowed from hardware and computer-game practices to denote *the practice of locally changing an existing website by the layman for the layman's purposes.*

So far, the most common DIY way to website modding is *JavaScript* using special weavers such as *Greasemonkey*[1]. A script can *react* to events when interacting or loading a page. The script can *access* any node of the page. And finally, the script can also *change* the page at wish. But this freedom has a trade off. Making use of the knowledge about a page implementation, can make the script bound to the actual page structure, data and style. If the page changes, all the scripting can fall apart. This is a main stumbling block for robust, scalable and widely-adopted Web modding. To this end, two main questions are posed,

1. how can *websites* be developed that facilitate layman modding while still permitting the website to evolve?
2. how can *scripts* be developed so that they do not interfere with the target website, hence, ensuring resilience to website changes?

This work introduces the *"modding interface"* for websites. This interface aims at shielding the script from *"design decisions that are likely to change*" in the website.

[1] *http://ww.greasespot.com*

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

2. A MOTIVATING SAMPLE

Consider the script in the left-hand side of Figure 1 adapted from *fav.icio.us2*[2]. It suffices to say this script leverages *del.icio.us* by supplementing posts (i.e. bookmarks) with their corresponding favicons. This example highlights some of the limitations of current scripting: (1) coarse-grained approach: the whole page is processed in a single run. On loading, the script iterates over each post and adds the corresponding favicon. If fine-grained processing is required for specific elements then, conditional statements need to be embedded, hidden within the script code; (2) data is recovered through scraping, which bounds the script to the current page structure; (3) pointcuts are identified through XPath expressions, which again couples the script to layout decisions, Therefore, current scripting exhibits a tight coupling with the website, making scripts fragile to website upgrades.

3. THE MODDING INTERFACE

The modding interface aims at shielding the script (i.e. the consumer) from *"design decisions that are likely to change"* in the current realization of the page. From a DOM-document viewpoint, design decisions are realized on how content is structured, rendered or browsed. Among these decisions, those related with content tend to be more stable than the other two. Hence, it comes as no surprise that modding should be mainly based on the page content.

Therefore, we abstract away from data into *"concepts"*. Concepts describe meaningful units of what (rather than how) is being rendered by an HTML page. They attempt to capture the essence of the notions being handled by the website. For instance,

[2] Script available at
http://userscripts.org/scripts/source/3406.user.js

```
//LOGIC
function add_favicon(link) {...}
//DIGGING OUT DATA + LOGIC CALL + MOD APPLICATION
function initialize() {
  var link;
  var matches = root.evaluate(
                "//li[contains(@class, 'post')]/h4[@class='desc']/a",
                document, null,
                XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE, null);
  for (var i = 0; link=matches.snapshotItem(i); i++){
    var favicon=add_favicon(link);
    link.parentNode.insertBefore(favicon, link);
  }
}
//ENTRY POINT
window.addEventListener("load", initialize, true);
```

GUI EVENT

```
//LOGIC
function add_favicon(link) {...}
//CALL LOGIC + MOD APPLICATION (DIGGING OUT DATA IMPLICIT)
function initialize(eventOcurrence) {
  var ref=eventOcurrence.getElementsByTagName("rel-bookmark").item(0);
  var img=add_favicon(ref);
  eventOcurrence.target.insertBefore(img,ref);
}
//REGISTER RULE
document.addEventListener("postLoad",initialize,true);
```

CONCEPTUAL EVENT

Figure 1: The *fav.ico.us* script: traditional (left-hand side) vs. interface aware (right-hand side).

"post" could be a concept for *del.icio.us*, *"book"* for *Amazon*, *"flight"* for *Expedia*, etc.

However, such approach does not match with the event-driven nature of script programming. This raises the notion of *"conceptual event"*. A conceptual event denotes an interaction with a concept being rendered. For example, *del.icio.us* can define three conceptual events on *"Post": PostLoad, PostMouseOver* and *PostMouseOut* based on the namesake events. Accordingly, the loading of a page can potentially raise as many *PostLoad* occurrences as *post* are present in this page.

The scripter now subscribes to conceptual events as he did before for GUI events. Figure 1 (b) shows the *fav.icio.us2* script but now rewritten in terms of this modding interface. Differences with the previous script are highlighted in bold: (1) the script subscribes to conceptual events; (2) data is recovered from event parameters rather than digging out the DOM document. This accounts for two main advantages. First, the script is easier to write and read since most scraping is removed from the script. Second, the script is more resilience to website upgrades.

This improves the resilience of the script to website upgrades, but does not preclude the script itself from changing the page in "unsafe ways". So far, the scripter can inject any HTML fragment on the premise that the disclosure of the page implementation makes him acknowledgeable about what would be the right fragment code. This approach may work for small scripts but is hardly scalable for complex pages. We can not rely on end users peering on HTML code to ascertain what would be a wrong fragment. This is the role of *"the modding constraints"*.

As their database counterparts, a *"modding constraint"* describes a constraint that should be obeyed no matter what and where the modification of the website is achieved. These constraints reflect invariants on the layout or aesthetics concerning a given *concept*. For instance, modifications of the concept *"Post"* should be compliant with the following constraints: if the layout is extended then, the supplemented code is restricted to be of type *HTMLParagraphElement* as defined by the W3C. Now, a script that subscribes to *"PostLoad"* must generate an *HTMLParagraphElement-compliant* fragment. The weaver (e.g. *Greasemonkey*) checks whether this constraint is fulfilled, and if not so, ignores the script but still render the rest of the page.

4. MAKING IT WORK

First, the website publishes its modding interface. This implies two meta links in the pages' header. One link points to the modding-interface document (i.e. *rel="moddingInterface"*) that describes the interface using OWL. The second link holds a transformation document (i.e. *rel="transformation"*). Using a GRDDL-like approach, this link keeps an *XSLT* file that recovers the potential event occurrences from the hosting page.

Second, scripters write the code based on conceptual events. Conceptual events are described in the *moddingInterface* file using OWL. To avoid the scripter handling directly OWL, an add-on for *Greasemonkey* has been developed: *APRON*[3]. When enabled, *APRON* monitors which websites provide a modding interface by looking for the *"moddingInterface"* metalink. If founded, the *APRON* icon highlights. Clicking on this icon, pops up a page that provides some script snippets that indicate how to subscribe to the conceptual events, and how to recover the event parameters. The scripter can then copy the corresponding snippet and paste it to his favourite *JavaScript* development environment.

Third, the user script is enacted. *Greasemonkey* keeps watching for URLs with an associated script as usual. Additionally, *APRON* also surveillance interface-aware websites. Once one is found, (1) *APRON* produces the conceptual-event occurrences from the page being rendered (by accessing the *"transformation"* metalink); (2) scripts are fired which results in HTML fragments being generated; (3) the script-generated fragment is validated against the corresponding modding constraints; (4) if violated, the fragment is ignored, and the page is rendered without the modding. Otherwise, the fragment is inlayed into the page.

5. CONCLUSION

Fostering a win-win relationship between website owners and laymen, substantiates the efforts from moving away from "fragile scripting" to scalable, robust scripting. To this end, we propose *the modding interface* as an attempt to isolate layman's script from upgrades in the website. From the website viewpoint, this interface realizes a controlled setting for modding. From a scripter perspective, the modding interface reduces the freedom but increases change resilience, and eases coding.

6. ACKNOWLEDGMENTS

This work was co-supported by the Spanish Ministry of Science & Education, and the European Social Fund under contract TIN2005-05610. Arellano has a doctoral grant from the Spanish Ministry of Science & Education.

[3]*APRON* itself is supported as a *Greasemonkey* script.