A Framework for Concern-Sensitive, Client-Side Adaptation

Sergio Firmenich[1, 2], Marco Winckler[3], Gustavo Rossi[1,2]
[1]LIFIA, Facultad de Informática,
[2]Universidad Nacional de La Plata and Conicet Argentina
{sergio.firmenich, gordillo, gustavo}@lifia.info.unlp.edu.ar
[3]IRIT, Université Paul Sabatier, France
winckler@irit.fr

Abstract. Currently the Web is a platform for performing complex tasks which involve dealing with different Web applications. However, users still have to face these tasks in a handcrafted way. While building "opportunistic" service-based software, such as mashups, can be a solution for combining data and information from different providers, many times this approach might have limitations. In this paper we present a novel approach which combines concern-sensitive application adaptation with user-collected data to improve the user experience while performing a task. We have developed some simple though powerful tools for applying this approach to some typical tasks such as trip planning. We illustrate the paper with simple though realistic examples and compare our work with others in the same field.

1 Introduction

As wisely pointed out in [6], 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. More recently different technologies provide users with tools allowing them to change the way Web content was presented. For example, using visual Mashups [5, 14], users can compose content hosted by diverse Web sites and they can run Greasemonkey scripts [9] 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 provided limited support to tasks that require navigation on multiple Web sites. 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 example, 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 [12]. 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-weight 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 he needs further explanation about a topic.

In a previous work [8], 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 work. The framework is fully described in section 3. Section 4 presents tools built upon the framework. Section 5 presents how we have validated our approach with end-users. Finally, section 6 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 [4]. 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 [5, 14] simplify the development of these applications. Since most Web applications do not provide Web services to access their functionality or information, [10] 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 [9], 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 [6] 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 [7] 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.

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 [8] 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 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 concrete information can be used to perform the adaptation, therefore improving the support for his task and providing support for building more complex adaptations.

3 A framework for concern-sensitive augmentation

For the sake of comprehension we first introduce some basic concepts and background work; next we make an overview of the approach and of our tool support.

3.1 Background for the framework

Our framework is based on the concept of concern-sensitive navigation (CSN). We say that a Web application (or specifically a Web page) is concern-sensitive (CS) when its contents, operations and outgoing navigation links can change (or adapt) to follow the actual situation (concern) in which it is accessed [8]. Concern-sensitive navigation is different from context-aware navigation, where other contextual parameters (location, time, preferences) are considered. Figure 1 illustrates the differences between flat and concern-sensitive navigation. 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 enriche their contents by taking into account what was the user concern in the previous page.

In [8] 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.

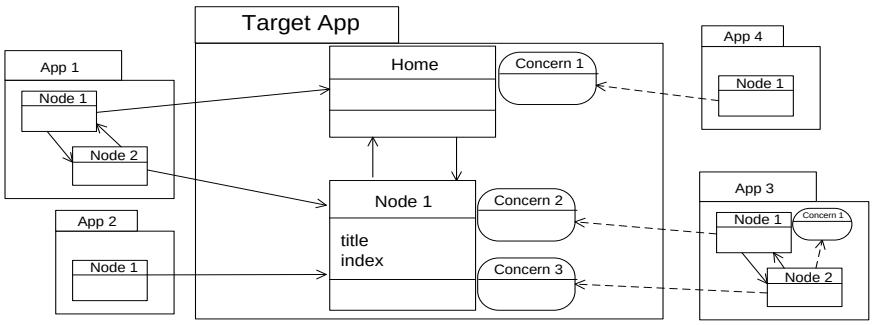

Fig. 1. Flat Navigation vs. Concern-Sensitive Navigation

Figure 2 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 2, augmented with the corresponding map and a set of links to those Wikipedia articles in the surroundings of the current one.

Fig. 2. 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.

3.2 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 3.3.1). By combining augmenters, the framework also supports *scenario engineering* for developing customized adaptations for specific domains such as trip planning (See section 3.3.2). 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 use 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*.

The framework is described at Figure 3 using the pyramid approach [11]. 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 *AbstractAdapter* 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]).

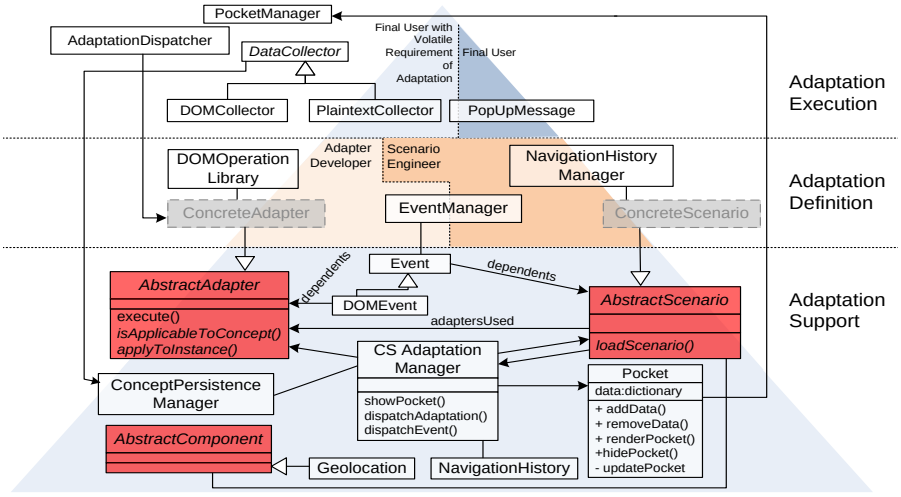

Fig 3. Framework structure

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.
  - 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.
  - AbstractAdapter and AbstractScenario: are abstract classes from which concrete augmenters and scenarios, correspondingly, developed by users must inherit.
  - AbstractCommponent: 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.
  - ConcreteAdapter and ConcreteScenarios: are scripts developed by users with programming skills. These classes are shown in Figure 3 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.

3.3 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 2, 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 4.1 details how users collect information during navigation.

3.3.1 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 4 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 right in the Figure). As Figure 4 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.

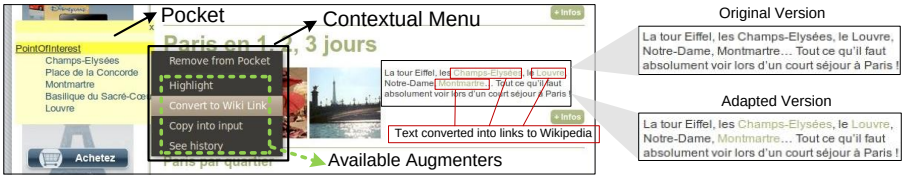

Fig 4. Plain text converted into links to add personal navigation

Since augmenters can be applied to different Web pages they must be developed without a dependence of a particular DOM, as described in [6]. Moreover, when using the framework, developers must:

- Construct an augmenter as a JavaScript object inheriting from *AbstractAdapter*, the hot-spot shown in Figure 3.
- Implement the methods defined as abstract in *AbstractAdapter*. This is necessary because the *execute()* method of *AbstractAdapter* (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.

Augmenters are executed when a user explicit triggers them or when a scenario is instantiated (see section 3.3.2). In Figure 5.a, 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* and *applyToInstance* messages. All augmenters developed by users must have these methods defined as in the augmenters showed in Figure 5.b (*Highligth* and *WikiLinkConvertion*).

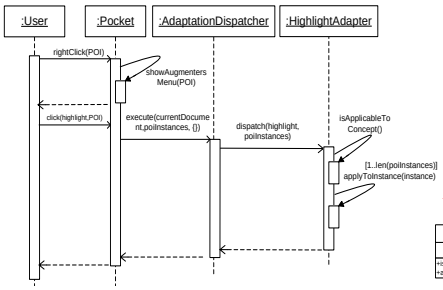

Fig 5.a. sequence diagram describing user triggering an augmenter.

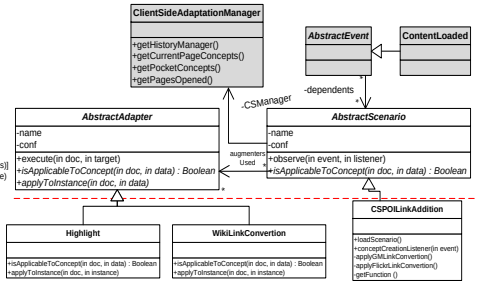

Fig 5.b. class diagram presenting the framework extensions with augmenters and scenarios.

3.3.2 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 6 shows. In Figure 6.a 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 6.a, 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 6.b shows a similar case, but now since GoogleMaps is the previously visited Web page, a link to GoogleMaps is added.

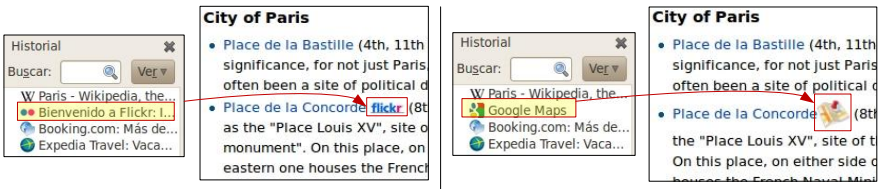

Fig 6.a. Navigation with Flickr concern. **Fig 6.b.** Navigation with GoogleMaps concern.

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 3.
- 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 4.3). To illustrate this, in Figure 7 we show how a scenario is executed when the user opens a Wikipedia article. At the left of this Figure, we show a sequence diagram for the scenario CSPOILinkAddition, a concrete scenario for the example of Figure 6. 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 7, 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.

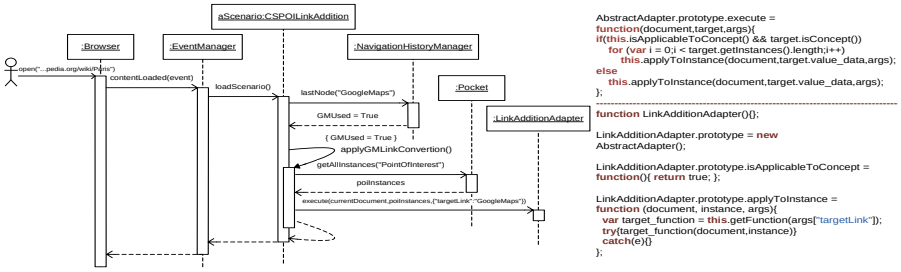

Fig. 7. Sequence diagram for a scenario execution and code of the augmenter applied

The right side of Figure 7 shows an excerpt of the augmenter code used in this scenario. The method *execute()* of the *AbstractAdapter* is first shown. This is a template method that sends both *isApplicableToConcept* and *applyToInstance* messages, which are defined in LinkAdditionAdapter. This augmenter has others method like *getFunction* that are not shown by the sake of conciseness.

4 Tool support

The framework was implemented as a Firefox extension that provides all components shown in the pyramid of Figure 3, plus other components such as some defaults augmenters. Hereafter, we illustrate the use of augmenters and scenarios by end-users.

4.1 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. In Figure 8.a the user stores different information elements, collected with the *PlaintextCollector* component, into the *Pocket*. *PlaintextCollector* has two options, "Put into the Pocket" and "Put into the Pocket as volatile data" 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 "Put into Pocket" option which opens the dialog.

4.2 Description of default augmenters in the framework

Currently, some augmenters are provided by default with 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.

The 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.

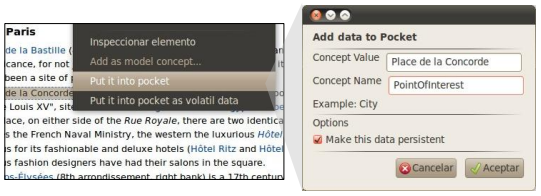

Fig 8.a. Information extraction from Wikipedia.

Fig 8.b. Resulting adaptation.

As an example we show how the CopyIntoInput augmenter is used in Figure 8.b. Here the data collected before as instances of "*PointOfInterest*" is available into the *Pocket* (the floating box at the left in Figure 8.b) 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).

4.3 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 9 shows the initial steps that a user would

possibly perform to satisfy the needs previously described at section 1. In Figure 9.a, 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 9.a 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 9.b 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 Booking.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.

Data Collection in Expedia.com

Fig 9.a. Information extraction from *expedia.com*

Form filling in Booking.comwith data collected previously

Fig 9.b. Form filling in *booking.com* with information collected in previous Web sites

Fig 10. Information in pocket used into a GoogleMaps scenario

Fig 10. Information in pocket used into a Flickr scenario

For example in Figure 10.a we show how, when the user arrives to Google Maps, the information in the pocket can be used automatically to create Google Maps links in the left bar. On the other hand, the same information can be used in another scenario if the user opens Flickr.com as shown in Figure 10.b where the points of

interest are offered as Flickr's tags. In this adaptation, the scenario engineer has used a framework tool (the floating box is a PopUpMessage) for a message suggesting a simple adaptation.

5 Evaluation of the approach

To validate our approach and actual usage of the tools, we have conducted a usability study with end-users. The goal of this evaluation was to investigate if client-side adaptation is usable for solving common tasks whilst navigating the web. The adaptations investigated in this study explored the following framework components: *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 study was run with 11 participants (6 males and 5 females, aged from 23 to 46 years old). All participants were experienced Web users (i.e. > 5 years using the Web) that spend a significant time browsing the Web as part of their daily activities (in average 4,1 hours of navigation on the Web per day, SD=2,4 h). We have focused on experienced users because we assume that they are more likely to formulate special needs for adapting Web pages than novices with the Web. Participants were asked to fill out a pre-questionnaire; following they were introduced to the system and asked to conduct five tasks at their workplace, followed by a final interview and a System Usability Scale questionnaire (i.e. SUS, [3]). The SUS has been used as a complement to user observation, as it is widely used in comparative usability assessments in industry. The five tasks were related to investigate the working hypothesis, on how usable our approach is for solving common tasks whilst navigating Web sites. All tasks were related to the following problem: the goal is to plan a trip to Paris to visit an exposition, which includes collecting information such as dates and location of the exposition and booking a hotel; for that purpose users should visit different Web sites and use our tools to perform client-side adaption on the page visited. In average, users spent 37 minutes to complete the test. Usability was measure in terms of time to accomplish tasks, number of tasks performed successfully, and user satisfaction (via a questionnaire).

The results show that, generally, participants appreciate the concept of client-side adaptation 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". Notwithstanding, all participants said that our tools for client-side adaptation are useful and that they are willing to use them in the future. Adaption across different Web site was described as "natural" by 7 participants and a "real need" by 5 of them. The component *DataCollector* was the most successfully applied by all participants; it was considered very useful and a "good substitute for post-its". However, success rate varied according to 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 could only apply it to the exact word previously selected, and users cannot choose the color

and/or the police used to highlight different pieces of information. Participants were very impressed by the augmenter allowing links to Wikipedia from concepts (the *WikiLinkConvertion*); despite the fact that it was considered extremely useful, the success rate with this augmenter was the lower in the study, 18%, due to two main issues: the fact that links can only be created from typed information and lack of visual feedback (i.e. an icon) indicating where that action was possible. Nine participants (81,8%) said that using the augmenters improved 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". 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.

6 Conclusions and future 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). In comparison with the usual client-side adaptations, 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. The first one is to improve 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. 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.

Besides that, and as indicated 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.

References

1. Araújo, S., Gao, Q., Leonardi, E., Houben, G. Carbon: Domain-Independent Automatic Web Form Filling. In: Proc. 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. Daniel, F., Casati, F., Soi, S., Fox, J., Zancarli, D., Shan, M. Hosted Universal Integration on the Web: The mashArt Platform. In Proceedings of ICSOC/ServiceWave (Stockholm, 2009), 647-648.
6. Diaz, O., Arellano, C., Iturrioz, J. Layman tuning of websites: facing change resilience. In.: Proc. of WWW2008 Conference, (Beijing, 2008), 127-1128.
7. 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.
8. 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.
9. Greasemonkey, At: http://www.greasespot.net/ (last visit on Feb. 11, 2011)
10. Han, H., Tokuda, T. A Method for Integration of Web Applications Based on Information Extraction. In Proceeding of ICWE (New York, 2008), Springer, 189-195.
11. 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, 496-510, 1997.
12. 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, pp. 639-644.
13. Yu, J., Benatallah, B., Casati, F., and Daniel, F. Understanding Mashup Development. IEEE Internet Computing, 12:44–52, 2008.
14. Wong, J. and Hong, J. I. Making Mashups wit Marmite: Towards End-User Programming for the Web. ACM, City, 2007.