Journal of Web Engineering, Vol. 10, No.4 (2011) 289-315

A CROWDSOURCED APPROACH FOR CONCERN-SENSITIVE INTEGRATION OF INFORMATION ACROSS THE WEB

SERGIO FIRMENICH, GUSTAVO ROSSI, SILVIA GORDILLO

Facultad de Informática, Universidad Nacional de La Plata and Conicet Argentina

[sergio.firmenich , gustavo, gordillo@lifia.info.unlp.edu.ar]

MARCO WINCKLER
IRIT, Université Paul Sabatier, France
winckler@irit.fr

Received November 3, 2011
Revised December 22, 2011

Currently, users navigate across the Web in order to accomplish complex tasks. Frequently, these tasks involve working with many different Web applications. For this kind of tasks users have to move information manually among Web pages. While in the field of Adaptive Web there is important and meaningful research about adapting Web sites according with the user´s activity most adaptation approaches are applied only in the boundaries of single Web applications. In this work we show to use Client-Side adaptation to provide new techniques for concern-sensitive, task-based adaptations across the Web. We show an architectural overview of our tool and illustrate the power of the approach with examples.

Key words: Concern-sensitive navigation, Information Integration, Client-Side adaptation
Communicated by: O. Diaz & S. Auer

1 Introduction

As wisely pointed out in [8], one of the most interesting facets of Web evolution is the kind of end-users interaction with Web contents. At first, users could only browse through contents provided by Web sites. Later, users could actively contribute with content by using tools (e.g. CMS, wikis) embedded into these sites. Currently, users are interested in adapting contents according to their preferences. There is meaningful research about the adaptive Web [5], but these works are applied within the boundaries of single Web applications. Currently, users navigate from one application to another one unpredictably, and some tasks performed in the Web are accomplished by using several applications. This current use of the Web creates new challenges regarded to adaptability on Web applications in order to integrate them in different ways.

To solve the lack of information integration among Web applications new technologies have arisen, especially tools that allows users to change the way Web content is presented and combined. For example, using visual Mashups [7], users can compose content hosted by diverse Web sites and

they can run Greasemonkey scripts [12] to change third part Web applications by adding content and/or controls (e.g. highlight search results in Amazon.com which refer to Kindle).

These tools built under the concept of Web augmentation [2] extend what user can do with Web contents, but they provide limited support for tasks that require navigation on multiple Web sites.

Imagine a user interested in participating in a conference. A possible scenario is shown in Figure 1. First he enters to the conference Web site in order to obtain the dates, and then he might possible navigate to Wikipedia for seeing information about the conferences place. After that, he could use a Web application in order to submit a paper, another one for booking flights, and finally the last system in order to make the registration in the conference, hotels, etc.

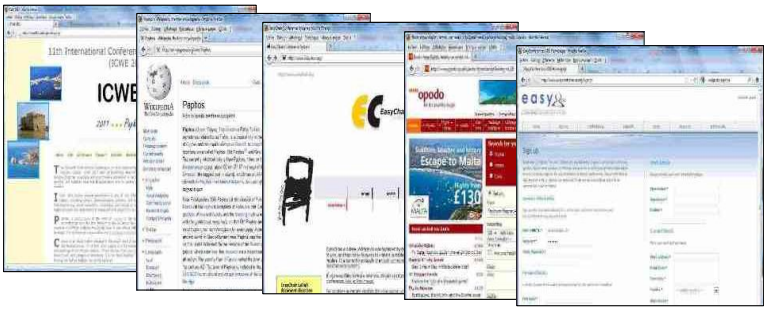

Figure 1: Inter-application navigation with a single concern

In cases like this, users are not supported with a mechanism for integrating information. When they change the application in use, their context is usually lost and they have to re-enter information, pay attention in which part of the new Web page this information is important, etc.

Previously mentioned techniques such as mash-ups and Web augmentation cannot be directly applied in these cases. For example, a user who is using the Web for planning a holiday trip to Paris might ultimately visit several sites such as expedia.com for flights, booking.com for hotels, wikipedia.org for general information about the city and parisinfo.fr for points of interest, current events or expositions in Paris. From the users' point of view, the navigation of all these sites is part of the same task. The existing augmentation techniques are of little help in this case. For instance, GreaseMonkey scripts can adapt the content on a specific Web site but it will require much effort to make it generic enough to integrate information provided by different applications. Mashups, meanwhile, can be used to integrate content from several Web sites; however, a Mashup for expedia.com will not necessarily integrate information from other users' preferred Web sites (e.g. airfrance.fr, venere.com...). If these sites provide public APIs, Mashups can be extended, but it does not prevent users to learn how to do it beforehand. Quite often, users' tasks are associated with opportunistic navigation on different Web sites, which is difficult to predict [17]. In this context, effective Web augmentation should overcome two main barriers: i) to take into account different applications which are visited by users (either through explicit navigation or just opening a new browser's window with the corresponding URL); and ii) to adapt the unknown target Web sites, considering that the user might need different kind of adaptations at different sites.

This paper proposes a framework for creating flexible, light-weighted and effective adaptations to support users' tasks during the navigation of diverse Web applications. Our goal is to support users'

tasks by keeping his actual concern (and related data) persistent through applications. For example, allowing that dates, used on expedia.com for booking a flight, could be reused as input for booking.com while booking hotels in the same period. Another example of adaptation that illustrate our approach is the inclusion of new links allowing users to easily navigate from parisinfo.fr to related articles at wikipedia.com whenever one needs further explanation about a topic.

In a previous work [10], we showed how to profit from the knowledge of the current user's concern to improve navigation in Web applications, by enriching the target page with information or links which are useful in that specific concern. In this paper, we push further this approach to allow adaptations that go beyond a single application's boundary. Moreover, we present a framework and a set of tools which allow simplifying the process of concern-sensitive Web augmentation, reducing the programming burden, and therefore allowing end-users to configure their own adaptations even when they are complex as in the example above.

This paper is organized as follows: in section 2 we provide an overview of related works. In section 3 we explain Concern-Sensitive integration of information. The framework is fully described in section 4. Section 5 presents tools built upon the framework. Section 6 presents how we have validated our approach with end-users. Finally, in section 7 we present conclusions and future work.

2 Related Work

The field of Web applications adaptation is broad; therefore, for the sake of conciseness we will concentrate on those research works which are close to our intent. The interested reader can find more material on the general subject in [5]. Some works like [4] are based on navigation in order to perform adaptations. As stated in the introduction we can identify two coarse-grained approaches for end-user development in Web applications: i) mashing up contents or services in a new application and ii) adapting the augmented application, generally by running adaptation scripts in the client side.

Mashups are an interesting alternative for final users to combine existing resources and services in a new specialized application. Visual and intuitive tools such as [7, 21] simplify the development of these applications. Since most Web applications do not provide Web services to access their functionality or information, [13] proposes a novel approach to integrate contents of third party applications by describing and extracting these contents at the client side and to use these contents later by generating virtual Web services that allow accessing them.

The second alternative to build support for users tasks is Web augmentation [2], where the target application is modified (adapted) instead of "integrated" in a new one. This approach is very popular since it is an excellent vehicle for crowdsourcing. Many popular Web applications such as Gmail have incorporated some of these user-programmed adaptations into their applications, like the mail delete button (See http://userscripts.org/scripts/show/1345). The most popular tool to support Web augmentation is GreaseMonkey [12], whose scripts are written in JavaScript. The problem with these scripts is their dependence on the DOM; if the DOM changes the script can stop working. In [8] the authors propose a way to make GreaseMonkey scripts more robust, by using a conceptual layer (provided by the Web application developer) over the DOM. In [9] the authors extend the idea to allow scripts developers to write their own conceptual abstractions to cut the dependency with unknown

developers; in this way, when the DOM changes, the maintenance is easier because only the matching between the concepts and the DOM need to be redefined.

There are some research works aimed to improve the user experience by supporting them in their tasks, particularly for repetitive ones. This problem has been tackled in [14] with the CoScripter tool. CoScripter is a Firefox Plug-in which allows users to record their interactions with a Web Site, and then, they can repeat the process automatically later. The approach is quite flexible; for example, the whole process can be repeated with different information in form inputs for each interaction with a site, but always using the same fixed Web sites. In this way, CoScripter is not useful when users need to change slightly the process, for example by changing which is the target Web application. However, both CoScripter's goals and our approach's goals are different, because with CoScripter Web applications are not adapted (not further that fill forms with values) and volatile requirements are not contemplated.

Other close research work is [18], which is also realized with a Firefox plug-in, to improve the user experience by empowering his browser with commands with different goals. With MozillaUbiquity users execute commands (developed by them) for specific operations, for example to publish some text from the current Web page in a social network. Anyway, these commands are executed under user demand, and adaptations are not made automatically. Although MozillaUbiquity shorten the distance between two distinct Web Applications, moving information from one of them to another is not fully exploited.

While we share the philosophy behind these works, we believe that it is necessary to go a step further in the kind of supported adaptations. In [10] we showed how to use the actual user concern (expressed in his navigational history) as an additional parameter to adapt the target application. By using the scripting interface approach we managed to make the process more modular, and by defining adaptations for application families (e.g. social networks) we improved the reuse of adaptation scripts. In the following sections we show how to broaden the approach allowing end users to select which information can be used to perform the adaptation, therefore improving the support for his task and providing support for building more complex adaptations.

3 Concern-Sensitive Integration

This work aims to fill the existing gap found when users navigate from one Web application to another one, or simply change the application in use by typing a new URL. We want to provide users with mechanisms to adapt and integrate information among the Web considering that several tasks they perform involve many Web sites. In this sense we need to be clear about what a concern is, and how the navigation is sensitive to it. First in this section we briefly explain what concern-sensitive navigation is. Then, we separate the explanation of Concern-Sensitive integration in two subsections. One section is dedicated to those cases in which CSN have been designed. In second subsection we show how our approach contemplates volatile requirements too. A new approach to tackle volatile concerns which requires dynamic and mechanisms for integrate information by performing adaptations over Web applications.

3.1 Background of Our Approach: Concern-Sensitive Navigation

In [10] we said that a Web application supports CSN when the contents, links and operations exhibited by the application pages are not fixed for different navigation paths, but instead can change when accessed in the context of different navigation concerns. To make this definition concrete and practical we assume that users navigate through *navigation objects* which are the realization of hypermedia nodes. Suppose a navigation object Nj which is an instance of a navigation object type N. While in conventional Web navigation, this object will exhibit the same contents and links regardless of how it was reached, in CSN its properties can be slightly adjusted according to the current user concern as shown in Figure 2. While Nj comprises two attributes (Figure 2.1.), when accessed in a specific concern C it also exhibits an additional attribute and an anchor for a link (Figure 2.2).

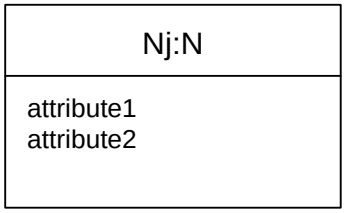

Figure 2.1: Conventional navigation to Nj

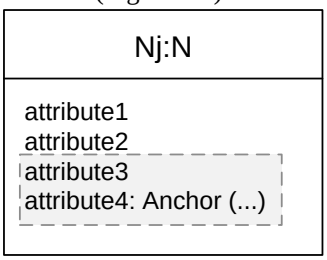

Figure 2.2: Nj accessed in C

While the basic definition of CSN does not impose limits to the kind of variations which navigation objects might suffer when accessed in different concerns, a disciplined use of these ideas (i.e. in the context of a solid design approach), have a stronger positive impact on application usability. As explained in the following subsection, the concept of CSN is strongly related to the concept of object roles, as applied to navigation objects.

Our framework is based on the concept of concern-sensitive navigation (CSN), but applied over existing Web applications like in [10]. Note that concern-sensitive navigation is different from context-aware navigation, where other contextual parameters (location, time, preferences) are considered.

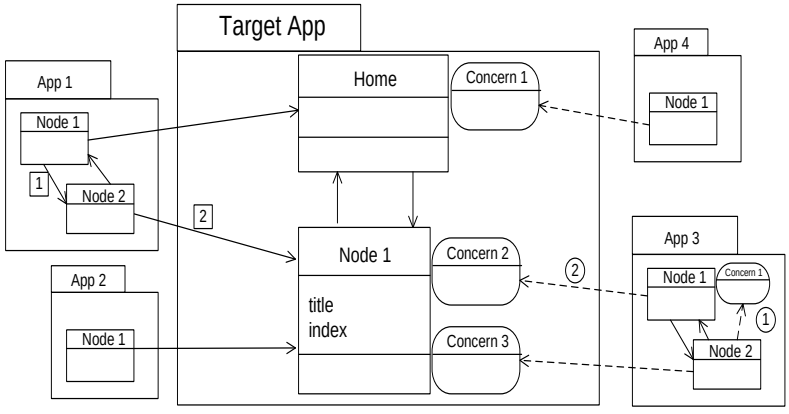

Figure 3: Flat Navigation vs. Concern-Sensitive Navigation

Figure 3 illustrates the differences between flat and concern-sensitive navigation. Basically, each arrow represents a link which enables users to navigate from one node to another one. Two simple navigational scenarios are defined; in one of them the navigational steps are marked with numbers into boxes while in the other steps are shown in circles. Both scenarios have the same final node, but they are different since one of them uses flat navigation (those marked with boxes) and the other one with concern-sensitive navigation (marked with circles). Note that there are two kinds of navigations: Flat navigations (represented with solid arrows) where the target Web pages show always the same information, without taking into account the source of navigation; in concern-sensitive navigations (represented with dashed arrows) meanwhile, the target pages adapt or enrich their contents by taking into account what was the user concern in the previous page.

In [10] we have argued that concern-sensitive navigation simplifies the user's tasks by providing him sensitive information or options according to his current needs. We have also introduced an approach to build smart client-side adaptations, implemented as browsers' plugins, which allow making specific Web applications aware of the concern in which they were accessed, changing contents and links in consequence.

In the following section we explain our current approach for Concern-Sensitive Integration, which was presented in [11], as an important part for support users tasks among the Web.

3.2 Designed Concern-Sensitive Integration

In [10] we have shown an approach for engineering CSN by developing GreaseMonkey (GM) scripts with our tools. With Concern-Sensitive Navigation we improve user experience by adapting Web pages with new functionality or content, or even adapting these, in corresponding with his concern in previous Web pages. Figure 4 shows an example of concern-sensitive navigation across two applications: Google Maps (as the source of navigation) and Wikipedia (as the target). The left-side displays Wikipedia links in the map of Paris; once selected, these links trigger the page at the right-side of Figure 4, augmented with the corresponding map and a set of links to those Wikipedia articles in the surroundings of the current one.

Figure 4: Inter-application CSN between Google Maps and Wikipedia

In general, the task of CS adaptation of a page P requires that we: (a) know the actual user's navigation concern (i.e. pages previous navigated, e.g. Google maps), (b) record the set of relevant information from previously visited pages that are needed for adaptation (e.g. the current map), and (c)

have the capacity for enriching P with contents or links related with (a) and (b) by intervening in P's DOM.

Note that if we are trying to adapt existing Web pages we need to know how to manipulate the underlying DOM structure. Clearly, this kind of enhancements requires the knowledge of Web sites involve. For instance, in the example from Figure 4, it is not possible to add the new navigational menu (which is based on the concern from google maps) without to know how to manipulate the Wikipedia DOM. In other words, it means that the concern-sensitive behaviour has to be designed and implemented to be executed under certain circumstances. In the example, it is executed when a user in Google Maps navigates to Wikipedia. Behind this navigation there is a software component watching what is happening and this acts in consequence. In our approach these components have in their scope abstract models like those described in [9]. With models like this DOMs can be manipulated in a more abstract way, and it not only makes more robust script but it allows scripts to be reutilized in more than one application if these have similar abstract models.

An approach to support Concern-Sensitive integration has to contemplate a flexible way to create new software component like this, in order to allow users to create new integrations among the Web.

These Concern-Sensitive components are part of our approach and we named them *scenario*, which are fully explained in section 4.

3.3 Volatile Concern-Sensitive Integration

In order to achieve concern-sensitive integration/adaptation automatically we need to know what users are trying to do in order to predict their next step, and then, help them in this task. To achieve this several approaches for observing user behaviour and data acquisition have been defined [6]. These mechanisms observe the user's activity to process it and infer the following step (adapting the target page). Based on the results of this observation process, adaptation approaches are used for customizing Web sites according; a classic example of this kind of dynamic navigation can be found in [4]. However, users' behaviour is not always predictable; in consequence, not always the data needed for adaptation can be collected with precision. Finally, adaptations may not be helpful or even may not be performed because this lack of information. Our approach is perhaps more humble and at the same time realistic and user-oriented.

In our approach, we use *scenarios* as the tool for collecting data about user activity and acting (automatically) in corresponding with that. But not always a scenario is available. Imagine the scenario from Figure 4, where a user is using GoogleMaps. Instead of navigating to Wikipedia (which the scenario would be expecting for) s/he decides to search something about Paris in google.com and then s/he opens any resulting Web site. Probably he did not know this Web page and in consequence, he does not know about any *scenario* to improve it with the GoogleMaps concern.

We think that scenarios are really useful, but actually, not all users concerns can be foreseen, much less the user activity. For example, many times users navigate unpredictably but still with a concern in mind. As part of a full approach, these kinds of concerns have to be contemplated in some way.

Therefore, we provide users with tools for adapting Web sites under demand (not automatically) and we resume this kind of behaviour as volatile requirements of adaptation.

It is not important if adaptations are made automatically or under user demand, an adaptation may require: a) a way to collect information related to the concern, b) a way for utilising the information collected for supporting user while performing some task. Clearly in automatic approaches these two parts are made automatically, meanwhile, in this work we present tools for doing it manually too.

We believe that it is important to provide users with tools for collecting and moving information among the Web. It is not casual that multi-tab Web browsers have had success, since users can open more than one application at the same time and then move information in a "handcraft" way, for example by copying & pasting.

As the counterpart, once users are provided with mechanisms for manual data collection, they need to utilize the collected information. As we said before, with automated mechanism for data collection, the use of these data for performing adaptations is usually automated too. However, when data is collected manually it would be important to provide users with tools for executing adaptations manually too.

In order to contemplate this kind of volatile concerns, our approach for integration of information allows users to both collect data among Web sites and perform adaptations over these Web sites with the previously collected data. On the one hand, in this work we have developed a tool named *Pocket* for data collection. On the other hand, tools named *augmenters* are defined to define adaptation actions which are performed by both user – under their demand – and *scenarios* – for automatic adaptations –.

In the following section we explain all these components with more details and show several examples about concern-sensitive integration of information.

4 A Framework for Concern-Sensitive Augmentation

4.1 The Approach in a Nutshell

The CSN approach works well for application families (e.g. plugins that work for similar applications which share some features). However, it "only" provides end-users with a fixed set of adaptations. We have developed a software framework which extends the concept of CSN by providing different kinds of users (end-users, developers, etc) a set of tools to augment Web applications by considering the actual user concern. Developers can use the framework to implement new adaptation functions, named augmenters. Augmenters are built as generic adaptations featuring behaviours such as automatic filling in forms, highlighting text, etc. End-users can benefit of these augmenters during navigating by "collecting" concern information to be used when adapting the user interface (see section 3.3). By combining augmenters, the framework also supports scenario engineering for developing customized adaptations for specific domains such as trip planning (See section 3.2). For example a scenario can be based in the use of the form filling augmenter when the user is navigating among several Web sites for booking flights and hotels. The same form filling augmenter can be used to fill forms related to a product search in different e-commerce Web sites, for example by taking the department (e.g. electronics) and the keyword (e.g. iphone4) used in amazon.com to complete the form automatically in fnac.fr.

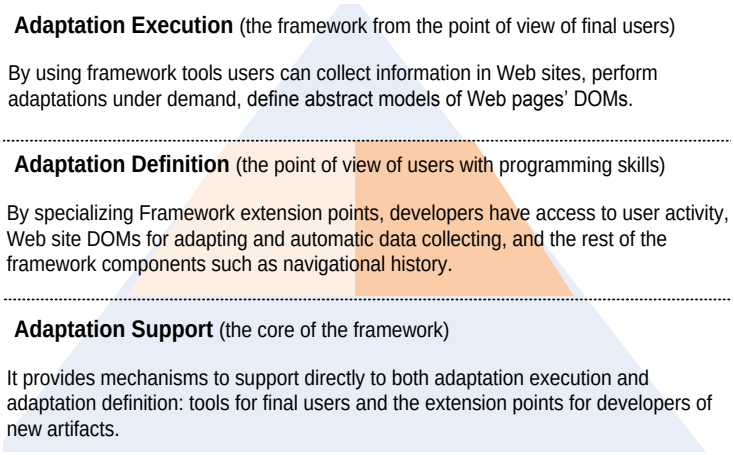

Figure 5: A functional overview of the framework

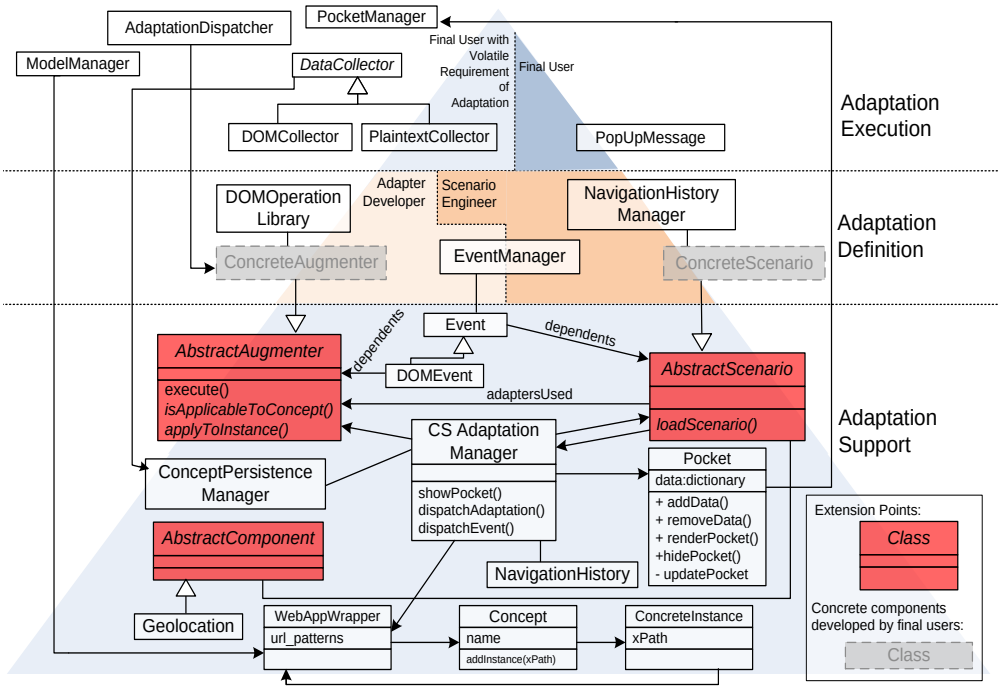

Figure 6: Framework structure

The framework is described using the pyramid approach [16]. First, we show in Figure 5 a generic scheme in order to provide a functional overview of the framework. In Figure 6 we complete this structure with the relevant components for each layer. The top levels are more abstract while lower ones are more detailed. At the top layer, final users can collect relevant information for their current task or concern by using the DataCollector tool. Then, when they navigate to other sites they are able

to execute augmenters using this information; in this way they can satisfy volatile requirements of adaptation (not foreseen by developers). At the middle layer, end users with programming skills can extend the framework by developing augmenters and scenarios as classes inheriting of AbstractAugmenter and AbstractScenario, two outstanding framework hot-spots. The bottom layer shows a more detailed view of the framework design; a third hot-spot, AbstractComponent abstracts concrete components used in scenarios; for example we developed a component which offers geo-location information; another tool could empower the scenarios by giving them auto fill forms capabilities (e.g. a component that implements carbon [1]).

Framework components act like libraries to be used for developing adaptations. We briefly outline the main framework components:

- Adaptation Support Layer
    - ClientSideAdaptationManager: is the Framework's core, whose functions are to coordinate others elements and to serve as communicator with the browser.
    - WebAppWrapper: is a wrapper which manages all information of a particular Web site (determined by URL patterns). For example, it manages the abstract model generated over a Web site DOM.
    - Concept: is the class which manages the DOM abstractions. When a model is created for a particular Web site, the corresponding WebAppWrapper is created and then, this has related a set of Concept which belongs to the model.
    - ConcreteInstance: is the concrete instance of a concept. A concrete instance of a concept has a xPath for a specific WebAppWrapper. For example, the concept *SearchResult* could have an instance (and its xPath) for the google.com WebAppWrapper and another one (with other xPath) for the WebAppWrapper of yahoo.com. In this way, an adaptation can be developed in terms of the *SearchResult* concept; then it will work in any Web site whose model has a ConcreteInstance of this concept.
    - NavigationHistory: is the navigation history object provided by the browser. We have developed a wrapper on top of it to ease scenarios development.
    - ConceptPersistenceManager: is responsible for saving and restoring user data into the local files system.
    - AbstractAugmenter and AbstractScenario: are abstract classes from which concrete augmenters and scenarios, correspondingly, developed by users must inherit.
    - AbstractComponent: is an abstract class used for extending the framework by developing components to support new capabilities (e.g. geolocation).
- Adaptation Definition Layer
    - DOMOperationLibrary: a library that operates with DOM elements; it raises the level of typical JavaScript sentences easing the development of augmenters.

  - EventManager: is the responsible of adding and removing listeners (Adaptation Definition Layer) of events from the lower layer.
  - NavigationHistoryManager: is a wrapper with which scenarios can make queries about navigation history.
  - ConcreteAugmenter and ConcreteScenarios: are scripts developed by users with programming skills. These classes are shown in Figure 6 in order to highlight their place in the hierarchy. Some concrete augmenters as HighlightAdapter, WikiLinkConverter, CopyIntoInputAdapter are included in our framework.

- Adaptation Execution Layer
  - DataCollector: is the tool to allow users collecting information while navigating. So far, two concrete DataCollectors have been implemented: one for selecting plaintext information, and another to handle DOM elements.
  - PocketManager: is our tool to allow users to move information among sites.
  - AdaptationDispatcher: is the responsible of executing an adaptation under user demand. It is useful to accomplish volatile requirements of adaptation.
  - ModelManager: it allows users to create and instantiate Concept and ConcreteInstance for the model of a particular Web site. Note that this tool is a bit different from DataCollectors. The models created by ModelManager are usually created before developing adaptations; it means that adaptations can be based on these models. With DOMCollector users can collect DOM elements under demand in an unpredictable way; meanwhile, the data from the model is always available.

4.3 Extending the Framework

The framework can be extended in two ways: by creating new augmenters (generic basic adaptations), and by building scenarios (for supporting specific user's tasks). Although we do not restrict the kind of adaptations, we fully support the development of adaptations which take into account the actual user concern. Since many times it is not enough to be aware of the user's navigation history to fully know his concern, further information about his current activity is often needed. The example given at Figure 4, shows how some information is moved from GoogleMaps to Wikipedia. Our framework offers two kinds of tools to move information among Web sites. The first one is the DataCollector with which users can select elements from the current Web page. The elements selected are added into the second tool named Pocket which can store either simple plain text or data with some semantic meaning as a concept name. Once the information is stored into the Pocket, it will remain available for any Web pages visited later on. Section 5.1 details how users collect information during navigation.

4.3.1 Creating Augmenters with the Framework

The simplest way to extend our Framework is to develop a new augmenter. An augmenter is an adaptation component developed by users with programming skills. Augmenters have two main contributions in our adaptation approach: they provide tools for satisfying end-users' volatile

requirements for adaptations and they support the development of sophisticated scenarios built by combining simpler augmenters.

An augmenter can be standalone or be executed with data collected as argument; in this case this data is assigned by the actor who triggers the augmenter execution (either a scenario or the user). For example, an augmenter aimed to highlight elements in the page, must be able to do it for an element (for example the City instance "Paris") or for a collection of elements (for example, all City instances). Therefore augmenters should be flexible with regard to the user's needs. Figure 7 shows an augmenter (WikiLinkConvertion) applied to parisinfo.com with the user coming from wikipedia.com with his PointOfInterest instances (these are strings collected from the Web pages visited and conceptualized or typed as PointOfInterest) in the Pocket (the floating box showed at left in the Figure). As Figure 7 shows, the augmenter WikiLinkConvertion is applied to any PointOfInterest occurrence in the page. Note that when the user right clicks over PointOfInterest, a menu with the available augmenters is opened and then he chooses "Convert to Wiki Link", so WikiLinkConvertion is executed with all instances of PointOfInterest as parameters.

Fig 7. Plain text converted into links to add personal navigation

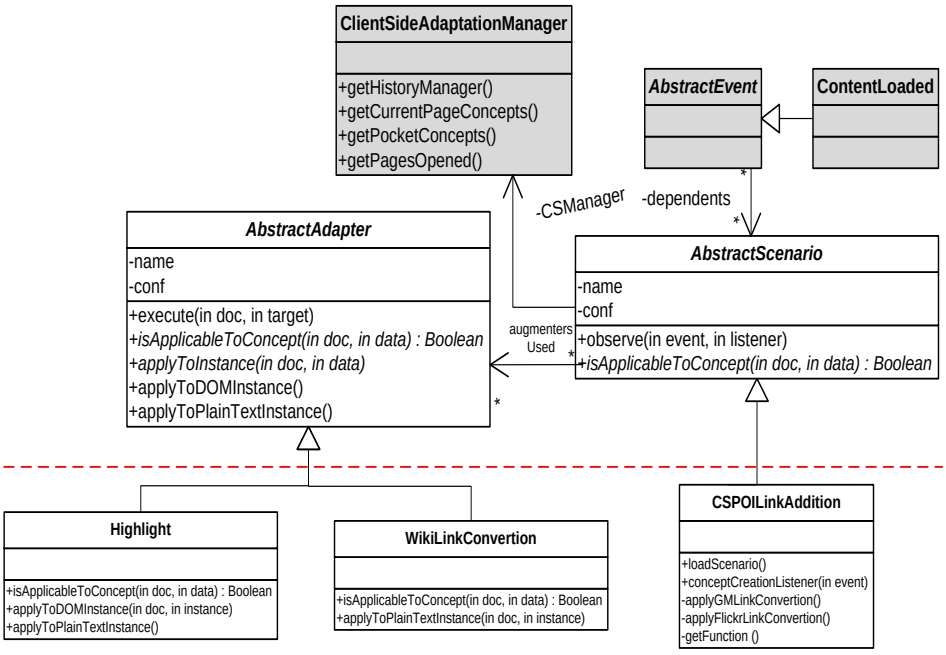

Figure 8: class diagram presenting the framework extensions with augmenters and scenarios.

Since augmenters can be applied to different Web pages they must be developed without a dependence of a particular DOM, as described in [8]. Moreover, when using the framework, developers must:

- As Figure 8 shows, developers have to construct an augmenter as a JavaScript object inheriting from *AbstractAugmenter* (the hot-spot shown in Figure 6).
- Implement the methods defined as abstract in *AbstractAugmenter*. This is necessary because the *execute()* method of *AbstractAugmenter* (a template method) sends messages to concrete augmenters. Since the method *execute()* is the starting point of an augmenter, if a message can not be dispatched, the execution will fail. The method *execute()* receives data as parameter which is used to perform the adaptation.

Manipulating the DOM to adapt the page is a responsibility of augmenters. Since DOM manipulation can be hard for users, the framework provides them with the DOMOperationLibrary, a component inspired in the most popular JavaScript libraries like Prototype (see http://www.prototypejs.org/) and jQuery (see http://jquery.com/) to make DOM manipulation simpler. In this way, target DOM elements (those that are abstracted by elements from the Pocket) are easily manipulated by operations like style changes, hiding, removing, or adding content.

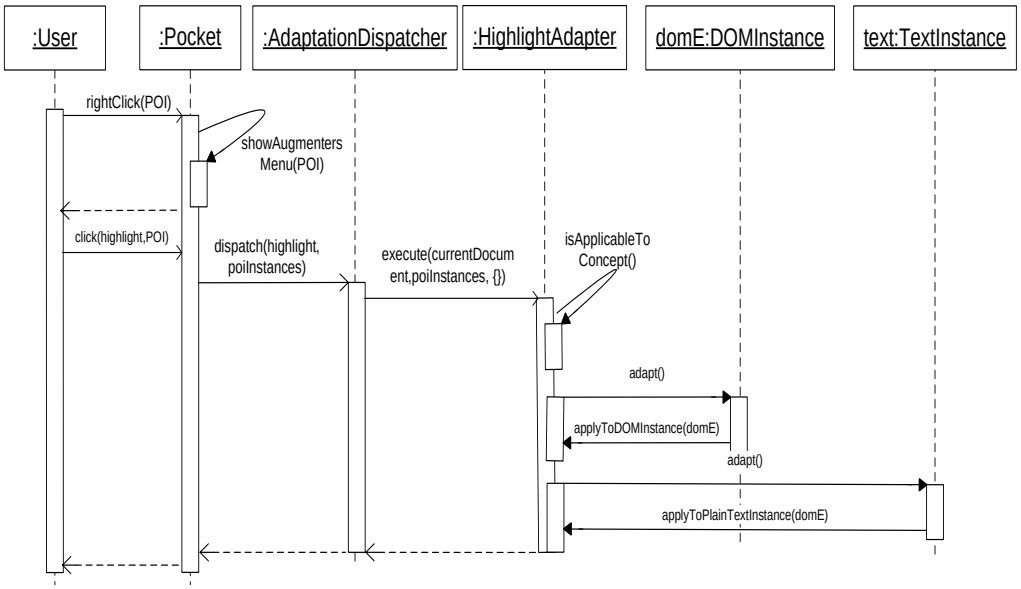

Figure 9: Sequence diagram describing user triggering an augmenter.

Augmenters are executed when a user explicit triggers them or when a scenario is instantiated. In Figure 9, we show a sequence diagram to demonstrate how the framework chains the execution of augmenters. The object User represents the real user. First, the user chooses an element from the Pocket and when he right clicks over it; a menu is opened with all augmenters available. When he selects one of them, the Pocket sends the *dispatch()* message to the *AdaptationDispatcher* that finally executes the augmenter with the *execute()* message. Note that when an augmenter receives the execute

message, it sends to itself both the *isApplicableToConcept* in order to determinate if the augmenter can be applied over a set of instances. In both cases when the target is only one instance and when the target is a set of them, the augmenter uses double dispatching for adapting each instance. It is because two instances of a concept can be different, while one of them could be plain text collected by the user, another one could be a DOM abstraction element. Clearly, the augmenter would have different behaviour for adapting each of them. For instance, highlight some text is different than highlight a DOM element. Then, the augmenter sends *adapt()* message to each target instance. In consequence, the instance responses by sending the corresponding message *applyToDOMInstance()* or *applyToPlainTextInstance()*. These are too hook methods of AbstractAugmenter, the concrete ones have to redefine them. All augmenters developed by users must have these methods defined as in the augmenters showed in Figure 8.

4.3.2 Creating Scenarios with the Framework

Augmenters are useful to perform simple tasks on a site; however, for complex tasks users perform sets of activities, many times following pre-defined patterns. For example, booking flights, and then booking a hotel is a common scenario. In different moments (and moreover for different users) the Web pages used to do these tasks may change. However, the information used during the task is similar and the kind of adaptation needed too. For example depart date, arrival date, and a destination are all the pieces of information needed to perform (in a simplified view) this task in any Web site of this kind.

A scenario is an event-driven script; it registers listeners for those events in which it is interested in. These events usually refer to the user activity as when he opens sites or collects new data. When an event occurs the scenario is loaded and it first checks that the information it needs is available; if so, the scenario is instantiated. Scenarios execute adaptations when some conditions (e.g. about the navigational history or collected data) are satisfied; to perform adaptations, they trigger augmenters that change the DOM. A scenario could execute the same augmenter, but with different arguments as Figure 10 shows. In Figure 10.1 a Wikipedia article is adapted in the context of a scenario. The scenario uses the LinkAdditionAdapter (an augmenter similar to the one described in the previous section) to add a link close to each occurrence of the target element. In Figure 10.1, the target elements are all instances of the PointOfInterst concept, and the adaptation is executed automatically when Flickr.com appears in the navigational history. Figure 10.2 shows a similar case, but now since GoogleMaps is the previously visited Web page, a link to GoogleMaps is added.

Figure 10.1: Navigation with Flickr concern. Figure 10.2: Navigation with GoogleMaps concern.

A scenario is realized in a quite similar way than augmenters (in the sense of being a JavaScript file) but with some distinct features to register its interest in different events. The scenario engineer has to respect these constraints:

- Construct the scenario as a JavaScript object inheriting from AbstractScenario, the hot-spot shown in Figure 6.
- Implement the methods defined as abstract in AbstractScenario. There are methods that will be executed during initialization when the browser is opened. Note, for example, that scenarios can be interested in different events; therefore they must register listeners which will be executed in order to instantiate the scenario when the events happen. The same kind of inversion of control occurs when the framework sends the loadScenario() message in order to wakeup the scenario.
- Specify which augmenters are necessary to carry out the scenario.
- Specify the set of concepts needed to instantiate the scenario and define them in the DataCollector tool; thus, when users collect data, the available concepts or types are those in which the scenarios are interested in (e.g. destination, dates, etc).

A scenario needs to manage more information than an augmenter. In this sense a Scenario Engineer can use some tools provided by our framework that give him:

- The capability to add listeners to different events which will take place in the user navigation context. For example a scenario could express interest in a Web page load (contentLoadedEvent), or even in the instantiation of some particular concept (cityInstantiatedEvent); this event occurs when the user has added a particular value typed as City into the Pocket.
- Knowledge about the navigation history.
- Knowledge about concepts and concepts instances stored into the Pocket.

Scenarios are not magically executed. A scenario is latent, waiting for the signal needed to be executed. For example, the *destinationInstantiated* event could trigger the scenario if it had registered a listener to be executed when instances of the *Destination* concept are created (see an example in section 5.3).

To illustrate this, in Figure 11 we show how a scenario is executed when the user opens a Wikipedia article. The Figure shows a sequence diagram for the scenario CSPOILinkAddition, a concrete scenario for the example of Figure 10. First, the scenario adds itself as the listener of the contentLoadedEvent. Then, once the content is loaded, the EventManager object loads all scenarios that are waiting for this event (in the example there is only one scenario). In the example of Figure 10, CSPOILinkAddition consults the NavigationHistoryManager to know if the previous node of the history is GoogleMaps and, as it is true, CSPOILinkAddition sends the *applyGMLinkConvert* message. The method *applyGMLinkConvert* gets all instances of the concept PointOfInterest by sending the message *getAllInstances* to the Pocket object. After that, it sends the message execute to the augmenter (LinkAdditionAdapter) with the current document (it is the DOM target), all the PointOfInterest instances and a dictionary with parameters that the augmenter needs.

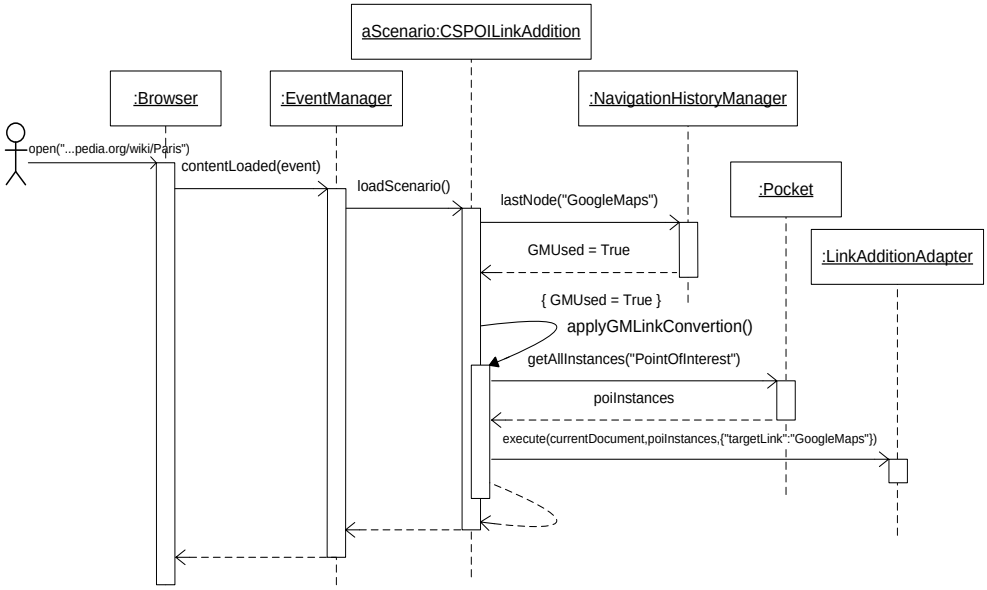

Figure 11: Sequence diagram for a scenario execution

```
AbstractAugmenter.prototype.execute =

function(document,target,args){
   this.setDataForAdaptation(document,target.value_data,args)

   if(this.isApplicableToConcept() && target.isConcept())
     for (var i = 0;i < target.getInstances().length;i++)
       this.adapt(this);

   else
     this.adapt(this);
};
```

```
function LinkAdditionAdapter(){};

LinkAdditionAdapter.prototype = new
AbstractAugmenter();

LinkAdditionAdapter.prototype.isApplicableToConcept =
function(){ return; };

LinkAdditionAdapter.prototype.applyToTextPlainInstance =
function (document, instance, args){
  var target_function = this.getFunction(args["targetLink"]);
  try{target_function(document,instance)}
  catch(e){}
};
```

Figure 12: Code of the augmenter applied

Figure 12 shows an excerpt of the augmenter code used in this scenario. The method *execute()* of the *AbstractAugmenter* is first shown. This is a template method that sends both *isApplicableToConcept* and *applyToPlainTextInstance* messages, which are defined in LinkAdditionAdapter. This augmenter has others method like getFunction that are not shown by the sake of conciseness.

5 Tool Support

The framework was implemented as a Firefox extension that provides all components shown in the pyramid of Figure 6, plus other components such as some defaults augmenters. Hereafter, we illustrate the use of augmenters and scenarios by end-users.

5.1 Data Collector

In our approach, end-users execute the adaptations. Although this can be made both explicitly (when users execute some augmenter) or implicitly (when a scenario is instantiated), some information is always needed since we aim to improve the user experience by adapting the Web pages he navigates according to the needs of his current concern. In this way users are empowered with tools to (when necessary) collect meaningful information while they visit Web sites. This information can be collected "automatically" when the user is instantiating a previously developed scenario, and the underlying tool is aware of the semantics of the pages' data, or might be collected "by hand" using tools provided by the framework (concrete DataCollectors). A DataCollector allows users to define untyped data (in order to quickly add information into the Pocket for volatile adaptations), and typed data (usually to add information for scenarios). The information collected is later available into the Pocket and it can be used to perform adaptations. Although users can extend the framework with new concrete DataCollectors, we provide three of them. One it is for collecting plain text without attaching to it a semantic meaning. A second one is for collecting plain text but giving a semantic meaning, for example for the text "*Paris*", the tag "*City*" can be specified. Finally we provide a DOM data collector which allows users to collect DOM elements that have not been abstracted in the Web page model. In Figure 15.1 the user stores different information elements, collected with the PlaintextCollector component, into the Pocket. There are two kinds of PlaintextCollector, "Put into the Pocket" and "Create instance into the Pocket" as it is shown. In this figure he collects several points of interest that he would like to visit (from the Wikipedia article) and keeps them in the pocket. Since he wants to type them as "PointOfInterest" he uses the "Create instance into the Pocket" option which opens the dialog.

As the Figure 15.1 shows, a third DataCollector is DOMDataCollector, which allows users to collect elements from the Web page's DOM.

Concrete DataCollectors shown in the menu depend on the context. When the user right-clicks on one Web site, the DataCollector analyzes if the target item can be collected by itself. In correspondence with that, the concrete DataCollector appears in the context menu or it is hidden. For example, if any text is selected the PlainTextCollector are not shown (see Figure 14).

5.2 Creating Models Over Web Pages

Parts of the information used to execute augmenters (by both end-users and scenarios) are models abstracted from DOMs. Basically a model is a set of concepts which gives a semantic meaning to a certain DOM element. A concept in a particular Web page has a related xPath to access to this element. Note that the same concept in other Web page would have another xPath. Since some adaptations like scenario can be developed in terms of concepts, the scenario could be reused in several Web applications.

In this section we introduce our tool for defining models over the DOM. Basically this is done by choosing meaningful concepts from the application's UI as shown in Figure 13.1 on the left. On the right of Figure 13.1 we show the dialog used to complete the specification of the concept, including properties and events. This process can be incremental, i.e. the model can be defined "opportunistically" to be useful for the current adaptation, and completed later when new adaptations require more concepts to be defined. Additionally, and even though the script scope (the applications

in which it "runs") is configurable from GreaseMonkey, our tool allows defining that scope, by setting a set of URLs or URL patterns as in GreaseMonkey which is useful for applications sharing similar concepts.

The result of this process is the ontology describing the application model and an XSLT file – this is the *data extraction layer* - which is used to extract the occurrences of the concepts defined in the application model (i.e., its ontology), from the concrete application. Figure 13.2 presents the Model Viewer which shows all the defined concepts with their corresponding properties and events.

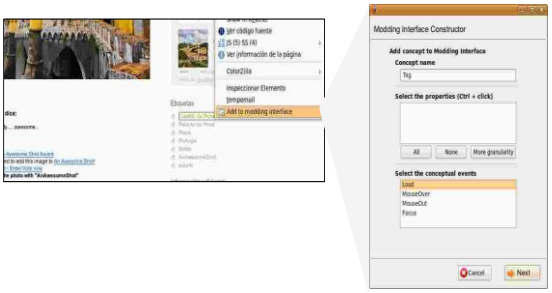

Figure 13.1 Adding concepts to the abstract model

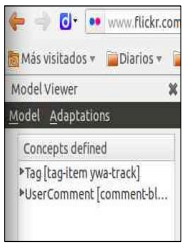

Figure 13.2 Model Viewer

The tools in Figure 13 are thought for complex model where concepts have associated properties and behaviour. Besides that, this allows users to define concepts which can have more than one instance in a particular Web page, because these kinds of concepts are defined by the *class* attribute. This means that different DOM elements with the same *class* name would be considered instances of the same concept. It is a powerful mechanism for specifying a concept which has several instances (represented by different DOM element but with the same structure) in the same Web page.

Anyway, it can happen that users would be interested in non-repetitive structures from the DOM. For example, a user could be interested just in an image, or an anchor, and even not being interested in defining a concept as part of the Web page's model. In these cases, users can collect DOM elements in a volatile way. For example, in Figure 14 a user is collecting picture's anchors from Flickr with the DOMDataCollector. Note that the Figure shows an anchor which had been collected before.

Figure 14: DOMDataCollector used to collect DOM elements in a volatile way

Finally, the anchors collected can be used later in other Web pages, for instance to share them in Facebook or just for having a navigational context for the user.

5.3 Description of Default Augmenters in the Framework

Currently, some default augmenters are provided by the framework. Some remarkable ones follow:

- Highlight: it highlights the occurrences of the data received by parameter.
- CopyIntoInput: it pastes the value received as parameter into an input form field. Once the augmenter is executed, it adds a listener to the click event which is removed after the first time in which the target is an input.
- WikiLinkConvertion: it creates links to wikipedia.com pages using as input any occurrences values received as a parameter. For example if the parameter is "Paris" then the link would be to the Wikipedia article about Paris.

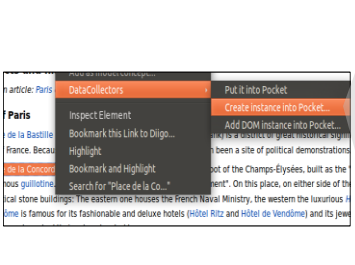

Figure 15.1: Information extraction from Wikipedia.

Figure 15.2: Resulting adaptation.

Both augmenters Highlight and WikiLinkConvertion can perform adaptations for a single value (e.g. "Paris") or for a collection of values, instances of a concept (e.g. City). The augmenter CopyIntoInput can only be executed with a single value.

As an example we show how the CopyIntoInput augmenter is used in Figure 15. Here the data collected before as instances of "PointOfInterest" is available into the Pocket (the floating box at the left in Figure 15.2) when the user opens Google Maps. In order to use one of these instances the user right clicks over the target point of interest to open the contextual menu with the augmenters available for the current site. Once he has chosen the CopyIntoInput augmenter, the AdaptationDispatcher triggers it. Since the augmenters provided as defaults are generic, they will always appear in the contextual menu. However which augmenters are available depend on the current site, because augmenters can be generic enough to be applied to any page (e.g. highlight) or specific for a single site (e.g. search the target value as a location in GoogleMaps).

As we said before, augmenters can be executed with both kind of information (text plain collected by users, or concepts abstracted from DOM elements). As an example, in Figure 16 we show how the augmenter for adding links to Flickr is applied with both kinds of data collected.

Figure 16.1 shows the abstract model for france-voyage.com, which includes the concept T*own* with the property *name*. At the left of Figure 16.2, we show the Pocket in which the user had collected the *Town* instance: "*Tamnièss*". So far, in the scope of this france-voyage.com page, there are several instances of the concept *Town*. One of them is the plain text collected "*Tamnièss*", but, some of them are instances of the concept from the corresponding abstract model in Figure 16.1.

Finally, at the right of Figure 16.2, we show the effect of applying the augmenter over all instances of *Town* (plain text instances and DOM abstraction instances). Note that the PlainTextInstance "*Tamniès*" is adapted in a determined way, while all instances of the abstract model are adapted in a different one.

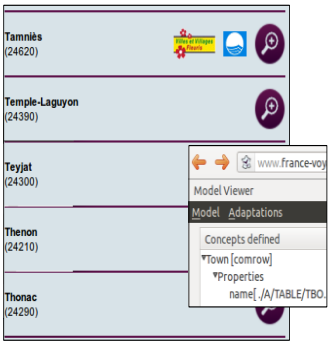

Figure 16.1 Abstract model of France-Voyage.com

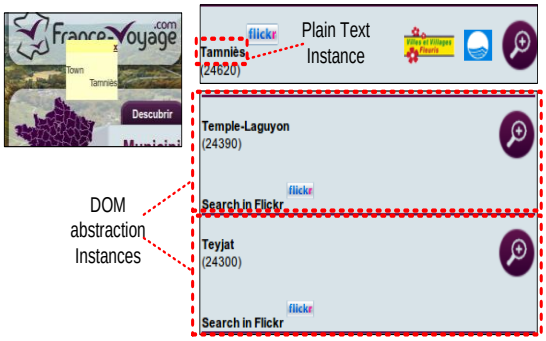

Figure 16.2: Augmenter applied over pocket data collected and concepts abstracted from DOM

5.3 Scenario Instantiation by End-Users

A scenario waits for an event to be loaded; when the event occurs and all conditions are satisfied, it is instantiated. Figure 17 shows the initial steps that a user would possibly perform to satisfy the needs previously described at section 1. In Figure 17.1, while he books (or just explore) the flights to Paris, he collects some data which will be useful in the following steps. When users collect data they can give them a conceptual meaning, by assigning a type to the selected value. In the example, the types used are *departDate*, *arriveDate* and destination. When some data is collected the corresponding event is triggered (e.g. *destinationInstantiated*). In Figure 17.2 we show how the scenario shows a popup message to offer users to use the information collected for booking hotels; this message is showed after the dates and destination were collected. Figure 17.2 shows how the form field destination is filled in with the information previously collected. This scenario is executed once the user reaches the page booking.com (either by following a link or entering a new URL). Notice that the scenario can be instantiated, because the information needed is available into the Pocket. For form filling cases, the adaptation could be automatic when the adaptation is developed particularly for an application (in this case for Venere.com) or even by using other tools like carbon [1] in order to automatically fill forms in any Web site. This use of concern information improves the user experience by allowing him to "transport" critical data among Web applications and use these data to adapt them.

Although in the example in Figure 17 the user has followed a link (which was suggested into a Pop-up Message by the adaptation) it is not the only way to use the collected information. For example, if the user enters a new URL, a new Web application will be loaded without following a link, and the information would be available too as Figure 18 shows. In this figure, we show how to extend the task in Figure 17 by adding one more subtask: renting a car in Hetz.com. It is not important if Hertz.com was opened before or after Venere.com (see Figure 17); the information is still available in any Web application without taking into account how the application was reached (by following a link, when the user enters a different URL in the browser, etc.).

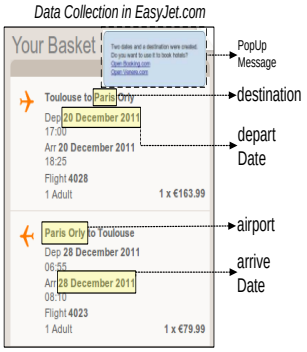

Figure 17.1: Information extraction from easyjet.com

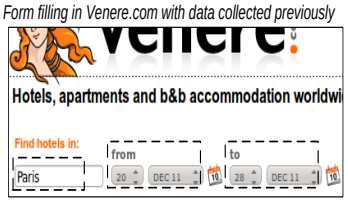

Figure 17.2: Form filling in venere.com with information collected in previous Web sites

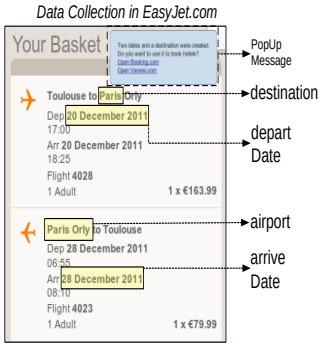

Figure 18.1: Information extraction from easyjet.com

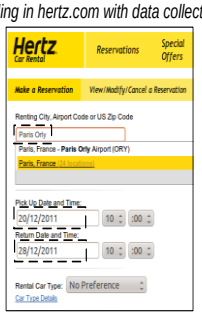

Figure 18.2: Form filling in hertz.com with information collected in previous Web sites

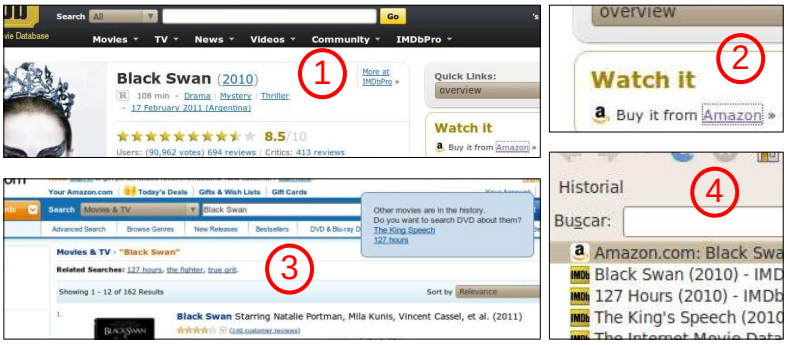

Figure 19: Use of navigational history for performing adaptations

In Figure 19 we show another adaptation to depict the potential of our approach. Here a user is using IMDB. As image 19 shows, the page of the movie “Black Swan” is loaded and from it the user navigates (with the link showed in image 2) to Amazon.com in order to buy the “Black Swan” DVD.

When Amazon.com is loaded a menu is added (image 3) to offer searches about other movies which the user had visited before in IMDB (image 4).

Scenarios have in their scope the models abstracted from Web pages' DOMs. When a Web site loads in the browser, the framework makes the abstract model of the corresponding Web site automatically available. Then, a scenario can be developed using specific models. For instance, in Figure 20 we show a simple scenario which uses the *SearchResult* concept abstracted from Web searchers. Figure 20.1 shows how the same concept, *SearchResult*, was defined for both google.com (left) and ar.yahoo.com (right). Note that the concept has a property named *target*, a string containing the *href* attribute value of an anchor. Based on the concept *SearchResult*, the scenario can be used indistinctly in both Web applications. Just as an example, in Figure 20.2 we show how the scenario highlights the search results in corresponding with the navigational history. In order to perform this adaptation the scenario carries out several previous steps. First, it queries to the navigational history in order to determinate which search results it will highlight. For example, on the top of Figure 20.2, the scenario highlights (in google.com) the *SearchResult* instances which correspond to Wikipedia pages. On the bottom, the scenario highlights (in ar.yahoo.com) the instances of *SearchResult* whose target property value match with a URL from Google Maps.

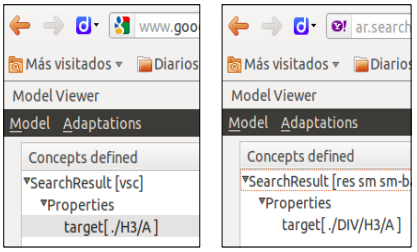

Figure 20.1: SearchResult concept for Google and Yahoo

Figure 20.2:Highlight SearchResult instances related with Wikipedia articles

In these cases, the scenario has to filter the *SearchResult* instances in order to get only those whose target property is a link to, for instance, Google Maps. Then, with the corresponding *SearchResult* instances, the scenario executes the Highlight augmenter.

6 Evaluation of the Approach

The feasibility of the tools developed around the CSA framework has been duly illustrated by the case studies presented in previous section. In order to investigate the usability of our tools we have run an empirical user testing with eleven participants. Such study encompassed the following components of the framework: *Highlight* for changing color of important information, *WikiLinkConvertion* for creating new links to Wikipedia, *DataCollector* for recording information for later usage, and *CopyIntoInput* for automating filling in forms with dates previously collected by the user. The user testing was aimed to investigated usability aspects such as effectiveness (if users are able to

accomplish their tasks using the augmenters), efficiency (if users' performance was improved using the tools) and satisfaction (how user perceive the usage of tools).

6.1 Set-up, Method and Procedure

The user testing basically consisted of observing users' performance and comments whilst users accomplish some tasks [19]. The study was run in a controlled room equipped with a PC running Window 7 and Firefox 5.1 to which the CSA framework was previously installed. None video-recording tools have been employed but user were observed by an evaluator who took notes during the whole test.

Upon their arrival, participants got a sort introduction about the goals of the study and both users and the evaluator signed a non-disclosure agreement concerning privacy issues. Participants were then asked to fill out a pre-questionnaire that was used to collect demographic and their experience with Web applications. Participants were shown a short demonstration (2-5 minutes) of the tools accordingly to the scenario described in the introduction. During this period, all participants' questions related to the CSA framework, the tools and the user testing itself were answered by the test leader. For the next step, the test leader was not allows to reply to any user's question.

The initial setup was a Web page advertising the art exhibition "De Stijl et Mondrian" (available at: http://www.parisinfo.com/sorties/1193219/de-stijl-et-mondrian, February 1st 2011). The test started by telling the user to consider the following scenario: "*Let's assume you are planning a trip to Paris for visiting an art exhibition called "De Stijl et Mondrian". For accomplish this goal, I will ask you to perform five tasks in different Web sites. You can use whatever tool you think useful. Please let me know when you are ready*". Once users felt ready, the tasks were presented to him, one by one. We asked users to describe in the beginning of the task, what they intended to do, and if possible to think aloud during task completion (i.e. thinking aloud protocol [15]). If participants naturally stopped the thinking aloud because of the speech interaction, we did not force them to continue. After users have finished every task, they were asked to assess from 1 to 5 the difficulty to perform the tasks (1 from very easy to 5 very difficult). The tasks were:

1. To collect required data for planning the trip including dates, keywords and locations;
2. To book a hotel in Paris near the exposition for the week-end of February 18th 2011;
3. To select a hotel in the neighborhood of "Les Marais";
4. To record information about the hotel;
5. To create a relationship between the actual Web of the exhibition and the Web site wikipedia.com."

Efficiency and effectiveness were respectively measured in terms of time to accomplish tasks (i.e. efficiency) and number of tasks performed successfully (i.e. effectiveness). Satisfaction was assessed by a questionnaire including: what they liked in the tools, what they didn't like, whether they experienced difficulties in activating the tools, which task were difficult to perform, whether they were aware of they actions using the tools, what kind of client-side adaptations they did on the Web pages, whether or not they want to change the actual tools, which new client-side adaptation tools s/he could envisage, which error s/he did during the test, things s/he want to change in the tools and thinks s/he

don't want to change. The study was concluded by a final interview and a System Usability Scale questionnaire (i.e. SUS, [3]). The SUS has been as a complement to user observation as it is widely used in comparative usability assessments in the industry.

6.2 Participants

For this first usability study about the augmenters, we have focused on experienced users because we assume they are more likely to formulate special needs for adapting Web pages than novices with the Web. Moreover, based on the previous experience on Web navigation that could provide qualitative information about whether or not augmenters could help them to perform their tasks. Eleven participants (6 males and 5 females, aged from 23 to 46 years old) were recruited at the department of Informatics of the University Paul Sabatier, Toulouse, France. Among the participants we count four assistant professors, three PhD students, two master students and two technicians. All participants were experienced Web users (i.e. > 5 years using the web) that browse the web as part of their daily activities (in average 4,1 hours of navigation on the web per day, SD=2,4 h). For the purposes of our work, the selected group of user was pretty homogenous so that we have tested the tool with a limited the number of participants (i.e. eleven) which is still in accordance with the practice for user testing experiments as described in [19].

6.3 Results

The eleven participants used the tools presented during the training period to perform the tasks. Users completed the test in approximately 37 minutes (SD=9 minutes) including the training session period, presentation of testing scenario to the users and the time required for filling pre- and post-questionnaires. Table 1 summarizes the main findings in terms of success rate (number or users that complete successfully a task using the augmenter) and time required for using each augmenter (average time and standard deviation), perceived difficulty (ranging from 1 to 5, lower score is the best) and overall comments. Its is noteworthy that some tasks required the combined use of augmenters, however Table 1 only reports the time spent by the users for using an augmenter in the first occurrence in the sequence of tasks. The standard deviation might indicate that some users would need more time to understand the use of augmenters and employ them successfully. This hypothesis can be related to the perceived difficulty which was given by all participants regardless the fact they succeeded or failed the task. Further studies are required to explain this result but we might expect that standard deviation would be lower once users used the augmenters more often. It is noteworthy that average time only takes into account successful uses for the augmenters.

The tool *DataCollector* was the most successful applied by all participants; it was considered the very useful and a "good substitute for post-its". However, success rate varied according the augmenter employed: *CopyIntoInput* was considered very easy to use by participants and employed successfully by 10 of them (90,9%).

The augmenter *Highlight* (72% of success rate, 8 participants) was considered easy to use but 5 users blamed it because they only can be applied to the exact word previously selected and users could not choose the color and/or the policy used to highlight different pieces of information. Whilst the use of this augmenter requires a few actions some users fail to use it successfully as they it might be

confusing to use the highlight only with data available at the pocket instead of applying it direct over the text available on the Web page.

Table 1: Summary of results concerning the user of augmenters by participants.

Augmenter	**Success rate**	**Time**	**Perceived difficulty** (1 easy ↔ 5 difficult)	**Comments**
DataCollector	100% (N=11)	~12,3 s (SD=3,7)	very easy ↔ easy ~1,6 (SD=0,7)	The use of this augmenter requires the selection of text in Web page, selection of the augmenter and time for typing a keyword for identified the data collected into the pocket. Participants used short keywords (from 5 to 11 characters) for identifying data but the number of characters was not analyzed in the time for completing the task.
Highlight	72% (N=8)	~20,4 s (SD=18,2)	easy ↔ neutral ~2,4 (SD=0,7)	The use of this augmenter requires the selection of data previously collected in the *pocket* and the selection of the augmenter from the menu.
WikiLinkConvertion	18% (N=2)	~52,5 s (SD=32,1)	neutral ↔ difficult ~3,6 (SD=1,4)	The use of this augmenter requires the selection of text in a Web page (that will become the anchor for the link) and then the selection of augmenter in the corresponding menu.
CopyIntoInput	90,9% (N=10)	~8,8 s (SD=7,6)	easy ↔ neutral ~2,4 (SD=1,2)	The use of this augmenter requires the selection of data previously collected in the *pocket* and then a click on the form field that should receive that data.

OBS.: Participants in the study=11, N=number of participants using the augmenters successfully, ~= average, SD= standard deviation.

Participants were very impressed by the augmenter allowing links to Wikipedia from concepts, i.e. *WikiLinkConvertion*; despite the fact it was considered extremely useful, the success rate with this augmenter was the lowest in the study, i.e. 18%, due to two main issues: the fact that links can only be created from typed information; lack of visual feedback (i.e. an icon) indicating where that action was possible. Only two users succeed to apply this augmenter that was also considered unnatural and difficult to use.

Nine participants (81,8%) mentioned that using the augmenters improve their performance with tasks, one user said it could be faster without the augmenters and the other one didn't see any difference. This user perception has been confirmed by the time recorded during task execution using augmenters *WikiLinkConvertion* and *CopyIntoInput*.

This study also revealed some usability problems that motivate further development in the tool. For example, users requested to have a visual indicator allowing them to distinguish where augmenters have been applied (ex. links on the web site x links created with the augmenter *WikiLinkConvertion*). Users intuitively tried to activate some of the augmenters using Drag & Drop which is an indicator for further research of more natural interaction with augmenters. The most frequent suggestions for new

augmenters include “automatic filling forms”, “create links to other web sites than Wikipedia”, and “automatic highlight at the web page of information previously collected”.

The results show that, generally, participants appreciate the concept of CSA and the tool support. In the pre-questionnaire, when asked if they would like to modify the web pages they visit, 2 of 11 participants said no because “it could be very time consuming”. Nonetheless, all participants said that our tools for client-side adaptation are useful and that they are willing to use it in the future. Adaption across different web site was described as “natural” by 7 participants and a “real need” by 5 of them. This positive analysis is confirmed by a SUS score of 84,9 points (SD = 5,5), which is a good indicator of general usability of the system.

7 Concluding Remarks and Further work

In this work we have presented a novel approach for Client-Side adaptation which takes into account the tasks that users perform while navigating the Web. We aim to support complex concern-sensitive adaptations in the Client-Side in order to improve the users’ experience. We have developed a support framework which can be extended with two kinds of adaptations: atomic augmenters (realizing simple adaptation actions) and scenarios which comprise the use of different augmenters on somewhat predefined Web pages. These adaptations can be executed either manually, e.g. when the user triggers an adaptation action explicitly, or automatically when some scenario is instantiated. Being built on solid engineering principles, the framework can be extended and/or used both by end-users or developers (e.g. by developing JavaScript code). Our approach is focused in concern-sensitive integration the information, in comparison with the usual Client-Side adaptation approaches; we provide a flexible mechanism to integrate information while users navigate the web, instead of “just” providing tools to statically adapt Web sites. Our approach is based in two main types of developers interventions: the first one (augmenters) supports generic scripts with specific adaptation goals to be applied over any Web page, and the second one (scenarios) can be used when the goal is to support users tasks among several Web sites. We have performed a small but meaningful evaluation with end-users with excellent results.

We are working in several directions to improve the approach. As it was explained in Section 5, we have detected usability problems in some of our tools when users are trying to adapt the Web sites or even while they are collecting data. In this sense, we are developing not only new tools but also tuning the existing ones and performing new evaluations with them.

So far, although we propose an approach for developing scenarios, augmenters and components in a crowdsourced way, we do not provide a repository available for sharing artefacts developed by users. It is part of our future work to provide a repository where users can publish, share and download augmenters, scenarios, components, and abstract models developed users.

On of the most important things is to improve both the development process and tools for developers using the framework. Although we have defined guidelines for both augmenters and scenarios development, these must still be written in a quite similar way to bare JavaScript programming. The code showed in Figure 12 can be developed only by users with JavaScript skills. Our goal is to raise the abstraction level for developers by creating a domain specific language that will simplify the specification of both augmenters and scenarios; this will let users without JavaScript

knowledge to develop adaptations easily. Currently we have developed an internal DSL for providing developers with the guidelines for creating scenarios. We aim to obtain an external DSL, particular a visual language to model scenarios behaviour and then generate the corresponding JavaScript code.

References

1. Araújo, S., Gao, Q., Leonardi, E., Houben, G. Carbon: Domain-Independent Automatic Web Form Filling. In: Proceedings of ICWE2010, (Vienna, 2010), Springer, 292-306.
2. Bouvin, N. O. Unifying Strategies for Web Augmentation. In: Proc. of the 10th ACM Conference on Hypertext and Hypermedia, 1999.
3. Brooke, J. (1996) "SUS: a 'quick and dirty' usability scale". In: Usability Evaluation in Industry. London: Taylor and Francis.
4. Brusilovsky, P. Adaptive Navigation Support. In: The Adaptive Web: Methods and Strategies of Web Personalization, Springer, 2007, 263-290.
5. Brusilovsky, P, Kobsa, A., Nejdl, W. The Adaptive Web: Methods and Strategies of Web Personalization, Springer, 2007.
6. Cooley, R., Tan, P., Srivastava, J. Discovery of Interesting Usage Patterns from Web Data. In: Proc. of WEBKDD'99 (San Diego, 1999), 163-182.
7. Daniel, F., Casati, F., Soi, S., Fox, J., Zancarli, D., Shan, M. Hosted Universal Integration on the Web: The mashArt Platform. In: Proc. of ICSOC/ServiceWave (Stockholm, 2009), 647-648.
8. Diaz, O., Arellano, C., Iturrioz, J. Layman tuning of websites: facing change resilience. In.: Proceedings of WWW2008 Conference, (Beijing, 2008), 127-1128.
9. Diaz, O., Arellano, C., Iturrioz, J. Interfaces for Scripting: Making Greasemonkey Scripts Resilient to Website Upgrades. In Proceeding of ICWE2010 (Vienna, 2010), Springer, 233-247.
10. Firmenich, S., Rossi, G., Urbieta, M., Gordillo, S., Challiol, C., Nanard, J., Nanard, M., Araujo, J. Engineering Concern-Sensitive Navigation Structures. Concepts, tools and examples. JWE 2010, 157-185.
11. Firmenich, S., Winckler, M., Gordillo. A Framework for Concern-Sensitive, Client-Side Adaptation. In: Proc. of ICWE (Paphos, 2011), Springer, 198-213.
12. Greasemonkey, At: http://www.greasespot.net/ (last visit on Oct. 25, 2011).
13. Han, H., Tokuda, T. A Method for Integration of Web Applications Based on Information Extraction. In: Proc. of ICWE (New York, 2008), Springer, 189-195.
14. Leshed, G., Haber, E., Matthews, T., Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. In: Proc. of the Twenty-Sixth annual SIGCHI conference on Human factors in computing systems (Italy, 2008), 1719-1728.
15. Lewis, C. H. Using the "Thinking Aloud" Method In Cognitive Interface Design. IBM Research Report, RC-9265, 1982.
16. Meusel, M., Czarnecki, K., Köpf, W. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In: Proceedings of ECOOP'97, 1997, 496-510.
17. Miller, C. S., and Remington, R. W. Modeling an Opportunistic Strategy for Information Navigation. In: Proc. Of 23th Conference of the Cognitive Science Society, 2001, 639-644.
18. MozillaUbiquity, At: http://mozillalabs.com/ubiquity/ (last visit on Oct. 25, 2011).
19. Rubin, J. 1994. Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests. Theresa Hudson (Ed.). John Wiley & Sons, Inc., New York, NY, USA.
20. Yu, J., Benatallah, B., Casati, F., and Daniel, F. Understanding Mashup Development. IEEE Internet Computing, 12:44–52, 2008.
21. 21. Wong, J. and Hong, J. I. Making Mashups with Marmite: Towards End-User Programming for the Web. ACM, City, 2007.