

# IFML by Example: Modeling GMail

## 1 Introduction

This document exemplifies the modeling constructs and the expressive power of IFML by modeling a popular Rich Internet Application: Gmail ([www.gmail.com](http://www.gmail.com)).

## 2 The Content Model

Gmail is an application for managing mail messages and contacts of users.

A **User** possesses a set of MailBoxes. A **MailBox** (aka System Tag) consists of a set of **MailMessages**, MailMessages are organized not only in MailBoxes but also in user-defined clusters, called **Tags**. Therefore, MailBoxes and Tags can be seen as special cases of a common concept of **MailMessageGroup**. A user can also manage **ChatConversations**, which are composed of **ChatMessages**. A User is also associated with a set of **Contacts**. Contacts are clustered in **ContactGroups**.

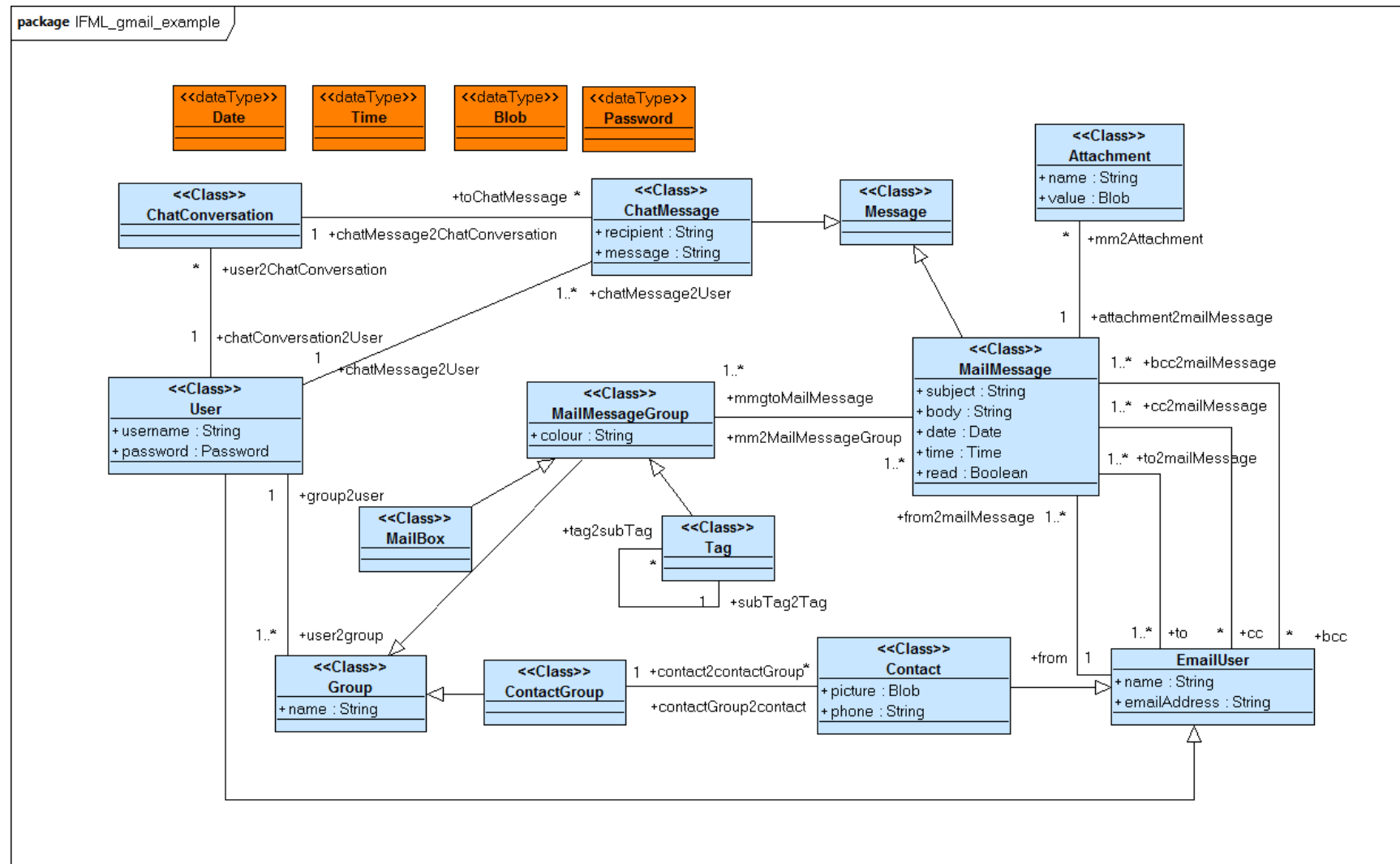


Figure 1: the content model of the GMail application

### 3 Model of the Interface

The Gmail interface consists of a top-level container, which is logically divided into two *alternative* sub-containers one for managing *MailMessages* and one for managing *Contacts*<sup>1</sup>.

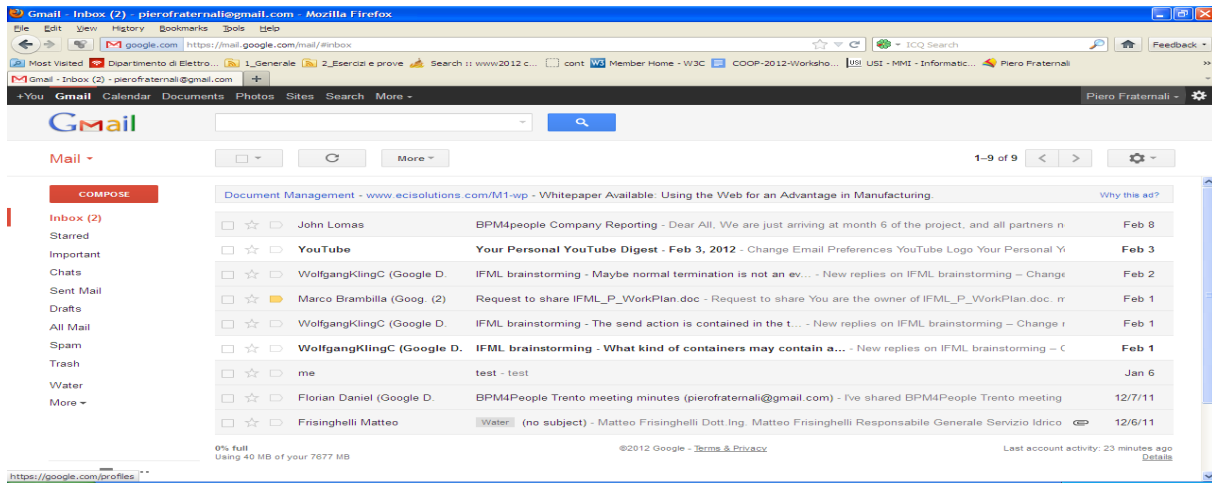


Figure 2: The Gmail view container for MailMessages

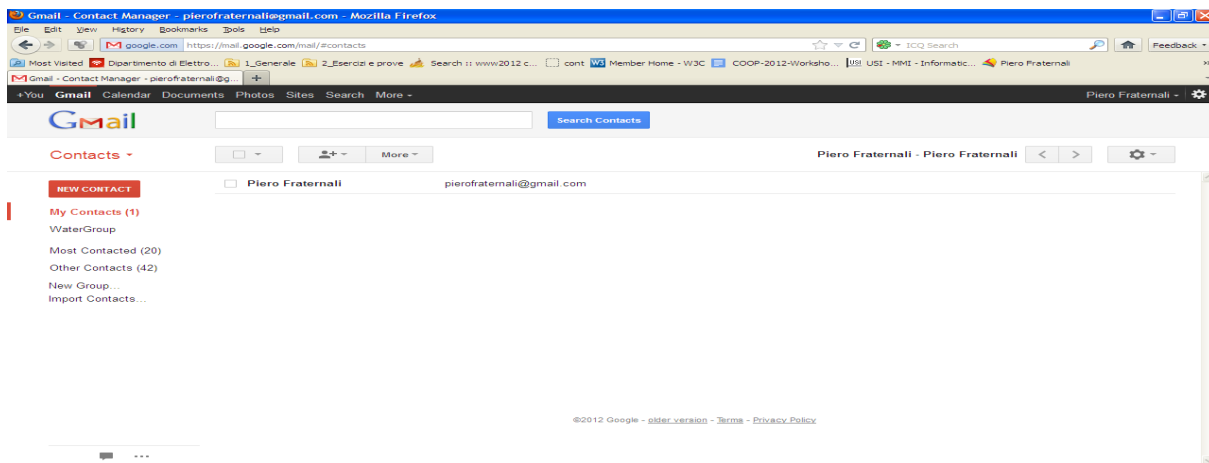


Figure 3: Gmail view container for Contacts

By default, when Gmail is accessed, the container for managing *MailMessages* is presented. At any moment, it is possible to Switch from the *MailMessages* to the *Contacts* view components, by means of a menu, shown in Figure 4.

<sup>1</sup> For simplicity we do not consider the activity management functionality of Gmail.

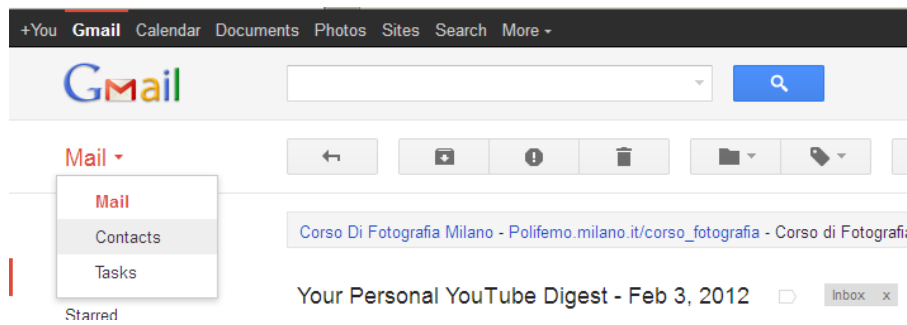


Figure 4: A menu allows one to switch from the MailMessages to the Contacts view components

The model of the top level container of Gmail is shown in Figure 5



Figure 5: IFML model of the Top Container of Gmail.

## Notations

1. The nesting of mutually exclusive view containers into a view container (*isXOR* property equal true) is denoted with a [XOR] icon.
2. The default view container (*isDefault* property equal true) of a set of mutually exclusive view sibling containers is denoted with a [D] icon on container.
3. The global reachability of view container from all the other sibling containers and their children sub-containers is denoted with a [L] (Landmark) icon on container.

## Model usability

- The use of the [L] (Landmark) icon reduces the number of navigation events that need to be explicitly represented (otherwise one event should be necessary in all the view containers from which the target view container is reachable), resulting in simpler models.

The *MailMessages* view container comprises five main nested elements:

- a view component (*MboxList*) showing a list of *MailBoxes* and *Tags*;
- a view container (*MessageSearch*) permitting the user to input search keywords to be matched against the *MailMessages*;
- a *MailBox* view container, permitting one to access the messages of a specific *MailBox* or

associated with a specific Tag and the details of a specific message;

- a *MessageWriter* view container, permitting one to access the details of a specific message;
- a *Settings* view container, permitting one to modify the settings of Gmail.

The *MailBox*, *MessageWriter*, and *Settings* view containers are in alternative: only one at a time is displayed. None of these alternate view containers is the default one, because they are all accessed as a consequence of an explicit user's choice. The *MessageWriter* and *Settings* view containers are denoted as landmark, because they are reachable from all the other sibling view containers of the *MailMessages* view container. Conversely, the *MailBox* view container is not denoted as landmark, because it is accessed only by means of a specific interaction event: the selection of a *MailBox* from the *MboxList* view component.

The *MailBox* view container comprises the view component (*MessageList*) showing the *MailMessages* associated to a given *MailBox* or *Tag*. The *MboxList* allows user interaction: selecting a specific *MailBox* or *Tag* the user produces a navigation event that results in changing the content of the *MessageList*, so to display the messages of the selected *MailBox* or *Tag*. This behavior is represented in the model fragment shown in Figure 6.

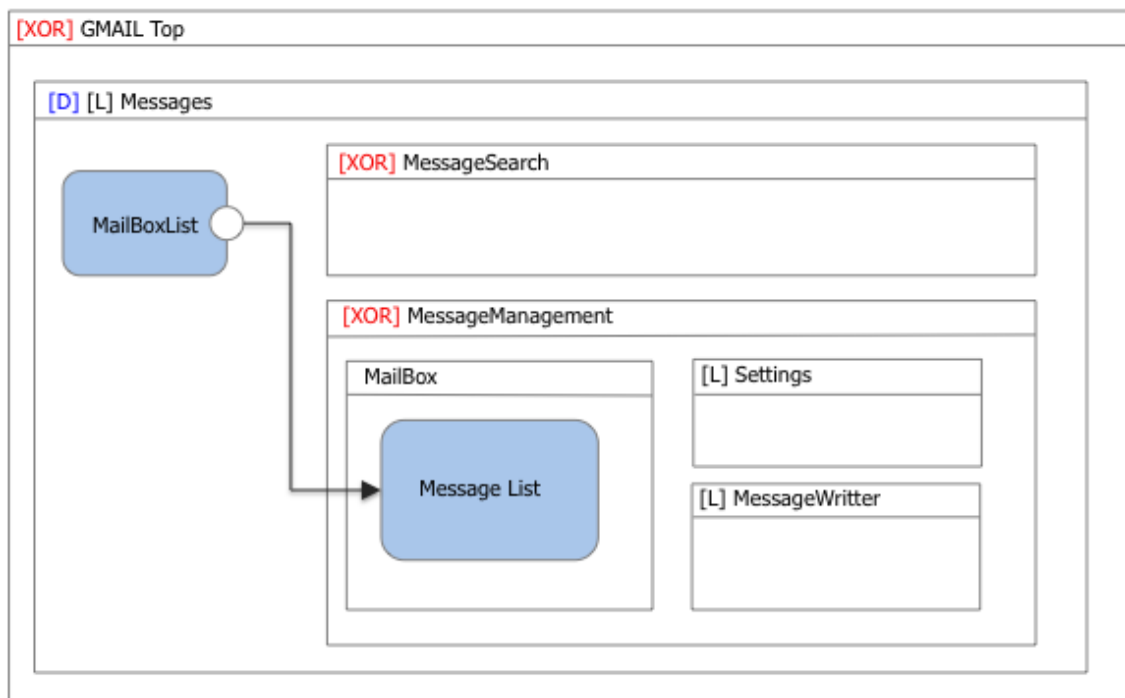


Figure 6: Model of the *MailMessages* view container: a navigation event and parameter passing flow between the *MboxList* view component and the *MessageList* view component denote that the user can select one mail box and view a list of its messages

## Semantics

1. The *MBoxList* view component is associated with an event, denoted by a circle. A interaction flow connects the event to the target components affected by it: *MessageList*. The semantics of this pattern is that a user's interaction with the *MBoxList* view component determines: 1) the display of the view container that comprises the *MessageList* view component (the *MailBox* XOR child of the *MessageManagement*) the computation and 2) the display of the target view

component (in this case, the *MessageList* component is computed with the selected *MailBox* as input parameter and displayed).

The model of Figure 6 can be refined to show the parameter binding that binds the selection of a *MailBox* in the *MBoxList* component and the display of the messages of that MailBox in the *MessageList* view component.

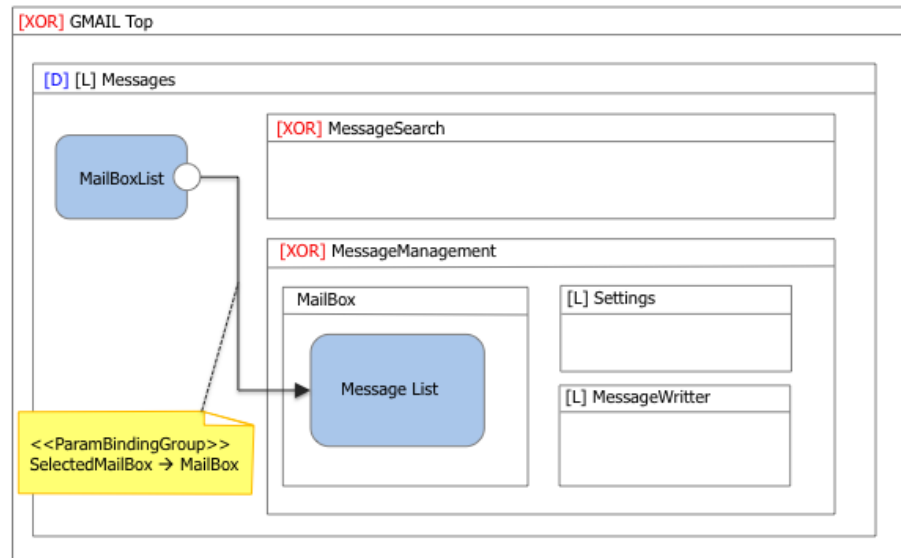


Figure 7: Notations to express (or infer) parameter dependencies between view components.

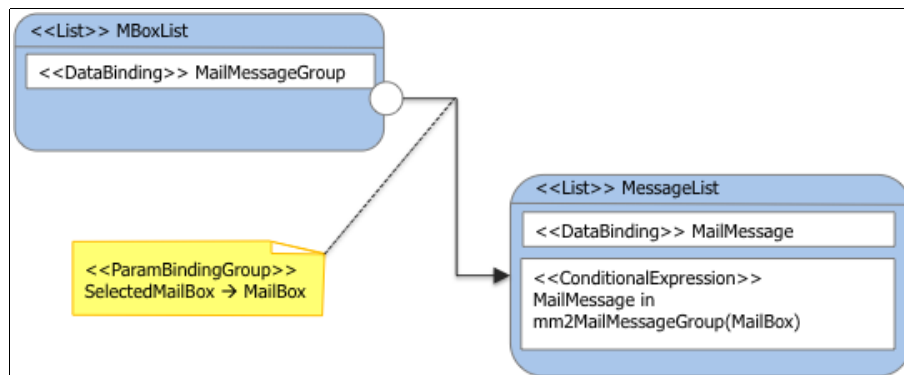


Figure 8: Notations to express (or infer) parameter dependencies between view components with extension mechanism.

### Language extension and notation

1. In the upper part of Figure 8, a UML-style annotation explicitly expresses that an output parameter of the source component is associated with an input parameter of the target component.
2. In the lower part of Figure 8, the model makes use of the IFML extension mechanism. An `<<List>>` component is introduced, which extend the basic view component to represent a list of dynamically extracted data objects. The component refers a content binding of the content model where the objects of the list belong; it may also refer to an expression to denote a filter on the instances to display. In this case, the join expression on relationship

*mm2MailMessageGroup* (see content model) dictates that only the messages of the mail box received as an input parameter are displayed. The semantics of the component may specify default input and output parameters, so that the parameter binding can be inferred and need not be explicitly represented: the default output of the *MboxList* list component is defined as the selected object of type *MailBox*: the default input of the *MessageList* list component is an object of type *MailMessageGroup*, as specified by the join expression on the relationship *mm2MailMessageGroup*. Since these two parameters match, there is no need of expressing the parameter binding explicitly.

The *MessageList* component supports the interaction with mail messages, individually or in sets. On the entire set of messages, the *MarkAllAsRead* event permits the user to update the message in the current *MailBox*, setting their status to “read” (see Figure 9).

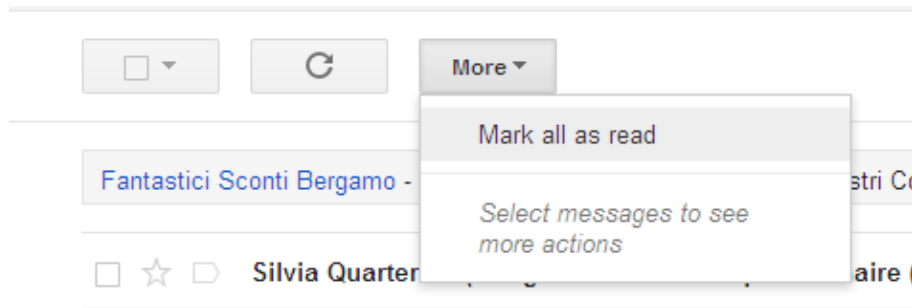


Figure 9: The *MarkAllAsRead* user-generated event marks all messages in the current mail box as “read”

As shown in Figure 10, the *MessageList* supports a second kind of interaction: the selection of a subset of messages; when there is at least one selected message, a view container is displayed (*MessageToolbar*), which permits the user to perform several actions in the selected messages: archiving, deleting, moving to a *MailBox/Tag*, reporting as spam, etc.

In summary, the *MessageList* component supports three types of interactive events:

1. an event for selecting the entire set of messages and triggering an action upon them, marking all as read (Figure 9);
2. an event for selecting/deselecting one or more messages (Figure 10);
3. an event for selecting an individual message and opening it for reading.

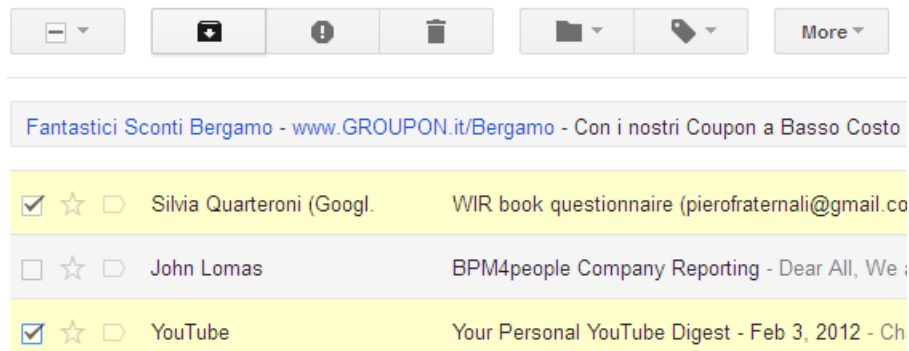


Figure 10: When one or more messages are selected in the *MessageList* component, the *MessageToolbar* view container is displayed, which allow the user to perform several actions of the selected set of messages. If all messages are deselected, such view container is no longer displayed

### Language extension and notation

1. For making the model more self-explaining and supporting code generation better, it is possible to further extend IFML with a specific view component: the *MultiChoiceList* (Figure 11). The multi choice list would extend the behavior of the list view component with more event types: the default type (denoted by the default notation) expresses the selection of one element of the list; the selection/de-selection event type, denoted by a ticker icon, expresses the selection or de-selection of any number of elements; the set selection event type, denoted by an asterisk, denotes the triggering of an action on the entire set of element of the list.

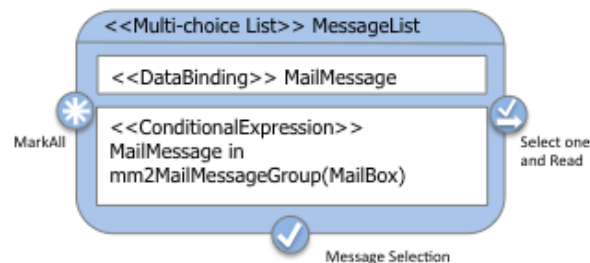


Figure 11: The *<<Multi-choice List>>* view component extends the *<<List>>* view component to enable more types of interaction events with the element of the list

The behavior of the *MessageSelection* event of the *MessageList* view component that triggers the display of the *MessageToolbar* view container is modeled as shown in Figure 12.



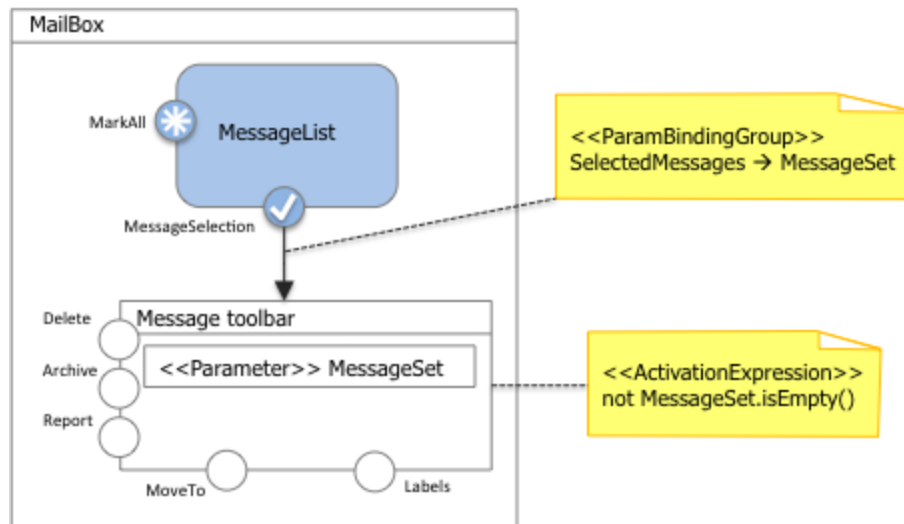


Figure 12: User events that mark one or more messages in the current mail box produce the display of the *MessageToolbar* view container, which remains visible/active if at least one message is selected

The *MessageSelection* event has a parameter binding, which associates the (possibly empty) set of currently selected messages with an input parameter of the *MessageToolbar* view component. The *MessageToolbar* view component is associated with an (activation) expression, which tests that at least one message is selected.

### Notation

1. For better readability of the model, it is possible to name the events, as shown in Figure 11 and in Figure 12. This annotation can be a guide for producing the implementation, for example it can be used to generate the labels of buttons and links, the tool tips of commands, and other similar usability aids.

### Semantics

1. The association of a boolean expression to a view container means that the view container is active/visible if the expression evaluates to true.

The actions performed by the user on the messages (all, or a subset thereof) are represented as shown in Figure 13. An interaction flow arrow connects the event responsible of triggering the action to the action itself, supporting the specification of parameter bindings.

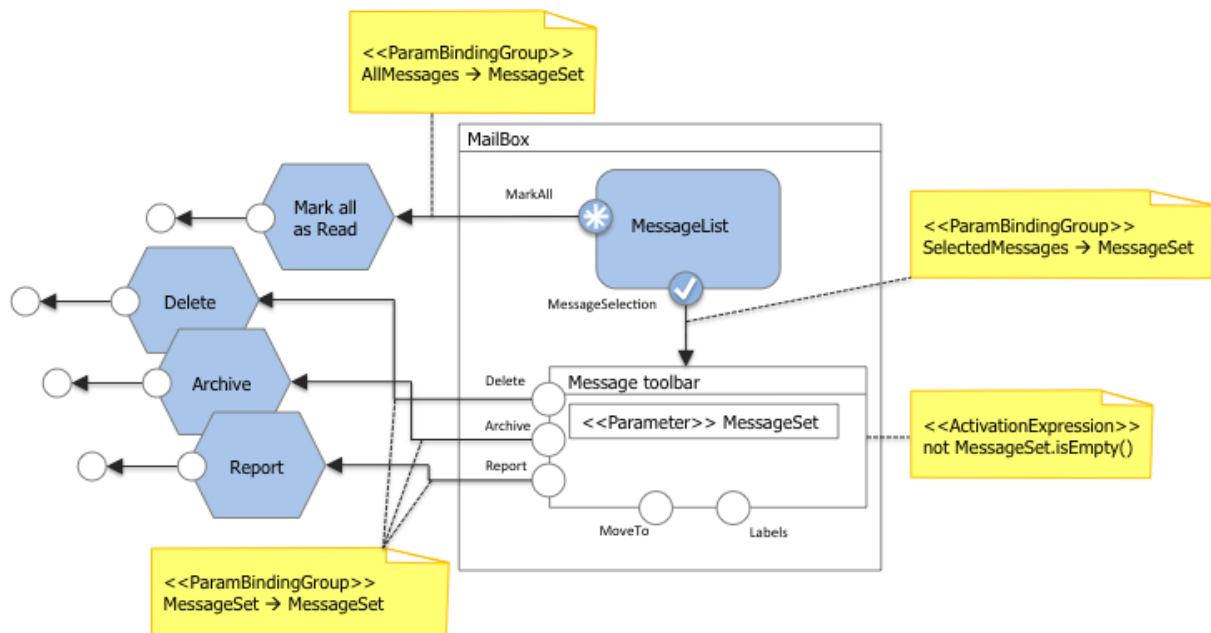


Figure 13: The MessageList view component and the MessageToolbar view container are associated with events that trigger actions on messages. Actions are represented as components placed outside the view containers, with input and output parameters

For example, the output parameter (*MessageSet*) of the *MessageToolbar* view container is associated with an input parameter of the business actions *Delete*, *Archive*, and *Report*.

The execution of an action produces an action completion event and the sending of an asynchronous notification, denoted as a circle linked to the action box. Such a notification sending event is matched by a system event, which triggers the display of a *MessageNotification* view component, shown in Figure 14.

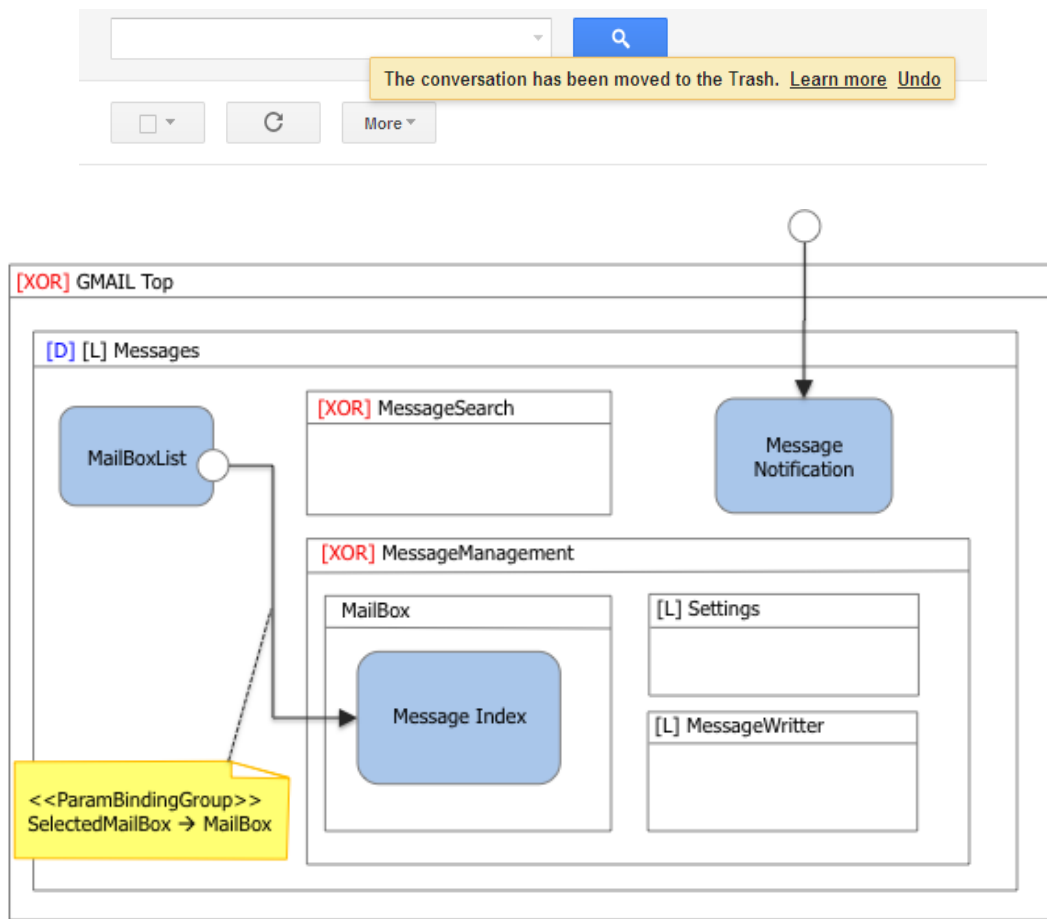
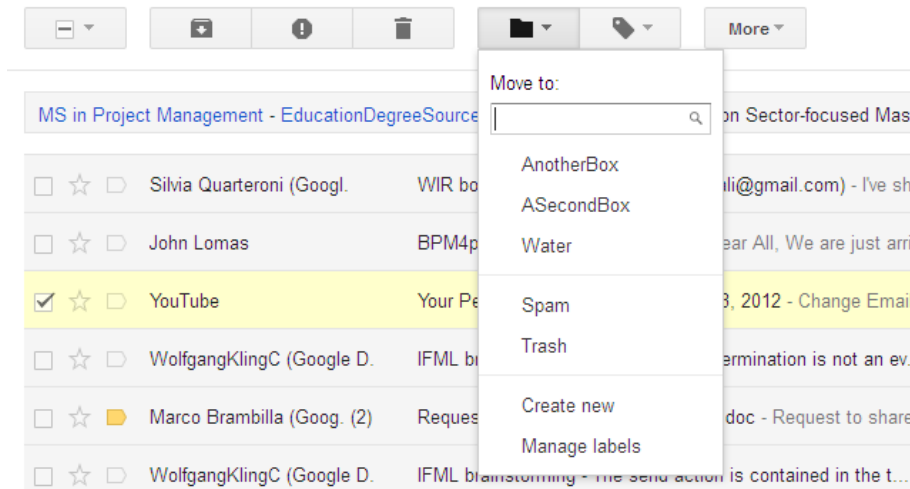


Figure 14: The MailMessages view container comprises a message notification component, which displays notifications of executed actions on MailMessages (illustrated above)

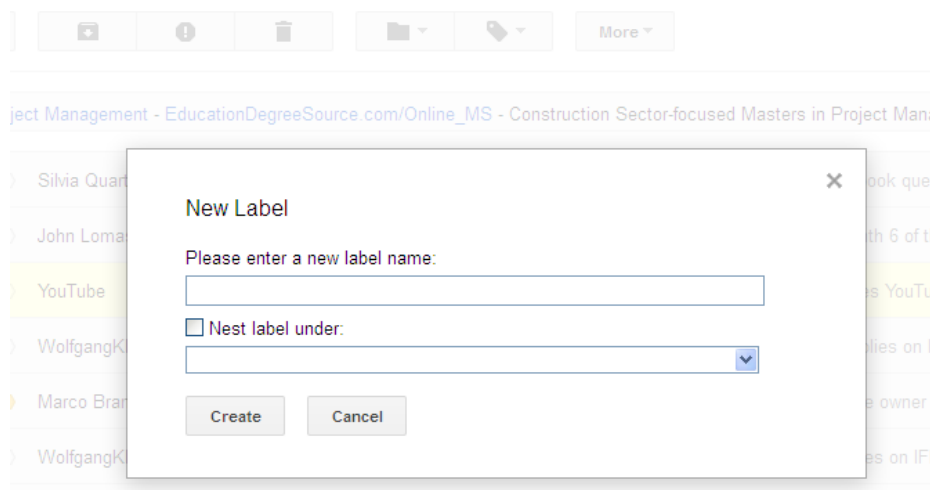
Note that the notification reception event is associated with the parameter *MessageSet*, which can be used in the *MessageNotification* component, e.g., to support the undo of the action (not modeled for brevity).

Some actions on mail messages require a more elaborate interaction flow: *Move to folder* and *Associate with tag* (see Figure 18). For example, moving a set of selected messages to a folder is done by first accessing a view container in a new window with the list of available *MailBox* and *Tags* (shown in Figure 15) and then selecting from such list the destination *MailBox* or *Tag*.



*Figure 15: The MoveTo action is activated by first accessing a modal view container with the list of the available MailBoxes and Tags and then selecting the target one. The view container comprising the list of MailBoxes and Tags is also associated with navigation events for creating new tags and managing existing tags*

The view container comprising the list of *MailBoxes* and *Tags* is also associated with navigation events for creating new tags and managing existing tags. For example, the *Create New* event causes a modal view container to be displayed, whereby the user can create a new tag and associate the selected messages with it (see Figure 16).



*Figure 16: The Create New event causes a modal view container to be displayed, whereby the user can create a new tag and associate the selected messages with it*

The interaction flow for moving a message to an existing or newly created tag is represented in Figure 17. The view container ([Modal] and [Modeless]) icons annotate the view containers to specify that they open in a new window and are modal or modeless.

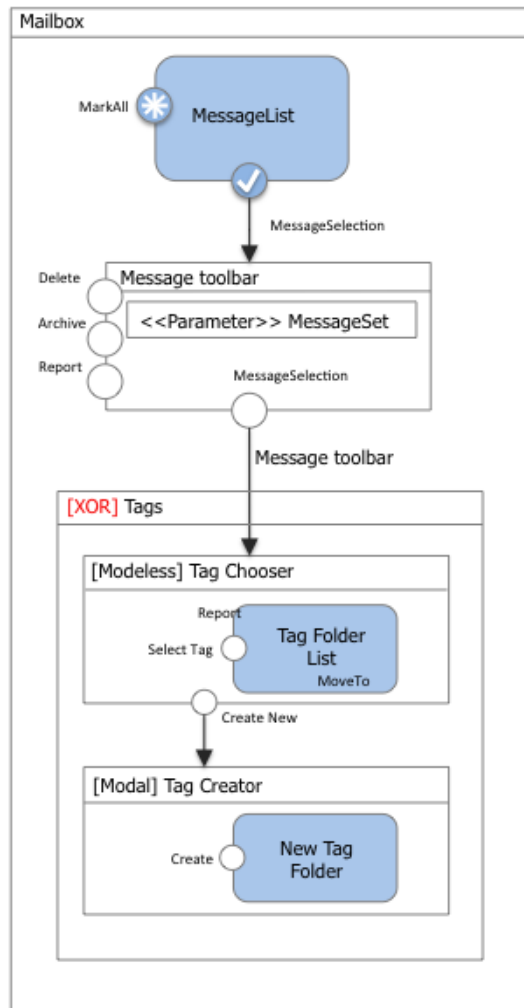


Figure 17: The model of the interaction flow for moving a message to an existing or newly created tag. The view container TagChooser is a modeless view container (which hides when clicking outside of it) and the TagCreator is a modal view container.

Archiving, reporting, and associating messages to existing/new tags imply the invocation of business logic components, as shown in Figure 18.

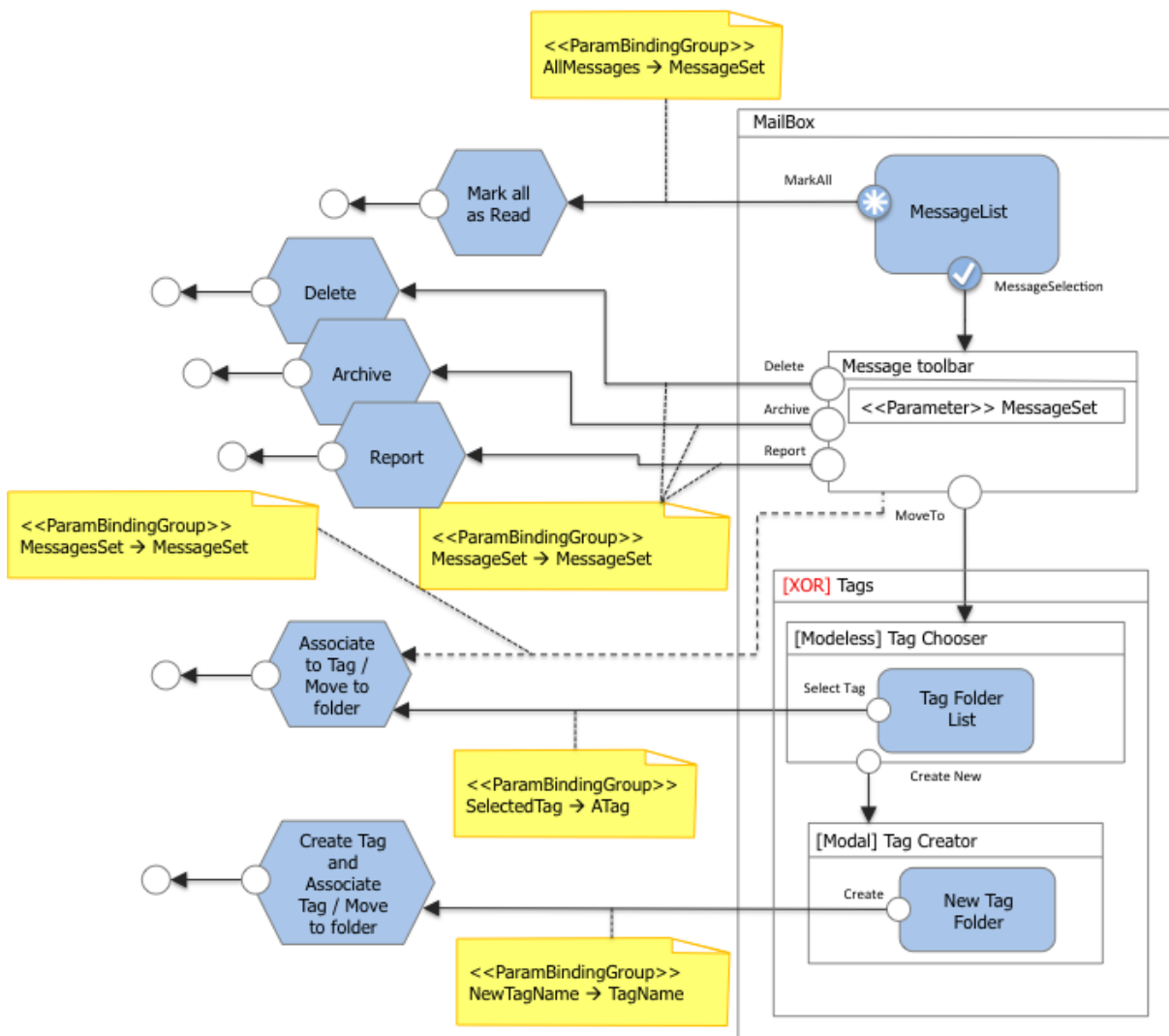


Figure 18: The model of the interaction flow for moving a message to an existing or newly created tag

In Figure 18 the parameter bindings are modeled explicitly: 1) the selected mail messages are associated with the input of the *Delete*, *Archive*, and *Report* actions; 2) the *SelectedTag* parameter, which corresponds to the user's choice of a tag to associate with a set of messages, is the input of the *AssociateToTag* action. Note that the *AssociateToTag* action receives the selected message set through a DataFlow (dashed arrow) coming from the MessageToolbar ViewContainer; 3) the *NewTagName* parameter, which corresponds to the new label entered by the user, is the input of the *CreateTag* action.

The specification of composite action flows is not allowed but the internal functioning of an action could be specified with an orchestration model (e.g, a UML activity diagram, a SOAML specification, etc.).

The access to the messages can also occur through a search functionality. An input field supports simple keyword based search; with a click, the user can also access a more powerful search input form, where he can specify several criteria to be matched, as shown in Figure 19.

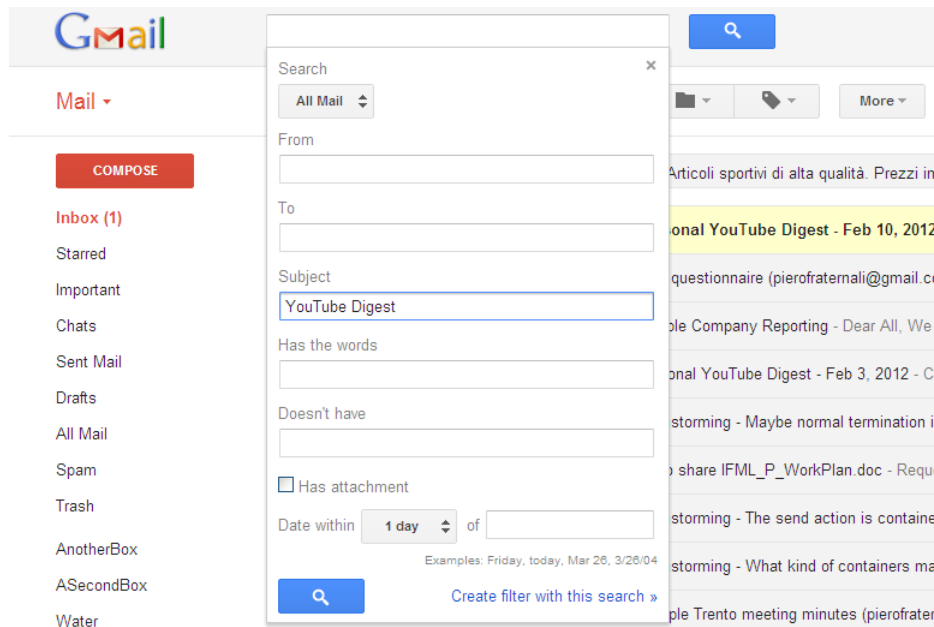


Figure 19: The message search functionality (full search modal view container)

The IFML model of the search functionality (shown in Figure 20) comprises a view component (*MessageKeywordSearch*) for entering a string to be matched to the mail messages and filter those to be displayed in the *MessageList* view component. Such an interaction flow can be represented with an event associated to the *MessageKeywordSearch* and a interaction flow to the *MessageList* view component; a parameter bindings specifies that the output parameter of the *MessageKeywordSearch* view component is associated with the input parameter of the *MessageList* view component. From the *MessageKeywordSearch* another event (*Show search options*) opens a modal view container (*FullSearch*), where the user can input more information to drive the search. In this latter case, the parameter binding associates each field value of the entry view component to a respective input parameter of the *MessageList* component. Note that after giving the input of the *FullSearch* two navigations occur. One for the *MessageList* for showing the search result and another to the Search container for passing and displaying the keyword search.

The example shown in the right part of Figure 20 illustrates how extending the basic IFML view components with domain specific view and business logic can make the model more self-descriptive. For instance, one could define a view component abstracting the notion of input forms for data entry (denoted by the stereotype <<Entry>>), composed of a set of typed fields (e.g., denoted as nested view components of type <<SimpleField>>); an <<Entry>> component could expose as default parameters, the values of the contained fields. The parameter binding would then couple each input field with the respective parameters of the ConditionalExpression expression of the dynamic list component (as shown in the right part of Figure 20). Note that the <<List>> view component is associated with multiple ConditionalExpression expressions, which are used to compute the component when different navigation events occur. Which expression has to be evaluated is dictated by the parameter binding associated with the interaction flows of the event triggering the computation.

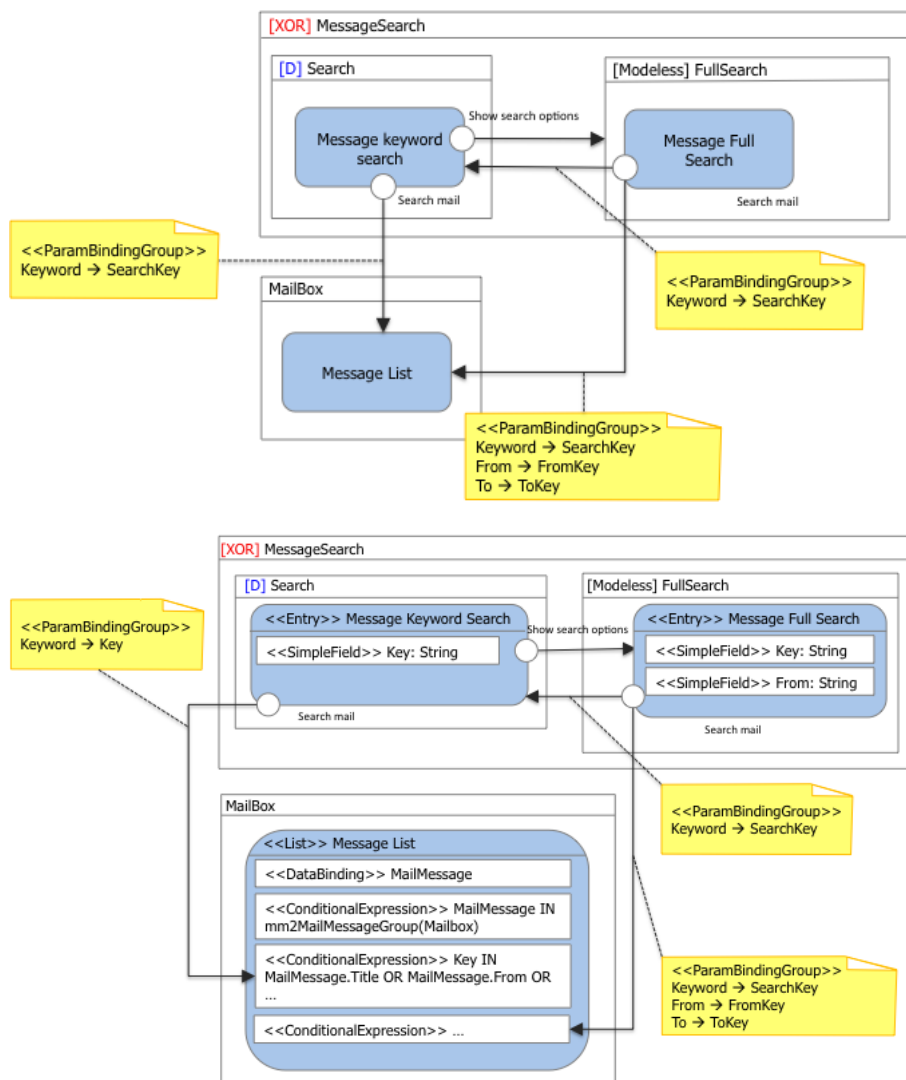


Figure 20: The model of the message search functionality (top). The same model refined with the use of the extended view components **<<Entry>>** and **<<List>>** (bottom)

The selection of a message from the *MessageList* view component causes the *MessageDetails* view component to be displayed. Such a component permits the user to access one specific message at a time. This corresponds to the XOR (MessageManagement and MessageReader) nesting of view components shown in Figure 21.





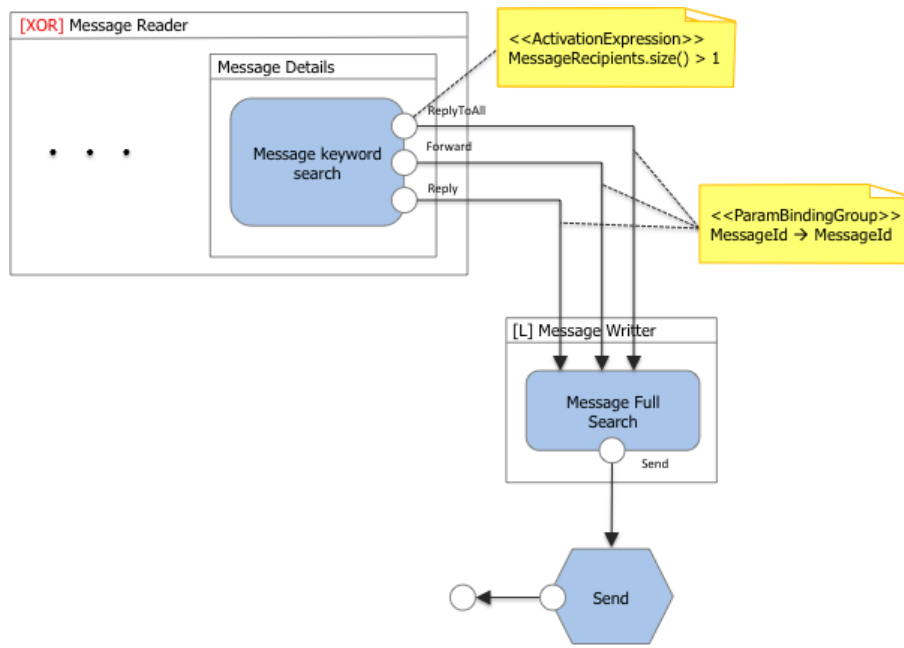


Figure 22: The different ways to access the *MessageWriter* view component

The link *ReplyToAll* is active only when the message displayed in the *MessageDetails* view component is associated with more than one recipient. This can be expressed as a activation expression associated with the *ReplyToAll* event (see Figure 22). The *MessageWriter* view component has an internal structure, shown in Figure 23.

Buttons: Reply, Reply to all, Forward, Send, Save Now, Discard

To: Piero Fraternali <piero.fraternali@polimi.it>

Cc: Marco Brambilla <mbrambil@elet.polimi.it>

Links: Add Bcc, Edit Subject, Attach a file, Insert: Invitation

Rich formatting » Check Spelling ▾

On Sat, Feb 18, 2012 at 3:54 PM, Piero Fraternali <piero.fraternali@polimi.it> wrote:  
 > This mail is a non sense message to be used in the document illustrating how  
 > IFML can model the GMail client application.  
 > Have fun!  
 >  
 > Piero Fraternali

Figure 23: The internal structure of the *MessageWriter* view component

The view component permits the user to edit a new message, reply to an existing message (to the sender only or to all) and to forward an existing message. The view component can be represented as a form composed of different fields: *To*, *Cc*, *Bcc*, *Subject*, *Body*, and *Attachment*.

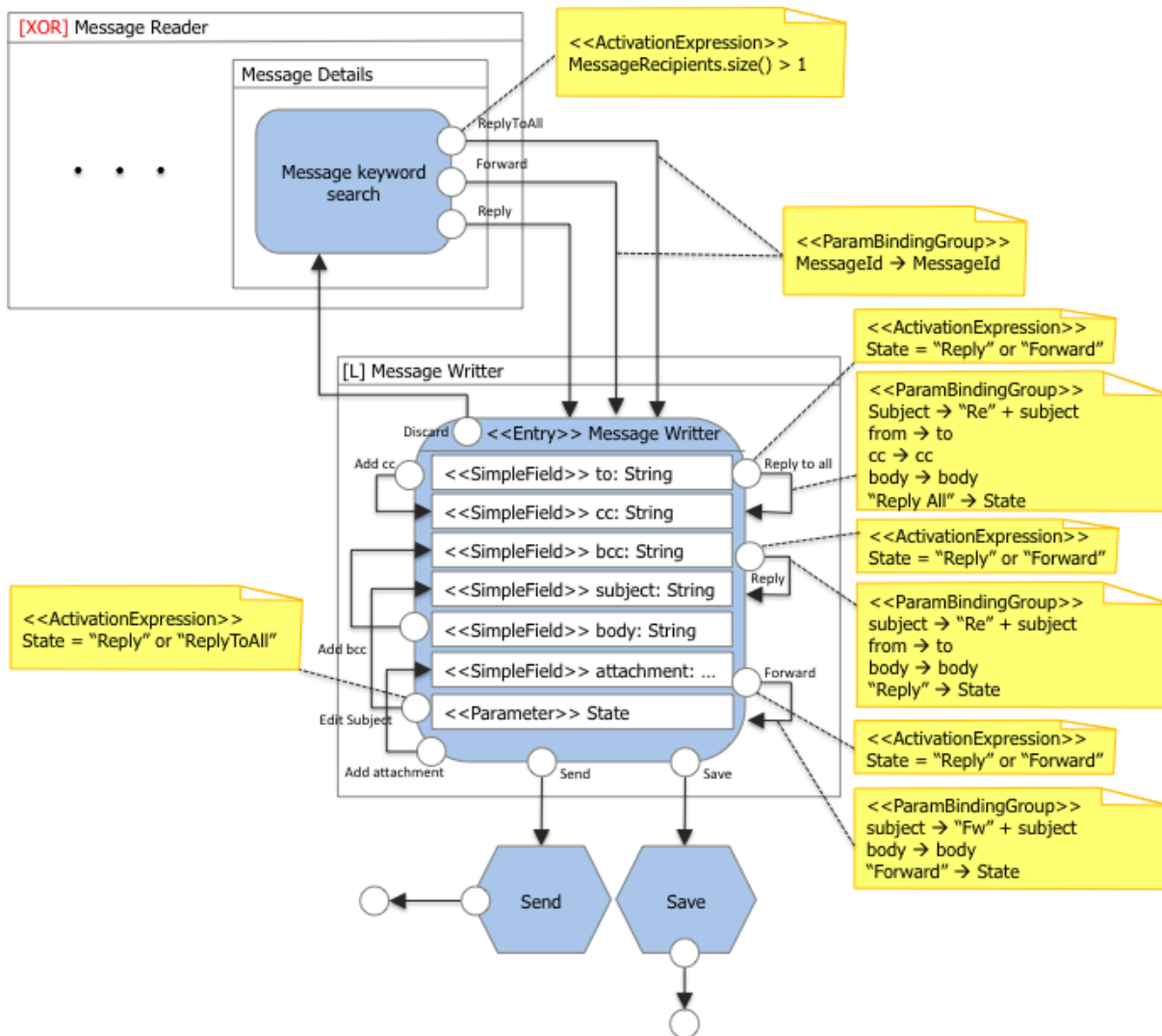


Figure 24: The IFML model of the internal structure of the MessageWriter view component, with the names of the event displayed for clarity

Note that some form fields can be automatically filled with content (e.g., the *To* field is automatically set to the mail address of the sender when the *ReplyTo* event is raised). This is modeled by considering that each `<<SimpleField>>` component of a `<<Entry>>` component is associated to an implicit input parameter that denotes the value of the field.

In addition to the form fields view component parts, the *MessageWriter* view component has an explicit parameter (*State*), which denotes four different edit configurations: 1) when the user is editing a new message, 2) replying to the sender of an existing message, 3) replying to the sender of an existing message and to all recipients in copy, or 4) forwarding an existing message. These edit configuration differ in the subset of fields that are automatically filled-in and in the commands that are enabled: for example Figure 23 shows the edit configuration when the user is replying to the sender of an existing message and to all recipients in copy.

The *MessageWriter* view component is associated with three events (*Reply*, *ReplyToAll*, *Forward*) for switching from one of the *ReplyTo*, *ReplyToAll*, and *Forward* editing configurations to the other two

ones. For example, Figure 24 shows that the the event *ReplyToAll* is active only when the *State* parameter has the value *Reply* or *Forward* and that its effect is to assign a value to the *Subject*, *To*, *Cc* and *Body* field, and set the *State* parameter to the value *ReplyToAll*.

Another example of conditional event is the *EditSubject* one: the event for editing the subject field is available only when the *State* parameter is *ReplyToAll* or *Reply*.

The model refinement of the *MessageWriter* view component can go on, by zooming-in inside the *Body* field. The Body field can be refined by a nested component, which supports client-side business logic like the rich formatting and the spell checking of the text.

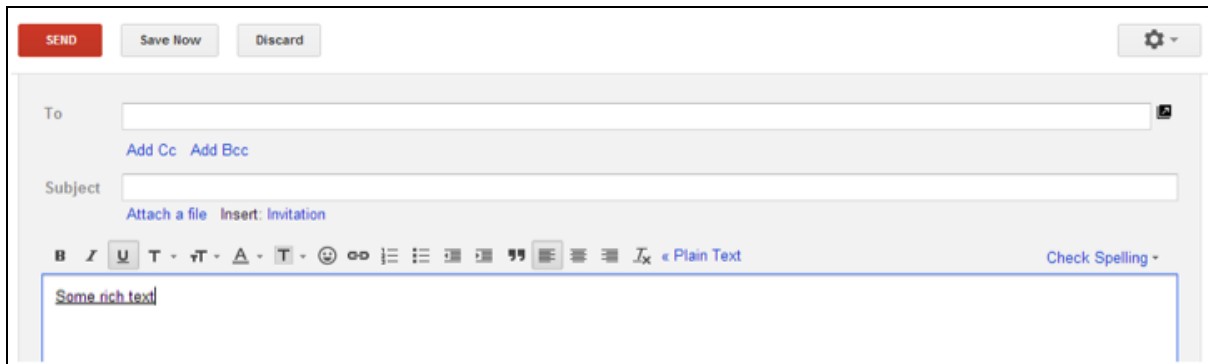


Figure 25: The rich text editing toolbar in the Body input field of the MessageWriter view component

Figure 25 shows the rich text editing toolbar in the Body input field of the *MessageWriter* view component, which appears when the user clicks on the *RichFormatting* link shown in Figure 23.

A number of editing commands apply to the text, which rewrite the content of the view component at the client side. Similarly, the *CheckSpelling* command triggers a client-side action that highlights in red the misspelled words.

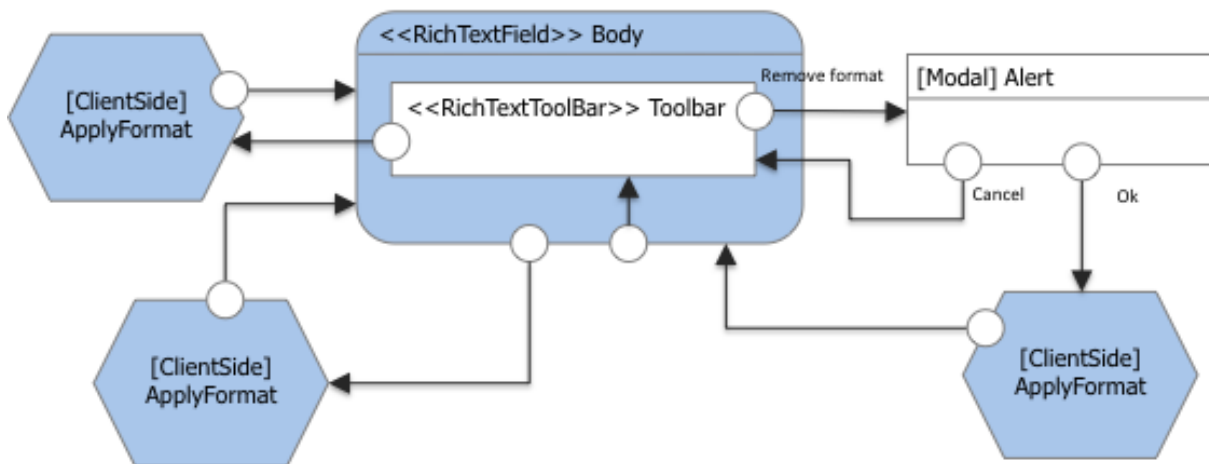


Figure 26: The rich text editing toolbar in the Body input field of the MessageWriter view component

Figure 26 shows the IFML model of the rich text editor field. An event corresponding to the *RichFormatting* interaction flow permits the user to access the *Rich Text Toolbar* view container, which comprises a number of commands for applying formatting to the text; for brevity, we summarize these commands as the invocation of the *ApplyFormat Action*, which is shown with the [ClientSide] icon to denote that it actuates at the client side. Similarly, an event permits the user to trigger the *SpellCheck*

*Action*, which is also client-side. Finally, from the *RichText Toolbar* view container an event (the *PlainText* link visible in Figure 25) permits one to remove the formatting and go back to the plain text mode; before firing the action, though, an alert modal view container is presented where the user can confirm or discard the format removal action. Discarding the action leads one back to the *Body* component and to the *Rich Text Toolbar*.