Facing Interaction-Rich RIAs: the Orchestration Model

Sandy Pérez, Oscar Díaz
ONEKIN Group, University of the Basque Country
San Sebastián, Spain
sandy-perez@ikasle.ehu.es, oscar.diaz@ehu.es

Santiago Meliá, Jaime Gómez
IWAD Group, University of Alicante
Alicante, Spain
{santi,jgomez}@dlsi.ua.es

Abstract

Promptness, efficiency and stickiness are among the advantages exhibited by the new crop of Rich Internet Applications (RIAs). These advantages came at the cost of increasing the complexity of development. Additionally, the plethora of RIA frameworks can lock this code into a specific platform. This scenario advises for using model-driven development (MDD). This paper focuses on interaction-rich RIAs by addressing two issues: (1) interaction dependencies among widgets, and (2) grouping of widgets into Ajax pages. These concerns are captured in the Orchestration Model. MDD wise, OO-H metamodels accounts for the PIMs whereas Google Web Toolkit is the selected PSM. During transformation, a "message broker" pattern is introduced to decouple widgets from their dependencies. When Ajax pages are generated, heuristics are introduced to find a balance between communication overhead, presentation readiness and maintainability. A running example is used throughout.

1. Introduction

Rich Internet Applications (RIAs) strive to leverage the Web with the engaging interactivity found in traditional desktop applications. Combined with the Software-as-a-Service (SaaS) delivery model, web applications are empowered to compete with desktop interfaces [3]. Moreover, RIAs provide a new client-server architecture that reduces significantly the network traffic using more intelligent asynchronous requests. Faster performance, readiness and engaging interactivity are the hallmarks of this new crop of applications such as Google's GMail, Google's Docs or Yahoo!'s Mail.

The web engineering community is well-aware that RIA development imposes new stringent demands to traditional methods [22]. On one hand, it must introduce/enrich models that account for the new concerns raised by RIAs. On the other hand, the youth of the area advises to use model-driven approaches to abstract away from the plethora of RIA platforms currently competing to find their way. In this context, this paper focuses on the interactivity wealth brought by RIAs. Specifically, we strive to find systematic engineering principles to (1) define interaction dependencies among the GUI widgets and (2) grouping of widgets into Ajax pages. Other main concerns such as data replication [7] are outside the scope of this work.

Interaction Dependencies. Traditional GUI methodologies advise to begin with a mock-up of the interfaces to be presented to the user. IDEs such as Visual Studio and JDeveloper follow such approach providing first a design of the pages which are next enriched with the supported functionality and content. Model-driven web methods tend to use a similar manner where a first draft of the presentation is generated after the navigation model. This draft is then enriched with presentation concerns. However, describing such mock-ups for RIAs is a more demanding process. The wealth of an Ajax page can not be captured by just a GUI mock-up as those generated by Visual Studio and the like.

Grouping of widgets. The page is traditionally both the unit of delivery and the unit of presentation. However, this is no longer so in RIAs. The fat-client approach that characterises RIAs permits a single page to convey the interactivity that would have required multiple pages in a traditional setting. This recommends to distinguish between "*page*" as the unit of delivery from "*screenShot*" as the unit of rendering (i.e. the set of artefacts being simultaneously rendered). This defines a spectrum on the way presentation is delivered. At one end of the spectrum is to equate *page* and *screenShot* (i.e. the traditional approach). The other end is the single-page approach where all *screenShots* are embodied in a unique *page* (e.g. Google's GMail, Reader or Maps follow this pattern). However, something in between is also possible. For instance, the website *www.a9.com* is distributed among distinct Ajax pages.

This paper addresses how existing model-based web methods can account for these two issues. Basically, the *Presentation Model* specifies the application *screenShots*. This model is then complemented with an ***Orchestration***

978-0-7695-3261-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWE.2008.12

IEEE computer society

Model that captures the *interaction dependencies* among presentation widgets. In turn, such relationships guide the designer to distribute the *screenShots* among Ajax pages. Although the approach is illustrated for the OO-H method [13], the insights can be easily extrapolated to other methods.

As an additional contribution, an MDD approach is followed. OO-H metamodels accounts for the Platform Independent Models (PIM) whereas Google Web Toolkit (GWT) [14] is the selected Ajax platform (i.e. the Platform Specific Model or PSM). During transformation, a "message broker" pattern is introduced to decouple widgets from their dependencies and in so doing, bringing the advantages of "separation of concerns". Furthermore, distribution of widgets among Ajax pages is addressed through heuristics that attempt to find a balance between communication overhead, presentation readiness and maintainability. A running example is used throughout.

The paper begins with an outline of the whole process to better frame the role of the *Orchestration Model*.

2. An MDD process for RIAs

The attractive interactions exhibited by RIAs came at the cost of increasing the complexity of development. Furthermore, the plethora of RIA frameworks can lock this code into a specific platform and to make things worse, this framework is likely to evolve due to the youth of the technology. This advocates for MDD.

MDD achieves decoupling by creating distinct (meta) models of a system at different levels of abstraction. Then, transformations are applied that eventually produce code. Hence, direct code programming is substituted by first modelling, next transforming. This permits facing development complexity by distributing concerns among distinct models and different abstraction layers while decoupling from technological subtleties.

MDD focuses on the construction of models, specification of transformation patterns, and automatic generation of code. For our case, the main models include:

- the *Conceptual Model*, which indicates the main entities and relationships found in the application domain. It addresses which data is to be handled.
- the *Navigation Model*, which specifies the data to be presented (as a view on the *Conceptual Model*) and the order in which this data is to be presented. It considers sequencing and data flow.
- the *Presentation Model*, which considers data rendering and layout. It introduces the notion of *widget* and *screenShot*.
- the *Orchestration Model*, which addresses widget coordination. It introduces the notion of *orchestral widget*, *SignalBroadcast*, *SignalHandler* and *page*.
- the *Google Web Toolkit* (GWT), which is a PSM for RIAs. Within this framework, a message-broker architecture is generated through transformation from the previous models.

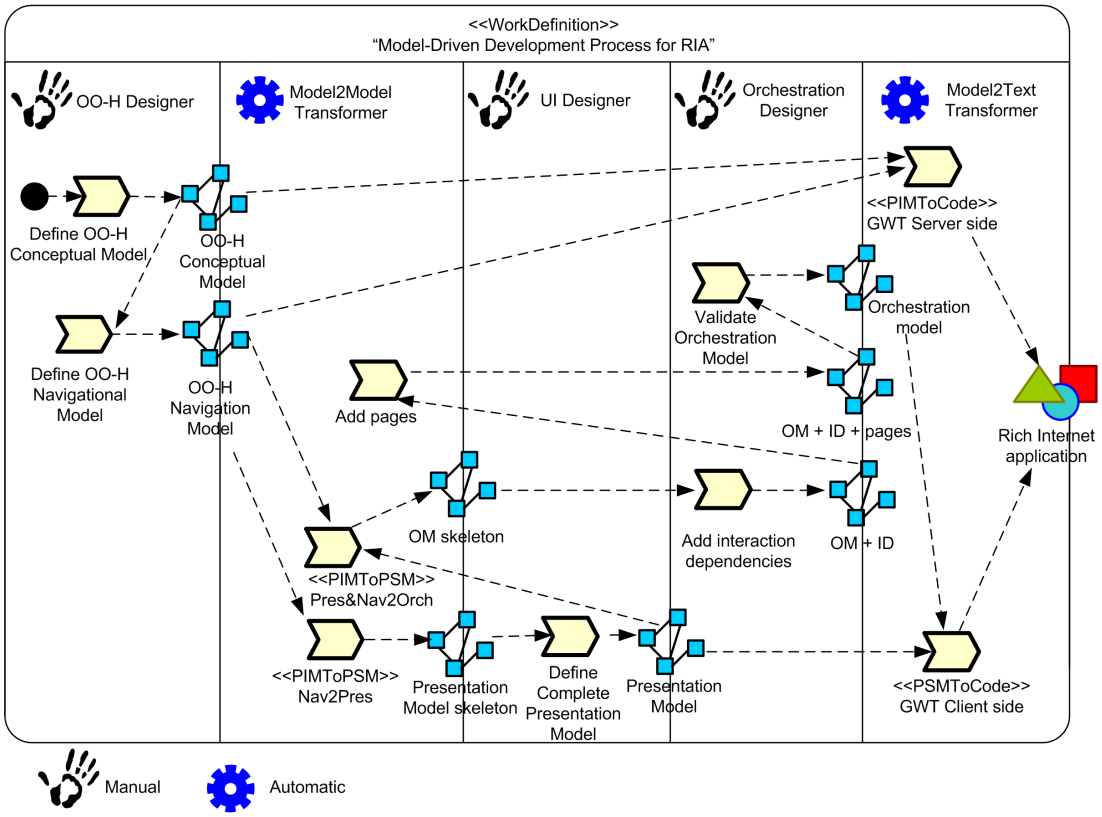
The GWT code is obtained through transformation. Indeed, MDD conceives development as transformation chains where the artefacts that result from each phase must be models. SPEM (Software Process Engineering Metamodel) is a notation for defining processes and their components whose constructs are described in UML notation [19]. Hereafter, SPEM terminology is used to specify the milestone, roles and dataflow that go with producing a RIA using OO-H. Stereotypes are introduced to account for MDD specificities. Specifically,

- *actor* stereotypes are introduced to account for the manner a transformation can be conducted: automatic (<<*gear*>> stereotype) or manual (<<*hand*>> stereotype),
- *activity* stereotypes are supported to model different types of transformations: *PIMToPIM*, *PIMToPSM*, *PIMToCode*, *PSMToCode*, etc.

Figure 1 outlines the OO-H process. First, the *OO-H Designer* defines the *Conceptual Model* which serves as input to obtain the *Navigation Model*. The latter in turn serves to generate a first skeleton of the *Presentation Model*. This draft is then completed by the *UI Designer* using VisualStudio-like tooling (i.e. conceiving the page as a *screenShot*).

RIAs detach the unit of delivery (i.e. a page as the result of an HTTP request with an addressable URL) from the unit of presentation (i.e. a set of widgets simultaneously available or *screenShot*). In so doing, allows different ways of distributing the widgets into a single page or different pages. Some Ajax applications have only one page (e.g. Google's Mail) whereas other Ajax applications go for several pages (e.g. *a9.com*). However, additional information about widget dependencies is required to assess how *screenShots* will be finally grouped into pages. This is the matter of the *Orchestration Model*.

The *Orchestration Model* captures *interaction dependencies* among presentation widgets. Using state machines, widgets are modelled as states. These widgets can react to events raised directly by the user. Additionally, some widgets can be affected by operations conducted in other widgets leading to *interaction dependencies*. This provides the desktop-like interactivity that characterises most RIA. Notice however, that the *Orchestration Model* is not defined



<<WorkDefinition>>
"Model-Driven Development Process for RIA"
OO-H Designer
Model2Model Transformer
UI Designer
Orchestration Designer
Model2Text Transformer
Define OO-H Conceptual Model
OO-H Conceptual Model
Define OO-H Navigational Model
OO-H Navigation Model
Add pages
<<PIMToCode>> GWT Server side
Validate Orchestration Model
Orchestration model
OM + ID + pages
Rich Internet application
OM skeleton
Add interaction dependencies
OM + ID
<<PIMToPSM>> Pres&Nav2Orch
<<PIMToPSM>> Nav2Pres
Presentation Model skeleton
Define Complete Presentation Model
Presentation Model
<<PSMToCode>> GWT Client side
Manual
Automatic

Figure 1. An MDD Process for RIA (OM stands for *Orchestration Model*).

from scratch (see Figure 1). A transformation model-to-model (i.e. *Pres&Nav2Orch*) obtains a first skeleton which is later enriched by the *Orchestration Designer* with the *interaction dependencies*.

Once *interaction dependencies* are manually specified, an automatic transformation takes this information to generate a new *Orchestration Model* but now, enhanced with pages. The tag value "*pageTag*" is used to mark those *screenShot* states that are going to be packaged into a single Ajax page. The transformer provides a first packaging that is later validated/extended by the *Orchestration Designer*.

The validation of the *Orchestration Model* completes the specification of all models. Now, a model-to-code transformation obtains a first skeleton of the GWT project. Unfortunately, the whole application can not be fully obtained from the models. Specifically, widget dependencies require events to be propagated through signals. This implies a mapping between event parameters and signal parameters that we could not derive automatically from the models, and needs to be provided by the user. This mapping is supported as a transformation parameter so that it can be reused in posterior transformations. Next sections introduce the details with the help of a sample case.

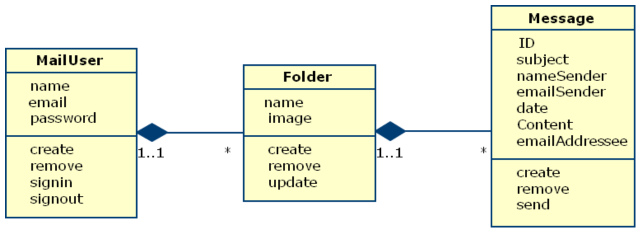


Figure 2. *Mail*'s *Conceptual Model*

3. A running example

A common example of interaction-rich RIA is the *Mail* application. For completeness sake, next paragraphs introduce the main models for this case. OO-H notation is used although the insights can be easily extrapolated to other methods.

Conceptual Model. Figure 2 depicts the *Conceptual Model* for our sample case. This model represents the domain entities and their relationships, free from any technical or implementation details. *MailUser* and *Message* capture data about users and their e-mail messages, respectively. The e-mails are stored in *Folders* which permits to classify

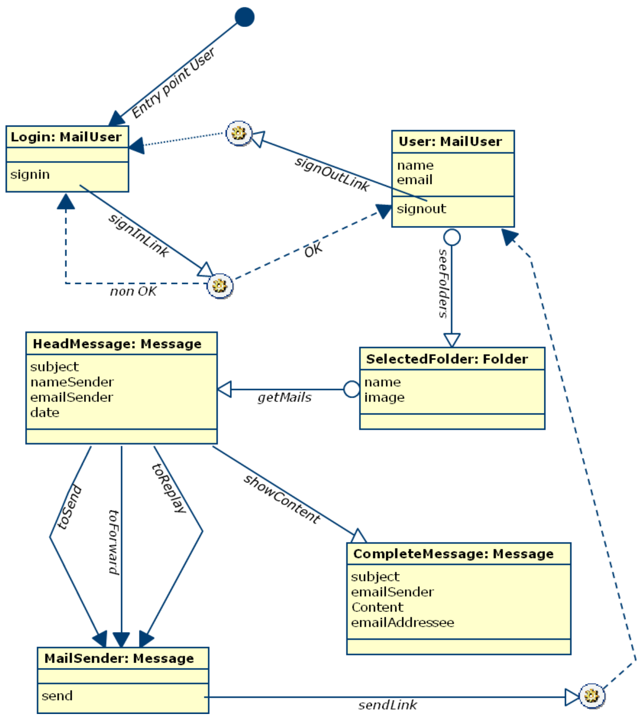


Figure 3. *Mail*'s *Navigation Model*.

the e-mails according to an established criterion (e.g. received e-mails are stored in the "inbox" folder, sent e-mails in the "sent" folder, etc.).

Navigation Model. This model establishes the most relevant semantic paths along the domain space providing views upon the *Conceptual Model*. Figure 3 depicts the *Navigation Model* for our sample case. The navigation starts with the *Login navigationalClass* where *MailUser* verifies his credentials by enacting the *signInLink* link (called *serviceAssociation* link in OO-H parlance). If satisfied, the user moves to a different context where the user is presented with data required to send e-mails. This data is gradually presented through the so-called *navigationalClass*. First, the *User navigationalClass* provides the name and e-mail of the *MailUser* and shows him automatically a set of his own *Folders* (*SelectedFolder*). A specific folder can now be selected to obtain the content of this folder (i.e. *HeadMessage navigationalClass*). By selecting a *HeadMessage*, the user moves to *CompleteMessage*.

Presentation Model. The *Presentation Model* is made up of a set of *screenShots*. A *screenShot* is used like a container that allows the *UI Designer* to realize a spatial distribution of different widgets rendered at a given moment. There are two kinds of widgets: (1) simple widgets (e.g. *Button*, *TextBox*, *Label*, etc.) and (2) containers that can contain other widgets (e.g. *Tree*, *Panel*, etc.). Figure 4 depicts the *MailReader screenShot* for our sample case. A root panel is divided into three areas: NORTH that contains the heading; WEST, which keeps the menu; CENTER, which holds the widgets for e-mail handling[1].

The *Presentation Model* is just a static representation of the widgets as structural components of a page. But widgets do not only render content, they can also have an interactive side. Indeed, this is one of the added-value of RIAs. Next section delves into the details.

4. The Orchestration Model

The *Orchestration Model* captures *interaction dependencies* among widgets. However, not all widgets are liable to be orchestrated. Widgets can play different roles in the *Presentation Model*. Some widgets just render some static content (e.g. on the right upper corner of the figure 4, widgets that show the user's name and e-mail), other widgets can realize navigation (e.g. on the right upper corner of the figure 4, the *Hyperlink* widget with the text *"Sign Out"*). Here, the focus is on widgets that support a functional unit of interaction (e.g. displaying e-mails) liable to be orchestrated with other units. An "***orchestral widget***" provides a unit of interaction with the user to achieve a meaningful task (e.g. sending an e-mail). *Orchestral widgets* are the subject matter of the *Orchestration Model*.

RIA applications are characterised by a rich-interaction setting that surpasses the boundaries of a single widget to spread along distinct widgets. This allows widgets to react to actions not directly related to a user interaction with this widget. Interactions on a widget *A* can also affect a nearby widget *B*. Indeed, inter-widget interactions embody important interaction patterns about how widgets can be synchronized.

The "separation of concerns" motto advises orchestration to be modelled and supported independently of the widgets themselves: widgets are seen as black-box components where an orchestration template is superimposed to provide additional interactivity. Orchestration is then an orthogonal concern from widgets themselves: distinct orchestration templates can be provided from the very same *Presentation Model* (where widgets are defined).

Orchestration is modelled through UML behavioural state machines [20]. This comes as no surprise since the use of state machines (and their neighbours, statecharts) is

[1]On the top side of CENTER, a *CustomWidget* called *NavigationalGrid*, shows the message headings using a paging mechanism. Properties of this widgets accounts for paging data (i.e. size, previous and next). The *NavigationalGrid* is associated with the *HeadMessage navigationalClass*, and supports message retrieval and displayal of the fields for each head message (i.e. *nameSender*, *emailSender* and *subject*). On the bottom side, two *HTML* widgets show the content of the selected e-mail being associated with a *CompleteMessage navigationalClass*: the widget above shows the heading of the selected message while the widget below shows the content.

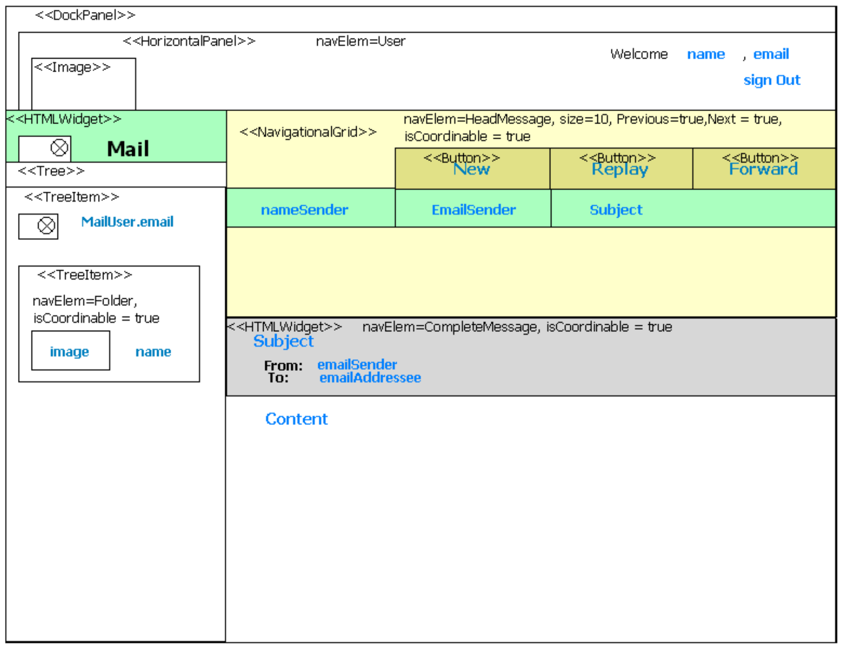


Figure 4. *Mail*'s *Presentation Model*.

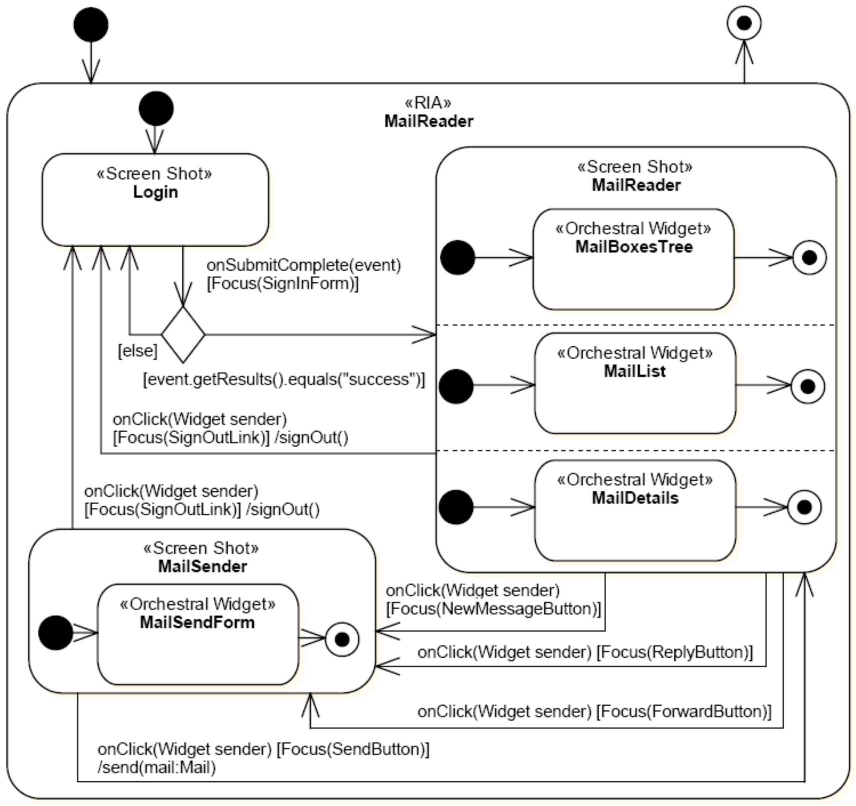

«RIA» MailReader
«Screen Shot» Login
onSubmitComplete(event) [Focus(SignInForm)]
[else]
[event.getResults().equals("success")]
onClick(Widget sender) [Focus(SignOutLink)] /signOut()
onClick(Widget sender) [Focus(SignOutLink)] /signOut()
«Screen Shot» MailReader
«Orchestral Widget» MailBoxesTree
«Orchestral Widget» MailList
«Orchestral Widget» MailDetails
«Screen Shot» MailSender
«Orchestral Widget» MailSendForm
onClick(Widget sender) [Focus(NewMessageButton)]
onClick(Widget sender) [Focus(ReplyButton)]
onClick(Widget sender) [Focus(ForwardButton)]
onClick(Widget sender) [Focus(SendButton)] /send(mail:Mail)

Figure 5. *Mail*'s *Orchestration Model*: **first skeleton.**

common for modelling hypermedia applications [10], web service composition [4, 8] and reactive systems.

Figure 5 shows a skeleton[2] *Orchestration Model* as generated from the *Presentation Model* in Figure 4. The application is represented as a non-orthogonal composite state stereotyped as < <*RIA*> > (i.e. the root of the statechart). Every *screenShot* derived from the *Presentation Model* is modelled through a state stereotype: < <*Screen Shot*> >. If the page contains no *orchestral widget* then, its state counterpart will be atomic (e.g. *Login* page). Otherwise, an AND state is defined that encompasses a state for each *orchestral widget* which can be simultaneously rendered in this state (e.g. *MailReader* state). Notice that widgets other than orchestral do not have a counterpart in this model.

This skeleton is then enriched with interactive dependencies by the *Orchestration Designer*. As example, consider the following dependencies:

1. if a box in the *MailBoxesTree* is selected then, the *MailList* should be refreshed with the e-mails of the selected box;
2. if an e-mail in the *MailList* is selected then, the *MailDetails* widgets should be refreshed with the contents of this e-mail, provided the e-mail is not spam. Otherwise, a confirmation from the user is required;
3. if an e-mail in the *MailList* is selected then, the *MailSentForm* widget should be pre-loaded with this e-mail's content.

As this example highlights, now widgets not only react to direct user interactions but they can also be reactive to events raised by other widgets.

To capture this functionality, "*orchestral transitions*" and "*orchestral states*" are introduced. Transitions are described as "*event[condition]/action*". To denote *orchestral transitions*, two transition stereotypes are introduced:

- **< <SignalBroadcast> >**. Rationale: for disseminating state changes to interested states (i.e. widgets). This leads to two types of events based on the event source, namely, user events (i.e. those fired by the user while interacting), and signals, system events used to communicate state changes. A *SignalBroadcast* transition is triggered by a user event. However, the associated action stands for a *BroadcastSignalAction* [20] (i.e. an action that raises a signal) where the name and the form of the signature determine the signal to be fired by the action.
- **< <SignalHandler> >**. Rationale: for a state to capture signals. The difference with traditional transitions rests on the event that now stands for a signal.

[2]The initial pseudostate and the final state are introduced in order to be UML-compliant.

The *Orchestration Designer* can now use these stereotypes to capture *interaction dependencies*. Figure 6 depicts this situation for our sample case. The first dependency makes *MailBoxesTree* and *MailList* to work in sync: if a box in the *MailBoxesTree* is selected then, the *MailList* should be refreshed with the e-mails of the selected box. This is achieved through two *orchestral transitions*:

- the transition that outgoes the *MailBoxTree* widget broadcasts the *onSelectBox* signal when the *onTreeItemSelected* event occurs;
- the transition that outgoes the *MailList* widget is enacted by this signal which causes the display of the e-mails of the selected mail box (through the *getMails* action).

However, this synchronization is not always automatic but additional user interaction can be required for orchestration sake. This leads to ***orchestral states*** (i.e. states whose rationales rest on some orchestrational purposes). In some case, these states prompt the user for his acknowledgment. In other cases, the user needs to resolve parameter mismatches between the broadcast signal and the triggered action (e.g. if the parameter of the signal is a set, and the receiving action is a singleton). These situations are very common so that it pays off the introduction of the following stereotypes:

- **confirm**, which serves to verify acceptance from the user (e.g. supported as a *Confirm* JavaScript dialog box).
- **alert**, of help when we want to make sure that the information is passed to the user (e.g. realised as an *Alert* JavaScript dialog box).
- **prompt**, which is used when a user's input value is required (e.g. implemented as a *Prompt* JavaScript dialog box).
- **selectFromRange**, of use when there are several options and only one is permitted (e.g. embodied as a *GWT DialogBox*).

Figure 6 resort to the < <*confirm*> > state to describe our second dependency: if an e-mail in the *MailList* is selected then, the *MailDetails* widgets should be refreshed with the contents of this e-mail, provided the e-mail is not spam; otherwise, a confirmation from the user is required. The latter is modelled through the < <*confirm*> > pseudostate: a choice vertex which, when reached, result in the dynamic evaluation of the guards of its outgoing transitions.

Interaction dependencies are not restricted to be between widgets which are simultaneously rendered in the same *screenShot*. This is the case of our last example: if an e-mail in the *MailList* is selected then, the *MailSentForm* widget

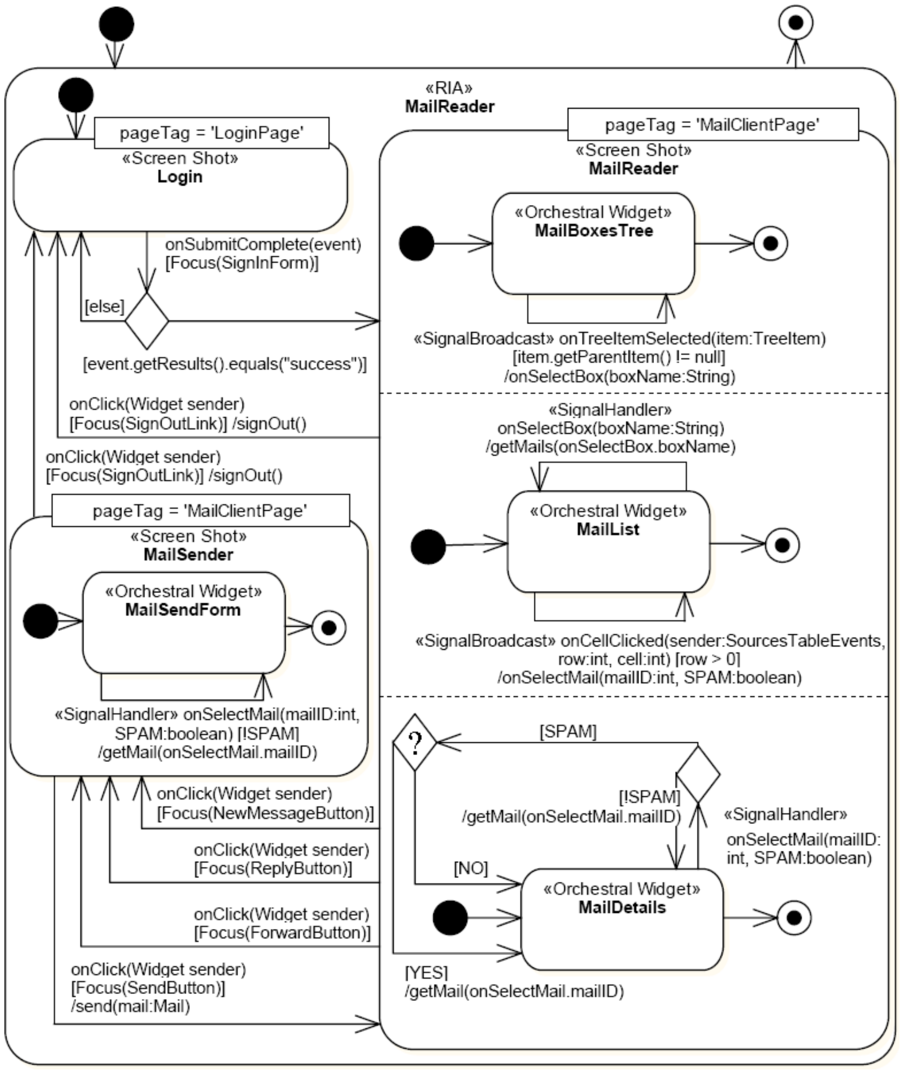

«RIA»
MailReader
pageTag = 'LoginPage'
«Screen Shot»
Login
pageTag = 'MailClientPage'
«Screen Shot»
MailReader
«Orchestral Widget»
MailBoxesTree
onSubmitComplete(event)
[Focus(SignInForm)]
[else]
[event.getResults().equals("success")]
«SignalBroadcast» onTreeItemSelected(item:TreeItem)
[item.getParentItem() != null]
/onSelectBox(boxName:String)
onClick(Widget sender)
[Focus(SignOutLink)] /signOut()
«SignalHandler»
onSelectBox(boxName:String)
/getMails(onSelectBox.boxName)
onClick(Widget sender)
[Focus(SignOutLink)] /signOut()
«Orchestral Widget»
MailList
pageTag = 'MailClientPage'
«Screen Shot»
MailSender
«Orchestral Widget»
MailSendForm
«SignalBroadcast» onCellClicked(sender:SourcesTableEvents,
row:int, cell:int) [row > 0]
/onSelectMail(mailID:int, SPAM:boolean)
«SignalHandler» onSelectMail(mailID:int,
SPAM:boolean) [!SPAM]
/getMail(onSelectMail.mailID)
[SPAM]
?
onClick(Widget sender)
[Focus(NewMessageButton)]
[!SPAM]
/getMail(onSelectMail.mailID)
«SignalHandler»
onSelectMail(mailID:
int, SPAM:boolean)
onClick(Widget sender)
[Focus(ReplyButton)]
[NO]
«Orchestral Widget»
MailDetails
onClick(Widget sender)
[Focus(ForwardButton)]
onClick(Widget sender)
[Focus(SendButton)]
/send(mail:Mail)
[YES]
/getMail(onSelectMail.mailID)

Figure 6. Enhanced *Orchestration Model*: *interaction dependencies* **are provided by the user &** *pageTag* **values are heuristically suggested by the transformer.**

should be pre-loaded with this e-mail's content. Here, *MailList* and *MailSentForm* are placed into two different *screenShots*. If *MailList* is the one on display, *MailSentForm* will keep receiving *MailList*'s signals even if hidden. When *MailSentForm*'s *screenShot* turns on display, the fields of its form will be already filled up with the contents of the last e-mail being selected by *MailList*.

This latter mechanism allows for asynchronous parameter passing between widgets. Most web methods (e.g. WebML or OO-H) allow for synchronous parameter passing as navigational association. By contrast, *interaction dependencies* allow for asynchronous passing of parameters while interacting with the current *screenShot*. When moving to the next *screenShot*, one or distinct widgets can already be initialised as a results of previous interactions. This approach decouples parameter passing from *screenShot* navigation, and in our opinion, it is a main departure from traditional web applications.

At this stage, *screenShots* and *interaction dependencies* are defined. Next issue addresses grouping of *screenShots* into pages.

4.1. How many Ajax pages?

RIAs detach the unit of delivery from the unit of presentation (i.e. a set of widgets simultaneously available). In so doing, allows different ways of distributing the widgets into pages. Some Ajax applications have only one page (e.g. Google's Mail) whereas other Ajax applications go for several pages (e.g. *a9.com*). This section delves into the criteria that can help the designer in this decision.

A first criterion is to minimize HTTP round-trips. From this perspective, the fewer the pages, the better. Additionally, having the whole application is a single page will certainly speed up the enactment of the *interaction dependencies* so that events from one widget will be readily account for by the dependant widget. This accounts for readiness as all interactions are controlled at the client. But it is not only a matter of promptness. As stated in [1] "*this kind of technique isn't just pure eye-candy, there has been a fair bit of research to demonstrate that users perceive the wait time as the time when nothing is happening on the screen, some simple animation can make the web application feel faster even when it's not*". Improving the user experience is the final goal.

Therefore, a first heuristic is:

> If an *interaction dependency* exists between widgets W1 and W2 then, affected *screenShots* are candidates to be enclosed in the same page.

When widgets are heavily coupled through *interaction dependencies* (and this tends to be the case) such heuristic will lead to the single-page pattern. Most of the Ajax designs coming out of Google (GMail, Reader, Maps) are single-page applications. Some implications follow:

- only one URL for the whole application. In Ajax, most server communication occurs through *XMLHttpRequest* which do not affect the page URL. Hence, the application sticks with the same URL no matter how many transfers occur. The problem is that URLs are a main mechanism for locating and bookmarking application functionality and content which is becoming even more important with the advent of social bookmarking sites such as *del.icio.us*. Additionally, the *Back Button* and history mechanism of browsers is based on URL. Changing page content through *XMLHttpRequest* will provide the new content but will not allow the user to go back through the *Back Button* which is contra-intuitive for many users. This means no *Back Button* and bookmarking facilities [18].

- page size increases. Coarse-grained widgets could affect page downloading time. This can not be an issue if the cost of widget fetching is spread across user-time (using asynchronous request). However, this increases the complexity of the application. Similar remark can be made for handling user permission across different part of the application. As this control is now achieved through JavaScript, it is open to malicious hacking. While you can always come up with a way to defeat such attacks, it increases the complexity of the scripting.

The URL issue is partially solved in some advanced frameworks. For instance, GWT provides a history mechanism. The designer defines "major state changes" as history tokens. When the state is reached, the corresponding token is introduced. When the *Back Button* is clicked on, the application uses the history to recover the previous state. Notice that this still requires for the designer to define the "major state changes". This solution allows for finer-grained bookmarking but search engines still fail to properly index the URL so generated[3].

In summary, the advantages of the single-page pattern are clear. However, stubbornly sticking with this pattern can lead to convoluted solutions difficult to develop and maintain. Although frameworks such as GWT permit to use Java (and the associated tooling for testing, documenting, etc) rather than cryptic JavaScript for client-side development, similar principles used for Java development should hold here. And a main principle is modularity.

Hence, page-partition guidelines are required that help in finding a right balance. To this end, we introduce the notion of "*weakest link*". Let *S* be a graph whose nodes stands for widgets, and arcs denote *interaction dependencies*. Both nodes and arcs can be weight. The weight of a node is the approximate size of the corresponding widget (e.g. video or images weight more than raw html). The weight of an arc is an estimate of the occurrence frequency of the associated signal (i.e. the higher the frequency, the larger the payoff to have the corresponding widgets simultaneously available in the client without involving an HTTP request). Finally, the weight of a graph is worked out from the weight of its nodes and arcs. The *weakest link* is the one that splits the graph into two subgraphs with similar weight. This process iterates till the weights of all sub-graphs are below a certain threshold set by the designer. If the threshold is high, the algorithm comes up with coarse-grained pages. Finer-grained pages can be obtained by setting the threshold to lower values.

Based on the previous observations, the algorithm comes up with a first partition into pages which is presented to the designer for validation. Our preliminary insights are that it very much depends on the application at hand. In some cases, "fine-grained" pages can cause a minimum delay while easing development and maintainability.

These design heuristics are realised as endogenous transformation rules: the input *Orchestration Model* is enriched with a new tag value: *pageTag* (see Figure 6). The transformer provides a first packaging that is later validated/extended by the *Orchestration Designer*. *ScreenShots* with the same *pageTag* value will belong to the same Ajax

[3][18] suggests the use of a Site Map page that links to all the "major states" (addressable through fragment identifiers, i.e. those optional components of URLs that follows the # character) that want to be indexed with the link text containing suitable description.

page. Otherwise, they are kept in different pages, and hence, moving among them requires a new server request.

The validation of the *Orchestration Model* completes the specification of all models. Next section addresses the generation of code from the *Orchestration Model* using *Google Web Toolkit* (GWT) as the technological platform. The focus is on client-side code.

5. Down to code: generating code for the Google Web Toolkit

The *Orchestration Model* captures interactive richness through *interaction dependencies*. These dependencies have a scope (i.e. the enclosing context that contains the widgets tightened by the dependency). The scope can be "*screenShot*", "*ajaxPage*" and "*application*" based on whether the widgets belong to the same *screenShot*, the same Ajax page or rather, and they are distributed among distinct Ajax pages, respectively.

Different dependency scopes will lead to different coding. This section focuses on GWT code generation for dependencies with scope "*screenShot*" and "*ajaxPage*" which can be handled with client scripting. By contrast, the "*application*" scope also involves server programming, and hence, it is left outside this paper's content. The section begins with a brief on Google Web Toolkit (GWT).

5.1. An outline on GWT

GWT is an open source Java software development framework whose most outstanding feature is that it allows Ajax applications to be thoroughly coded in Java without resorting to JavaScript. When the application is deployed, the GWT compiler translates the Java application to browser-compliant JavaScript and HTML. This is a main departure compared with other Ajax frameworks.

A *GWT application is* enveloped into Java packages where client and server artefacts are kept. We focus on the client side. A *GWT application* has a **configuration file**, *App.gwt.xml*, which is used to define the *entry-point* class, compiler directives, the application module and dependencies with other external modules. The *entry-point* class is executed when the module loads into the browser. Some elements (e.g. html or jsp pages, stylesheets, images, and so on) of a GWT application must be included in the **public** folder and only one of them, an HTML page (*App.html*), is required. This HTML page is in charge of loading and executing the application.

GWT applications arrange Java code into two categories: (1) those that will be compiled into JavaScript to be executed at the **client** side (at least one is required, *App.java*) and (2) those that will be compiled into bytecode to be executed at the **server** side. To this end, a GWT application contains two **scripts**: *App-compile.cmd* and *App-shell.cmd*. The former executes the Java to JavaScript compiler. *App-shell.cmd* launches the application by executing the hosted browser that ships with GWT.

As stated in previous sections, Ajax pages can break the browser's *Back Button*. GWT overcomes this pitfall through the so-called *History* object. This object behaves as a breadcrumb record: when the application reaches a certain state a token is left in the history so that we can emulate the *Back Button* functionality by going backwards through the token history. The history also supports a special widget called *Hyperlink*[4] that permits to move among *screenShots* packaged in the same Ajax page.

5.2. Client-side architecture

This section introduces the main classes and relationships that support the *Orchestration Model* at the client side. The *Orchestration Model* strives to separate widget functionality from widget dependencies so that each concern can be conceived, developed and maintained as separate as possible.

To this end, a first approach would be the use of the *Observer* pattern where the subject (publisher) notifies every observer (subscriber). That is, widgets that publish events must notify all those widgets interested in such events. Such an approach, defeats the transparent principle whereby coordination should go unnoticed for those widgets participating in the coordination. Furthermore, it jeopardizes maintainability as the system would quickly explode into an unmanageable number of cross-widget relationships, resulting in integration spaghetti.

A more decoupled approach is offered by the "message broker" pattern which is widely adopted in EAI (Enterprise Application Integration) for message routing [15]. In this pattern, a central *Message Broker* receives events notifications, determines the correct destinations, and routes the events to the correct destinations.

Figure 7 shows the class diagram for this pattern. Widgets can play two compatible roles: *Subscriber* and *Publisher*. Available GWT widgets have a pre-established set of events they can react to. By introducing the *Subscriber* role, widgets can be notified of events other than those directly derived from user interactions. On the other side, GWT widgets are isolated components where their actions are limited to the widget itself. The role *Publisher* extends this behaviour by allowing widget events to be propagated outside this widget's realm. Additionally, the broker is made unique in the system (along the lines of the *Singleton* pattern [12]) which eliminates the need for subscribers and publishers to have a reference to the broker.

[4]If we want a normal hyperlink to another HTML page then, we must use the *HTML* widget rather than the *Hyperlink* widget.

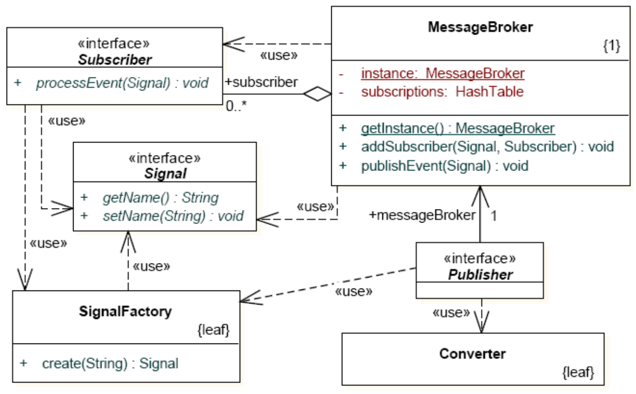


Figure 7. Class diagram for supporting *interaction dependencies* **in GWT**

From a maintainability perspective, the benefits of this pattern includes easy addition/removal of dependencies between widgets, and "separation of concerns" between the functionality of the widget themselves and their *interaction dependencies*.

5.3. From the Orchestration Model to GWT using MOFScript

A web application is embodied as a *GWT application*. This section focuses on the model-to-code transformation that obtains GWT-based Ajax pages from *screenShot* states in the *Orchestration Model*. Table 1 outlines the mapping between constructs of each model.

First, a page (i.e. a compound of *screenShots*) outputs a GWT module. This module encompasses a Java class for each < <*Screen Shot*> >. Figure 8 provides a snippet of the generated code. A class *MailClientPage* is generated from the namesake *pageTag* tag value in the *Orchestration Model*. This class implements the interfaces *EntryPoint* (i.e. making it an Ajax page) and *HistoryListener* (i.e. providing bookmarking facilities). The latter implies that this class recognises different "major state". These states are supported as tokens on the history, and so are they declared (see 8(a)). In this example, *MailClientPage* defines two major states, namely, *MailReader* and *MailSender* that stand for the namesake *screenShots*. This implies that *Back Button* facilities will be provided to go back and for between these two *screenShots*.

Besides widgets and panels, this GWT module also contains a special Java class: the *Message Broker*. This broker regulates the interaction between publishers and subscribers. This role is played by < <*Orchestral Widgets*> >. If an < <*Orchestral Widget*> > has a < <*SignalBroadcast*> > transition then, its Java class counterpart should implement the *Publisher* interface. Likewise, if an < <*Orchestral Widget*> > has a < <*SignalHandler*> > transition then, its Java class counterpart should implement the *Subscriber* interface.

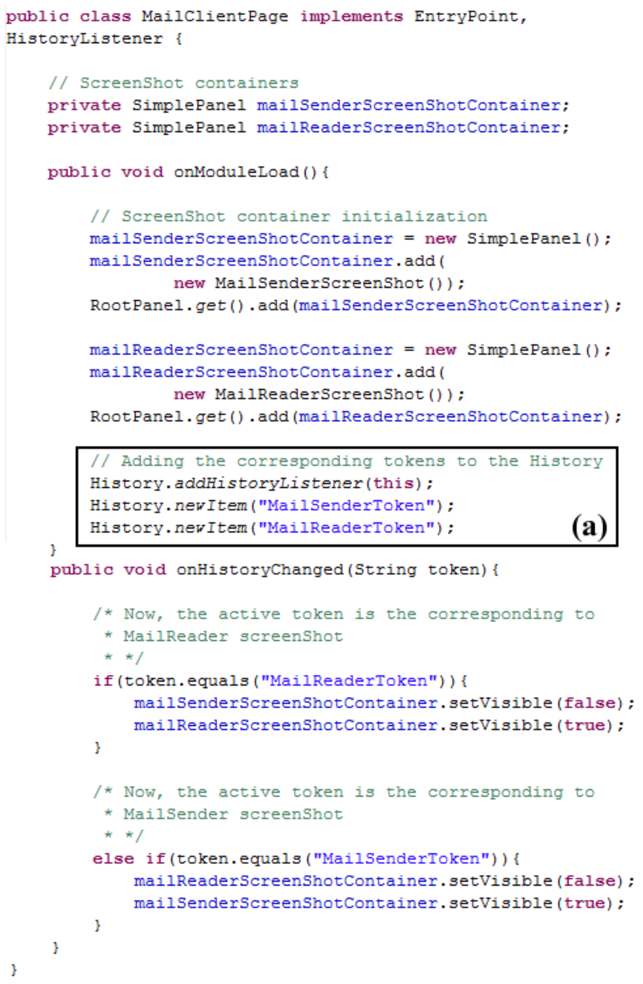

```
public class MailClientPage implements EntryPoint,
HistoryListener {

    // ScreenShot containers
    private SimplePanel mailSenderScreenShotContainer;
    private SimplePanel mailReaderScreenShotContainer;

    public void onModuleLoad(){

        // ScreenShot container initialization
        mailSenderScreenShotContainer = new SimplePanel();
        mailSenderScreenShotContainer.add(
                new MailSenderScreenShot());
        RootPanel.get().add(mailSenderScreenShotContainer);

        mailReaderScreenShotContainer = new SimplePanel();
        mailReaderScreenShotContainer.add(
                new MailReaderScreenShot());
        RootPanel.get().add(mailReaderScreenShotContainer);

        // Adding the corresponding tokens to the History
        History.addHistoryListener(this);
        History.newItem("MailSenderToken");
        History.newItem("MailReaderToken");              (a)
    }
    public void onHistoryChanged(String token){

        /* Now, the active token is the corresponding to
         * MailReader screenShot
         * */
        if(token.equals("MailReaderToken")){
            mailSenderScreenShotContainer.setVisible(false);
            mailReaderScreenShotContainer.setVisible(true);
        }

        /* Now, the active token is the corresponding to
         * MailSender screenShot
         * */
        else if(token.equals("MailSenderToken")){
            mailReaderScreenShotContainer.setVisible(false);
            mailSenderScreenShotContainer.setVisible(true);
        }
    }
}
```

Figure 8. *MailSender* **and** *MailReader screenShots* **from the** *Orchestration Model* **(see figure 6) map into the same Ajax page in the GWT PSM.**

The former case is illustrated for the *MailBoxesTree* widget (see Figure 9). This < <*Orchestral Widget*> > has a < <*SignalBroadcast*> > transition. Hence, the *MailBoxesTree* class implements the *Publisher* interface. A transition is specified as *event[condition]/action*. The event becomes the selector of the method. The condition outputs a condition statement in the method's body. Finally, the action produces the publishing of a namesake signal through the *Message Broker*. Additionally, this action produces a namesake class that implements the *Signal* interface.

The subscriber case is exhibited by the *MailList* state which has a < <*SignalHandler*> > (see Figure 10). Hence, the *MailList* class implements the *Subscriber* interface. The

Orchestration	**GWT**
Page	- A GWT module that contains a set of *screenShots* and has a *MessageBroker*.
ScreenShot	- A simple panel in the module's entry point class, - a token in the History - and a Java class implementing the *screenShot*.
Orchestral Widget	- A Java class implementing the widget that... [if has *SignalBroadcast* transitions] –> implements *Publisher* interface. [if has *SignalHandler* transitions] –> implements *Subscriber* interface.
Transition	- Inter-page navigation: traditional navigational links. - Inter-screenShot navigation inside the same page: the token in the History changes.
SignalBroadcast	- A new Java class is created, if it does not exist, that implements the signal. - The Java class that implements the widget is modified in order to implement the corresponding listeners. - In the corresponding listener's method... - [if the transition has constrains] –> *if* statements are created, - a signal object is created using the *SignalFactory* class, - signal's attributes are given values - and the *MessageBroker* is notified of the signal.
SignalHandler	- In the constructor of the class that implements the widget... - some code is introduced in order to subscribe the widget to the signal. - and the *processEvent* method is added where... - the content of an *if* statement will process the signal.
Choice pseudostate	- In the *processEvent* method, an *if* statement (or *if-else* if there are at least two outgoing transitions) is added.
Confirm	- In the *processEvent* method... - a JavaScript confirm dialogue box is added - as well as an *if-else* statement, where *if* will be executed if user clicks on *Yes* and *else* when user clicks on *No*.
Alert	- In the *processEvent* method, a JavaScript alert dialogue box is added.
Prompt	- In the *processEvent* method, a JavaScript prompt dialogue box is added.
SelectFromRange	- A new Java class is created, which extends the *DialogBox* class. This specialized *DialogBox* will be a modal dialog box and it will allow the user to select one option from a range. - In the *processEvent* method, a call to the created Java class is added.

Table 1. Mapping from the *Orchestration Model* to GWT code.

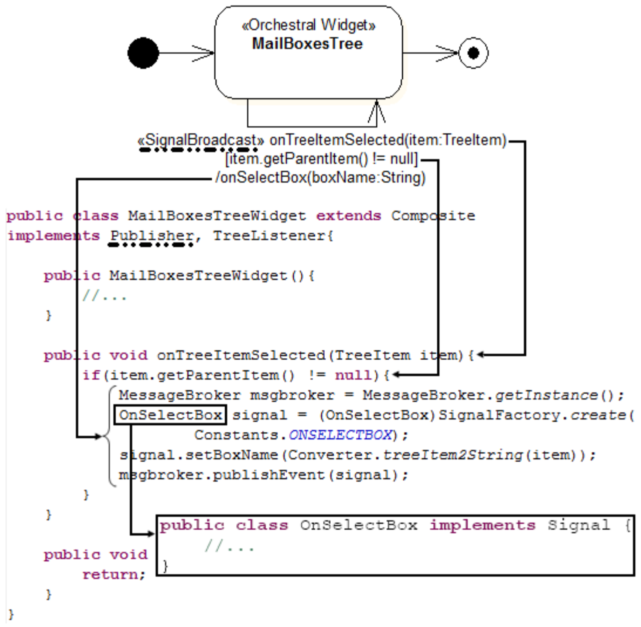
constructor of the class includes some code to subscribe to the corresponding signal. The transition's condition outputs a conditional to be satisfied before processing the signal. Finally, the transition's action is mapped into a call to the business logic.

6. Related work

Web modelling approaches, which were originally intended for traditional web applications, are now being updated to take into account RIAs' specificities. To the best of our knowledge, these efforts have been mainly conducted for WebML and OOHDM (see Table 2).

WebML. This method is mainly focus on data-intensive web applications. It is then a natural evolution for WebML to tackle how data can be distributed between the client and the server in RIAs. This issue is first posed in [5], and next, further developed in [7] where they provide some insights about using a push or a pull approach to keep replicated data in sync.

As for sophisticated interaction modelling, [9] extends WebML's "hypertext model in the small" with the notion of "*computation sequence*". *Computation sequences* play a similar role to our "*interaction dependencies*": keeping "*content units*" (i.e. WebML abstraction for widgets) in sync. *Computation sequences* are associated with links so that when *"a link is navigated, the dynamic model dictates*


«Orchestral Widget»
MailBoxesTree
«SignalBroadcast» onTreeItemSelected(item:TreeItem)
[item.getParentItem() != null]
/onSelectBox(boxName:String)

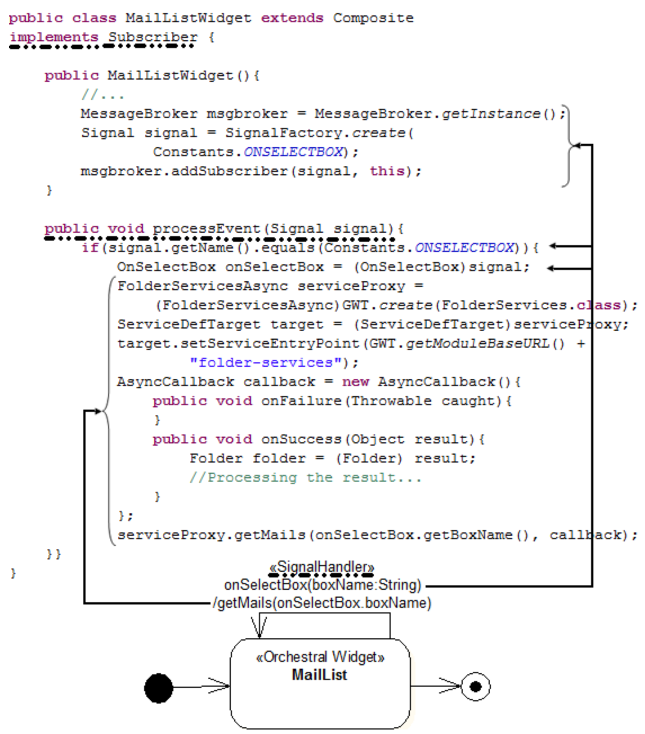
```
public class MailBoxesTreeWidget extends Composite
implements Publisher, TreeListener{

    public MailBoxesTreeWidget(){
        //...
    }

    public void onTreeItemSelected(TreeItem item){
        if(item.getParentItem() != null){
            MessageBroker msgbroker = MessageBroker.getInstance();
            OnSelectBox signal = (OnSelectBox)SignalFactory.create(
                    Constants.ONSELECTBOX);
            signal.setBoxName(Converter.treeItem2String(item));
            msgbroker.publishEvent(signal);
        }
    }

    public void
        return;
    }
}
```

```
public class OnSelectBox implements Signal {
    //...
}
```

Figure 9. *MailBoxesTree* **widget maps into a namesake class which implements the** *Publisher* **interface.**

explicitly the effects on all the components of the page" [9]. The difference stems from the coupling. In *computation sequences*, the triggering widget (i.e. the link) knows about the triggered widgets (i.e. *content units*). By contrast, our approach uses a broker to decouple both concerns. This accounts for maintainability. For instance, if a new widget is to be somehow synchronized, WebML will require to change the involved *computation sequences*. In our approach, this new widget just subscribes to the event. Furthermore, state machines also allows for defining the *interaction dependency* at different level of abstractions. Examples in the paper always have atomic states -that stand for *orchestral widgets*- as the triggering source and, in this way, it is similar to WebML. However, the triggering source can also be an AND state. In this case, the occurrence of the signalling event in any of the enclosed atomic states will trigger the signal.

WebML presentation does not scale to account for RIA rich presentation. Hence, recent work is enhancing WebML hypertext model with RUX [21]. RUX splits presentation into three models: *Abstract*, *Concrete* and *Final Interface* models [17]. *Abstract Interface* is intended to be reusable by all RIA platforms. *Concrete Interface* is split into *Spatial Presentation* (specifies the spatial arrangement and look&feel), *Temporal Presentation* (allows the specification of those behaviours which require a temporal synchronization) and *Interaction Presentation* (models user's



```
public class MailListWidget extends Composite
implements Subscriber {

    public MailListWidget(){
        //...
        MessageBroker msgbroker = MessageBroker.getInstance();
        Signal signal = SignalFactory.create(
                Constants.ONSELECTBOX);
        msgbroker.addSubscriber(signal, this);
    }

    public void processEvent(Signal signal){
        if(signal.getName().equals(Constants.ONSELECTBOX)){
            OnSelectBox onSelectBox = (OnSelectBox)signal;
            FolderServicesAsync serviceProxy =
                (FolderServicesAsync)GWT.create(FolderServices.class);
            ServiceDefTarget target = (ServiceDefTarget)serviceProxy;
            target.setServiceEntryPoint(GWT.getModuleBaseURL() +
                    "folder-services");
            AsyncCallback callback = new AsyncCallback(){
                public void onFailure(Throwable caught){
                }
                public void onSuccess(Object result){
                    Folder folder = (Folder) result;
                    //Processing the result...
                }
            };
            serviceProxy.getMails(onSelectBox.getBoxName(), callback);
    }}
}
```

Figure 10. *MailList* **widget maps into a namesake class which implements the** *Subscriber* **interface.**

interactions). Finally, *Final Interface* describes the final user interface according to RIA selected rendering technology. This approach lets RUX support multi-device RIAs.

RUX focuses on stepwise definition of user interaction. By contrast, our work addresses crosscutting interactions among widgets. Furthermore, RUX models are XML documents where hard-coded generators obtain the application code. Our work stick with MDD practises.

OOHDM. This method proposes the use of *ADVcharts* for modelling widget interaction [24]. *ADVcharts* and statecharts have a similar expressiveness [6], but *ADVcharts* are claimed to be more legible. Urbieta et al. also addresses widget orchestration but here the difference with our approach is on the implementation side. OOHDM takes JavaScript as the PSM, and conceives widget coordination as a crosscut. Hence, an aspect-oriented approach is proposed using [2]. By contrast, our approach realises on the transformer itself to inlay the cross-cutting coordination using a broker. This is a more general approach that using aspectual programming, and permits the use of existing Java tooling for testing and documentation, a not neglectable advantage if we are tackling complex applications.

Outside the web engineering community, HCI work is also of much relevance here. *UsiXML* [16] is a user-

Orchestration in RIA	**OOHDM** [24]	**WebML** [5, 7, 9]	**This paper**
Orchestral transitions	Yes	Yes	Yes
Orchestral state	Yes	No	Yes
Page partitioning	No	No	Yes
Development approach			
MDD-compliant	No	Yes	Yes
Technological Platform	XHTML + JavaScript	Open Laszlo and Flex*	GWT

**Integrated with RUX*

Table 2. Related work

interface description language which is applied to RIA in [23]. This framework structures the development process along four levels of abstraction (captured as XML files): *Task and Concepts*, *Abstract User Interface* (independent of any modality of interaction), *Concrete User Interface* (independent of any computing platform) and *Final User Interface*, which depends on a particular computing platform. Being XML models, moving from one layer to a lower-abstraction layer is achieved through XSLT transformations. Besides the development framework (XML vs. UML/Ecore), the difference rests on the stress given to orchestration. Whereas *UsiXML* captures orchestration somehow as part of the *Concrete/Abstract User Interface*, our claim is that orchestration should be a first concern during RIA design and as such, capture as a separate model.

The use of statecharts is also common in HCI. StateWebCharts (SWC) [25] is a case in point. SWC is mainly used to describe navigation between pages (documents) rather than interaction between widgets (objects). Although SWC supports both server-side and client-side execution, it does not support inter-widget interactions. Additionally, SWC focuses on the design stage, does not address implementation concerns.

UML's state machines are also the selected formalism to support the so-called *Guide Model* in [11]. As a refinement of the Task Model, the *Guide Model* provides navigation and synchronization details on user interaction. This is similar to the aim of our *Orchestration Model*. The difference stems from the mechanism used to achieve such orchestration. *Guide Model* uses "traditional" transitions, whereas the *Orchestration Model* resorts to the event broadcasting mechanism of UML's state machines. Additionally, the *Orchestration Model* views widgets as standalone components. This implies that widget composition can not be taken from granted but additional concerns are raised by gluing widgets together. This is the rational behind the *orchestral states* introduced in the *Orchestration Model*.

7. Conclusions

It is expected that a large number of web applications will exhibit RIA-like features in the near future. This will improve the user experience but it will increase the complexity of development too. Such scenario grounds the need for taking RIA concern in existing web methods.

This paper proposes the introduction of the *Orchestration Model* in existing web methods to face interactive-rich RIA. The use of an MDD approach accounts for facing in a stepwise manner the different issues risen during orchestration, mainly, *interaction dependencies* and *interaction scope*. These decisions are decoupled from the chosen technological platform. As a proof-of-concept, an MDD process is defined using OO-H metamodels as PIMs, and GWT as the PSM. QVT and MOFScript are used as the model-to-model and model-to-code transformation languages.

Future research includes gaining further insights on the partition criteria for page definition that find a balance between promptness and maintainability. Also, we plan to address support for *interaction dependencies* when split into distinct Ajax pages (i.e. the scope of the dependency being "*application*").

8. Acknowledgments

This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contracts TIN2005-05610 (WAPO) and TIN2007-67078 (ESPIA). Perez enjoys a doctoral grant from the Basque Government under the "Researchers Training Program".

References

[1] Ajax: Single-page vs. Multi-page. Published at http://getahead.org/dwr/ajax/single-page-design.

[2] Aspect Oriented Programming and JavaScript. Published at http://www.dotvoid.com/view.php?id=43.

[3] J. Allaire. Macromedia Flash MX-A next-generation rich client. Technical report, Macromedia, March 2002.

[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services that Export their Behavior. In *1st International Conference on Service Oriented Computing*, 2003.

[5] A. Bozzon, S. Comai, P. Fraternali, and G. T. Carughi. Conceptual Modeling and Code Generation for Rich Internet Applications. In *6th International Conference on Web Engineering*, 2006.

[6] L. M. F. Carneiro, D. D. Cowan, and C. J. P. Lucena. ADVcharts: A Visual Formalism for Interactive Systems. *ACM SIGCHI Bulletin*, 26:74–77, 1994.

[7] G. T. Carughi, S. Comai, A. Bozzon, and P. Fraternali. Modeling Distributed Events in Data-Intensive Rich Internet Applications. In *8th International Conference on Web Information Systems Engineering*, 2007.

[8] F. Casati and M. C. Shan. Dynamic and Adaptive Composition of E-Services. *Information Systems*, 26(3):143–163, May 2001.

[9] S. Comai and G. T. Carughi. A Behavioral Model for Rich Internet Applications. In *7th International Conference on Web Engineering*, 2007.

[10] M. C. Ferreira de Oliveira, M. A. Santos Turine, and P. Cesar Masiero. Statechart-based Model for Hypermedia Applications. *ACM Transactions on Information Systems (TOIS)*, 19(1):28–52, January 2001.

[11] P. Dolog and J. Stage. Designing Interaction Spaces for Rich Internet Applications with UML. In *7th International Conference on Web Engineering (ICWE)*, 2007.

[12] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[13] J. Gómez, C. Cachero, and O. Pastor. Conceptual Modeling of Device-Independent Web Applications. *IEEE MultiMedia*, 8(2):26–39, 2001.

[14] Google. Google Web Toolkit (GWT). Published at http://code.google.com/webtoolkit/.

[15] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging*. Addison-Wesley Professional, 2003.

[16] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins, and D. Trevisan. USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In *Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages at the ACM Advanced Visual Interfaces*, 2004.

[17] M. Linaje, J. C. Preciado, and F. Sánchez-Figueroa. A Method for Model Based Design of Rich Internet Application Interactive User Interfaces. In *7th International Conference on Web Engineering*, 2007.

[18] M. Mahemoff. *Ajax Design Patterns*. O'REILLY, 2006.

[19] Object Management Group (OMG). Software Process Engineering Metamodel, version 1.1. Published at http://www.omg.org/docs/formal/05-01-06.pdf, January 2005.

[20] Object Management Group (OMG). Unified Modeling Language: Superstructure (version 2.1.1). Published at http://www.omg.org/docs/formal/07-02-03.pdf, February 2007.

[21] J. C. Preciado, M. Linaje, S. Comai, and F. Sánchez-Figueroa. Designing Rich Internet Applications with Web Engineering Methodologies. In *6th International Conference on Web Engineering*, 2006.

[22] J. C. Preciado, M. Linaje, F. Sánchez, and S. Comai. Necessity of Methodologies to Model Rich Internet Applications. In *7th IEEE International Symposium on Web Site Evolution*, 2005.

[23] F. J. Martínez Ruiz, J. Muñoz Arteaga, J. Vanderdonckt, and J. M. González Calleros. A First Draft of a Model-Driven Method for Designing Graphical User Interfaces of Rich Internet Applications. In *4th Latin American Web Congress (LA-Web)*, 2006.

[24] M. Urbieta, G. Rossi, J. Ginzburg, and D. Schwabe. Designing the Interface of Rich Internet Applications. In *5th Latin American Web Congress*, 2007.

[25] M. Winckler and P. Palanque. StateWebCharts: A Formal Description Technique Dedicated to Navigation Modelling of Web Applications. In *International Workshop on Design, Specification and Verification of Interactive Systems*, 2003.