Refactoring to Rich Internet Applications. A Model-Driven Approach

Gustavo Rossi[1], Matias Urbieta[1], Jeronimo Ginzburg [2], Damiano Distante[3], Alejandra Garrido[1]

[1]*LIFIA, Facultad de Informática, UNLP, La Plata, Argentina and Conicet*
{gustavo, matias.urbieta, garrido}@lifia.info.unlp.edu.ar

[2]*Departamento de Computación, Universidad de Buenos Aires, Argentina*
jginzbur@dc.uba.ar

[3]*RCOST – Research Centre on Software Technology*
Department of Engineering, University of Sannio, Italy
distante@unisannio.it

Abstract

Rich Internet Applications (RIAs) combine the simplicity of the hypertext paradigm with the flexibility of desktop interfaces. The quick emergence of these applications is driving a new (r)evolution in the Web field. Building RIAs from scratch is often unfeasible because companies do not want to loose their investments in legacy Web software; additionally, most users are still accustomed to the "old" Web interaction style. In this paper we present an evolutionary approach to transform conventional Web software into RIAs; we show how to apply the well-known refactoring concept to seamless introduce rich interface functionality in a Web application. By applying refactoring at the model level, we make the transition more systematic and less prone to error. We briefly introduce the problem with a simple example, and then we describe two refactorings and present our approach to specify these refactorings at the interface design level.

1. Introduction

In the last years, we have witnessed a fast growth of Web applications exhibiting sophisticated user interface behaviors. These applications, known as RIAs, have introduced the richness of desktop interfaces into the peaceful world of the navigational Web. Many frameworks have emerged to ease the construction of this Web software, such as Ajax [6] and Open Lazlo [15]. Once again, designers are facing a nightmare: not only they have to accompany the rapid pace of Web applications evolution; now, they have to transform the "old" Web software into the fashionable RIA interfaces. To make matters worse, RIAs seem to be always in "beta" state: new interface features are introduced, tested and then consolidated or discarded.

In some cases, the added behaviors belong to new design concerns with respect to the "legacy" application, e.g., the addition of a chat window in a mail program (see for example Gmail or Yahoo mail). This improvements may also introduce crosscutting behaviors (i.e., the new functionality affects the old one). Though dealing with these concerns separately is feasible [9], many other problems arise as shown later in the paper.

Unfortunately, while there is still no common agreement or measure on the impact of these changes on final users, the transition to RIA has already started, and designers, developers and maintainers need to face it.

There are many alternatives to manage the migration of a Web application to a RIA. For example, the RUX-model approach [11] extracts existing data and business logic from the Web application being adapted, and provides a set of user interface abstractions to specify the structure and behavior of the new RIA. In [12], the authors propose a semi-automatic approach that supports the migration of classical multi-page Web applications to single-page AJAX RIAs. Additionally, when the new application is radically different from the old one, the two must co-exist, as it is the case in Yahoo mail.

However, many well-known applications, such as Amazon.com, eBay.com and CNN.com, have used a radically different strategy. Instead of undergoing a thorough migrating process yielding a complete RIA version of them, they were subject to a smooth evolutionary approach. They started changing limited parts of their sites, introducing rich interface functionalities in a step by step way, evaluating them with customers and enriching the application seamlessly. In some cases, the enrichments were discarded after some time and the "old" style preferred.

As an example of this type of evolution, we show in Figure 1.a and 1.b the "old" and new styles, respectively, of a product list in Amazon.com. In Figure 1.a, we can see a conventional vertical index that scrolls with the page. Meanwhile, Figure 1.b shows a horizontally scrollable index where mouse hovering on one of the elements allows previewing part of the target contents and even execute some operations on the target product.

Figure 1.a: An index with a conventional interface

We have formalized this evolutionary approach by using the concept of *Web Model Refactoring* [7]. A Web Model Refactoring (WMR) is a change applied on the navigation or interface model of a Web application, aimed at improving its external quality while preserving the application's behavior. This paper focuses on WMRs over the application's graphical interface design that may transform a legacy Web application into a RIA. We call them *RIA refactorings*. RIA refactorings are described as compositions of RIA interfaces [18], which in turn are formally specified with Abstract Data Views (ADVs) [3]. As an example, we present two RIA refactorings that introduce typical RIA interaction styles by using oblivious composition of ADVs.

Figure 1.b: A RIA version for the index in Figure 1.a

We also show how a clear separation of structural and behavioral concerns simplifies the specification of RIA refactorings, which can be represented as weavings of simpler interface behaviors. Additionally, our style recognizes the volatile nature of the evolutionary process, in which new features might be either discarded or consolidated, and therefore aims at making evolution non-intrusive by preventing, when possible, model editions.

The main contributions of our approach are the following:

- We present a novel model-based approach for systematically transforming a Web application into a RIA, by applying small design improvements that we call RIA refactorings.
- We formalize the approach by showing how to specify the resulting Web interfaces using ADVs.
- We show how to represent some meaningful changes as weavings of oblivious interface models, therefore simplifying the process of evolution.

Regarding the first contribution, we extend our refactoring catalogue (initially presented in [7]) to include some new refactorings towards well-known RIA patterns. The second contribution presents a minor variant of the approach presented in [18]. The third contribution is the major novelty of this paper, since it shows how to use separation of concerns to introduce RIA refactorings as weaving of oblivious interface atoms.

The rest of the paper is structured as follows. In Section 2 we present a short introduction to Web model refactoring. In Section 3 we show how to apply this concept to introduce RIA interface features; we present our approach to specify and compose RIA interfaces and show how to specify refactorings as

compositions; we also discuss some issues on mapping interface refactorings to implementation. In Section 4 we discuss some related work. Finally, in Section 5 we conclude the paper and present some further work we are pursuing. For space reasons we concentrate on interface transformations, and ignore back-end changes, although Section 5 presents some comments on these aspects.

2. Model-Based Refactoring in Web Applications

Refactoring was originally defined in the context of object-oriented systems to "factor out" new abstractions by applying small changes to the source code of an application that preserve its behavior [14]. These changes aim at improving the internal structure of the code, making it more reusable and maintainable [4]. Refactorings are usually motivated by "bad smells" in design [4], i.e., heuristics that may indicate poor design quality.

Model refactoring has also been proposed to support the process of continuous improvement of an application's design [23], for example to introduce design patterns [5] in the context of an agile approach [10].

In the case of Web applications, we have defined *Web model refactorings* as those changes that can be applied to the navigation and interface models of a Web application that preserve the application's behavior [7]. Web model refactorings differ from conventional model refactoring in that they are targeted at improving the external quality of the application instead of the internal structure. Moreover, conventional refactorings preserve the "observable behavior" of the application [4], i.e., the mapping from input to output values. In the case of WMRs, they change "observable models" (i.e., models of the user interaction with the application) so none of them would be legal if they had to preserve "observable behavior". Instead, they preserve the behavior defined in the underlying application or domain model, and preserve the "availability" of this behavior, which means that neither nodes with operations may become disconnected nor their interfaces may be discarded [7].

The aim of WMRs is to apply slight changes to the navigational or interface structure of the application in order to make it easier to use (e.g., by improving the interface look and feel, by reducing the navigation steps needed to perform a task, etc.). They are also triggered by "bad smells" in design, in this case, the navigation and user interface design, and are usually motivated by well-known Web patterns as those in [19,21].

Navigation model refactorings may change, among others: the contents of a node, the set of outgoing links of a node, the navigation topology between a set of nodes (guided tour, index, etc.), and the user operations accessible from a node [7]. Meanwhile, presentation model refactorings aim at improving the look and feel of a page by changing the arrangement or type of widgets, the number of sections in a page, the interface effects, etc. We have described a number of navigation and presentation model refactorings using a simplified template comprising motivation, mechanics and example [7]. Notice that the mechanics can be described at different levels of abstraction, independently of the underlying design method or approach. We next present an example of a navigation model refactoring.

Turn Information into Link

***Motivation*:** During the process of completing a business transaction, some Web pages may show intermediate results or a succinct review of the information gathered until a certain point of the transaction. A common example occurs when checking the status of the shopping cart during the process of buying some products in an e-commerce site. Such Web pages should provide the user with the chance to review the choices and the information provided in previous steps of the process (e.g., items in the shopping cart, shipping and payment data, etc.) by means of direct links to the pages showing details on them.

Mechanics: In the navigation model of the Web application, find the node corresponding to the intermediate results page. Select the portion of information about the target item that better distinguishes it. Add a link from the node representing the intermediate results towards the target node; the anchor of the link would be the selected portion of information.

Example: This refactoring may be used to add links from names of products in a shopping cart, to the pages showing detailed information about the products.

The catalogue of refactorings we have identified includes: Replace Widget (to make an interface object look close to its intent), Split List (dividing the entries into several pages), Add Information, etc. Their rationales and descriptions can be read in [7,13].

3. Model-Based Refactoring to RIA

We can use the concept of Web model refactoring to seamlessly introduce RIA features into a Web application; we call these specific WMRs *RIA refactorings*. RIA refactorings may be guided by RIA patterns, and they usually arise when we discover some "bad smell" in the application interaction style that can be eliminated by introducing some richer behavior.

Some RIA refactorings are pure interface refactorings, but most of them combine transformations of the navigation and interface models. In fact, introducing RIA features often results in changing both the interaction styles and the hypertext structure; this is the case, for example, when the target of a link is "transcluded" into the same page of the link.

The main message of this paper is that a new application supporting RIA features can be obtained by:

a) Identifying the desirable changes by exploring a catalogue of refactorings and detecting "bad smells" in the corresponding application.
b) Applying the corresponding refactoring mechanics to the involved ADVs.
c) Generating the new application from the transformed ADVs.

In this section we will concentrate on items a) and b) of the above list, by showing that RIA refactorings can be obtained by oblivious composition of interface objects and interaction styles, which are modeled as belonging to different concerns. We introduce our specification constructs in Section 3.1; we show the basic composition style in Section 3.2 and its application to refactoring in Section 3.3. To illustrate our approach, we describe in Section 3.4 two RIA refactorings which, when applied in sequence, yield the transformation from Figure 1.a into Figure 1.b. We discuss our contributions in Section 3.5 and provide some comments on implementation in Section 3.6.

3.1. Modeling RIA Interfaces

In [18] we presented a systematic approach for designing the interface of RIAs; the proposed approach extends the OOHDM [17] interface design model by allowing separation of independent or crosscutting concerns in the interface specification. For each concern we specify a set of Abstract Data Views (ADVs) [3]. An ADV is a composite interface object intended to specify a navigational object's (the ADV's owner) look and feel. ADVs can be easily mapped, either manually or automatically, into running interface objects (e.g., XML/XSL specifications).

An ADV behavior can be exercised by traditional method calls and also by interface or internally generated events (such as "mouse click"). ADVs can be composed or grouped in generalization/specialization hierarchies therefore allowing some level of reuse, when defining recurrent interface object types (like buttons, maps, etc.). ADVs promote separation of concerns because they do not deal with data or business logic, which is usually managed in the ADVs' owners (i.e., nodes). However, being full fledged objects, they can contain arbitrary behaviors, including part of the business logic which in some RIAs might be also allocated in the interface [2].

ADVs specify the interface aspects of its owner, i.e. how we intend the owner to be perceived by the user. ADVs may also relate with their owners not just to indicate the owner's look and feel but to trigger the owners' behaviors (which is the case with buttons, menus, list of options, etc.). The relationships with application objects are specified using configuration diagrams, which are similar to UML class diagrams emphasizing the messages that clients send to servers. ADVs are also used to indicate how interaction will proceed and which interface effects take place as the result of user interaction. These behavioral aspects, which are specified using ADV-charts [2] (a kind of State charts), are of great importance for RIA modeling. ADVcharts generalize Statecharts to deal with aspects of design specific to interactive systems, such as using pointing devices to associate events with particular ADVs or focus of control. Besides, different from Statecharts, which only provide behavioural (state) nesting, ADV-charts also support structural nesting by allowing ADVs inside states and vice versa. This greater communication power does not imply a loose of computational power as it has been shown elsewhere [3] that an ADV-chart can be translated into an equivalent Statechart.

An ADV chart comprises a set of transitions which rules interfaces transformations. Each transitions specifies an event that must be handled, a precondition that must be satisfied for transforming the user interface once the event is triggered, and a postcondition that specifies the resultant interface state. Both pre and post conditions are Boolean expressions. The Boolean expressions may be formed by objects methods and Boolean operations: and "&", or "|", and negation operation; it is also possible to use short-circuit operations. The examples that follow use the function Focus(), which indicates the position of the cursor. They also use a pseudo-variable called perCont

(referring to the perception context) to indicate the objects that are perceivable; these objects are "added" or "subtracted" from the perception context.

In Figure 2 we show the ADV corresponding to a video user interface (UI) component, whose real interface is shown at the right. This ADV is composed of inner ADVs that may belong to primitive types (like the buttons or the scrollbar) or other user-defined ADVs (like the playList). The progress bar ADV is specified outside the VideoADV because of its complexity and to facilitate its reuse. The position of inner ADVs in the diagram gives a cue to graphic designers and may also be used to improve automatic layout generation.

A simplified ADV-chart specifying the behavior of the Video UI component is presented in Figure 3. In this chart we can see how the interface reacts when the Play, Pause or PlayList buttons are pressed. These behaviors are specified at the right of Figure 3 indicating, for each state transition, which event causes the transition, under which preconditions it can occur, and which are the side effects, expressed as post-conditions. Transition labeled 1 occurs when the Play button is clicked; the Play button becomes disabled and the stop button becomes enabled. On the other hand, transition 3 makes the Video Title and the Play List perceivable minimizing the video. Finally, transitions 2 and 4 invert transitions 1 and 3 respectively.

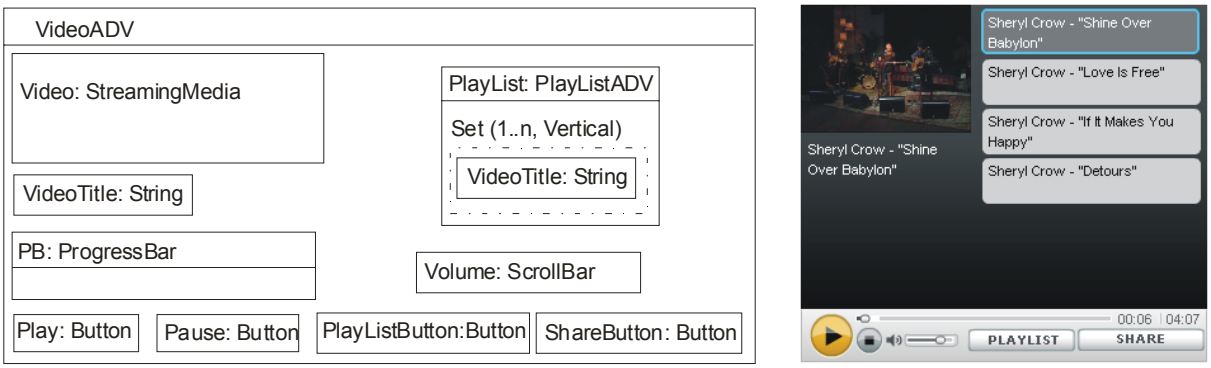
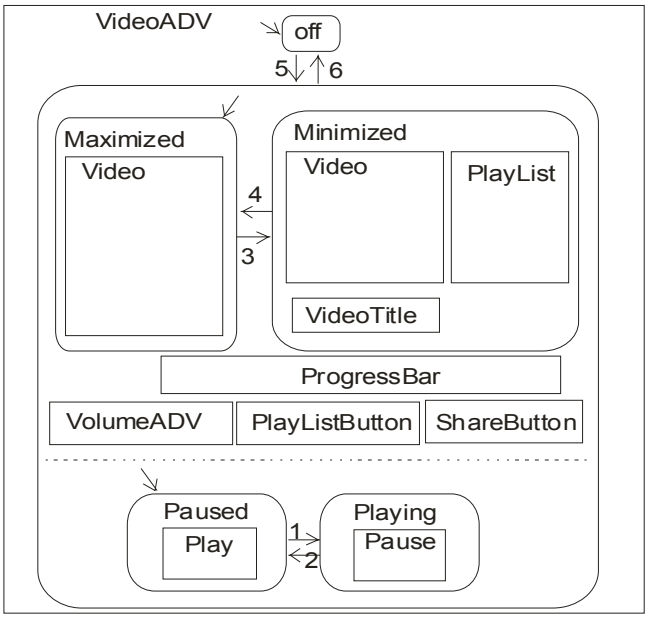


Figure 2: Video ADV



1:
Event: MouseClicked
Pre-Cond: Focus(Play)
Post-Cond: owner.isPlaying()
2:
Event: MouseClicked
Pre-Cond: Focus(Pause)
Post-Cond: ! owner.isPlaying()
3:
Event: MouseClicked
Pre-Cond: Focus(PlayListButton)
Post-Cond: perCont=perCont+PlayList+VideoTitle
4:
Event: MouseClicked
Pre-Cond: (Focus(PlayListButton) || Focus(Video))
Post-Cond: perCont=perCont-PlayList-VideoTitle
5: Event: Display
Pre-Cond:
Post-Cond: perCont=perCont+VideoADV
6:Event: Hide
Pre-Cond:
Post-Cond: perCont=perCont-VideoADV

Figure 3: ADV-Chart for Video ADV

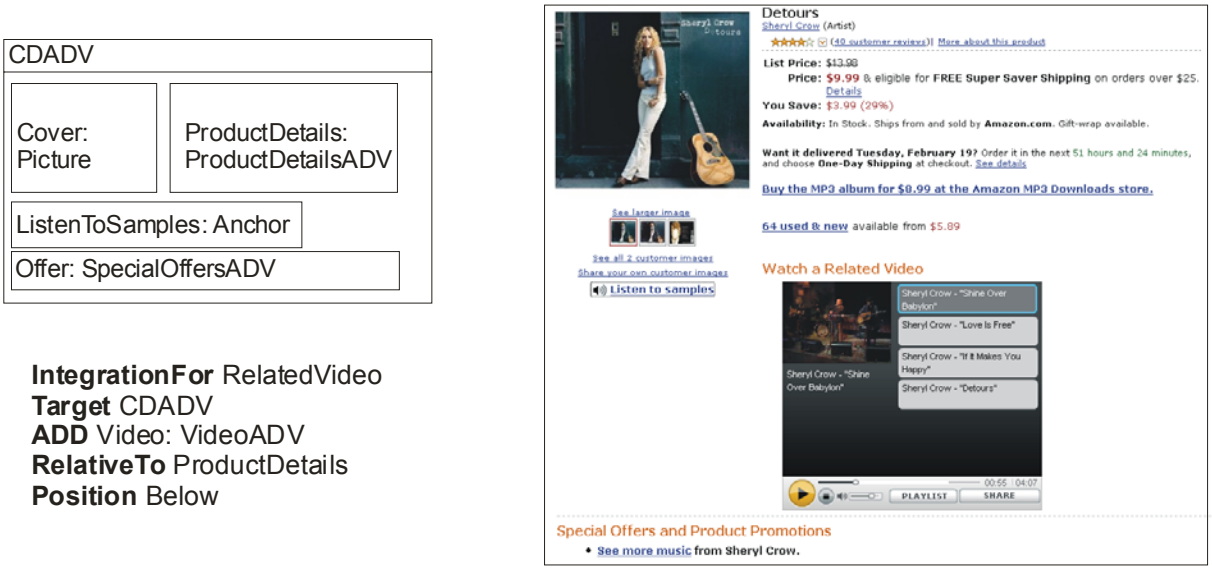


Figure 4: Integrating the Video into the CD ADV

3.2 Composing RIA Interface Designs

As mentioned before, most complex RIAs deal with different application concerns. In a previous paper [8], we presented an approach to modularize interface concerns in order to make compositions seamless and unobtrusive. User interfaces belonging to different concerns are designed separately and, when possible, obliviously from each other; then they are composed using an integration specification. This solution facilitates the application's evolution by allowing each concern to be developed regardless of the others.

As an example, we show at the left of Figure 4, a fragment of the ADV corresponding to a CD page and the integration specification to seamlessly compose the Video ADV (shown in Figure 2) with it. At the right of Figure 4 we show the resulting interface. This way of specification helps to deal with unstable or volatile functionality (e.g., when the video might be later removed from the page), because the integration is specified separately from the other ADVs, which remain independent from each other. The integration specification indicates the position in which the new ADV (VideoADV) will be inserted.

3.3 Refactorings as Compositions

RIA refactorings introduce new interaction facilities on existing interface objects and usually might introduce new interface objects as shown in Figure 1.b. Instead of changing the original interface specification, we apply a RIA refactoring by composing the new ADV (which represents the item details hovering in Figure 1.b) using the approach presented in Section 3.2,. The composition might involve "just" changing the behavior of the original ADV, as it is necessary when transforming a typical hypertext index into a scrollable one, or it might involve more complex compositions, as we will show next.

Specifying a RIA refactoring as a composition has several advantages:

- First, it allows reversing the refactoring as the original model was not edited;
- Second, it allows defining composition templates (as we show in Section 3.4);
- Finally, we can have libraries of interaction styles defined as ADVs (with their corresponding ADV-charts) to apply the refactorings.

3.4 RIA Refactorings

RIA refactorings are applied to concrete applications, by transforming specific interface elements into others, adding new interaction facilities, etc. However, it is possible to express the mechanics of refactorings at a higher-level, by using what we call "ADV templates".

An ADV template comprises the roles played by the different objects and the involved behaviors, i.e., interface objects and their behaviors, and interaction

styles. The template provides hooks to apply the refactorings by replacing static references at: ADVs, ADVCharts and Integration Specifications. The hooks may be interface components such as Pictures, TextFields, or more complex components; in our interface model, these elements are also ADVs.

Parameters Types are introduced by "<>"and hooks become template's parameters. Therefore, the changes proposed in the mechanics of RIA refactorings are realized by instantiating the corresponding parameters as defined in the ADV template.

We next present two RIA refactorings: Make Index Scrollable and Add Link Target Anticipation. They can be considered as specializations of the Web model refactorings Introduce Scrolling and Anticipate Target presented in a previous work [7]. In this paper we precisely describe the mechanics of these refactorings by composition of ADVs.

I) Make Index Scrollable

***Motivation*:** A Web page presenting a long linearly ordered index of items in an e-commerce site, may require the user to repeatedly scroll the whole page. In order to improve navigability and highlight some index items, it is possible to make the index horizontal and scrollable. A pair of buttons (Left and Right) are introduced, which trigger a shift between items to the corresponding left or right.

In this way, instead of using the browser scroll-bar, we introduce two application scroll controls that help to reduce the space devoted to the index. Alternatively, the scroll could be vertical or circular, i.e., in a carrousel style. The latter might be preferable for a small number of elements because we can keep them all visible.

Mechanics: The refactoring starts by selecting the target user interface component (StaticIndex) that will be enriched. Following with our running example, Figure 5 shows the interface for a "Recommendations" page, which contains an ADV for the products' index. In turn, the index ADV contains a set of Product ADVs, each comprising all the information of a product. We do not show the ADV-Chart for this interface since it does not exhibit behaviors beyond navigation, i.e., it only reacts to the MouseClick event by triggering a link.

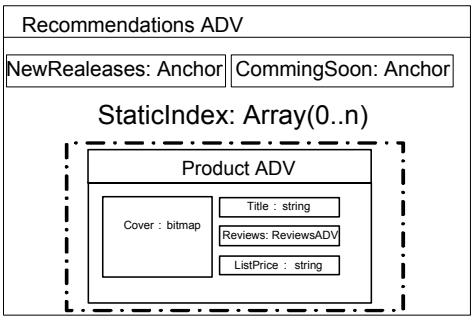


Figure 5: Index ADV

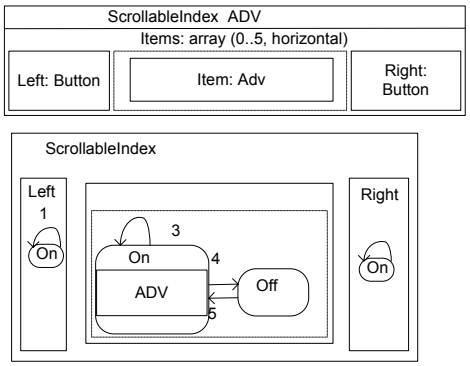
The second step is defining the behavior of a generic scrollable index component, whose ADV and ADV-Chart are (partially) specified in Figure 6.

The third step consists of composing the Recommendations ADV with the new ADV for index entries in the way described in Section 3.2. For this purpose we define the following integration specification:

IntegrationFor Make Index Scrollable
Target Recommendations
Add Index: IndexScrollable
RelativeTo StaticIndex
Position Replace

This integration specification can be used in order to replace the traditional index of Figure 5 with the new index. Here we indicate that we wish to substitute the original index with the scrollable one. By changing the integration specification we can obtain other results. For example, if we use another generic template like *Carrousel* (instead of stopping at the last index item it start again with the index's first item) we could obtain a different index look and feel.

Notice that the changes produced by the refactoring do not remove any operations nor data from the interface, therefore preserving the applications' behavior.



1
Event: MouseClicked
Pre-Cond: Focus (Left)
Post-Cond: ShiftLeft(),

2
Event: MouseClicked
Pre-Cond: Focus (Left)
Post-Cond: ShiftLeft(),

3.
Event: MouseClicked
Pre-Cond:
Focus (Items[i])
Post-Cond:
Items[i].AnchorSelected()

4,5)
Event: ShiftLeft
Pre-Cond: OFFSET>0;
((∀ i) (0<= i < 5 && Items.get(i).getModel() == owner.Items.get(OFFSET+i))
Post-Cond:
(∀ i) (0<= i < 5)
Items.get(i).getModel()==owner.Items.get(OFFSET +i - 1))

Event: ShiftRight
Pre-Cond: OFFSET<owner.Items.size()-5
((∀ i) (0<= i < 5 && Items.get(i).getModel() == owner.Items.get(i+OFFSET))
Post-Cond:
(∀ i) (0<= i < 5)
Items.get(i).getModel()==owner.Items.get(OFFSET +i + 1))

Figure 6: Scrollable Index template definition

II) Add Link Target Anticipation

Motivation: Some UIs present a long list of elements to the user, for example as the result of a search operation. Choosing among the elements may imply navigating to the target item, exploring the desired features and eventually backtracking to the list. A better solution is to provide a summary of the target item (including some possible actions), for example by means of the hovering details shown in Figure 1.b.

Mechanics: The first step of this refactoring involves making the corresponding interface object sensible to the "mouseover" event. Then, the second step is to associate to the mouse over event the action of displaying some interface objects corresponding to the link target (which might contain operations as shown in Figure 1.b). These objects disappear from the perception space when the mouse is over another sensible area or clicked on another interface object.

As an example, we show in Figure 7 the template for introducing hover details into a generic ADV implementing the Anchor interface. We also show in Figure 8, the Preview ADV, which defines the interface of the anticipated target. Using the integration specification described below, we intend decorating the Product ADV, by putting it inside a Hover Details ADV template. This change will also impact on the Recommendation interface, by incorporating a summary of the target of the index items as hover details.

IntegrationFor HoverDetails
Target Product ADV
ReplaceWith HoverDetails<Product ADV, Preview>

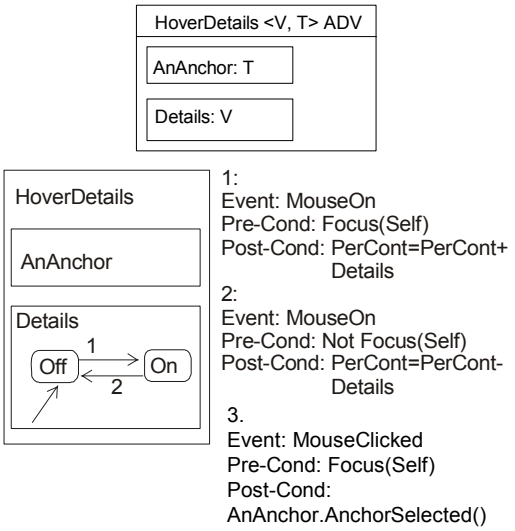


Figure 7: Introducing Hover Details

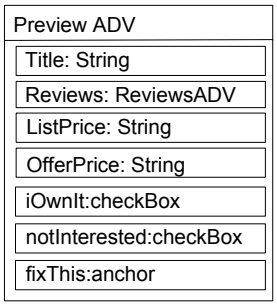
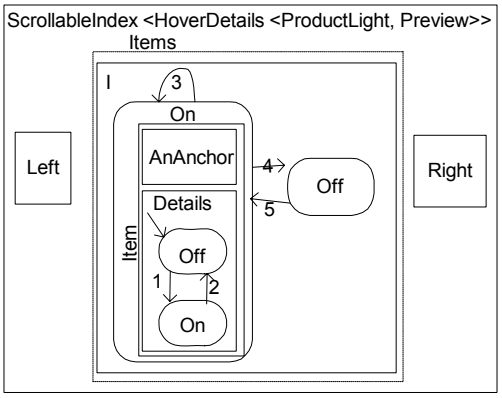


Figure 8: ADV for the Target Anticipation

Figure 9 shows the composed ADV-Chart, which represents the user interface of Figure 1.b. The Details ADV and Events 1' and 2' are incorporated as the result of the previous transformation. The rest of the event's behaviors remains equals to the original.

Note that this refactoring is not removing operations from the interface, but just may be reducing the number of steps necessary to reach them.



0. Event: MouseClicked.... // Same events
...
...
...
1':
Event: MouseOn
Pre-Cond: Focus(Items[i].Item)
Post-Cond: PerCont = PerCont + Items[i].Item.Details
2':
Event: MouseOn
Pre-Cond: Not Focus(Item[i].Item)
Post-Cond: PerCont = PerCont - Items[i].Item.Details

Figure 9: Resulting Composed ADV

3.5 Discussion

The novel contributions discussed in the previous section can be summarized as:

- An evolutionary improvement process based on refactorings;
- A compositional approach for dealing with these improvements in a model-driven way.

The first contribution is an adaptation of well-known software engineering practices to deal with evolution (in fact we borrowed the term refactoring from agile approaches).

Regarding the second one, a reader might argue that it strongly depends on a rather proprietary notation (ADVs) and as such difficult to be universally adopted. It is true that for completely profiting from the approach it is necessary to use this formalism (as with other model-driven approaches) and that existing Web applications might have been designed with others approaches. However, our view goes beyond this particular notation; specifically it is possible to subscribe to these ideas by "just" relying on the transformational approach at a lower level of abstraction, i.e., dealing with HTML/XML code. To show the feasibility of applying refactorings as XML transformations we devote the following sub-section to briefly show how to map the previous ideas to an implementation setting.

3.6 Mapping to Running Applications

As said above, even though our main concern is to focus on design issues, we briefly show in this section that our approach is suitable for obtaining running RIAs.

ADVs can be mapped in a straightforward way onto concrete interface implementations, which support event-driven actions such as HTML/JavaScript (AJAX), GWT, XUL, etc. To make the discussion concrete, we show how to map our abstract components into HTML/JavaScript ones. Different heuristics can be defined for other languages or tools.

We suppose that the source ADVs have been already mapped onto HTML documents and that their behavior has been coded using JavaScript.

The first step is to code each component of the template, namely ADVs and ADV-Charts, using HTML/Javascript inside an XSL document with a template style, which allows referencing the parameters for instantiating the refactoring. At instantiation time, the XSL engine replaces the parameters' references with the corresponding instantiation values.

Next, as we have discussed in [18], integration specifications can be mapped into XSL Transformations [20]. These transformations are capable of inserting, deleting or replacing fragments of code belonging to the user interface and implemented with XML-compliant languages like HTML, JSP, JSF, XSL, etc. Using XSL transformations, rich behavior can be incorporated in the interface by inserting blocks of JavaScript functions.

Once we have specified the ADV templates, or we have got the templates from a catalogue, the refactoring process can be automatized; templates are instantiated for each refactoring declaration. The instantiation mainly gets a template, replaces its hooks with real components, and finally merges the resultant block of code with the target source code of the refactoring declaration.

In some cases, existing interface behavior is overridden due to some interface enhancement, such as intercepting a mouse click for popping up a banner. We profit from a JavaScript facility that allows to redefine functions at runtime and to wrap one function into another [1].

Next we are going to refactor a simple index (like presented at figure 1.a) into a new scrollable index. In Figure 10, we show the original code block corresponding to the index in Figure 1.a, developed

using Struts2. We present it in a simplified way to concentrate on the changes. In Figure 11, meanwhile, we show the specification of the scrollable index template of Figure 1.b.

```
<html>…
<body>…
<table id=”index”>
<s:iterator value="index" status="status"><tr>
   <td><s:property value="top.name" /></td>
   <td><s:property value="top.price" /></td>
   <td><s:property value="top.imageName"/></td>
</tr></s:iterator>
</table>
<body></html>
```

Figure 10: Original code of Index HTML

The template implementation comprises three main generic components: Scrollable Index ADV XSL document, an item view ADV implementation adapter, and the integration specification mapped to a XSL document.

In order to enrich the conventional index in Figure 1.a, we will use the scrollable index's toolkit [24] as part of template implementation which, after its instantiation, will give as a result a RIA index such as the one in Figure 1.b.

The first template component is the index item ADV adapter (named ItemViewAdapter and coded into ItemViewAdapter.xsl file); its implementation just contains few lines that render a ProductADV by taking the ADV implementation from the target code; it then applies low level changes for a correct fit. In this example, the adapter transforms a HTML table-based index item to an unordered list item (using UL and IL HTML Tags), which is required by the scrollable index artifact. This low level code adaptation may not be needed and it just can be an idempotent function.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="…">
<xsl:template name="scrollableIndex" >
<!-- Right Button component definition -->
<table>
<s:iterator value="index" status="status">
    <xsl:call-template name="ItemViewAdapter" />
</s:iterator>
</table>
<!-- Left Button component definition -->
<!-- Carrousel initialization -->
</xsl:template>
</xsl:stylesheet>
```

Figure 11: Scrollable Index ADV's template

The Next step in the implementation of the template is the coding of Scrollable index ADV skeleton using XSL. The template introduces all the code related to Scrollable index and inserts within the ***Item*** artifact adapter (using xsl:call-template routine) which will render the index item.

This new structure is coded into a file called scrollable.xsl. For a matter of space, part of the code was replaced by XML comments.

The last template component is the integration specification; this implementation is realized, again, using XSL transformations corresponding to the integration specification. The XSL document specifies, using a XPATH expression, that the widget component with `id` equals to `'index'` in the original Index code, will be replaced by the Scrollable Index ADV template wrapping the item view. This specification is shown at Figure 12.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
        <xsl:include href="scrollable.xsl"/>
        <xsl:include
href="ItemViewAdapter.xsl"/>
        <xsl:template match="//*[@id='index']">
        <xsl:call-template
name="scrollableIndex">
        </xsl:call-template>
        </xsl:template>
</xsl:stylesheet>
```

Figure 12: Template's integration implementation

Figure 13 shows the final interface and its runtime composing components: Scrollable index and new item interface.

In order to reuse this refactoring in other context, we need to specify a new integration specification that points out to the new target index in the `match´ attribute of the XSL `template´ tag, and a new ItemViewAdapter for adapting the source code of original index items.

In Figure 12, the XSL engine has a pipeline with two integration specifications, for Scrollable Index and Hover Detail refactorings. Each specification sets corresponding ADV parameters. Taking as input the index code presented in Figure 10, the XSL engine processes the Scrollable index's integration specification (Figure 12), applying over its input the changes proposed by the refactoring (coded in Figure 11). Then it applies the second refactoring by introducing a preview popup into the first transformation result, which ends the refactoring process.

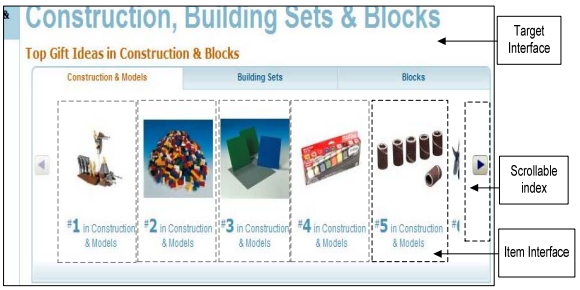


Figure 13 : Resulting woven interface

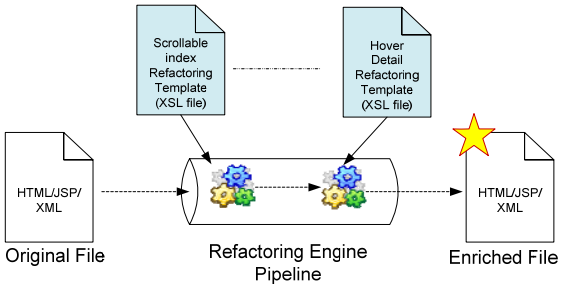


Figure 14: Refactoring process

4. Related Work and Discussion

Refactoring is a rich research subject both in conventional software and also in Web applications [16]. Our view is original as we consider refactoring of navigation and interface models to improve usability; in the case of RIA, our aim is to enhance the interaction facilities by small changes that make evolution seamless.

In [11] the authors present a method for engineering the adaptation of model-based Web 1.0 applications to Web 2.0 UI. The first phase of this model extracts the relevant information from the adapted Web model, to automatically build an initial version of an abstract interface common to all RIA devices. In the next phase, by means of transformation rules, it provides a draft version of the concrete interface for a given set of RIA-capable devices. Finally it automatically provides the final interface based on the RIA technology chosen by the modeler. The RUX-Tool [11] implements the RUXModel method.

In [12] meanwhile, a navigational model of web applications is extracted and then candidate user interface components are identified to be migrated to a single-page AJAX interface. They propose a migration process, consisting of five steps: retrieving pages, navigational path extraction, user interface component model identification, single-page user interface model definition, and target model transformation. This approach is implemented by a tool called RETJAX.

These two approaches imply a complete reengineering of the whole legacy application, while in this paper we propose to apply incremental refactorings at the model-level to transform conventional Web software into RIA. Another original aspect of our approach is that we do not conceive transformations to incorporate new features as editions on a model but rather as compositions with orthogonal features. In this way, not only we can reuse the ADV templates in different applications but we do not pollute the original code making the transformation easily undoable if the new feature will not be finally incorporated.

5. Concluding Remarks and Further Work

We have presented a model-based approach to transform conventional Web applications into RIA in a step by step way. The approach is based on the well-known concept of refactoring applied to Web interface models. We have shown elsewhere [13] that, by applying simple refactorings, it is possible to improve the external quality of a Web application. In particular, in this paper we showed how a single refactoring allows introducing a richer interaction style in a Web application. More complex transformations can be obtained by composing refactorings, e.g. to introduce RIA patterns such as those in [22].

While the refactoring process involves recording and reusing design experience, it also requires human intervention, for example to choose a specific refactoring from a catalogue. However, refactorings can be applied at the modeling level by systematically changing the original model into the refactored one. In this paper, we have shown that by representing the mechanics of refactorings using templates, and using a compositional approach, we can ease the refactoring process. To achieve this objective, we have used our approach for representing RIA interfaces [18], and the compositional approach presented in [8]. We have shown with very simple examples that this process is feasible, though it still requires further research. We are currently working on the following issues:

- Extending the catalogue to include more RIA refactorings.
- Studying relationships among similar refactorings (e.g. Index to Scrollable or to Carrousel) to improve catalogue organization and specification.
- Improving tool support for ADV representation and composition.

- Automatizing the transformation from woven ADVs into running code.
- Analyze the quality of design and implementation artifacts when refactorings are applied in sequence.
- Extending the approach to incorporate more complex navigational refactorings, particularly those which require back-end changes.
- We are working on the adaptation of a more standard notation (such as UML models) for the whole process.

We consider that the current trend to migrate Web applications into RIA deserves further attention. Particularly, the seamless evolutionary style that we are proposing is appropriate to face this situation, since it guarantees that introducing new facilities may be easily de-activated.

10. References

[1] Aspect Oriented Programming and Javascript. In http://www.dotvoid.com view.php?id=43 (2007)

[2] Bozzon, A., Comai, S., Fraternali, P., Toffetti Carughi, G.: Conceptual modeling and code generation for rich internet applications. ICWE 2006: 353-360.

[3] Cowan, D. Pereira de Lucena, C.: Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. IEEE Trans. Software Eng. 21(3): 229-243 (1995).

[4] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns. Elements of reusable object-oriented software" , Addison Wesley (1995).

[6] Garrett, J, "Ajax: A new approach to web applications". Adaptive path, 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[7] Garrido, A., Rossi, G., Distante, D.: Model Refactoring in Web Applications. In Proceedings of the 9th International Symposium on Web Site Evolution (WSE 2007: Oct. 05-06, 2007; Paris, France). Los Alamitos, CA: IEEE Press, 2007.

[8] Ginzburg, J., Rossi, G., Urbieta, M., Distante, D.: Transparent Interface Composition in Web Applications. In Proceedings of the 7th International Conference on Web Engineering (ICWE2007: July 16-20, 2007; Como, Italy), pp. 152-166 (2007).

[9] Gordillo, S., Rossi, G. Moreira, A., Araujo, A.,Vairetti, C., Urbieta, M.: Modeling and Composing Navigational Concerns in Web Applications. Requirements and Design Issues. LA-WEB 2006: 25-31.

[10] Kerievsky, J. Refactoring to Patterns. Addison-Wesley, 2005.

[11] Linaje, M., Preciado, J., Sanchez-Figueroa, F.: "Engineering Rich Internet Application User Interfaces over Legacy Web Models", Internet Computing, IEEE, Volume: 11, Issue: 6, pp. 53-59, Nov.-Dec. 2007.

[12] Mesbah, A., Van Deursen, A. "Migrating Multi-page Web Applications to Single-page AJAX Interfaces," csmr, pp. 181-190, 11th European Conference on Software Maintenance and Reengineering (CSMR'07), 2007.

[13] Olsina, L., Rossi, G., Garrido, A., Distante, D., Canfora, G. "Incremental Quality Improvement in Web Applications Using Web Model Refactoring". 1st Int. Workshop on Web Usability and Accessibility. Nancy, France, December 2007., Springer Verlag, LNCS.

[14] Opdyke, W. Refactoring Object-Oriented Frameworks. Ph.D.Thesis, University of Illinois at Urbana-Champaign, 1992.

[15] Openlaszlo, http://www.openlaszlo.org/

[16] Ricca, F., Tonella, P., Baxter, I.: Restructuring Web Applications via Transformation Rules. SCAM 2001: pp. 150-160, Firenze, Italy, 2001.

[17] Schwabe, D., Rossi, G. "An object-oriented approach to web-based application design". Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, v. 4#4, October, 1998, 207-225.

[18] Urbieta, M., Rossi, G., Ginzburg, J., Schwabe, D.: "Designing the Interface of Rich Internet Applications" Proc. of LA-WEB 07, Santiago, Chile, IEEE Press, 2007.

[19] Van Duyne, D., Landay, J., Hong, J. The Design of Sites. Addison-Wesley, 2003.

[20] XSL. The Extensible Stylesheet Language Family. In http://www.w3.org/Style/XSL/, 2008

[21] http://www.welie.com/patterns/, Welie

[22] http://developer.yahoo.com/ypatterns/, Yahoo! Patterns

[23] Zhang, J., Lin, Y., Gray, J. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In Model-driven Software Development, (S. Beydeda, M. Book, and V. Gruhn, eds.), Springer, Chapter 9, pp. 199-218, 2005.

[24] UI-Prototype: http://www.prototype-ui.com/