Interfaces for Scripting: Making Greasemonkey Scripts Resilient to Website Upgrades

Oscar Díaz, Cristóbal Arellano, and Jon Iturrioz

ONEKIN Research Group, University of the Basque Country,
San Sebastián, Spain
{oscar.diaz,cristobal.arellano,jon.iturrioz}@ehu.es
http://www.onekin.org/

Abstract. Thousands of users are streamlining their Web interactions through user scripts using special *weavers* such as *Greasemonkey*. Thousands of programmers are releasing their scripts in public repositories. Millions of downloads prove the success of this approach. So far, most scripts are just a few lines long. Although the amateurism of this community can partially explain this fact, it can also stem from the doubt about whether larger efforts will pay off. The fact that scripts directly access page structure makes scripts fragile to page upgrades. This brings the nightmare of maintenance, even more daunting considering the leisure-driven characteristic of this community. On these grounds, this work introduces *interfaces for scripting*. Akin to the *JavaScript* programming model, *Scripting Interfaces* are event-based, but rather than being defined in terms of low-level, user-interface events, *Scripting Interfaces* abstract these DOM events into *conceptual events*. Scripts can now subscribe to or notify of *conceptual events* in a similar way to what they did before. So-developed scripts improve their change resilience, portability, readability and easiness to collaborative development of scripts. This is achieved with no paradigm shift: programmers keep using native *JavaScript* mechanisms to handle *conceptual events*.

Key words: Greasemonkey, JavaScript, Maintenance, Web2.0

1 Introduction

Traditional adaptive techniques permit to adjust websites to the user profile with none (a.k.a. adaptive techniques) or minimum (a.k.a. adaptable techniques) user intervention [13]. No design can provide information for every situation, and no designer can include personalized information for every user. Hence, traditional customization techniques do not preclude the need for do-it-yourself (DIY) approaches where users themselves can locally modify websites for their *own* purposes.

A popular client-based DIY technology is *JavaScript*, using special plugins such as *Greasemonkey (GM)* [1]. A *GM* script resides locally, and it has a scope, i.e. the websites to be subject to scripting (identified through URL patterns). *GM* silently watches whether the current URL matches the URL patterns, and if so, *locally* executes the script that leads to on-the-fly changes on the current page. Scripts can be uploaded

to script repositories such as *userscripts.org*. With thousands of members and scripts, *userscripts.org* registers hundreds of downloads everyday! These remarkable figures stem from both the usefulness of scripts created by anyone, and the easiness of installation (a two-click process). This success evidences how scripting is moving from being a solitude practice to become a community phenomenon where laymen can enjoy scripts even if ignorant about *JavaScript*.

Unfortunately, current scripting practices do not scale up. Scripts directly access the structure of the page being rendered (i.e. the DOM tree). If the page changes, all the scripting can fall apart. And the page can change due to either upgrades on the website or changes made by previously enacted scripts. The problem is that websites are reckoned to evolve frequently, and the number of simultaneously enacted scripts tends to increase. This brings the nightmare of maintenance, even more daunting considering the altruistic, leisure-driven characteristic of many script programmers. *Our base premise is then that the maintenance burden is hindering GM scripters from becoming a mature community, not so in size but on the complexity of the scripts available.*

On these grounds, this work aims at shielding scripts from changes in the scripted pages. This implies separating the stable part of the script from the one more exposed to page changes. The former stands for the "mod logic", i.e. the code that supports the additional functionality provided by the script. The unstable part corresponds to code that weaves this mod logic to the current page, i.e. the code that consults/updates the scripted page.

We propose to encapsulate the fragile part of current scripts through interfaces. *Interfaces for scripting* encapsulate the current realization of HTML pages in terms of the *concepts* these pages convey. Now, scripts can subscribe to *conceptual events* (rather than subscribing to low-level, UI events) and notify of *conceptual events* (rather than directly modifying the page). This approach does not imply so much a change in the programming model but in the development methodology. Scripts keep using handlers but now upon *conceptual events* rather than UI events. The difference rests on the two-stage process. A first script implements the interface (so-called *Class Script*). A second script supports the mod logic on top of this interface (so-called *Mod Script*). In this way, the mod logic is decoupled from the concrete realization of the concepts this logic acts upon. Page changes only impact the *Class Script*.

From the perspective of the *Mod Script*, this approach accounts for change resilience (i.e. mod scripts are sheltered from changes in the website), readability (i.e. mod scripts are described in terms of *conceptual events* rather than HTML scraping), portability (i.e. mod scripts work for any website providing the same interface), and "collaborativeness" (i.e. the *Class Script* and the *Mod Script* can be developed by different users).

We regard as main contributions of this work, first, the rationales for bringing interfaces into the scripting realm, backed by examples taken from the *GM* community. And second, a non-disruptive approach that keeps current scripting practices without changes in neither the *JavaScript Engine* nor *GM*. Although the study is conducted for *Greasemonkey*, the approach can be generalized to any other *weaver*. Readers are encouraged to download the solution from *http://userscripts.org/users/61033*.

The paper begins by stating the problem, i.e. (1) script tight coupling to page rendering and, (2) companion script collision. Section 3 introduces *interfaces for scripting*. Its realization entails the introduction of three types of scripts: *Interface Scripts*, *Class Scripts* and *Mod Scripts* which are the subject of sections 4, 5 and 6, respectively. Section 7 revises the approach. Related work and conclusions end the paper.

2 Problem Statement

Fig. 1. *Amazon* enhanced with the *BookBurro* script: prices for the current book at other online bookshops are shown at the top-right side.

Greasemonkey (GM) is a *Firefox* extension that permits end users to install scripts at the client that make on-the-fly changes to the underlying HTML page structure (a.k.a. DOM tree) [2]. This is known as a *weaver*. This work focuses on *Greasemonkey* as a *weaver* for *Firefox*. Besides *Firefox*, *weavers* are available for Internet Explorer (e.g. *IE7Pro* or *Turnabout*), Opera (e.g. *User javascript*), Safari (e.g. *SIMBL + GreaseKit*) and natively supported in Google Chrome.

Reason	Times
Perfective maintenace	**22**
Adding new bookshops	18
Upgrades on bookshop websites	3
Remove sites	1
Corrective maintenance	**13**
Centralize scraping data	1
Bugfix	3
Improve code understandability	2
Improve UI	5
Add extra functionality (AJAX, caching)	2
Adaptive maintenance	**4**
Weaver-based no backward compatibility	3
Port to other browser weavers	1

Table 1. Rationales for upgrading *BookBurro* along a year period. (source: changelog in *BookBurro* script)

Weavers permit scripts to act upon Web pages at runtime. Pages are perceived as DOM trees[1]. The script is triggered by User Interface events (UI events) on this DOM tree (e.g. *load*, *click*). Event payloads provide the data that feed script handlers which in turn, update the DOM tree. Scripts are written in *JavaScript*. For instance, a popular script, *BookBurro*[2], embeds price comparison in *amazon.com* web pages. On loading, the script retrieves the book's ISBN being rendered, and next, embeds a panel with the price of this book at other online bookshops, e.g. *Buy, BN, Powell, Half,* etc (see Figure 1). At install-time, scripts are associated with URL patterns that denote the pages to which the script applies. You can keep the script for yourself or upload it into a script repository (e.g. *userscripts.org*) for others to download. *BookBurro*, with more than 18,684 downloads, illustrates this ripple benefit. Millions of downloads and thousands of uploads at *userscripts.org* provide anecdotal evidence of the profound impact that user scripting is having on a large number of users. Unfortunately, this practice is being hindered by maintenance burdens.

Next subsections provide two real scenarios where *BookBurro* is confronted with changes in either the scripted pages or the companion scripts. These scenarios are far from being just an academic exercise but they reflect similar settings as those faced up by *BookBurro* during its lifetime. Indeed, we examined the 18 different versions of *BookBurro* to assess the main reasons for the upgrades. Table 1 depicts the findings.

2.1 Upgrades on Scripted Pages

BookBurro embeds price comparison in *amazon.com* web pages from distinct online bookshops. The script is outlined in Figure 2 (left side). The process goes as follows:

- interacting with a page triggers UI events (e.g. *load*),
- the script can *react* to this event by triggering a handle (e.g. *"init"*, line 20). The association between event and handler (a.k.a. event listener) are achieved through the *addEventListener()* function (line 35),

[1] The Document Object Model (DOM) is a platform- and language-independent standard object model for representing HTML or XML documents as well as an Application Programming Interface (API) for querying, traversing and manipulating such documents.

[2] *BookBurro* is available at *http://userscripts.org/scripts/source/1859.user.js*

```
// ==UserScript==
// @name          Simplified version of Book Burro
// @namespace     http://overstimulate.com/userscripts/
// @description   Book prices from various book stores
// @include       http://www.amazon.com/*
// ...
// ==/UserScript==


//MOD-LOGIC
function createBookBurroPanel(isbn) {
 var priceListHTMLPanel=document.createElement("div");
 //A panel with a price list of selected book from
 //distinct bookstores is created and assigned to the
 //priceListHTMLPanel variable
 ...
 return priceListHTMLPanel;
}

function init() {
 //HTML SCRAPING
 var isbn=document.evaluate(
    "//body[@class='dp']"+
    "//td[@class='bucket']/div[@class='content']/ul/li[4]",
    document,null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,null
    ).snapshotItem(0).innerHTML.match(
    /[0-9]{10}/i )[0];
 //MOD-LOGIC CALL
 var bookBurroPanel=createBookBurroPanel(isbn);
 //HTML INJECTION
 document.body.appendChild(bookBurroPanel);
}

window.addEventListener("load",init,true);
```

```
// ==UserScript==
// @name          Interface-aware version of Book Burro
// @namespace     http://overstimulate.com/userscripts/
// @description   Book prices from various book stores
// @include       http://www.amazon.com/*
// ...
// ==/UserScript==
var doc=window.document;

//MOD-LOGIC
function createBookBurroPanel(isbn) {
 var priceListHTMLPanel=doc.createElement("div");
 //A panel with a price list of selected book from
 //distinct bookstores is created and assigned to the
 //priceListHTMLPanel variable
 ...
 return priceListHTMLPanel;
}

function init(loadBookOcc) {
 //EVENT PARAMETER RETRIEVE
 var book=loadBookOcc.currentTarget;
 var isbn=book.getElementsByTagName("isbn").item(0).nodeValue;
 //MOD-LOGIC CALL
 var bookBurroPanel=createBookBurroPanel(isbn);
 //EVENT DISPATCH FOR HTML INJECTION
 var appendChildBookOcc=doc.createEvent(
    "ProcessingEvents");
 appendChildBookOcc.initProcessingEvent(
    "appendChildBook",book,bookBurroPanel);
 doc.dispatchEvent(appendChildBookOcc);
}


doc.addEventListener("loadBook",init,true);
```

Fig. 2. Two versions of the *BookBurro* script: traditional (left side) vs. interface-aware (right side).

- a handler can access any node of the page (using DOM functions such as *"document.evaluate()"* in lines 22-26), and create HTML fragments (e.g. *bookBurroPanel* in lines 11-18),
- a handler can also *change* the DOM structure at wish by injecting HTML fragments. In the example, a *bookBurroPanel* is injected at a point identified by an *XPath* expression on the underlying DOM structure (i.e. the injection point). A DOM functions are used for this purpose (e.g. *"document.body.appendChild(bookBurroPanel)"* in line 32).

Scripts can do any change on the underlying page. But this freedom has a downside: makes the script bound to the actual page structure. If *Amazon* website is upgraded, all the scripting can fall apart. Back to our sample case, *BookBurro* first needs to retrieve the book's *ISBN* from the current page, and next, injects the *bookBurroPanel* at a certain location. This is normally achieved through *XPath* expressions (line 23-24). If the underlying page changes, *XPath* expressions could no longer recover/identify the right DOM portion which could make *BookBurro* stop working properly.

Recovering from these failures can be classified as perfective maintenance. For *BookBurro*, this accounts for 22 changes (see Table 1). This includes not only upgrades on consulted pages but also the need for *BookBurro* to run in bookshops other than *Amazon* (see user discussion thread at *http://userscripts.org/topics/15357*). These changes are not always straightforward which leads to delays on meeting these petitions by *BookBurro* programmers, more to the point if we consider that most programmers tend to do it altruistically. The problem is that websites are reckoned to evolve frequently, and programmers might not have the time to keep the script updated.

2.2 Changes in Companion Scripts

Scripts and scripted pages can exhibit an M:N relationship: a script can be applied to different pages, and a page can be the target of distinct scripts. *BookBurro* illustrates the first case where the very same script provides its mod (i.e. the *bookBurroPanel*) to different websites (*e.g. Amazon, Buy, BN...*). But, these websites can also be the substrate for other scripts. *Amazon* is a case in point. At the time of this writing, 261 scripts are reported to be available for *Amazon* at *userscripts.org*. If you are a regular *Amazon* visitor, it would be more than likely you have different scripts installed. These scripts (i.e. the companion set) will be enacted simultaneously when you visit *Amazon*. It is important to notice that script enactment is not in parallel but in sequence, i.e. scripts are launched in the order in which they were installed. This implies that the first script acts on the raw DOM tree, the second script consults the DOM once updated by the first script, and so on.

The problem is that programmers develop scripts from the raw DOM, being unaware of changes conducted by other companion scripts. This can end up in a real nightmare where code developed by different authors with different aims, is mixed up together with unforeseen results. Even worse, the final DOM tree can even be dependent on the order in which scripts are intermingled! This is not unusual for popular websites that enjoy a large set of scripts. The larger the set of (companion) scripts installed, the higher the likelihood of clashed. And the number of scripts available is steadily increasing which will likely lead to an increase in the number of scripts in each *user* installation. We then perceive "this DOM-based interaction model" as a main stumbling block for scripting to scale up. As learnt from previous experiences in Software Engineering, the approach is to abstract the way scripts are developed by moving away from "the implementation platform" (basically, the DOM document, and the UI events) to a more abstract platform. This is the aim of the *interfaces for scripting*.

3 The Big Picture

So far, scripts act directly upon DOM trees. We strive to abstract away from the DOM tree and the UI events through *Scripting Interfaces*. Interface reckons to hide "design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed" [12]. *Scripting Interfaces* aim at shielding scripts from layout/presentation decisions *"that are likely to change"* in current HTML pages.

Scripting Interfaces characterise pages in terms of the concepts these pages convey, hiding the circumstantial representation of these concepts in HTML. For instance, *"Bookmark"* could be a concept for *del.icio.us*, *"Book"* for *amazon.com*, *"Article"* for *acm.org*, etc. Now, scripts do no longer access directly the DOM tree but through the interface: you can subscribe to *loadBook* (rather than the DOM event, *load*) and obtain book data through the event payload rather than scraping the DOM tree; you can publish the event *appendChildBook* to add an HTML fragment as a child of a *Book* (rather than using an *XPath* expression). The right side of Figure 2 shows the *BookBurro* script but now supported through a *Scripting Interface*. The mod logic is the same (lines 11-18). Differences rest on (1) page scraping being substituted by event payload recovering

(lines 22-23) and (2), injection points described by the point where a conceptual event occurred (lines 27-31) rather than an *XPath* expression.

Figure 3 outlines the main notions of the problem space (in bold). **"UserScripts"** act upon **"BaseWebsites"** but rather than accessing websites directly, scripts are now specified in terms of a **"ScriptingInterface"**. Programming languages clearly distinguish between the interface and the realization of this interface (a.k.a. *class*). Likewise, *Scripting Interfaces* are implemented through **"ScriptingClasses"** which implement interfaces based on websites.

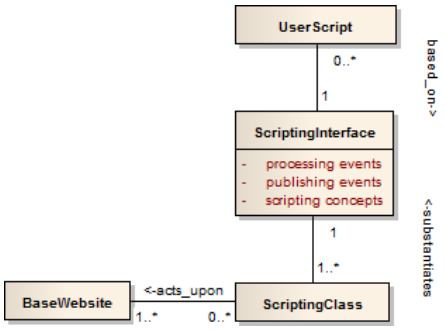

Fig. 3. The Problem Space.

How are *Scripting Interfaces* described? Interfaces are commonly specified in terms of operations defined upon data types. However, *JavaScript* favours event-based programming, i.e. listeners are associated to UI events. Unlike operations, listeners are not explicitly called but *triggered* when the associated event occurs. Akin to the *JavaScript* approach, *Scripting Interfaces* are to be described in terms of events rather than operations, but they will act upon *concepts* rather than DOM nodes.

A (scripting) concept *is a data compound whose rendering is liable to be enhanced as a unit.* This approach resembles that of *microformats* [10] (e.g. *hCalendar*). However, there exist two main differences. First, *microformats* are designed to be widely applicable, i.e., they are targeted at general-purpose agents. By contrast, scripting is website-specific, i.e. each website can have its own concepts. Second, *microformats* are server-based, i.e. they are annotated by the site owner. Conversely, scripting is client-based, i.e. it is up to the scripter to decide which the concepts of interest are. While *microformats* aim at re-using existing HTML tags to convey metadata, "scripting concepts" can be website-specific.

Additionally, components distinguish between the provided and the required interfaces to differentiate between services facilitated or necessitated by the component, respectively. Likewise, a *Scripting Interface* encapsulates a DOM tree, and offers a set of services to "read" and to "write" this DOM tree. From the interface perspective, the "read" part realizes the required interface as the set of events the interface realization just signals but leaves to scripts their processing (a.k.a. *Publishing Events*). This basically implies abstracting away from UI events (i.e. those rised when manipulating HTML elements) to *conceptual events* (i.e. those signalled when acting upon *concepts*). For instance, *amazon.com* can publish events on book loading (e.g. *loadBook*). Scripts can now subscribe to *loadBook* rather than listening to the UI event, *load*. From this perspective, *Publishing Events* are to concepts what UI events are to DOM nodes: means to operate on the underlying structure. But while DOM nodes are implementation-dependant, concepts are design notions, more stable under website upgrades.

```
// ==UserScript==
// @name          Scripting Interface for Books
// @include       *
// ==/UserScript==
var bookInterface={
 "ScriptingConcepts":[{
  "conceptId":"Book",
  "attributes":[{"attributeId":"title","type":"string"},
                {"attributeId":"author","type":"array"},
                {"attributeId":"isbn","type":"string",
                 "pattern":"/[0-9]{10}/"},
                {"attributeId":"price","type":"number"}]
 }],
 "PublishingEvents":[{"id":"loadBook",
                      "payloadType":"Book",
                      "uiEventType":"load",
                      "cancelable":false}],
 "ProcessingEvents":[{"id":"appendChildBook",
                      "payloadType":"HTMLDivElement",
                      "operationType":"appendChild",
                      "targetConcept":"Book"}]
}
```

Fig. 4. The *bookInterface* script.

As for the "write" part, scripts directly modify the DOM structure, coupling the script to the current page implementation. This coupling is now eliminated by identifying the place to be modified in terms of concept occurrences rather than through DOM nodes: the *Processing Events*. *Processing Events* are to concepts what DOM operations are to DOM nodes: means to operate on the underlying structure. For instance, Figure 2 (right side) shows the new version of *BookBurro*. Now, the location to place the *BookBurro* panel is specified in terms of event dispatching: *appendChildBook* (line 31). It is up to the *Scripting Interface* to map this event to the specific physical location (i.e. DOM node). From this perspective, *Processing Events* are the means to operate on the encapsulated realization of concepts. *Processing events* then realize the provided interface[3].

So far, the main notions of the problem space have been introduced. Next, we move to the solution space by addressing how previous concerns are engineered. *Our main requirement is non-disruptiveness from current practices*. The implications are twofold:

- from a programmer perspective, this implies *Mod Scripts* to be developed in a similar way to traditional scripts. This entails native *JavaScript* mechanisms to be used to notify/subscribe *conceptual events*,
- from the user perspective (i.e. users who install scripts), non-disruptiveness implies minimum diverge with current practices for script installation.

As a result, our proposal is *uniquely* based on traditional *GM* scripts where no plugin for neither *Firefox* nor *Greasemonkey* is required. Specifically, three types of *GM* scripts are introduced: **Interface Scripts,** which *specify* the interfaces; **Class Scripts,** which *implement* interfaces for a given base websites, and finally, **Mod Scripts** which built the mod logic on top of an interface. Next sections delve into the details.

[3] The terminology of "processing events" and "publishing events" is widely used for event-based components such as portlets [9].

4 Interface Scripts

Interfaces for scripting are specified through *Interface Scripts* using *JSON* (*JavaScript* Object Notation). *JSON* is a text format which is less verbose than XML, and whose syntax is familiar to *JavaScript* programmers [3]. *JSON* document structure can be constrained through a *JSON Schema* specification [4] (much like what *XML Schema* provides for *XML*). An interface is a collection of *"ScriptingConcepts", "PublishingEvents"* and *"ProcessingEvents"* instances (see Figure 4 for an example).

Concepts have a *conceptId* and a set of attributes. Each attribute has an *attributeI*d, a *type* and other features that constraint the set of possible values (e.g. *min*, *max*, *pattern*, etc). The sample case includes *Book* as a concept with four attributes: *title, author, isbn* and *price,*

Publishing Events are described through (1) the event payload, (2) when the event arises and (3), whether it can be cancelled or not. The event payload i.e. the type of parameter the event conveys (*"payloadType"* property), corresponds to one of the interface's *Concepts*. This concept is communicated at loading time, mouseover time, etc as specify by the *"uiEventType"* property whose values are taken from the W3C's DOM Level 2 Events specification [5]. Finally, the *"cancelable"* property mimics the namesake property available for *JavaScript* events whereby an event is liable to be called off by a handler so that the occurrence is no longer propagated to other handlers. The *bookInterface* exhibits *loadBook* as a publishing event to be raised when a page containing a *Book* is loaded,

Processing Events are specified through (1) the event payload and (2), how the signalled event is going to be processed. *Processing Events* carry a piece of *HTML markup*, i.e. its payload type is described along the W3C's DOM Level 2 HTML and Style specification [5] (e.g. *HTMLParagraphElement*). In this way, the *Scripting Interface* can restrict the type of the markup to be injected to those that can be safely injected (e.g. if the concept is realized through an *HTMLTableElement* <*table*>, only *HTMLTableRowElement* <*tr*> could be permitted). As for the processing mode, it specifies *what* is affected (i.e. the *"targetConcept"* property), and *how* is affected (i.e. the *"operationType"* property). The latter through a reference to the W3C's DOM Level 2 Core operations [5] (e.g. *appendChild*). The *bookInterface* provides an example: *appendChildBook* is raised when *HTMLDivElement* fragments are to be added as children of a *Book* concept.

Interface Scripts just contain the description of an interface. Additionally, they can be uploaded into script repositories so that a URL is generated. For instance, *bookInterface* can be found at *userscripts.org* where the following URL was generated: *http://userscripts.org/scripts/source/60315.user.js* . This is important for others to univocally refer to this script, e.g. *Class Scripts*.

```
// ==UserScript==
// @name          Scripting Class for Amazon Books
// @include       http://www.amazon.com/*
// @require       http://userscripts.org/scripts/source/60315.user.js
// @require       http://userscripts.org/scripts/source/60318.user.js
// ==/UserScript==
var bookAmazonClass={
 "baseWebsite":"http://www.amazon.com/*",
 "implements":"http://userscripts.org/scripts/source/60315.user.js",
 "scrapers":[
  {"scrapedConcept":"Book","XPath":"//body[@class='dp']",
   "attributeScrapers":[
    {"scrapedAttribute":"title",
     "XPath":"//span[@id='btAsinTitle']"},
    {"scrapedAttribute":"author",
     "XPath":"//div[@class='buying']/span/a"},
    {"scrapedAttribute":"isbn",
     "function":function(book){
      var isbnNode=document.evaluate(
       ".//td[@class='bucket']/div[@class='content']/ul/li[4]",book,
null,XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,null).snapshotItem(0);
      return isbnNode.innerHTML.match(/[0-9]{10}/)[0];}},
    {"scrapedAttribute":"price",
     "function":function(book){
      var price=document.evaluate(".//b[@class='priceLarge']",book,
null,XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,null).snapshotItem(0);
      return price.innerHTML.match(/[0-9]+(\.[0-9]+)?/)[0];}}]}]
}
window.registerScriptingClass(bookAmazonClass);
```

Fig. 5. The *bookAmazonClass* script.

5 Class Scripts

A *Class Script* implements an interface based on a specific website. A *Class Script* contains mappings that indicate how interface concepts are obtained from the circumstantial representation of those concepts in a concrete website. Figure 5 shows the *bookAmazonClass* script that implements the *bookInterface* for the *Amazon* website along the following characteristics:

- *"baseWebsite"*, which holds a URL expression for the base website (e.g. *www.amazon.com/**) [4],
- *"implements"*, which keeps the URL of the *Interface Script* whose interface is being realized. This URL is obtained from *userscripts.org* (see later),
- *"scrapers"*, which contains a tripplet <*scrapedConcept, expression, attributeScrapers*> that indicates that *scrapedConcept* is to be located by applying the *expression* to the *baseWebsite* page, and its properties obtained by applying *attributeScrapers*. The latter is a set of pairs *(attribute,expression)* that denotes that *attribute* can be obtained by applying *expression* on the *baseWebsite* page. Expressions can be either functions or *XPath* expressions. In the latter case, *XPath* expressions for attributes are relative to the location of the concept. In the example, *XPath* is used to identify *"title"* and *"author"* while *JavaScript* functions are needed to extract *"isbn"* and *"price"*. It is worth noticing, how the function to extract the ISBN coincides with the one embedded in the traditional script of Figure 2 (left side, lines 22-28).

[4] This URL expression should coincide with the one specified at the time the *Class Script* is installed.

Greasemonkey allows for scripts to have *require* dependencies (specified through the *@require* comment tag). At install-time, *Greasemonkey* will download and keep a locally cached copy of the required files. This facility is used for *Interface Scripts* to be simultaneously downloaded with the installation of the *Class Scripts* so that consumers of *Class Scripts* do not have to install the interface separately. Figure 5 shows this dependency through the *@require* tag in line 4[5]. Also, this facility is used to import the library that manages *conceptual events* and initializes the environment. This library is shared among all *Class Scripts* and its description is omitted due to paper length restrictions. Finally, the *registerScriptingClass* instruction (line 29), which is defined inside this library, makes the environment aware of this class.

Class Scripts can be uploaded to *userscripts.org* and installed as traditional scripts. Once a *Class Script* is successfully installed, the environment will generate *conceptual events* in the very same way that traditional UI events (see later). Now, it is the turn of *Mod Scripts* to capitalize on these *conceptual events*.

6 Mod Scripts

This section addresses the definition of scripts based on *conceptual events*. Native *JavaScript* mechanism is used to notify/subscribe *conceptual events* with no variations w.r.t. traditional script development. This is the most important characteristic to ensure non-disruptiveness with current *JavaScript* practices.

Notification of Processing Events. *JavaScript* follows an event-based approach where listeners can be associated with DOM-based events. An event is a happening of interest. Event types include: *MouseEventTypes* (e.g. *click, mouseover, mousemove...*), *UIEventTypes* (e.g. *DOMFocusIn, DOMFocusOut* and *DOMActivate*), *MutationEventTypes* (e.g. *DOMSubtreeModified, DOMNodeInserted*) and *HTMLEventTypes* (e.g. *load, change*). Operations are available for creation of event occurrences (e.g. *createEvent("MouseEvents")*), assigning the payload to an occurrence (e.g. *initMouseEvent("eventInstance", "eventParameters")*), or raising the event manually (e.g. *dispatchEvent(eventOccurrence)*). The following code simulates a click on a checkbox:

```
var ev=document.createEvent("MouseEvents");
var cb=document.getElementById("checkbox");
evt.initMouseEvent("click", true, true, window, 0, 0, 0, 0,
    0, false, false, false, false, 0, null);
cb.dispatchEvent(ev);
```

The snippet illustrates the pattern for dispatching an event occurrence: [*createEvent*, obtain DOM node, *initMouseEvent*, *dispatchEvent* on this node]. This is standard *JavaScript* code.

Conceptual events mimic this pattern. Back to our running example, a *bookBurroPanel* (i.e. an HTML fragment) is to be injected as a child of a *Book*. For this case, the pattern goes as follows: [*createEvent*, obtain concept, *initProcessingEvents*,

[5] This *@require* tag is only read at install-time, and the remote file is not monitored for changes. *Class Scripts* are not aware of changes made to interfaces once installed.

dispatchEvent on this concept]. The code follows (the complete script can be found at Figure 2):

```
var ev=document.createEvent("ProcessingEvents");
var book = loadBookOcc.currentTarget;
evt.initProcessingEvent("appendChildBook", book,
    bookBurroPanel);
doc.dispatchEvent(ev);
```

The only difference with traditional scripting is that now injection points are not DOM nodes but the current concept. This current concept is to be obtained through *Publishing Events*.

Subscription to Publishing Events. *JavaScript* achieves event subscription by registering a listener through the *addEventListener* method. An example follows:

```
function init(...){...}
var cb=document.getElementById("checkbox");
cb.addEventListener("click",init,true);
```

This code associates the script function *init()* with the occurrence of clicks on a *checkbox* node (a.k.a. the event target). From then on, a click on a checkbox will cause *init()* to be enacted. Since most *JavaScript* events are UI events, event occurrences are generated while the user interacts with the interface, raised by the *JavaScript Engine*, and captured and processed through script functions.

Subscription to *conceptual events* is accomplished in the very same way: associating a listener. For instance, instruction (line 35 in Figure 2 (right side)) *"doc.addEventListener("loadBook",init,true)"* adds a listener to the *loadBook* event, i.e. occurrences of *loadBook* will trigger the *init()* function. The difference rests on listeners being associated to the whole document (i.e. variable *doc*) rather than to DOM nodes (e.g. a *checkbox*). This highlights the fact of events happening on a document of books rather than on DOM nodes that are the circumstantial representation of these books.

7 Discussion

Change resilience. We advocate for traditional scripts to decouple the stable part (i.e. the mod logic which is realized through *Mod Script*s) from the unstable part (i.e. the logic that reads/writes the DOM which is supported through *Class Scripts*). In so doing, website upgrades will only impact *Class Scripts*. *Mod Scripts* are sheltered from the circumstantial realization of concepts in the website. For example, if *Amazon* decides to add more product details (e.g. other online bookshops include the book format: "printed" or "electronic"), *bookAmazonClass* needs to be rewritten but the *BookBurro* script itself is preserved. More to the point, the extreme change is moving to a different site altogether. For instance, *BookBurro* is initially thought to work upon *Amazon*. However, it can be made available for your favourite on-line bookshop as long as appropriate *Class Script* realizing *bookInterface* are provided for the target bookshop. This can be regarded as improving the portability of the script.

Script interferences. Companion scripts refer to those scripts that are simultaneously executed, then acting upon the very same DOM tree. The problem is that programmers develop scripts from the raw DOM, being unaware of changes conducted by other companion scripts. So far, scripts are enacted sequentially based on the time they were installed. This implies changes made by the first script *to be visible* to ulterior scripts. Two types of dependencies arise: read dependency (a script can accidentally read data written by a previous script), and write dependency (the injection point can be displaced by the writing of a node made by a previous script). As a result, the very same set of scripts can deliver different outcomes depending on the time they were installed.

Scripting Interfaces alleviate these problems. Read dependencies are obviated by making script changes transparent to other scripts. Scripts can only access the raw DOM, i.e. the DOM sent by the web server, previous to being updated by the scripts. Implementation wise, this is achieved by imposing *Class Scripts* (i.e. those accessing the DOM) to be executed before any *Mod Script* (i.e. those updating the DOM). Since *Class Scripts* are the first to be executed, the raw data is first captured as event payloads. Then, *Mod Scripts* take their data from these payloads. Therefore, there is no risk of a *Mod Script* reading data introduced by other *Mod Script*.

As for write dependencies, they are avoided by describing the writing point (i.e. the injection point) in terms of concepts (e.g. the location where a book is rendered) rather than directly addressing the DOM nodes. The injection point becomes logical rather than physical. If two *appendChild* scripts acts upon the very same concept, the "interference" is only noticeable in the order in which the children are rendered.

Readability. Both subscription and notification of *conceptual events* is easier to understand that their code counterparts (i.e. HTML scraping with XPath expressions). Additionally, both *Interface Scripts* and *Class Scripts* are *JSON* structures, hence, more declarative than *JavaScript* code. Besides improving legibility, this makes scripts liable to be automatically generated and processed. Indeed, *Class Scripts* can be automatically obtained using *MashMaker*'s data extractors [7] where XPath expressions are easily obtained by directly clicking on the page rendering (see section on related work).

Collaborative development. Compared with traditional practices, this proposal makes explicit a three-stage process during script development: interface definition, interface realization and mod-logic implementation. At first glance, this could look cumbersome compared with current practices where a unique script is needed. This additional effort can payoff in the following scenarios: the website suffers frequent updates, the very same mod logic is reused for distinct websites, or the page scraping part is complex enough to advice for separation of concerns. Additionally, this separation of concerns promotes collaborative development of scripts. For instance, a community interested in a certain topic (e.g. book price comparison) can provide interfaces that isolate this topic from its realization in distinct websites and then, releases *Class Scripts* for other programmers to capitalize upon (e.g. through *userscripts.org)*. This resembles the genesis of *microformats* where *microformat* tags are first introduced, and later, become *ad-hoc* standards as the rest of the community adopts them.

8 Related Work: Scraping, Scripting, Mashuping

Web scraping is the process of automatically converting Web resources into a specific structured format. For instance, *Piggy Bank* is a plugin for Web browsers that "lets Web users extract individual information items from within Web pages and save them in Semantic Web format" [8]. Both the extraction technique and the target format serve to characterize scripting tools [11].

Mashuping overcomes scraping by addressing not only data extraction but also how this data is combined in novel ways [14]. For the purpose of this work, mashup approaches can be classified as compositional and "customizational" based on the role played by the source applications. Compositional approaches are akin to *integration* efforts to built *new applications* out of existing resources. This is, most mashup examples aggregate data coming from different sources to conform a new application in its own right, detached from the source websites. *Yahoo Pipes* is a case in point [6]. By contrast, "customizational" approaches focus on a given application which is then leverage using mashups. The mashup can only be understood that by referring to the customized website. *MashMaker* is one of the few examples. According to its authors, *MashMaker* augments "the familiar web browsing interface that the user already uses to browse data, and enhances this with mashed up information" [7].

Scripting aligns with customizational mashup efforts. *Greasemonkey* scripts also act within the realm of Web pages. The difference stems for the target audiences. *MashMaker* is oriented towards end users. Do-it-yourself is a main tenant of the mashup movement. The downside is expressiveness. *MashMaker* amendments can only occur at the time the page is load. By contrast, scripts can be attached to any DOM event. *MashMaker* amendments tend to be gadgets as reusable components which end users can easily plug into the target application. By contrast, scripts can attach any HTML fragment where deletions are also possible (e.g. removing banners). Basically, any *MashMaker* amendment can be achieved through scripting but not the other way around. Notice however, that the very same amendment that takes some few minutes to complete in *MashMaker*, could become a lengthy scripting effort[6].

This also supports the rationale for this work, i.e. *Scripting Interface* as the means to preserve the costly development of scripts. Indeed, our work strives to detach scripts from the underlying Web pages. Events are the means to realize both data extraction (i.e. publishing events) and data injection (i.e. processing events). By contrast, loose coupling is not a priority in *MashMaker*. *MashMaker* couples data extractors (i.e. the counterpart of *Class Scripts*) and gadgets (i.e. the counterpart of the *Mod Script*). The data is extracted as a raw structure which should coincide with the parameters of the gadget to be plugged. In practice, this implies that each gadget has its own data extractor. More to the point, the location where the gadget is to be injected is limited to the place the data is extracted. This is in contrast to *Scripting Interfaces* where the very same *Mod Script* can inject its output at different locations raising distinct *Processing events*.

Is it possible to obtain the best of both worlds, i.e. the easiness of *MashMaker* and the flexibility of scripting? As an attempt, we were able to obtain *Class Scripts* out

[6] Of course, if you have to program your own *MashMaker* gadgets then this is a complete different matter.

of *MashMaker* data extractors. This permits *Greasemonkey* programmers to resort to *MashMaker* to obtain their *Class Scripts*, and move down to *JavaScript* to code their own amendments without being limited to reusing existing *MashMaker* widgets.

9 Conclusions

This work introduces *interfaces for scripting* to shelter user scripts from changes in the underlying Web pages. These interfaces are realised through standard scripts that generate *conceptual events* from UI events. The approach is aligned to *JavaScript* practices (event-based) and supported through standard *Greasemonkey* scripts. No plugin is required. Preliminary experiments suggest that this approach does not convey main disruptions for programmers while improving both change resilience and readability of scripts. The presumption is that improving change resilience will lead to greater user implication and more sophisticated scripts.

Next follow-ons include how to engineer sites to be more user-script friendly, with the possibility of providing the class/interface part directly from the site. At this regard, the possibility of automatically generating the class/interface parts using widgeting tools such as *MashMaker*, has so far being very encouraging.

Acknowledgements

This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2008-06507-C02-01/TIN (MODELINE), and the Avanza I+D initiative of the Ministry of Industry, Tourism and Commerce under contract TSI-020100-2008-415.

References

1. Greasemonkey Homepage. http://www.greasespot.net/.
2. Greasemonkey in Wikipedia. http://en.wikipedia.org/wiki/Greasemonkey.
3. JSON (JavaScript Object Notation). http://json.org/.
4. JSON Schema. http://json-schema.org/.
5. W3C DOM Level 2. http://www.w3.org/DOM/DOMTR#dom2.
6. Yahoo Pipes. http://pipes.yahoo.com/pipes/.
7. R. Ennals, E. A. Brewer, M. N. Garofalakis, M. Shadle, and P. Gandhi. Intel Mash Maker: join the web. In *SIGMOD Record*, 2007.
8. D. Huynh, S. Mazzocchi, and D. R. Karger. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. In *the 4th International Semantic Web Conference*, 2005.
9. Java Community Process (JCP). JSR 168: Portlet Specification Version 1.0, 2003. http://www.jcp.org/en/jsr/detail?id=168.
10. R. Khare and T. Çelik. Microformats: a pragmatic path to the semantic web. In *the 15th International Conference on World Wide Web*, 2007.
11. N. Kushmerick. *Encyclopedia of Database Systems*, chapter Languages for Web Data Extraction, page 1595. Springer, 2009.
12. D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15:1053–1058, 1972.
13. G. D. Magoulas S. Y. Chen, editor. *Adaptable and Adaptive Hypermedia Systems*. IRM Press, 2005.
14. J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12:44–52, 2008.