On the Implementation of Multiplatform RIA User Interface Components

Marino Linaje[1], Juan Carlos Preciado[1],
Rober Morales-Chaparro[2], Fernando Sanchez-Figueroa[1]
[1]QUERCUS SEG, Universidad de Extremadura, Cáceres, Spain
[2]HOMERIA Open Solutions. R&D Department, Cáceres, Spain
{jcpreciado, mlinaje, robermorales, fernando}@unex.es

Abstract

Nowadays, there are a growing number of Web 1.0 applications that are migrating towards Web 2.0 User Interfaces, in search of multimedia support and higher levels of interaction among other features. These Web 2.0 features can be implemented using RIA technologies. However, most of the current Web Models do not fully exploit all the potential benefits of Rich Internet Applications. Although there are interesting works that extend existing methodologies to deal with RIA features, they do not fully exploit presentation issues. RUX is a method that focuses on the enrichment of the User Interface while takes full advantage of the functionality already provided by the existing Web models. Far from explaining RUX-Method in detail, this paper focuses on the way the method deals with the definition and transformation of components at different levels of abstraction for different RIA rendering platforms.

1. Introduction

With the appearance of Web 2.0, the complexity of tasks performed via Web applications User Interfaces (UIs) has been increasing, in particular when high levels of interaction, client-side processing, and multimedia capacities have to be performed. In this context traditional HTTP-HTML-based Web (Web 1.0) applications are showing their limits, presenting several restrictions. To cite a few, they have *Process limitations* (e.g., complex Web Applications often require that the user navigates through a series of pages to complete only one task); *Data limitations* (e.g., interactive explorations of the data are not allowed); *Configuration limitations* (e.g., many Webs require the configuration of a product/service from multiple choices, but in general, they are unable to present the customized product/service to the user in an intuitive way and in a single step) and *Feedback limitations* (e.g., continued and ordered interaction without page refreshments is not allowed, so the interaction of the user is quite limited) [5].

These are some reasons why developers are building the future of the Web using Web 2.0 UIs technologies by means of Rich Internet Applications (RIAs). RIAs overcome the limits mentioned above, combining the benefits of the Web distribution architecture with the interface interactivity and multimedia support available in desktop applications.

Some of the novel features of RIAs affect the User Interface (UI) and the interaction paradigm; others extend to architectural issues, such as, the client-server communication and the distribution of the data and business logic. They support online and offline usage, sophisticated UIs, data storage and processing capabilities directly at the client side, powerful interaction tools leading to better usability and personalization, lower bandwidth consumption, and better separation between presentation and content [5].

Although traditional Web methodologies are been extended in several directions to cope with some of these new features, currently they do not cover RIA composition parameters fully at all [5]. Most Web methodologies do not support multimedia properly, their focus being on data intensive Web applications (e.g., WebML [1], OO-H [2], UWE [3], OOHDM [7], etc.). In addition, most of the HCI and multimedia methodologies are not data intensive and business logic oriented because they mainly focus on presentation, temporal specifications to support multimedia/animations and final-user interaction.

We can conclude that there is a need for methods and tools for the systematic development of RIAs, particularly for the Presentation level. In this sense, RUX-Method (Rich User eXperience) [4] is a model driven method which supports the design of multimedia, multi-modal, multiplatform and multi-device interactive Web 2.0 UIs for RIAs.

For this purpose, RUX-Method makes use of a Component Library for the definition of Components for different RIA rendering platforms. This definition is made at different levels of abstraction. This Library also specifies the transformation between Components placed at different Interface levels. The objective of this paper is briefly presenting this Library and the Components from the theory to the practice and the way in which RUX-Method deals with the generation of RIA components.

The rest of the paper is as follows. Section 2 shows several design issues regarding RIAs. Section 3 briefly describes RUX-Method and its Component Library while section 4 illustrates the implementation of Components. Finally, section 5 shows conclusions.

2. Concepts and technologies in RIAs

Designing RIAs with Web engineering methodologies requires adapting the Web development flow of traditional Web applications to consider the new client-side capacities, the new presentation features, and the different communication mechanisms between the client and the server. According to [5] the main issues to be taken into account for the design of RIAs can be grouped in four issues: *Data* (persistent and volatile content can be stored and manipulated at client and server side); *Business Logic* (in RIA both the client and the server can carry out complex operations); *Communication* (RIAs allow (a)synchronous communications. But also data and functionality distribution across client and server broadens the features of the produced events as they can originate, be detected, notified, and processed in a variety of ways); *Presentation* (RIAs offer enhanced presentations and user interactions, allowing to operate as single page applications).

Most of the new capabilities that RIA offers can be placed in one of these phases. This work deals with the Presentation features.

2.1. Technologies for RIAs

Some of the most wide and well-known technologies used for developing Rich Internet Applications over the Web are AJAX, Flash, FLEX, OpenLaszlo and Silverlight. However, new emerging technologies like JavaFx are gaining ground day by day.

AJAX (Asynchronous Javascript and XML) is mainly based on Javascript and acts at client side for creating better interactive Web applications avoiding page refreshment. In AJAX, the data can be retrieved asynchronously using the XMLHttpRequest object without reloading the entire UI. Many frameworks are available to develop AJAX trying to avoid cross-browser problems.

OpenLaszlo is a RIA technology that follows the open source philosophy and uses a declarative code (LZX) that mixes XML and ECMAScript. The client rendering technologies required by Laszlo are AJAX and Flash-Player.

FLEX is also open source and uses declarative code approach (MXML) to develop RIA based on XML and ActionScript. FLEX needs the open source Flash-Player installed to run the application.

Silverlight, a technology based on Windows Presentation Foundation, is the Microsoft's platform for building RIAs. Silverlight also follows a declarative approach (XAML) for the UI description.

Other technologies, such as JavaFx or Mozilla Prism are just emerging, so it is early to discuss about the implications to incorporate them in RUX-Method.

3. RUX-Method in brief

RUX-Method [4] is a model driven method which supports the design of multimedia, multi-modal and multi-device interactive UIs for RIAs. RUX-Method focuses on the enrichment of the User Interface while takes full advantage of the content and functionality already provided by the existing Web models. A RUX-Method overview is depicted graphically in Figure 1.

At design time, RUX-Method uses existing data, business logic and presentation models offered by the underlying Web model being enriched. This information provides a UI abstraction which is transformed until the desired RIA UI is reached. At run time, while a new UI is generated from RUX-Method, the data and business logic remain the same. To sum up, the responsibility of RUX-Method is providing a new UI with RIA features.

To facilitate the UI development process, RUX-Method is divided into three Interface levels: Abstract, Concrete and Final Interfaces. Each Interface level is mainly composed by Interface Components whose specifications are stored in the Component Library. One Component can only belong to one Interface level. The Library also stores how the transformations among Components of different levels are carried out.

There are two kinds of adaptation phases in the RUX-Method according to the Interface levels defined above. Firstly, the adaptation phase that catches and adapts Web 1.0 (data and navigation, as well as presentation when it is possible) to RUX-Method Abstract Interface that is called Connection Rules (CR). Secondly, the adaptation phase that adapts this Abstract Interface to one or more particular devices and grants access to the business logic that is called Transformation Rules 1 (TR1).

Finally, there is an additional transformation phase, Transformation Rules 2, (TR2) that completes the MDA life-cycle of RUX-Method supporting and ensuring the right code generation. Thus, in TR2, the Final Interface is automatically obtained depending on the chosen RIA rendering technology (e.g. Laszlo, AJAX). This process is performed automatically because TR2 establishes the way the matching takes place among Concrete Interface Components and Final Interface Components.

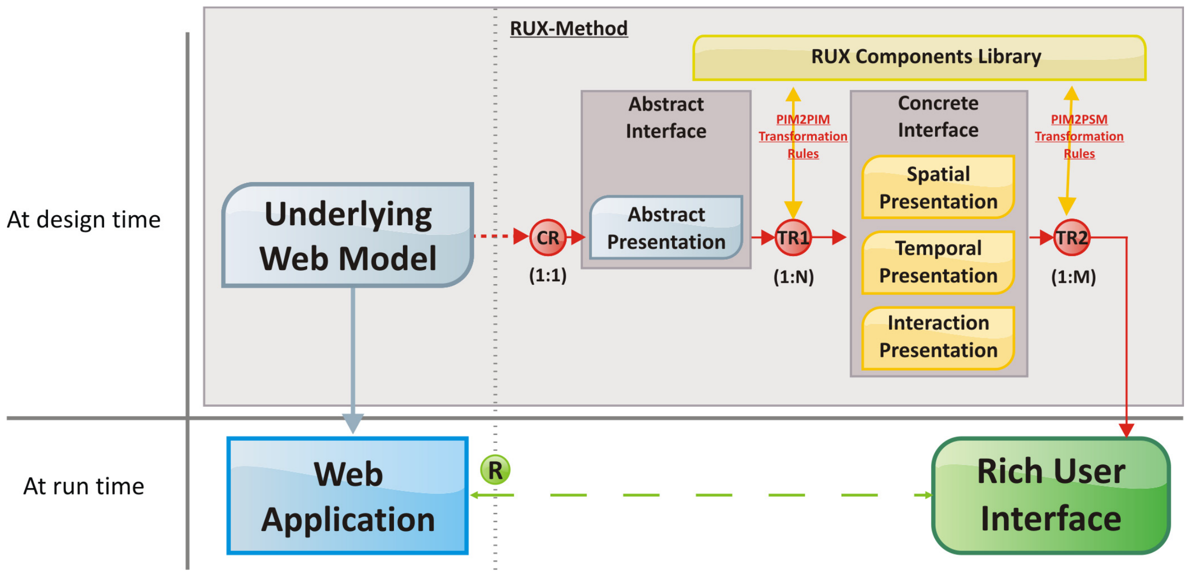

Figure 1. RUX-Method overview.

The cardinality in the relation among these phases for the Components is depicted in Figure 1. 1:N (Abstsract to Concrete Interface) and 1:M (Concrete to Final Interface).

In Figure 1 is depicted the full process at design time (at the top) and at run time (at the bottom). While at design time RUX-Method extracts the information from the underlying Web model, at run time the RIA UI obtained via RUX-Method communicates (marked R in Figure 1) with the Web application business logic obtained using the Web model.

3.1. RUX-Method Components Library

For a better understanding of the way in which the Transformation Rules are applied, it is necessary to know the role played by the Components Library. In this document and due to the RUX-Method nature, when we talk about components, we refer to User Interface Components. These Components can have different complexity levels and intrinsic functionality (e.g., Widgets, Gadgets, etc.).

The Components Library is responsible for:

1) storing the component specification (name, methods, properties and events),
2) specifying the transformation/mapping features for each Component from an Interface level into other Component in the following Interface level and
3) keeping the hierarchy among Components at each Interface level independently from other levels.

The set of Interface Components defined in the Library can be increased or modified by the modeller according to the specifications of the project. The set of available transformations can be also increased or updated according to the Interface Components included in the Library. For a given Component several transformations can be defined depending on the target interface level. The Component Library stores the interface level structure by means of skeletons.

3.2. Components and Transformations

Components are used for solving specific interface tasks. In the Abstract Interface the different kinds of media and views define the grouping and the type of elements that the user is going to perceive. These Abstract Interface Components are transformed (by means of TR1) into Concrete Interface Components according to the different ones available in RUX-Method: *Control* Components that are used for data I/O (e.g. *textcontrol*, etc.), *Layout* Components used to organize the content (e.g. *HDivideBox*, etc.) and *Navigational* Components that are used for navigating the content (e.g. *tabnav*, etc).

Each Component can be composed by different parts (not all of them required according to the Interface level of the Component): *Name* specifies the Component name and the list of Components from the previous Interface level that can be transformed in this Component. *Capability* expresses the functionality needed to express the Component behaviour. It is composed of a Header and a Body. While the Header is used and shared by all the instances of this Component in the application, the Body is specific for each instance. To clarify this abstract specification, let us show an example for an AJAX specific component: the Header can be a snippet CSS or JavaScript function placed once in the application. On the other hand, the Body is placed once for each instance defining different values in each case as necessary (e.g., example values for size, font color, etc), sharing all the instances of a component the same header. A *Property* indicates a Component characteristic that contains a value according to its Interface level.

For example, in the Abstract Interface a common property is *connectorid* that contains a reference (value) to the connector of the underlying Web model. In the Concrete Interface the *size* property, for example, is available in the *tabnav* Component. *Methods* express a way to communicate the whole interface with the Component (e.g. invoking a Component functionality). *Events* state for the list of events listened by the Component (e.g. ondrag).

From an abstract point of view a skeleton describes the basic structure of an application in a specific Interface level (meta-descriptor). There is a close relationship between skeletons and Components, since Components are placed in the Interface level according to the skeleton specified. A skeleton may include a set of common resources needed by Components.

Regarding the transformation among Components, both Transformation Rules (TR1 and TR2) follow the same steps:

1) use the skeleton defined (in the Component Library) for the target interface level,
2) transform each Component in the source interface level into its corresponding default Component in the target interface level,
3) enrich this skeleton including the Components obtained from step 2.

This process distributes the Component Headers and Bodies across the skeleton preserving the Component hierarchy.

4. Components in RUX-Tool

Currently, RUX-Method specifications are facilitated by a CASE tool called RUX-Tool [6] and real-life applications are being developed by the Homeria Open Solutions company. At the moment, RUX-Tool works with WebRatio 5.0 in commercial and academic terms allowing the design of rich UIs over Web 1.0 applications designed and generated with WebRatio (the WebML CASE tool). RUX-Tool is a browser-based RIA that works online and generates final UIs. The generated RIA is automatically deployed also online at server side, avoiding any kind of installation at client-side. Nowadays, the available UI code generators are FLEX, OpenLaszlo and AJAX. The RUX-Tool Component Library and code generators have a plug-in architecture for allowing the inclusion of new components and target platforms. Each component is a template based on Declarative Velocity Style Language (DVSL).

In practical terms, the skeleton indicates physically the set of files and folders to place the application and component resources and designates where and how to put the Component Header and each Component Body for a RUX-Method Component. This is carried out using skeletons written in XSL language for generating a XML file that describes the folders/files hierarchy for the whole application.

Far from explaining RUX-Tool Components in detail, we will use a little example for a better understanding.

4.1. Including Components in RUX-Tool

There is a list of steps to be followed for using Components in RUX-Tool:

1) Build the code of the component in all the platforms where the Component is going to be used,
2) Replace attributes used for spatial allocation (e.g., x, height) using DVSL variables,
3) Replace other attributes such as look&feel ones (e.g., background color, font size) using corresponding DVSL variables,
4) Replace data-based dynamically built elements using DVSL variables and
5) indicate the place where the content is placed if the component works like a container.

For illustrating this issue, we introduce the *vlistlayout* Component that stands for a vertical list layout and the *textcontrol* component that is a text output Component. *vlistlayout* is able to place several child elements (in the example *textcontrols*) acting like a vertical list (similar to HTML combobox).

For the sake of simplicity, we will define the Final Interface Component only for the AJAX (HTML-based) platforms. Firstly, we show the original component code which can be included in any HTML based application.

```
<table style="border: 1px solid black;
  overflow: hidden; width: 44px; height: 68px;
  position: absolute; left: 32px; top: 11px;
  background-color: rgb(245, 255, 244);">
  <tbody>
    <tr><td>
      <div style="border: 1px outset black;
        overflow: hidden; width: 31px;
        height: 21px; position: absolute;
        left: 10px; top: 35px;
        rgb(255, 255, 255);">
        Barcelona
      </div>
    </td></tr>
    <tr><td>
      <div style="border: 1px outset black;
        overflow: hidden; width: 31px;
        height: 21px; position: absolute;
        left: 10px; top: 35px;
        rgb(255, 255, 255);">
        Madrid
      </div>
    </td></tr>
  </tbody>
</table>
```

In general terms, this code will form part of the component Body after applying the corresponding DVSL dynamic replacing.

At this point, we need to replace the attributes used for spatial allocation using DVSL variables. The attributes *width*, *height*, *x* and *y* replaced are marked in bold text in the code below (these elements will be fixed dynamically in RUX-Tool when this Component is included in a Web application by the modeller). After this step, this Component can be used in RUX-Tool fixing the spatial arrangement.

```
<table style="border: 1px solid black;
  overflow: hidden;
  width: $attrib.w;
  height: $attrib.h;
  position: absolute;
  left: $attrib.x;
  top: $attrib.y;
  background-color: rgb(245, 255, 244);">
  <tbody>
    … (remains unmodified)
  </tbody>
</table>
```

In the next step, we show how other kind of attributes change, for example, the background color. In this case these attributes affect the look and feel of the Component, so they will appear in RUX-Tool inside the style properties menu.

```
<table style="border: 1px solid black;
  overflow: hidden;
  width: $attrib.w;
  height: $attrib.h;
  position: absolute;
  left: $attrib.x;
  top: $attrib.y;
  background-color:
   $node.Style.background-color">
  <tbody>
    … (remains unmodified)
  </tbody>
</table>
```

After this step any HTML tag that can be moved to <div> tag is moved and a CSS style is automatically assigned to maintain the previous appearance. One of the most interesting replacements comes when we need to add support for hierarchy and data-driven Components.

For the replacement of dynamic elements using DVSL variables, the first step is to prepare the Component to work with *n* items. In this case, we must indicate the specific portion of code that has to be iterated. In the example, code inside <tr> is wrapped for each item using the *#iterate_data()* directive.

The last step, deals with the insertion of Components that may contain other Components. Initially, we showed two items containing *Barcelona* and *Madrid*. In this case, it is necessary to introduce the *$context.applyTemplates(Node)* DVSL syntax directive, which applies templates to the specified node (similar to XSL) for generating each Component that can be a children of this node. DVSL *#match* directive is used to indicate that the code inside will be used for the output transformation when this template is applied.

```
#match("Part[@selected_component='vlistlayout']")
<div id="id_$!{attrib.id}"
  style="overflow: auto;
  position: absolute;
  left: $!{attrib.x};
  top:  $!{attrib.y};
  width:  $!{attrib.w}$!{attrib.wu};
  height: $!{attrib.h}$!{attrib.hu};
  background-color:
    $!{node.Style.background_color};">
  <div style="color:
    $!{node.Style.background_color};">
    #iterate_data()
    <div style="display: table-row;
      position: relative;
      width:100%;
      height: $!{node.Owned.item_size};">
      <div style="display:table-cell;
        position: absolute;
        width: 100%;
        height: $!{node.Owned.item_size};">
        #foreach(
            $child in $node.selectNodes("Part")
          )##composition
        $context.applyTemplates($child)##children
        #end
      </div>
    </div>
    #iterate_data_end()
  </div>
</div>
#end
```

The original HTML Component has been separated into two, the container (*vlistlayout* over these lines) and content (*textcontrol* under this paragraph). Following the same previous steps, the *textcontrol* Component can be also specified at Final interface level. The final description of this Component maintains the connection with the underlying Web model business logic using the *source* attribute.

```
<div style="width: $attrib.w;
  height: $attrib.h;
  overflow:hidden;
  position:absolute;
  left:$attrib.x;
  top:$attrib.y;">
    {$node.Source.attrib('connector_at')}
</div>
```

Of course, these two HTML-based Components of the Final interface level have a common representation to all the current RIA platforms at the Concrete Interface level. For the former, the representation of the Component in the Concrete Interface looks like this:

```
<Part selected_component="vlistlayout"
  rendering_component="vlistlayout"
  id="Rinu1_ai_ai"
  x="90" y="10" w="65" h="113"   wu="px" hu="px"
  type="Replicate"
  source_id="Rinu1_ai"
  name="cities">
  <Style>
    <background_color>#FFFFFF</background_color>
    <opacity>1</opacity>
    <font_family>Calibri</font_family>
    <font_style>plain</font_style>
    <font_weight>plain</font_weight>
    <font_size>12</font_size>
    <fgcolor>#000000</fgcolor>
    <stretches>none</stretches>
  </Style>
  …
</Part>
```

About this representation, we would like to clarify two attributes: *selected_component* that stands for the name of the component, *rendering_component* that RUX-Tool uses to facilitate the graphical manipulation of the Interface level (giving the component appearance).

Finally, there must be a correspondence also with the Abstract Interface level. The next snippet of code shows the Abstract Interface representation of the *textcontrol* that match with the Text that is a type of Media in the Abstract Interface level of RUX-Method.

```
<Source node_name="Media"
  x="10" y="27" w="55" h="22" wu="px" hu="px"
  is_output="true"
  type="Text"
  id="inu1_nameatt2_ai"
  source_id="inu1_nameatt2"
  name="nombre"
  connector_id="inu1_connector"
  connector_at="att2_name"/>
```

6. Conclusions

This paper has shown how RUX-Method allows the definition of Components at different levels of abstraction for the generation of multiplatform RIA User Interfaces. The definition of Components and the specification of transformation among Components are done in the Components Library. With the aid of this Library, it is possible to adapt an old Web 1.0 User Interface obtained following a Web methodology to a new Web 2.0 User Interface with multimedia support, richer user interactions and custom Components able to be rendered in different platforms. All these specifications are carried out by a module of RUX-Tool named Component Library editor, which makes possible to manage the whole set of Components and the relations among them.

Maybe a missing part of this work is related with the evaluation and comparison with other proposals. However, due to the originality of RUX-Method, it is not possible this type of content. Maybe the most relevant proposal regarding RIA UIs is [8]. Latter has also tool support, but the proposal miss some RIA UI features like temporal behaviour and do not use a previous Web model to full-fill the model-driven Web application development.

While theoretically RUX-Method can be combined with many of the existing Web models, at the moment specific CRs are just available for WebML [1] and UWE [3] while there is a work in progress with OO-H [2] and OOHDM [7].

Our aim at the workshop is discussing the best choices coming from the Web engineering and HCI fields for the implementation of multi-device, multi-modal and multi-platform components.

Acknowledgments: This work has been partially supported by TIN2005-09405-C02-02 and the Junta de Extremadura project PDT06A042. The authors express gratitude to the people working at Homeria Open Solutions.

7. References

[1] Ceri S., Fraternali P., Bongio A., Brambilla M., Comai S., and Matera M., *Designing Data-Intensive Web Applications*, Morgan Kauffmann, San Francisco, 2002

[2] Gómez J. and Cachero C., "OO-H Method: extending UML to model web interfaces", *Information modeling for internet applications*, Idea Group Publishing, 2003

[3] Koch, N., Knapp, A., Zhang, G., and Baumeister, H., "UML-Based Web Engineering: An Approach Based on Standards", *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, vol. 12, chapter 7, Springer-Verlag, 2007, pp 157-191

[4] Linaje, M., Preciado, J.C., and Sánchez-Figueroa, F. "Engineering Rich Internet Application User Interfaces over Legacy Web Models", *Internet Computing Magazine*, IEEE, vol.11, no.6, 2007, pp.53-59

[5] Preciado J.C., Linaje M., Comai S., and Sanchez-Figueroa F., "Designing Rich Internet Applications with Web Engineering Methodologies", *International Symposium on Web Site Evolution*, IEEE, 2007, pp. 23-30

[6] RUXProject Homepage: www.ruxproject.org

[7] Schwabe, D., Rossi, G. and Barbosa, S., "Systematic Hypermedia Design with OOHDM", *International Conference on Hypertext*, ACM Press, 1996, pp. 116 – 128

[8] Martínez-Ruiz FJ, Muñoz Arteaga J, Vanderdonckt J, et al (2006) A first draft of a Model-driven Method for Designing Graphical User Interfaces of Rich Internet Applications. Proceedings of LA-Web 32-38