Behaviour & Information Technology
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713736316

The role of HCI models in service front-end development

Fabio Paterno[a]; Carmen Santoro[a]; Lucio Davide Spano[a]
[a] CNR-ISTI, HIIS Laboratory, Pisa, Italy

Accepted uncorrected manuscript posted online: 04 March 2011

First published on: 27 April 2011

To cite this Article Paterno, Fabio , Santoro, Carmen and Spano, Lucio Davide(2011) 'The role of HCI models in service front-end development', Behaviour & Information Technology,, First published on: 27 April 2011 (iFirst)
To link to this Article: DOI: 10.1080/0144929X.2011.563795
URL: http://dx.doi.org/10.1080/0144929X.2011.563795

Behaviour & Information Technology
2011, 1–14, i*First* article

Taylor & Francis
Taylor & Francis Group

The role of HCI models in service front-end development

Fabio Paterno*, Carmen Santoro and Lucio Davide Spano

CNR-ISTI, HIIS Laboratory, Pisa, Italy

(*Received 27 February 2010; final version received 9 February 2011*)

This article discusses how human–computer interaction (HCI) models can support the development of interactive applications based on Web services. It also introduces a specific method exploiting such models for this purpose and the associated tool support. An example application of the method for an educational scenario is presented. The results of an early test of the development environment are reported as well. Lastly, some conclusions are drawn along with indications for future work.

Keywords: HCI models; Web services, service front-ends

1. Introduction

Nowadays, Web services are increasingly used to support remote and distributed access to application functionalities, which especially in business enterprises have already successfully produced results in sharing and leveraging the existing code between different business units. However, this trend has also emphasised the availability of a large gamut of services for which user interactive access has not been foreseen in advance, since Web services are, for the most part, provided for supporting computer-to-computer communications. In addition, in the event that the user support (the service front-end) is integrated with such services afterwards, very often it turns out to be based on ad hoc solutions that lack generality and cannot exploit the potentialities offered by such services.

Therefore, on the one hand, there is the need to include the user perspective in the design of interactive applications exploiting Web services, so as to enable users to reuse and compose such functionalities in building their own interactive applications. On the other hand, there is also the need to provide general approaches and methods to be used across different domains in order to enable users to exploit the possibilities offered by services in a large variety of contexts of use.

The use of logical descriptions of UIs has stimulated a great deal of interest in supporting the design and development of multi-device user interfaces (Myers *et al.* 2000). Indeed, one of their main advantages is that they allow developers to avoid dealing with a plethora of low-level details associated with the corresponding implementation languages. Now, the questions are whether and how such approaches can be improved (and possibly extended) when the application functionalities are provided by Web services, and what should be adapted in the overall design process in order to support such service-based functionalities and their composition.

In this article, we describe a method that we have identified for supporting model-based development of interactive applications based on Web services. The method exploits logical user interface descriptions, and it is also accompanied by an automatic tool, which should provide support in all the phases. The proposed approach exploits different user interface (UI) abstraction levels (task, abstract interface and concrete interface), which can also exploit additional UI information that is associated with Web services (so-called annotations).

In the article, after discussing related work, we provide background information on human–computer interaction (HCI) relevant models and their relationships with services. Next, we describe the approach that we have identified for supporting the development of service front-end in multi-device contexts, also providing information about Model-based Language for Interactive Applications Environment (MARIAE), the tool that has been developed to support such approach. Then, we discuss an example to show the application of the proposed concepts and better illustrate the approach, and report on a first user test. Lastly, some conclusions and indications for future work are provided.

2. Related work

Currently, the use of Web services is a mainstream trend in developing distributed applications. For

*Corresponding author. Email: fabio.paterno@isti.cnr.it

ISSN 0144-929X print/ISSN 1362-3001 online

DOI: 10.1080/0144929X.2011.563795
http://www.informaworld.com

enterprises, Web services have already provided several concrete advantages in terms of reusing the code and increasing productivity. In the research community, this widespread diffusion has represented a stimulus for investigating research opportunities in this area. An interesting perspective regards the use of Web services in the phase of requirement specification. In a study by Sawyer and Maiden (2009), the authors present an approach for exploiting Web services to improve the specification of requirements, together with the associated tool, which have been developed within the European SeCSE (http://www.secse-project.eu/) project. In their analysis, they claim that the use of Web services in the requirement process helps in discovering more effective requirements than those discovered with traditional approaches (like use cases). Indeed, the exploitation of Web services helps not only to refine the current set of requirements but also to discover new requirements that might derive from shared concepts across different domains. In addition, non-functional requirements have also been addressed in this work. Indeed, when specifying the properties depending on which the search of Web services will be carried out, it is also possible to specify the properties referring to quality of service, which describe service's non-functional properties which the user can be interested in and which can also represent a filter mechanism on the set of services that could be considered for the concerned application.

The HCI research area and especially the model-based approaches for UI design seem promising in this respect, since they are focused on complementary aspects (e.g. the human perspective, which is missing in Web services) and exploit the use of models that can find a natural counterpart in the descriptions that are associated with Web services. However, to date, the research issue of how to exploit Web services in the UI design approach, especially with model-based design, has not found final general solutions. In Vermeulen *et al.*'s study (2007), there is an attempt to extend the descriptions of services with user interface information: the Web Service Definition Language (WSDL) description is converted to Web Ontology Language for Services (OWL-S) format, which is then combined with two models typically found in HCI, a task model and a layout model. In our approach, the WSDL descriptions are analysed and exploited in the process of building corresponding UIs through the use of logical UI descriptions.

Model-driven design and deployment of service-enabled Web applications using WebML has been proposed as well (see, for example, Manolescu *et al.* 2005). Our work has a different focus since we propose an environment based on HCI models for generating usable service front-ends, which can be implemented in a variety of implementation environments and not only for the Web. The use of logical user interface languages to support service composition and user interface generation is explored in Dery-Pinna *et al.*'s study (2008). Our solution is able to provide a more systematic solution to such issues because it exploits task models that provide an integrated view of interaction and functional aspects.

The possibility of service functionalities of being combined together has also pushed much interest into analysing techniques for composing together portions, elements or both of UIs that are connected with such services (e.g. in graphical UIs, the UI widgets that allow the user to activate and control the functionalities associated with such services). To this respect, since concrete logical descriptions of UIs are often specified through eXtensible Markup Language (XML) user interface languages (Luyten *et al.* 2004), proposals to use XML tree algebra-based techniques for composing together portions of presentations have been put forward.

For instance in Lepreux *et al.*'s study (2006), fusion operation (composition of interactors with repetition of the intersection) and union operation (when the composition will not repeat the intersection) have been introduced for combining interactors, and the XML tree algebra presented in El Bekai and Rossiter's study (2005) is applied to composition or decomposition of graphical UIs specified in USer Interface eXtensible Markup Language (UsiXML) (Limbourg *et al.* 2004). In this method, the possible temporal relationships occurring between the interactions are not considered. To this regard, task models can provide useful support, as it will be shown in the next section.

Another approach that has considered the issue of user interfaces exploiting Web services is Dynvoker (Spillner *et al.* 2008). It is an environment that dynamically builds user interfaces for accessing Web services, by exploiting the service definition and also a GUI Deployment Descriptor (GUIDD) file, which contains information about how the internal operation can be made visible to the user. The result of the generation process is an Hyper Text Markup Language (HTML) form, which allows the user to invoke the service. However, there are two drawbacks in this approach: first, since the generation is made at runtime and taking into account a single service selected by the user, Dynvoker cannot support the generation of UI for composition of services. Also, the resulting form for activating the service cannot be customised by the designer.

The issues connected with reusing of shared UI structures in multi-device applications are considered in the Damask's approach (Lin and Landay 2008), which provides support for multi-device user interfaces

using patterns (providing information on what should be available in all platforms) and layers (information is provided regarding what is specific to a given platform). However, it does not provide any specific support for applications based on Web services.

3. HCI models and services

Model-based approaches are a well-known area in HCI. With such methods, the basic idea is that a UI can be described in terms of a number of relevant concepts; in this way, the designers can focus on relevant and logical aspects of a UI that allow them to reason and decide about the design rationale, without the need of handling all the low-level details of a UI from the beginning of the design.

Model-based approaches for UI design have evolved over time, depending on the issues raised by the current technological landscape. In the past they addressed basically issues connected with graphical UIs, evolving later on with introducing abstractions for UIs addressing other modalities apart from the graphical one. The most recent evolutions have covered the problems raised by the introduction of multiple modalities in the UI design, and also, the variation of UIs, depending on the specific device in use (multi-device applications). Nowadays, the issues are in the access to applications encapsulated into pre-existing web services, which should be possibly exploited regardless of the type of device in use.

This problem raises some interesting issues. Indeed, in order to be able to exploit the large availability of services, we cannot suppose anymore to develop ad hoc functionalities for the specific application to be developed. On the contrary, designers of interactive applications exploiting Web services should consider to build their applications by composing together even pre-existing pieces of software that are created by others. This affects the traditional well known top-down approach in which the designer builds the application from scratch by progressively refining through the various abstraction levels of the overall model of the application. With Web services, even if the designers want to create an interactive application from scratch, they are not completely free but already constrained at including the functionality already available, according to the service descriptions that come with the Web services. This generally implies composing together third-party functionalities in a more general and structured design, and therefore, designers should also be able to follow and include bottom-up steps when needed.

Another aspect that has to be taken into account as a further attempt to fill the lack of user perspective in handling Web services is the fact that *annotations* could be associated to Web services. Such annotations can also provide information about how the services will be finally rendered (e.g. they can specify labels or the format of fields). They can be even incomplete and provided at different levels of abstractions. In the remainder of this article, we will show how the annotations have been supported within our method and the supporting automatic environment.

However, before describing the approach presented in this article, we judge relevant to provide further background details on HCI models that could be exploited to create service front-ends in multi-device contexts.

In particular, the research community in model-based design of user interfaces has identified some logical levels for describing UIs (see Szekely 1996, Paternò 1999, Calvary *et al*. 2003). The logical levels include the following:

- The *task and object level*, which reflects the user view of the interactive system in terms of logical activities and objects manipulated to accomplish them.
- The *abstract user interface*, which provides a modality-independent description of the user interface.
- The *concrete user interface*, which provides a modality-dependent but implementation-language-independent description of the user interface.
- The *final implementation* in an implementation language for user interfaces.

In the next sub-sections, we will provide further details on each of such levels.

3.1. Task level

Task models are supposed to describe the various activities that are supported by an interactive application, together with their temporal relationships. One example of notation for task models is the ConcurTaskTrees (CTT) notation (Paternò 1999), widely used also for the availability of public tools supporting editing and analysis of such models. CTT allows designers to specify more flexible behaviour than traditional approaches, such as hierarchical task analysis (HTA) or goals, operators, method and selection rules (GOMS), because it includes a rich set of temporal operators among tasks. In this notation, tasks can be *elementary* tasks (basic logical entities that cannot be further decomposed) and *structured* tasks (tasks that can be decomposed into smaller sub-tasks). They can be of different categories depending on the allocation of their performance (e.g. system tasks, user tasks, interactive tasks and abstract tasks). Moreover,

tasks are hierarchically decomposed and are linked together by means of temporal operators, which specify the relationships occurring between them.

Such notation offers naturally some interesting integration opportunities for exploiting Web services in a model-based design approach. Indeed, it is a graphical hierarchical notation, and therefore, the refinement opportunities offered by the hierarchical task decomposition allows the user to select the most suitable granularity to be considered (for instance, low-level elementary application tasks could be easily associated with Web services' operations). In addition, with this notation, it is also possible to specify task patterns, which are task structures that are reusable across various applications. Thus, whenever designers find a problem that is similar to one that has been already solved, then they can reuse the solution previously developed. Therefore, by using this specification, it is possible to reuse portions of models at any level of granularity.

3.2. Abstract level

Another level of abstraction for describing UIs is represented by the abstract level. MARIA is an XML language supporting the abstraction layers indicated by a CAMELEON Reference Framework (Calvary *et al*. 2003), commonly referred in the HCI community.

At the abstract level, a UI is described in terms of abstract interactors, with no reference to a specific platform or modality. For instance, examples of abstract interactors are selection interactor (the one enabling a selection activity), edit interactor (the interactor enabling an editing task), etc. Therefore, describing a UI at an abstract level means to specify it in general terms, mainly referring to the semantical goal that various interactors will support. Each abstract UI is generally composed of a number of abstract presentations, which in turn consist of a number of interactors that can be composed together through some abstract operators (e.g. one operator enables the grouping of several interactors in one composed expression). The navigation among different abstract presentations is performed through a set of abstract connections, which specify the dynamic behaviour of each abstract presentation (how it is possible to move from one abstract presentation to another).

3.3. Concrete level

The concrete level provides further details on the specification carried out at the abstract level, depending on the specific platform at hand. Examples of platforms are the graphical desktop, the graphical mobile, the vocal, etc. So, all the devices that share common interaction capabilities (e.g. the screen size and the interaction modality) will be considered as belonging to the same cluster of devices. Therefore, the description of a UI at a concrete level inherits all the semantic information specified at the abstract level, and also adds further details about how the supposed activity is expected to be performed when considering a specific platform. This implies that a generic abstract interactor allowing a single choice can be mapped into several widgets when considering the graphical UI platform: for instance, this is the case of a radio button, a list and a pull-down menu.

3.4. Implementation level

The implementation level is the final level in which the UI is described providing a complete specification of all its parts by referring to a specific technology and implementation language. Therefore, this is a further refinement added with respect to the specification that the concrete level provides. Here, the same concrete UI can be mapped into different final UIs, if two different implementation languages are used (e.g. Java, XHTML, C#, etc.).

4. The method

As it has been previously pointed out, one of the impacts of including Web services in the design of multi-device interactive applications is the inability of following anymore a traditional, completely top-down approach going through the various abstraction UI layers. Since the task model of the interactive application to be developed is supposed to describe the various activities that should be supported, including the functionalities provided by the system, in case such functionalities are offered by services, there should be the possibility to connect such services with their correspondent counterparts in the task model. In more concrete terms, this means to perform an association between some of the elementary tasks existing in the task model and the operations specified in the Web services. As aforementioned, in our approach, we use the CTT notation for representing task models. This notation is a graphical notation that includes different icons for showing the allocation of tasks (system tasks are represented by a computer icon).

This association step has also the side effect of automatically determining how the Web services should be composed together (e.g. two Web services that are associated with two sibling application tasks will inherit the temporal relationships specified in the task model) while performing the first bottom-up step of our approach.

However, in order to make such association effective, it is important to use a level of granularity

suitable for expressing the details of the functionalities described in the Web services within the task model. Then, beyond associating system tasks to Web services, it is important to further decompose such system tasks into system sub-tasks, in which each sub-task will be associated with an operation defined in the Web service.

Once the task model and the Web services have been connected together, it is possible to perform the next top-down step, i.e to generate a first draft of an abstract UI description, which can be further edited by the designers. When a satisfactory customisation is reached, the designers can refine such abstract description into a concrete, platform-dependent one.

It is worth pointing out that different customisations can be performed and intervene at various abstraction levels, and therefore, at different steps of our approach. Indeed, the designer not only can freely edit the properties of the various UI elements at different levels of abstractions but also can freely include in the design some annotations that are associated with the Web services and which provide (even partial) hints about the possible user interfaces that can be used in the various levels of logical descriptions. Indeed, at the abstract user interface level, the annotations can specify groupings' definition, input validation rules, mandatory or optional elements, data relations (conversions, units and enumerations) and languages. At the concrete user interface level, the annotations can provide labels for input fields, content for help, error, warning messages and indications for appearance rules (formats, design templates, etc.).

The methodology proposed can be then summarised in the following steps:

(1) Creation of the CTT model, using the existing methodologies and tools.
(2) Association between the Web service operations and the corresponding basic system tasks.
(3) Generating the abstract description from the task model with a top-down transformation step and applying the semantic information of the CTT model.
(4) Generating a concrete description from the abstract description, possibly by editing it until a satisfactory result is obtained.
(5) Generating an implementation UI from the concrete UI description, also taking into account possible annotations.

The method has been summarised in Figure 1.

As you can derive from the right part of Figure 1, the method is based on two device independent languages (the task model language and the abstract user interface language), both expressing the involved concepts in a manner that is device-independent,

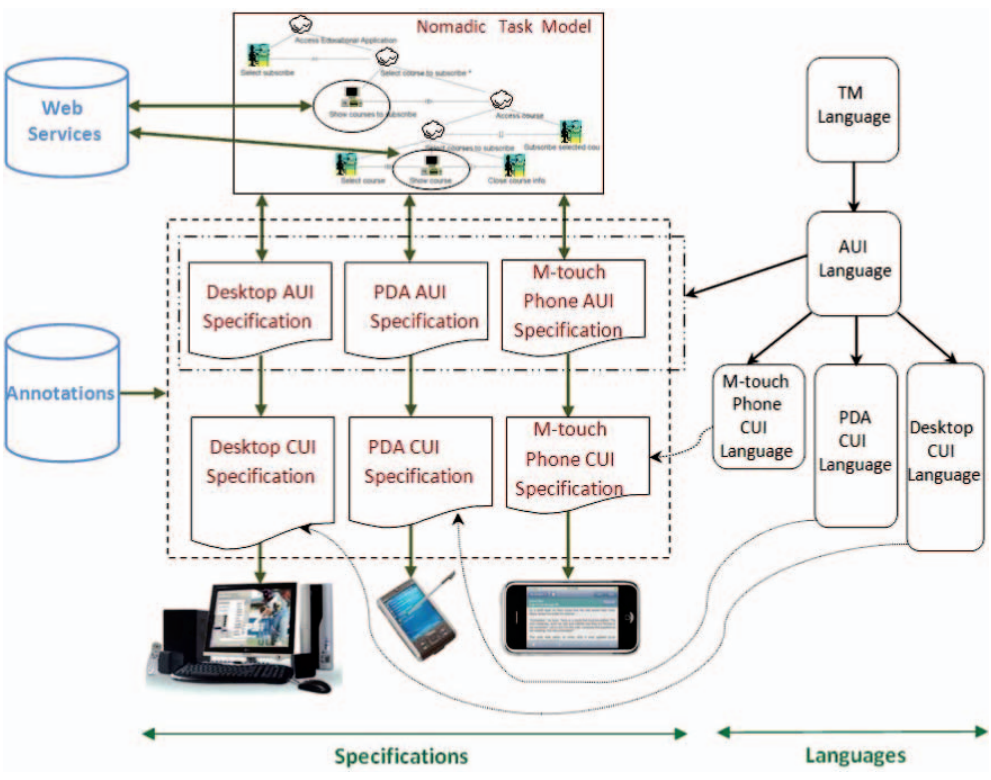

Figure 1. The method proposed.

and one concrete level, which is platform-dependent, and therefore, the associated languages define concepts in a way that depends on the specific platform in use. Therefore, for each different platform, we will have a different concrete language.

The fact that there is one task model language for specifying task models, and one single language for specifying abstract user interfaces means that each of such languages provides a vocabulary of concepts that are sufficient for specifying tasks (resp.: abstract user interfaces) in a way that is independent from a specific platform considered. However, it is worth pointing out that even if such languages provide concepts that are independent from the considered platform, they are still able to describe different interactions that can occur in different platforms. This is modelled in the central part of Figure 1, where you can see, for instance, that for the abstract level (which is platform independent), you can have different instances of abstract user interface (AUI) specifications that can be different. For instance, some abstract interactors that exist in an AUI specification could not exist in another specification if the supporting activity is not planned to be carried out in the targeted platform.

Indeed, when designing a specific multi-device application, even the models created with device independent languages should take into account the features of the target interaction modality. Thus, for example, in the task model, we can have tasks that depend on the actual modality (e.g. selecting a location in a map or showing a video), which thus are neglected if they cannot be supported (e.g. in the vocal modality).

Then, for each target platform, it is possible to filter the tasks relevant for it and then derive the corresponding abstract descriptions. Thus, abstract descriptions for user interfaces for versions for different platforms can differ, even if they use the same language. From the abstract description, it is then possible to derive more concrete ones in the specific concrete languages for each target platform. Since the concrete languages share a common core abstract vocabulary, this work is easier than working on a number of implementation modality-dependent languages.

5. Exploiting Web services and annotations in UI generation

In this section, we discuss how in our method it is possible to exploit the information contained in Web Services and related annotations for the generation of UIs, with particular reference to the supporting developed tool, MARIAE. The exploitation can be carried out at the different abstraction levels that may be involved in a multi-device UI specification, as is detailed in the following sub-sections.

5.1. Task level

As previously stated, the meeting point between the classical top-down approach of the user interface modelling and the bottom-up approach for composing services within the MARIAE tool is the binding between the system tasks and the service operations. Such correspondence allows the definition of the composition structure and the flow of information passing through the service operation chain.

Indeed, before creating any description of the user interface itself, the information contained in the service definition and the annotations enable improving the task model, describing more in detail the information exchanged between them. For instance, when a binding is created, the designer has to specify which interaction task provides the information related to all the input parameters. In addition, the designer has to specify which task exploits the information produced by the operation invocation, selecting from all the tasks that can be enabled by the bound system task.

Moreover, the annotations can be exploited for obtaining additional information with respect to that in the data types of the service definition.

5.2. Abstract level

At the abstract level, the annotations are exploited during the interactor generation process. Indeed, during the generation, the binding can provide information about those that are shown as only output interactors because they provide the results of the service execution and the objects that need to be shown as interactive interactors (e.g. editing or selection) because they require input from the users.

The selection of the input interactors depends on the data type analysis: Boolean: single choice with only a choice element; textual: text edit; numeric types: numerical edit and date types: text edit (a special treatment of such types cannot be done at the abstract level).

If the data type is shown as output, the correspondences are the following: Boolean: text; textual: text; numeric types: text and date types: text.

When the enumeration annotation is present, the interactor will be a single choice. The complex types are analysed recursively creating a grouping for each inner structure eventually reduced to simple types. For annotations like help, error messages, etc., a hidden group that contains them is created next to the interactor, creating a text for their textual representation or a description for the multimedia representation. For types representing lists or arrays, the input consists of a single choice between the already inserted elements, a group with the interface for the single element and two activators: one for inserting or

updating the current element and another for removing it from the list.

5.3. Concrete level

At the concrete level, the annotations are exploited for filling in the data corresponding to the concrete instance of the previously generated abstract interactors. More specifically, the annotations at the concrete level enable handling some information that can be useful for rendering the UI on a specific platform. Such information includes labels, visual properties, validation and multimedia information.

Within the MARIAE tool, it is possible to load annotations that are relevant for the design of the considered application. In particular, the user can specify either the address of a remote annotation repository in which the annotations can be found or, alternatively, a local file containing the annotations. In both cases, the annotations will be graphically displayed in a tree-like form within the MARIAE tool. Figure 2 shows how the annotations are displayed within the tool, for an example, banking application. As you can see, the tool shows annotations together with the related service operations they refer to.

Annotations regarding labels aim to suggest labels for presenting the associated data. They can be platform-dependent since, for instance, there might be short versions of labels to be used on mobile devices and longer version of the labels to be exploited on more capable devices such as the desktop PC.

Figure 2 shows how the information contained in the annotations is actually exploited within the MARIAE tool. The example considers the creation of a bank account for a new user, where the minimum number of transactions expected has to be specified. In the considered example, the annotation associated with the input operation for specifying the integer value of the minimum number of transactions is 'Min Transaction' (see Figure 2). Therefore, when the concrete user interface (CUI) (for graphical desktop) is built, such annotation information will provide the label of the spinbox UI object used to specify the numerical value (see Figure 3). A spinbox object is used in order to support the editing of a quantitative value within a predefined range.

Another type of annotation at the concrete level regards data validation. For instance, if at the abstract level we have a text-editing interactor, when we refine it at the concrete level, we can specify the format that the interactor requires in the associated annotation.

Another case is when an annotation provides more concrete information regarding the data types considered. For instance, it could happen that a Web service operation defines a value with a string data type as a parameter, while the annotation indicates that it is an enumerated value. In this case, the annotation information highlights that instead of using a concrete

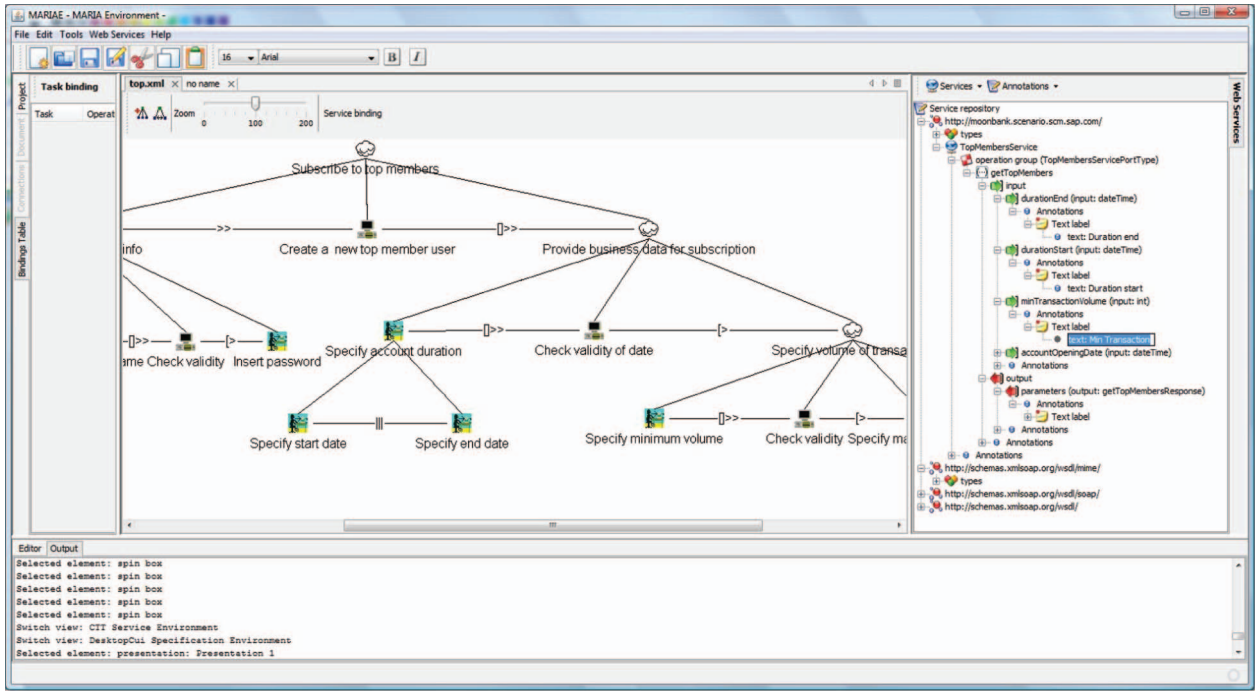

Figure 2. How the MARIAE tool supports the annotations.

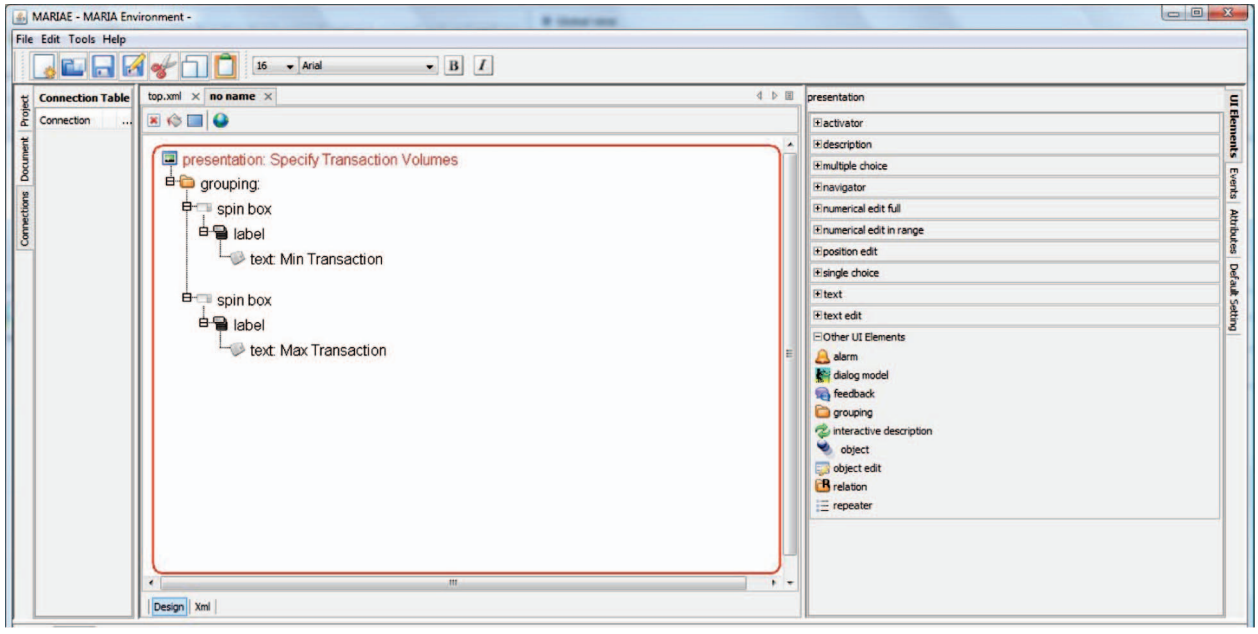

Figure 3. Using annotation information during the design of a concrete user interface.

refinement of an editing interactor (e.g. a text box in GUIs), a selection interactor would be more effective for choosing between a certain number of options (e.g. a list object or a pull-down menu in graphical UIs).

Regarding the visual properties of an interactor at the concrete level, such annotations could specify, for instance, whether or not a certain interactor can be hidden within the UI. Also, annotations at the concrete level regarding multimedia information allow specifying the type of multimedia object to be used to support the content associated with a certain interactor.

6. Example

The considered example is an educational scenario in which a student has to access a university website in order to get information on some courses. After logging in the system by providing username and password, she or he is presented with an overview page that displays a general summary of the scheduling of the courses in which she or he is currently enrolled. Such courses not only include technical university courses but also include language and sport.

Apart from a list of the courses, the system provides the user with the possibility to modify such list. For instance, she or he can decide to add some other courses or decide to withdraw from some courses in which she or he is no longer interested.

Figure 4 shows the task model describing the main functionalities of the system, and how users can access them. The model has been developed starting with an informal description of what the system should do and an early prototype highlighting the desired results. In the model, we can see a first part dedicated to the login followed by the access to the main functionalities, which are structured around three main tasks (overview, subscribe course and unsubscribe course), with the possibility to disable them (> operator) by the logout task.

In our case, the Web service used consists of a number of types and operations that implement the different functionalities offered by the service. More specifically, among the available operations, we have the following relevant ones:

- *Login* (input: login; output: loginResponse): This operation supports the login of the user. Its input parameter is a complex type composed of two string fields: username and password. The output parameter is a Boolean with the result of the process of login checking.
- *GetBasicData* (input: *getBasicData*; output: *getBasicDataResponse*): This operation provides some basic information about a specific student. The input parameter for this operation is the student's login. It outputs a parameter having a complex structure and composed of the following attributes: lastLogin (details about the last time the student logged in the system), name, surname and studentID.
- *SubscribeLanguage*/*SubscribeSport*/*Subscribe Lecture* (input: subscribeLanguage/subscribe Sport/subscribeLecture; output: subscribe

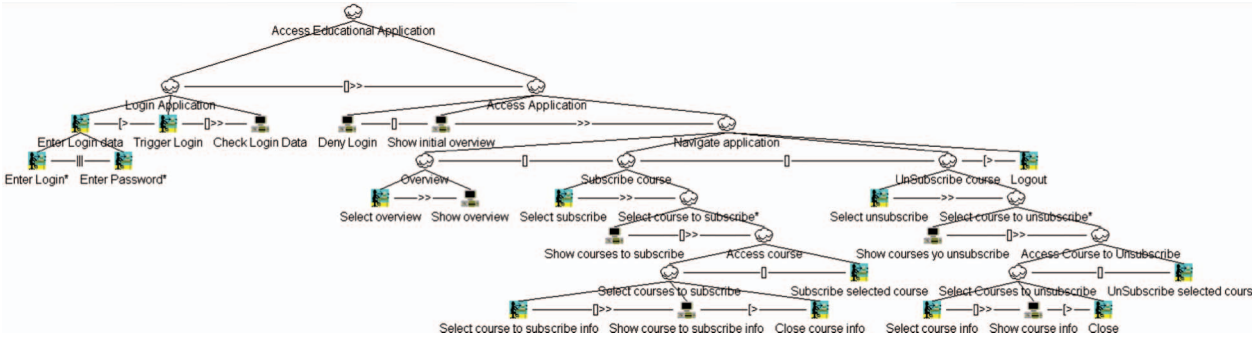

Figure 4. The initial task model.

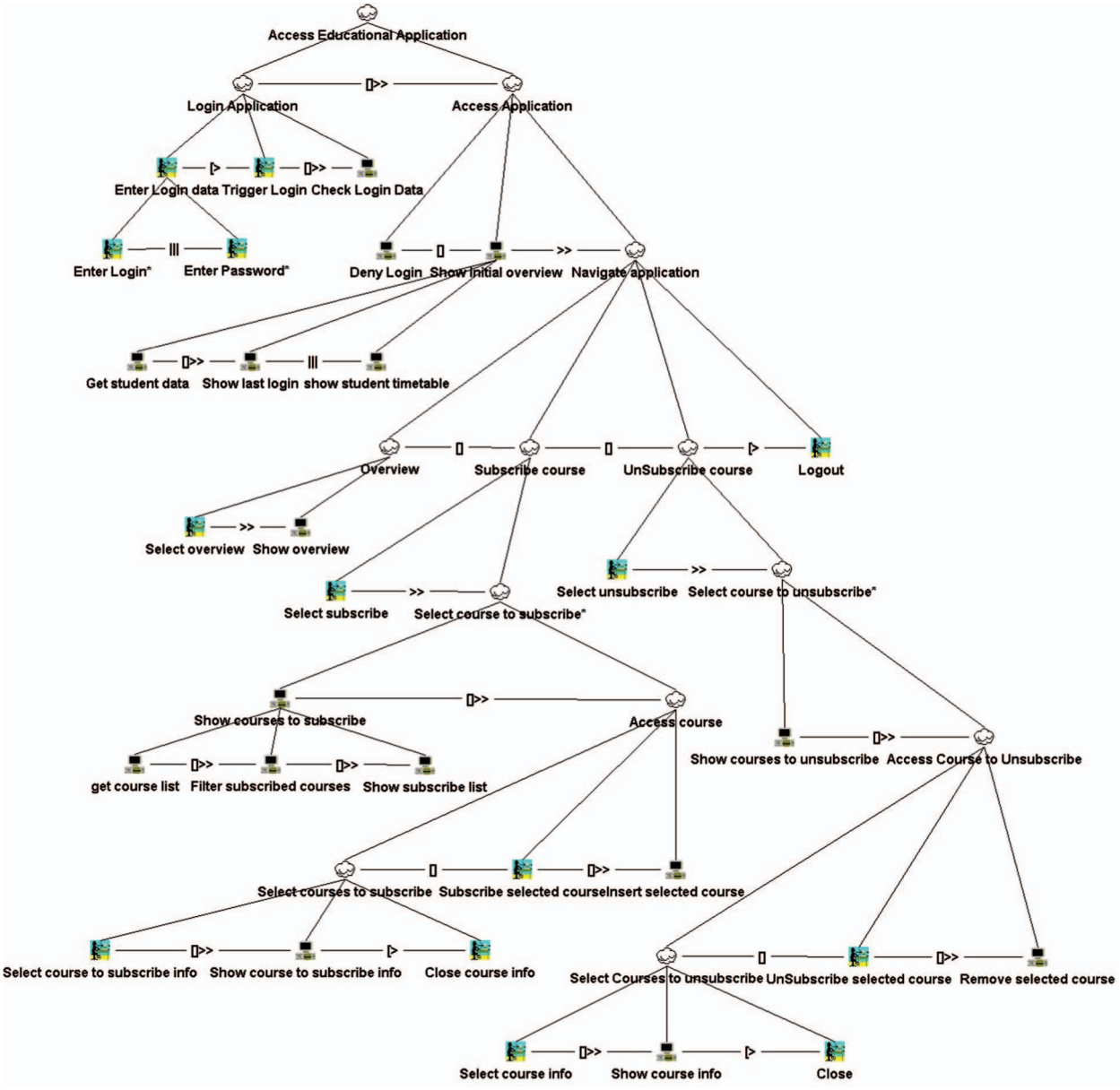

Figure 5. The revised task model.

Language Response): These operations support the subscription of the user to a specific language, course or lecture. They all share the same structure. Indeed, they have as an input parameter, a couple of integer fields: studentId and languageId (resp.: sportId/ lectureId). The studentId identifies the concerned student, and the languageId (resp.: sportId/lectureId) identifies the specific language, sport or lecture courses to which the student wants to subscribe. The output parameter in all three cases is a Boolean value (subscribed).

- *GetAllLectures* (input: *getAllLectures*; output: *getAllLectures*Response): This operation supports the delivery of all the lectures to which the student is currently subscribed. The operation delivers an array of objects as output, and each object correspond to a different lecture. The output object (*getAllLectures*Response) is a complex content defined through a number of properties or attributes: id, name, description, weekday, time, place, capacity or subscriptions.
- *GetStudentTimetable* (input: getStudentTime table; output: getStudentTimetableResponse): This operation provides some information about the student's timetable. As the input parameter, this operation gets the id of the student (which has integer type). As the output parameter, this operation gets an object of type 'timetable', which presents the student's weekly schedule.

Figure 5 shows the task model refined in order to take into account the specific Web service and the associated operations that have been chosen. In particular, the granularity of the system tasks has been refined in order to better describe the access to the operations' Web service. In particular, the task Show Initial Overview has been refined into three tasks to describe what happens in case of successful login. Indeed, in this case, the system first gets basic data regarding the student through a specific operation (student id, name, etc.), then with these data, the system looks for the last access by that user through another operation, and lastly through a further operation retrieves the student's course plan.

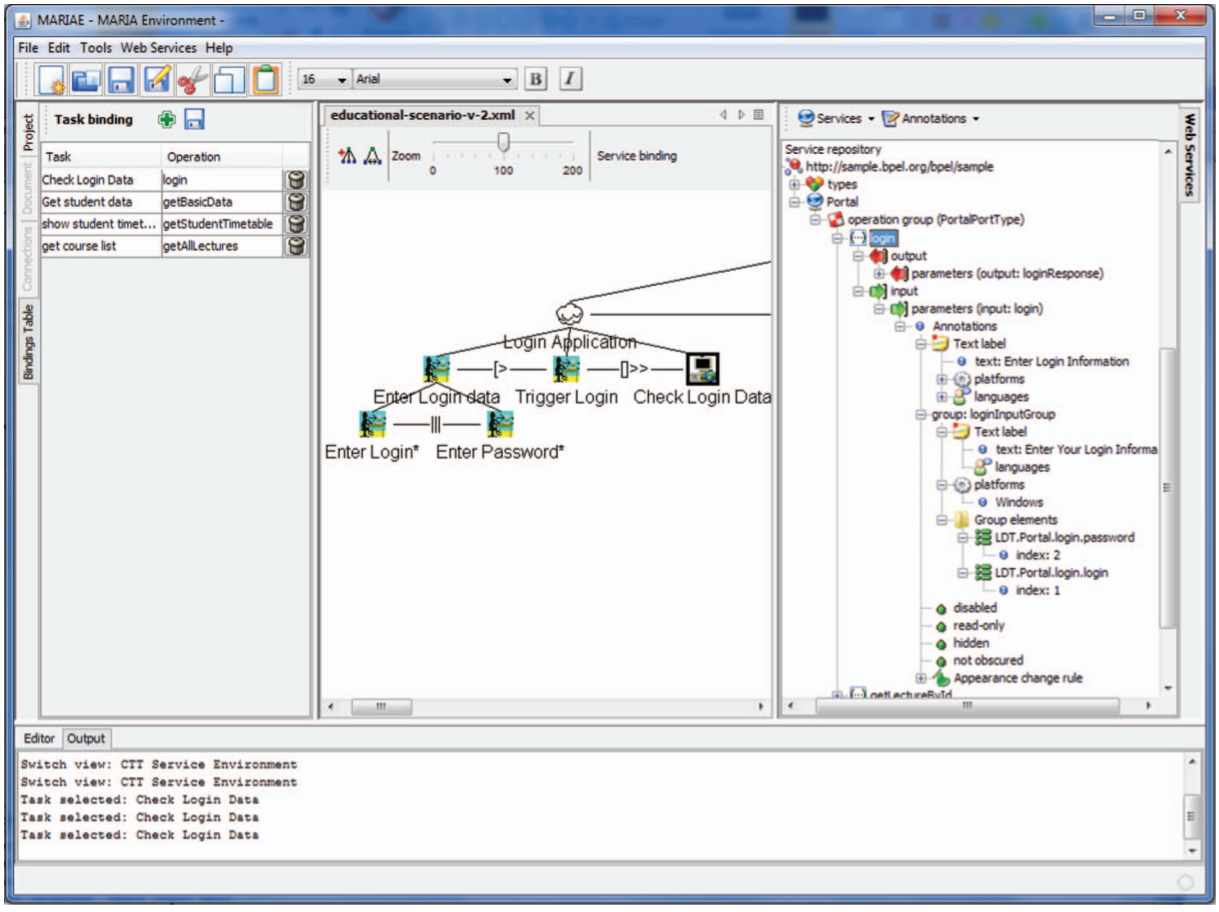

Figure 6. The display of services (with associated bindings), tasks and annotations in MARIAE.

Figure 6 shows the supporting tool displaying the services, the operations and the annotations. On the left side, the tool shows the coupling interactively performed by the designer among basic system tasks and operations in the Web services. In the central part, the task model is shown along with some controls to interactively configure its presentation. On the right side, the Web services and the associated annotations (when available) are shown. In such right side, it is possible to recognise the login operation with a number of annotations that provide the text for some labels, an indication of the supported platform, and the content language. There is also an indication of interface elements that should be grouped and elements that should be hidden.

When the tool performs the automatic transformation from the task model to the abstract user interface, a number of heuristics can be applied in order to avoid excessively fragmented presentations. In this case, from the task model shown before, we have obtained an abstract user interface with four main presentations. Figure 7 shows one such presentation in the tool. At this point, the tool interface is again divided into three parts but with different contents: on the left, an interactive tree view of the presentations and the interface elements that they contain; in the centre, the current abstract specification; on the right, the interface elements supported by the language that can be interactively inserted by drag-and-drop in the main area. In the current presentation, there are three main groups of elements and one interactor to perform the logout. One grouping is dedicated to showing information regarding the selected courses. This is obtained by a Repeater construct supported by the MARIA language that allows repeating the same type of information for each course. One grouping supports course selection and one access to course information

Figure 8 shows how the above abstract specification is transformed into a concrete description for a graphical desktop platform. In this case, we can see the concrete specification in the graphical syntax supported by the tool instead of the XML version. The concrete description shares the same structure of the abstract one, but provides more detail on the type of

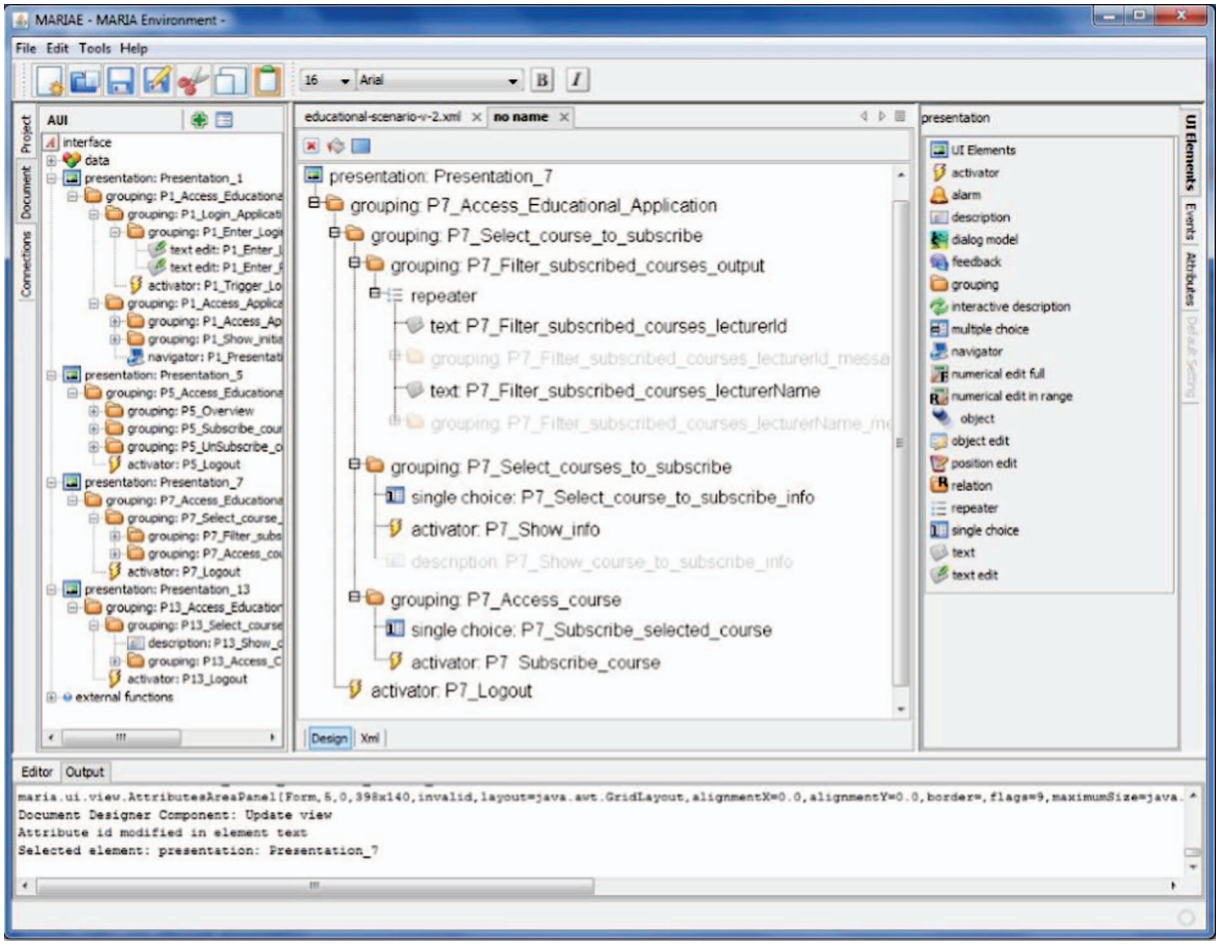

Figure 7. The AUI generated for the example.

Figure 8. The CUI generated for the example.

techniques to use for their implementation without assuming any specific implementation language. For example, it indicates that a single choice is carried out through a drop-down list or that an activator is performed by a button with a label.

The final transformation generates an implementation, in this case, in HTML and some scripts (see Figure 9). The final implementation shares the logical structure of the logical descriptions.

7. Evaluation

An early user test was performed on the developed environment. For this evaluation, users had to read a brief introduction to the development of UI models, explaining the different abstraction levels considered (task, abstract and concrete user interface). Then, they read the instruction for performing the test. After carrying out the test, they were asked to fill in a questionnaire. For the test, users were asked to:

(1) Import a CTT task model generated beforehand into the tool. Then, the user had to open an annotation file previously created for the Web services involved in the case study considered, and they were asked to associate the system task to the needed operations of the relevant Web services (so-called 'binding').
(2) Generate the first draft of the AUI based on the task model and the annotations about the web service operations bound to the model, and then possibly refine it.

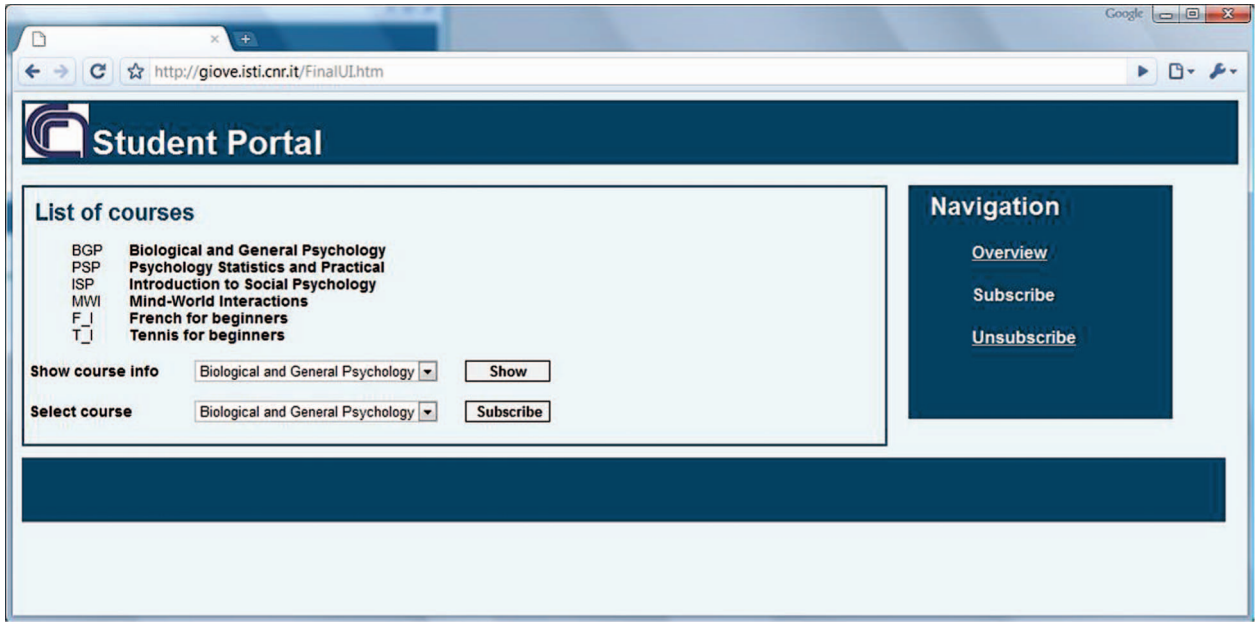

Figure 9. The final UI generated for the example.

(3) Create a CUI by using a transformation previously created for the test, and after that possibly refine the resulting CUI using the tool.

The test was performed by seven people, who were asked to rate the aspects of the methodology and the tool on a scale of one to five, with the possibility to add comments to their answers. The results obtained are reported in the following paragraph as arithmetic means plus or minus standard deviations.

The participants were all males graduated in Computer Science. The mean age of the participants was 27 $\pm$ 1.15 years old, with a good experience in the development of user interfaces (4.28 $\pm$ 0.48). On average, their experience in developing models for user interfaces is sufficient (3.57 $\pm$ 0.97), though a bit heterogeneous through the various UI levels (someone created only CUIs, someone only task models, etc.). Their knowledge about the development of applications based on Web services was rated sufficient (3.28 $\pm$ 0.75). The possibility to use annotations on Web services for creating the front-ends was really appreciated (4.57 $\pm$ 0.53), since the users judged that they can really speed-up the development. The visualisation of annotation and services was appreciated by all users (4.71 $\pm$ 0.48), who found very easy to navigate within the tree of annotation or services. The users liked the interface support for the binding between system tasks and operations (4.42 $\pm$ 0.53). From the comments it came out the suggestion to highlight the (systems) task that it is possible to bind, in order to facilitate the work of the designer.

The automatic creation of a first draft of the abstract interface starting from the task model was considered useful (4.85 $\pm$ 0.37), and the AUI editing interface was appreciated (4.42 $\pm$ 0.53). However, a user asked for improving the drag-and-drop mechanisms allowing the designer to move elements within the presentation, since he judged, not very intuitive, where the object was going to be placed. Regarding the transformations, the result of the transformation was judged really close to users' expectations (4.71 $\pm$ 0.48). The editing of the concrete level of the interface was rated good (4.85 $\pm$ 0.37). The overall impression of the methodology and the supporting tool was good, and the comments underlined the benefits for the creation of service front-ends. There were also some suggestions for trying to make more straightforward the overall process, since sometimes it happens to be too over-elaborated.

8. Conclusions and future work

In this article, we have discussed how HCI models can be exploited in the design of service-based applications and shown a method supported by an associated tool for this purpose. While such models have shown to be useful to support the development of multi-device user interfaces, in this article, we have focused on how they can support the design and development of service front-ends by also exploiting information from the Web service descriptions and associated annotations. We have described an example application of the method and the tool for an educational application. A first test of the environment has been reported as well.

Future work will be dedicated to performing more empirical validation of the approach proposed amongst developers of interactive service-based applications.

Acknowledgements

The authors gratefully acknowledge support from the EU ServFace Project (http://www.servface.du)

References

Calvary, G., *et al*., 2003. *The CAMELEON reference framework*. Deliverable 1.1, CAMELEON project. Available from: http://giove.isti.cnr.it/projects/cameleon/deliverable1_1.html [Accessed March 2011].

Dery-Pinna, A.-M., *et al*., 2008. ALIAS: a set of abstract languages for user interface assembly. *In*: *Proceedings of Software Engineering and Applications (SEA 2008)*, 16–18 November 2008 Orlando, Florida, USA. ACTA Press. Available from: http://www.actapress.com/Content_of_Proceeding.aspx?proceedingID=502

El Bekai, A. and Rossiter, N., 2005. A tree based algebra framework for XML data systems. *In*: *Proceedings of the seventh international conference on enterprise information systems*, 25–28 May 2005 Miami, USA. Available from: http://www.scitepress.org./DigitalLibrary/SignUp.aspx

Lepreux, S., Vanderdonckt, V., and Michotte, B., 2006. Visual design of user interfaces by (de)composition. *In*: *Proceedings of DSV-IS 2006, LNCS*, 26–28 July 2006, Dublin, Ireland. Berlin: Springer-Verlag, pp. 157–170.

Limbourg, Q., *et al*., 2004. USIXML: a language supporting multi-path development of user interfaces. *In*: Proceedings of the ninth IFIP working conference on engineering for human-computer interaction jointly with eleventh international workshop on design, specification, and verification of interactive systems, 11–13 July 2004 Hamburg. Berlin: Springer-Verlag, 200–220.

Lin, J. and Landay, J., 2008. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. *In*: *CHI '08 proceeding of the twenty-sixth annual SIGCHI conference on human factors in computing systems*, 5–10 April 2008, Florence, Italy. New York: ACM, 1313–1322.

Luyten, K., *et al*., 2004. Developing user interfaces with XML: advances on user interface description languages. *In: Advanced Visual Interfaces* 2004, 25–28 May 2004 Gallipoli, Italy.

Manolescu, I., *et al*., 2005. Model-driven design and deployment of service-enabled Web applications. *ACM Transactions on Internet Technology*, 5 (3), 439–479.

Myers, B., *et al*., 2000. Past, present, future of user interface tools. *ACM Transactions on Computer–Human Interaction*, 7 (1), 3–28.

Paternò, F., 1999. *Model-based design and evaluation of interactive applications*. London: Springer-Verlag.

Sawyer, P. and Maiden, N., 2009. How to use Web services in your requirements process. *IEEE Software*, 26 (1), 76–78.

Spillner, J., *et al*., 2008. Ad Hoc usage of Web services with Dynvoker. *In*: *ServiceWave '08 proceedings of the first european conference on towards a service-based internet, LNCS 5377*, 10–13 December, Madrid, Spain. Berlin: Springer-Verlag, 208–219.

Szekely, P.A., 1996. Retrospective and challenges for model-based interface development. *In*: *Design, specification and verification of interactive systems '96, proceedings of the third international eurographics workshop*, 5–7 June 1996 Namur, Belgium. Berlin: Springer-Verlag, 1–27.

Vermeulen, J., *et al*., 2008. Service-interaction descriptions: Augmenting services with user interface models. *In*: J. Gulliksen *et al*., eds. *Engineering interactive systems*, Lecture Notes in Computer Science, Vol. 4940, Berlin/Heidelberg: Springer-Verlag, 447–464.