Interaction Flow Modeling Language (IFML)

FTF – Beta 2- Revision 21

_____

This OMG document replaces document ptc/2013-03-08.

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.)

Table of Contents

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications
- UML
- MOF
- XMI
- CWM
- Profile specifications

OMG Middleware Specifications
- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)

Platform Specific Model and Interface Specifications
- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Ave
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

1   Scope

This specification defines the Interaction Flow Modeling Language (IFML). The objective of IFML is to provide system architects, software engineers, and software developers with tools for the definition of Interaction Flow Models that describe the principal dimensions of an application front-end: the view part of the application, made of view containers and view components; the objects that embody the state of the application and the references to business logic actions that can be executed; the binding of view components to data objects and events; the control logic that determines the actions to be executed after an event occurrence; and the distribution of control, data and business logic at the different tiers of the architecture.

2   Conformance

There are five ways in which a tool may demonstrate conformance to the IFML metamodel.

1. *Abstract syntax conformance.* A tool demonstrating abstract syntax conformance provides a user interface and/or API that enables instances of concrete IFML metaclasses to be created, read, updated and deleted. The tool must also provide a way to validate the well-formedness of models that corresponds to the constraints defined in the IFML metamodel.

2. *Concrete syntax conformance.* A tool demonstrating concrete syntax conformance provides a user interface and/or API that enables instances of IFML notation to be created, read, updated and deleted.

3. *Model interchange conformance.* A tool demonstrating model interchange conformance can import and export conformant XMI for all valid IFML models. Model interchange conformance implies abstract syntax conformance.

4. *Diagram interchange conformance.* A tool demonstrating diagram interchange conformance can import and export conformant DI for all valid IFML models with diagrams. Diagram interchange conformance implies both concrete syntax conformance and abstract syntax conformance.

5. *Semantic conformance.* A tool demonstrating semantic conformance provides a demonstrable way to interpret IFML semantics, e.g. code generation, model execution, or semantic model analysis.

A tool can claim conformance with the IFML metamodel if and only if the software fully implements the IFML metamodel in one or more of the above ways. A tool that only partially implements the metamodel can claim only that it is based on this specification, but cannot claim conformance with the specification.

A tool already conforming to the UML specification may demonstrate conformance with the UML Profile for IFML by providing the means to apply the profile to a UML model, as specified in Clause 11.

3   Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC2119, http://ietf.org/rfc/rfc2119, March 1997
- *OMG Unified Modeling Language (OMG UML), Infrastructure*, Version 2.4.1, formal/2011-08-05, August 2011.
- *OMG Unified Modeling Language (OMG UML), Infrastructure*, Version 2.4.1, formal/2011-08-06, August 2011.
- *OMG Meta Object Facility (MOF) Core Specification*, Version 2.4.1, formal/2011-08-07, August 2011
- *OMG MOF 2 XMI Mapping Specification, Version 2.4.1*, formal/2011-08-09, August 2011
- *Diagram Definition (DD), Version 1.0,* formal/2012-07-01, July 2012

4 Terms and Definitions

There are no formal definitions of terms in this specification.

5   Symbols

There are no symbols defined in this specification.

6   Additional Information

6.1   Business Motivation

In the last twenty years, capabilities such as form-based interaction, information browsing, link navigation, multimedia content fruition, and interface personalization have become mainstream in many business-to-consumer (B2C), business-to-business(B2B), and business-to-employee (B2E) applications. These are implemented on top of a variety of technologies and platforms: desktop applications, client-server applications, web applications, rich internet applications, mobile applications, and even human machine interfaces for industrial control, where more and more embedded systems are equipped with browser-based GUIs. This convergence in technologies is reflected in the HTML 5 initiative, which aims at establishing a unified set of concepts and a common technological platform for the development of a broad spectrum of interaction front-ends.

However, the emergence of such an unprecedented range of devices, technological platforms, and communication channels is not accompanied by the advent of an adequate approach for creating a Platform Independent Model (PIM) that can be used to express the interaction design decisions independently of the implementation platform. This causes front-end development to be a costly and inefficient process, where manual coding is the predominant development approach, reuse of design artifacts is low, and portability of applications across platforms remains difficult.

Using IFML for PIM-level interaction flow modeling, brings several benefits to the development process of application front-ends:

- It permits the formal specification of the different perspectives of the front-end: content, interface composition, interaction and navigation options, and connection with the business logic and the presentation.
- It separates the stakeholder concerns by isolating the specification of the front-end from its implementation-specific issues.
- It improves the development process, by fostering the separation of concerns in the user interaction design, thus granting the maximum efficiency to all the different developer roles.
- It enables the communication of interface and interaction design to non-technical stakeholders, permitting validation of requirements from subject matter experts (SMEs) and clients sooner in the development process.

6.2   Design Principles

Front-end design is a complex and multidisciplinary task, where many perspectives intersect. Therefore, IFML is particularly attentive to model usability and understandability, by explicitly addressing all the factors that contribute to making a PIM quickly learned, easy to use, and open to extensibility:

- It is concise, avoiding redundancy and reducing the number of diagram types and concepts needed to express the salient interface and interaction design decisions.
- It includes extensibility in the definition of new concepts (e.g., novel interface components or event types).
- It ensures implementability, that is, it supports the construction of model transformation frameworks and code generators that can map the PIM into a suitable PSM and ultimately into executable applications for a wide range of technological platforms and access devices.
- It ensures model-level reuse, that is, it supports the definition of reusable design patterns that can be stored, documented, searched and retrieved, and re-used in other applications.
- It allows the application of inference rules at the modeling level that automatically apply default modeling patterns and details whenever they can be determined from the context, giving the possibility to avoid the need for modelers to specify inferable information (e.g., automatic inference of the parameters that need to be passed from a component to another at the modeling level).

6.3   IFML Artifacts

The IFML specification consists of four main technical artifacts:

- *The IFML metamodel* specifies the structure and semantics of the IFML constructs using MOF.
- *The IFML UML profile* defines a UML-based syntax for expressing IFML models. In particular, the UML profile extends concepts of the following UML diagrams: class diagrams, state machine, and composite structure diagrams.
- *The IFML visual syntax* offers a dedicated visual syntax for expressing IFML models in a particularly concise way. Specifically, it provides a unique diagram capable of compacting the aspects of the user interface that are otherwise expressed separately with UML class diagrams, state machine and composite structure diagrams.
- *The IFML XMI* provides the IFML model exchange format, for tool portability.

6.4   Acknowledgements

The standardization initiative and the FTF of IFML have been lead by Marco Brambilla.

This specification was originally authored by:

- Marco Brambilla (WebRatio and Politecnico di Milano)
- Piero Fraternali (WebRatio and Politecnico di Milano)

Other authors that contributed to the current version include:

- Aldo Bongio (WebRatio)
- Stefano Butti (WebRatio)
- Adriano Comai (Soluta.net and WebRatio)
- Wolfgang Kling (Ecole des Mines de Nantes and WebRatio)
- Manfred R. Koethe (88Solutions)
- Andrea Mauri (WebRatio and Politecnico di Milano)
- Emanuele Molteni (WebRatio)
- Ed Seidewitz (Model Driven Solutions and Ivar Jacobson International)

We wish to thank all the other contributors that provided useful input, feedback and discussions on the IFML specification.

7   IFML Specification

7.1   Key Concepts of IFML

The Interaction Flow Modeling Language (IFML) supports the platform independent description of graphical user interfaces for applications accessed or deployed on such systems as desktop computers, laptop computers, PDAs, mobile phones, and tablets. The focus of the description is on the structure and behavior of the application as perceived by the end user. The description of the structure and behavior of the business and data components of the application is limited to those aspects that have a direct influence on the user's experience.

With respect to the popular Model-View-Controller (MVC) model of an interactive application,[1] the focus of IFML is on the view part. Furthermore, IFML describes how the view references or is depended on by the model and control parts of the application. In particular:

- With respect to the view, IFML deals with the view composition and the description of the elements that it exposes to the user for interaction.
- With respect to the controller, IFML lets the designer specify the effects of user interactions and system events on the application by defining the relevant events that the controller must take care of.
- With respect to the model, IFML allows for specification of the references to the data objects that embody the state of the application and are published in the user interface, as well as of the reference to the actions that are triggered by the interaction of the user.

IFML can be complemented with external models for the complete specification of applications with aspects that are not directly connected with the user interface and interaction:

- The internal functioning of the actions triggered by the user's interaction can be described using any action model. For example, if the action refers to the invocation of an object's method, this can be described using UML class and collaboration diagrams; if the action refers to the invocation of a web service, this can be described using a SoaML diagram.[2]
- The object model underlying the application can be described with any structural diagram, for example with a UML class diagram or a Common Warehouse Metamodel (CWM) diagram.[3]

Modeling the user interface and interaction with IFML amounts to addressing the following aspects:

- The composition of the view, in terms of its partition into independent visualization units, which can be displayed simultaneously or in mutual exclusion, and can be nested hierarchically.
- The content of the view, in terms of both the data elements published from the application to the user and of the data elements input from the user to the application.
- The commands enabling the user's interaction and the corresponding events.
- The reference to actions triggered by the user's commands.
- The effects of the user's interaction and of the action execution on the state of the user interface.
- The parameter binding between the elements of the user interface and the triggered actions.

Consequently, an IFML model supports the following design perspectives:

- The *view structure specification*, which consists of the definition of view containers, their nesting relationships, their visibility, and their reachability.
- The *view content specification*, which consists of the definition of view components, i.e., content and data entry elements contained within view containers.
- The *events specification*, which consists of the definition of events that may affect the state of the user

---

[1]   See, for example, http://en.wikipedia.org/wiki/Mode-view-controller.
[2]   See http://www.omg.org/spec/SoaML.
[3]   See http://www.omg.org/cwm/

interface. Events can be produced by the user's interaction, by the application, or by an external system.

- The *event transition specification*, which consists of the definition of the effect of an event on the user interface. The effect can be the change of the view container or of the content displayed, the triggering of an action, or both.

- The *parameter binding specification*, which consists of the definition of the input-output dependencies between view components and between view components and actions.

7.2   IFML in a Nutshell

An IFML diagram consists of one or more top-level *view containers*. For example, a desktop application or a rich Internet application (RIA) can be modeled as having one top-level container, the main window; instead, a Web application can be modeled as having multiple top-containers, one for every dynamic page template.

Each view container can be internally structured in a *hierarchy of sub-containers*. For example, in a desktop or RIA application, the main window can contain multiple tabbed frames, which in turn may contain several nested panes. The child view containers nested within a parent view container can be displayed simultaneously (e.g., an object pane and a property pane) or in *mutual exclusion* (e.g., two alternative tabs). In case of mutually exclusive (XOR) containers one could be the *default container,* displayed by default when the parent container is accessed.

A view container can contain *view components,* which denote the publication of content or interface elements for data entry (e.g., input forms). A view component can have *input and output parameters.* For example, a view component for showing the properties of an object can have as an input parameter the identifier of the object to display; a data entry form or a list of items can have as output parameters the values input or the item selected by the user.

A view container and a view component can be associated with *events,* to denote that they support the user's interaction. For example, a view component can represent: a list associated with an event for selecting one or more items, a form associated with an event for input submission, or an image gallery associated with an event for scrolling though the gallery. Events in concrete are rendered as interactors, which depend on the specific platform and therefore are not modeled in IFML but produced by the PIM to Platform-Specific Model (PSM) transformation rules. For example, the scrolling of an image gallery may be implemented as a link in an HTML application and as a flip gesture in a mobile phone application.

The effect of an event is represented by an *interaction flow* connection, which connects the event to the view container or component affected by the event. For example, in an HTML web application the event caused by the selection of one item from a list may cause the display of a new page with the details of the selected object. This may be represented by an interaction flow connecting the event associated with the list component in a top-level view container (the web page) with the view component representing the object detail, positioned in a different view container (the target web page). The interaction flow expresses a change of state of the user interface: the occurrence of the event causes a transition of state that produces a change in the user interface.

An event can also cause the *triggering of an action,* which is executed prior to updating the state of the user interface; for example, in a web content management application the user can select from a list the elements to delete; the selection event triggers a delete action, after which the page with the list is redisplayed. The effect of an event triggering an action is represented by an interaction flow that connects the action to the view container or component affected by the event.

An *input-output dependency* between view elements (view containers and view components) or between view elements and actions is denoted by *parameter bindings* associated with navigation flows (interaction flows for navigating between view elements). For example, in Figure 1, the navigation flow that goes from the event denoting the selection of an item of the Artist Index view component to the Artist view component (showing the selection details), has a parameter binding that associates an output parameter of the Artists Index view component with an input parameter of the Artist view component. See also further examples in Figure 2, Figure 3 and Figure 4.
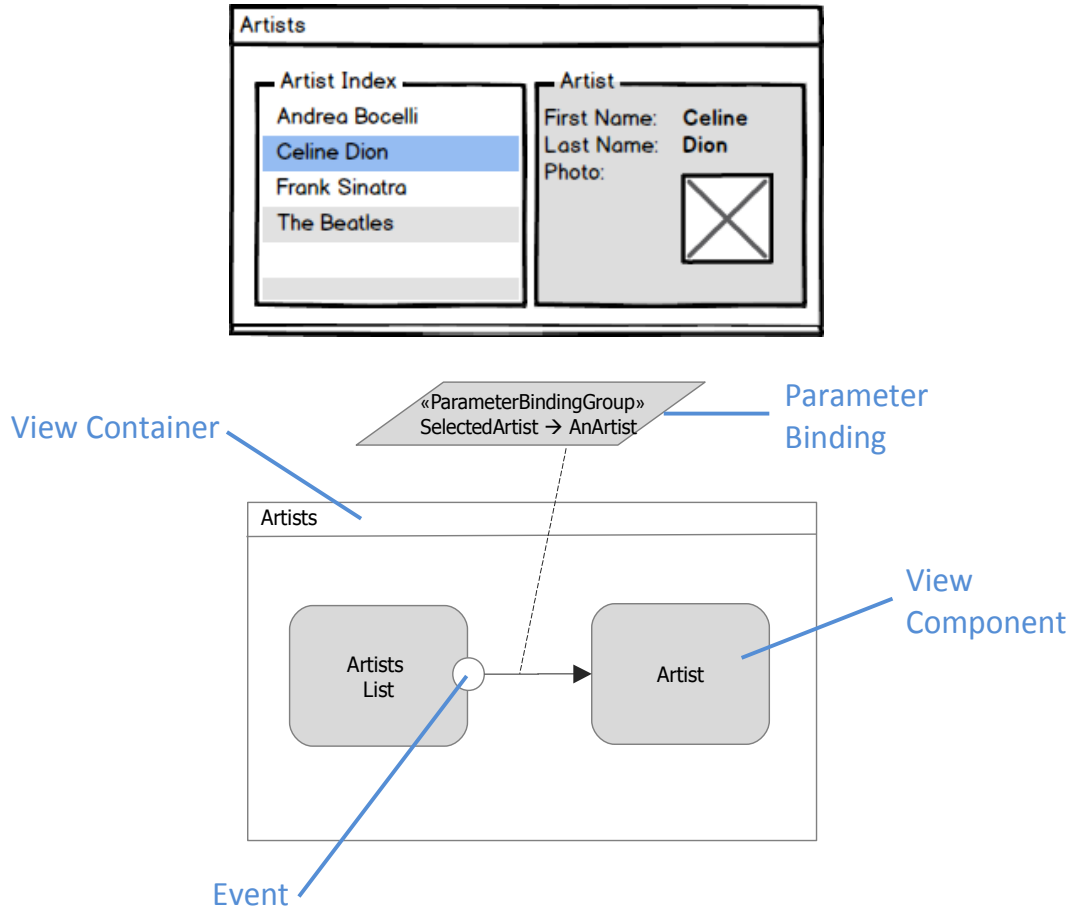
Figure 1: Example of user interface (top) and corresponding IFML model (bottom). The user selects an item in the list and displays its details in the same view container.
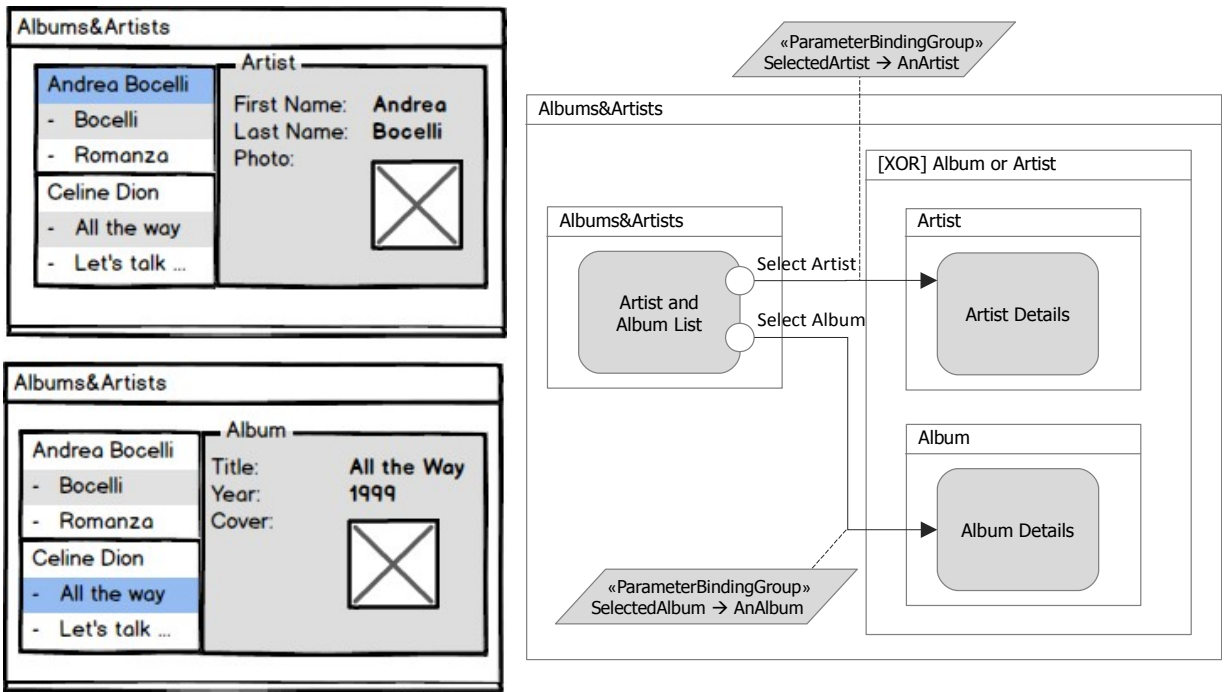
Figure 2: Example of user interface (left) and corresponding IFML model (right). One top-level container comprises three view containers: one with a list of artists and of their albums, one with the details of an artist, and one with the details of an album. The latter two view containers are mutually exclusive: only one at a time is displayed.
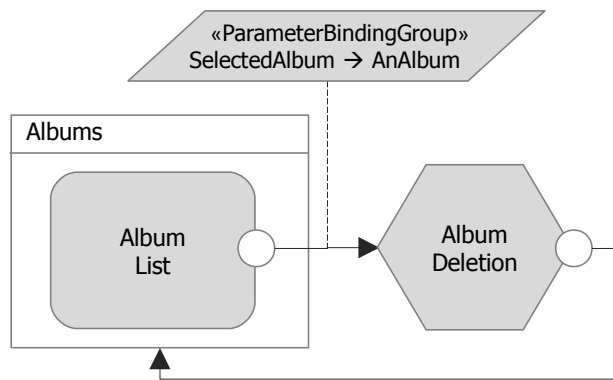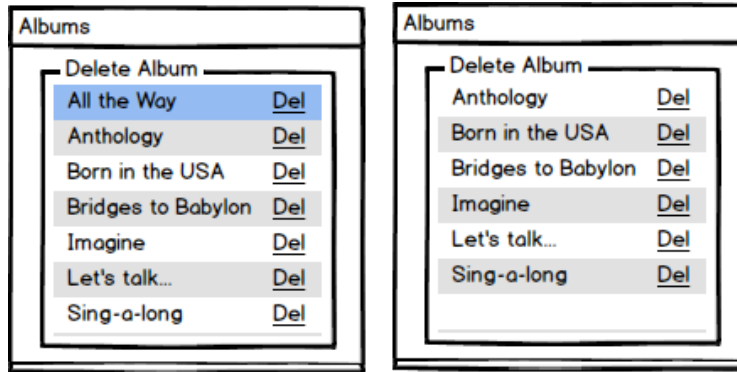
Figure 3: Example of user interface supporting action invocation (top) and corresponding IFML model (bottom). The user can select an item from a list of objects; the selection causes a delete action to be triggered after which the updated list of objects is redisplayed.
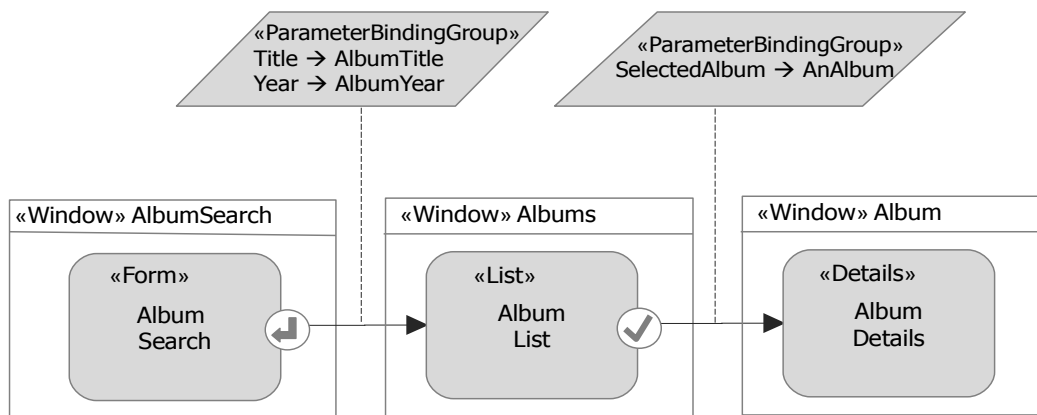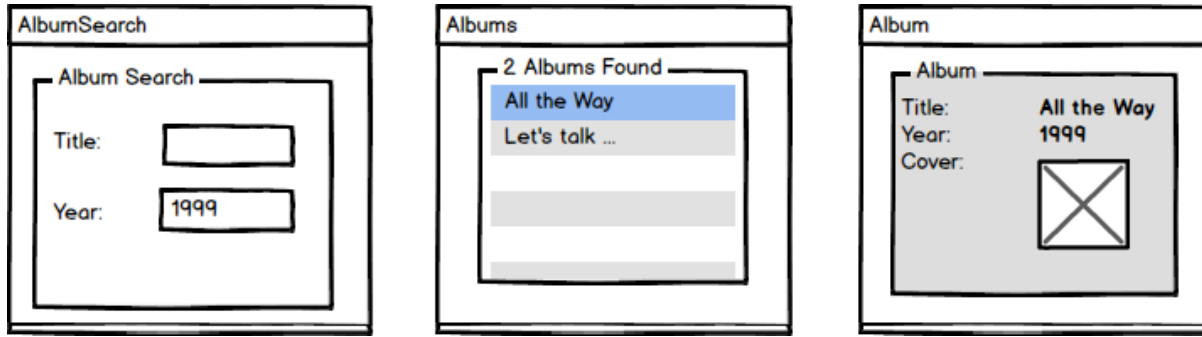
Figure 4: Example of user interface (top) and corresponding IFML model (bottom). The user enters data into an input form and submits them; this event causes a distinct view container to appear with a list of matching objects; finally, the selection of an item in the list causes the display of the corresponding details in a third view container.

7.3   Extensibility

IFML uses the extensibility mechanisms of UML to allow the definition of stereotypes, tagged values and constraints. The Extensions package exemplifies how the extension mechanism works: it contains concepts that extend concepts from the Core package. In the same way, new packages may be introduced containing new constructs, to model platform-independent or platform-specific concepts.

Extensions are meant to refine the semantics of the core concepts or to provide specific cases of core concepts. As such, they must therefore refine the semantics of the IFML concepts, and not modify it. The following concepts (and their extensions) can be extended in IFML while still achieving compliance to the standard:

- ViewContainer
- ViewComponent
- ViewComponentPart
- Event
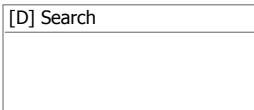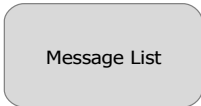
- DomainConcept and FeatureConcept
- BehaviorConcept and BehavioralFeatureConcept

Extensions of other elements are not allowed.
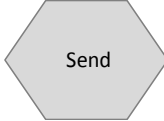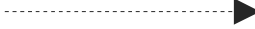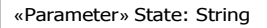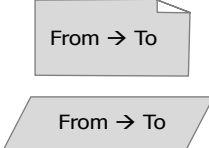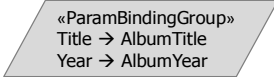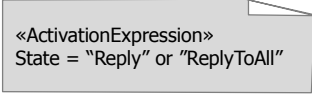
7.4 Concept List

Table 1 lists the core concepts of IFML and Table 2 lists a set of extension concepts provided as an example for the IFML extension mechanism.
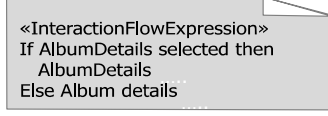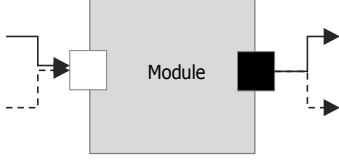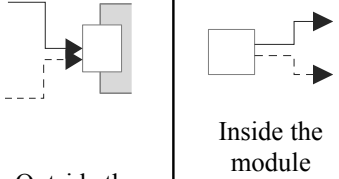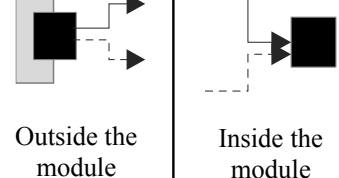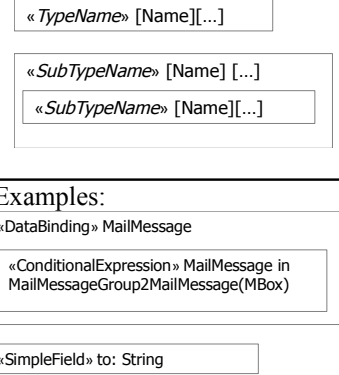
Table 1: Essential IFML Concepts

Concept	Meaning	IFML Notation	Example at implementation level
View Container	An element of the interface that comprises elements displaying content and supporting interaction and/or other ViewContainers.	Mailbox	Web page Window Pane.
XOR View Container	A ViewContainer comprising child ViewContainers that are displayed alternatively.	[XOR] MessageSearch	Tabbed panes in Java Frames in HTML.
Landmark View Container	A ViewContainer that is reachable from any other element of the user interface without having explicit incoming InteractionFlows.	[L] Message Writer	A logout link in HTML sites which is visible in every page.
Default View Container	A ViewContainer that will be presented by default to the user, when its enclosing container is accessed.	[D] Search	A welcome page.
View Component	An element of the interface that displays content or accepts input	Message List	An HTML list. A JavaScript image gallery. An input form.
Event	An occurrence that affects the state of the application	Name Name	
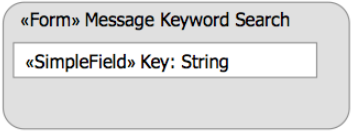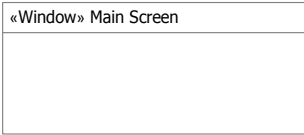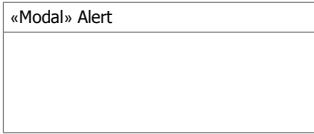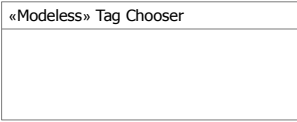
Table 1: Essential IFML Concepts

Concept	Meaning	IFML Notation	Example at implementation level
Action	A piece of business logic triggered by an event; it can be server side (the default) or client-side, denoted as [Client]	Send	A database update. The sending of an email. The spell checking of a text.
Navigation Flow	An input-output dependency. The source of the link has some output that is associated with the input of the target of the link	(solid arrow)	Sending and receiving of parameters in the HTTP request
Data Flow	Data passing between ViewComponents or Action as consequence of a previous user interaction.	(dashed arrow)	
Parameter	A typed and named value	Optionally shown. If necessary can be denoted as follows: «Parameter» State: String	HTTP query string parameters HTTP post parameters JavaScript variables and function parameters
Parameter Binding	Specification that an input parameter of a source is associated with an output parameter of a target	From → To From → To	
Parameter Binding Group	Set of ParameterBindings associated to an InteractionFlow (being it navigation or data flow)	«ParamBindingGroup» Title → AlbumTitle Year → AlbumYear	
Activation Expression	Boolean expression associated with a ViewElement, ViewComponentPart or Event: if true the element is enabled	«ActivationExpression» State = "Reply" or "ReplyToAll"	

Table 1: Essential IFML Concepts

Concept	Meaning	IFML Notation	Example at implementation level
Interaction Flow Expression	Determine which of the InteractionFlows are going to be followed as consequence of the occurrence of an Event.	«InteractionFlowExpression» If AlbumDetails selected then AlbumDetails Else Album details	Event triggered after selecting a given value in a ComboBox.
Module	Piece of user interface and its corresponding actions, which may be reused for improving IFML models maintainability	Module	
Input Port	An interaction point between a Module and its environment that collects InteractionFlows and parameters arriving at the module.	Outside the module / Inside the module	
Output Port	An interaction point between the Module and its environment that collects the InteractionFlows and parameters going out from the module.	Outside the module / Inside the module	
View Component Part	A part of a ViewComponent that may not live by its own. It can trigger Events and have outgoing and incoming InteractionFlows. A ViewComponentPart may contain other ViewComponentParts.	«*TypeName*» [Name][...] «*SubTypeName*» [Name] [...] «*SubTypeName*» [Name][...] Examples: «DataBinding» MailMessage «ConditionalExpression» MailMessage in MailMessageGroup2MailMessage(MBox) «SimpleField» to: String	Fields in a form

Table 2: Extension IFML Concepts

Concept Extension Examples	Meaning	IFML Notation	Example at implementation level
Select Event	Event denoting the selection of a single item of the user interface	(checkmark icon)	A selection of a row in a table.
Submit Event	Event that triggers a parameter passing between interaction flow elements	(submit icon)	A form submission button in HTML.
List	ViewComponent used to display a list of DataBinding instances	«List» MBoxList «DataBinding» MailMessageGroup	Table with rows of elements of the same kind.
Form	ViewComponent used to display a form that is composed of Fields	«Form» Message Keyword Search «SimpleField» Key: String	HTML form.
Details	ViewComponent used to display details of a specific DataBinding instance	«List» MBoxList «DataBinding» MailMessageGroup	
Window	A ViewContainer rendered as a window.	«Window» Main Screen	An HTML page or a desktop window.
Modal Window	A ViewContainer rendered in a new window that, when displayed, blocks interaction in all other previously active containers.	«Modal» Alert	A modal pop-up in HTML.
Modeless Window	A ViewContainer rendered in a new window, that when displayed, is superimposed over all other previously active containers, which remain active	«Modeless» Tag Chooser	A modeless pop-up in HTML.

8   IFML Metamodel

8.1   High-Level Description

The IFML metamodel is divided in three packages: the Core package, the Extension package and the DataTypes package. The Core package contains the concepts that build up the interaction infrastructure of the language in terms of InteractionFlowElements, InteractionFlows and Parameters. Core package concepts are extended by concrete concepts in the Extension package with more complex behaviors. The DataTypes package contains the custom data types defined by IFML.

The IFML metamodel uses the basic data types from the UML metamodel, specializes a number of UML metaclasses as the basis for IFML metaclasses, and presumes that the IFML DomainModel is represented in UML.

The high level description of the IFML metamodel given in the remainder of this subclause is structured into the following areas of concern:

- IFML Model
- Interaction Flow Model
- Interaction Flow Elements
- View Elements
- Events
- Specific Events and View Components
- Parameters
- Expressions
- ContentBinding

Subsequent subclauses provide detailed descriptions of the content of each of the three packages.

8.1.1   IFML Model
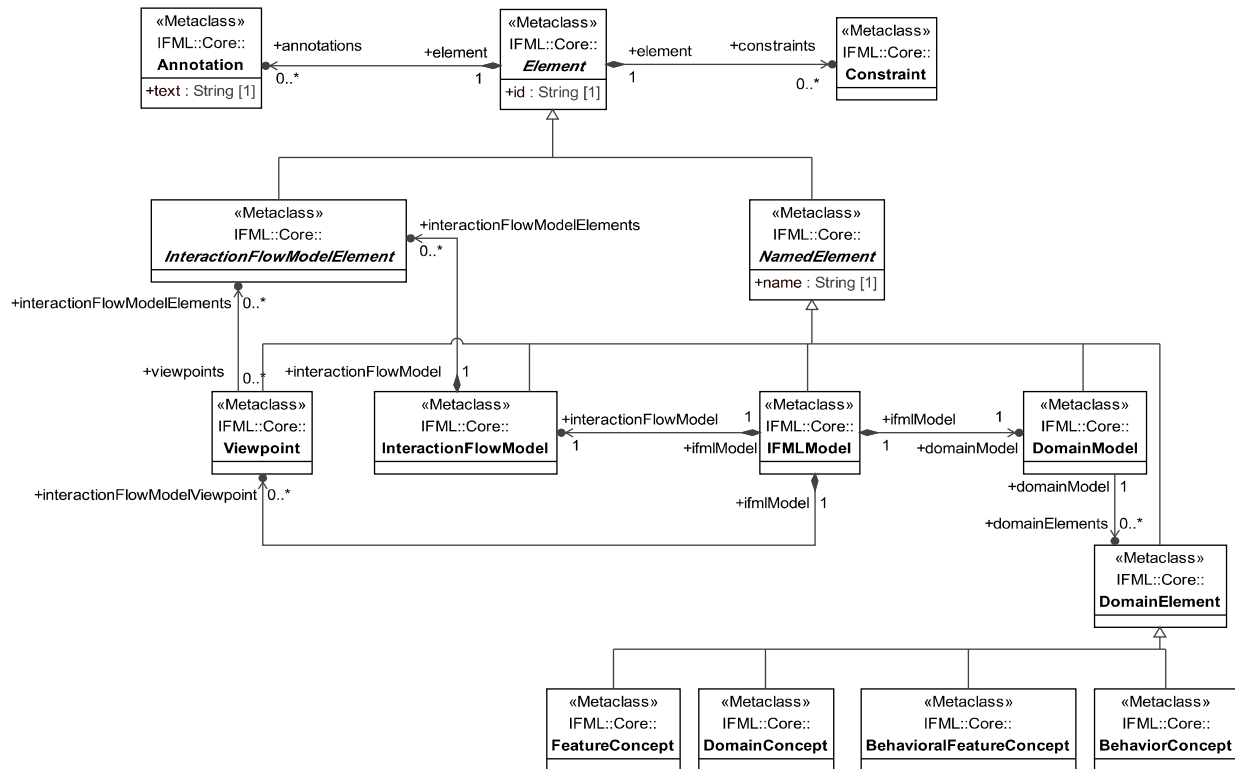
Figure 5: IFML Model

IFMLModel, as its name suggests, represents an IFML model and is the top-level container of all the rest of the model elements. It contains an InteractionFlowModel, a DomainModel and may optionally contain ViewPoints.

InteractionFlowModel is the user view of the whole application while ViewPoints present only specific aspects of the system by means of references to sets of InteractionFlowModelElements, which as a whole define a fully functional portion of the system. The purpose of a ViewPoint is to facilitate the comprehension of a complex system, to allow or disallow access to the system by a specific UserRole, or to show an adapted piece of the system to a specific context change.

InteractionFlowModelElement is an abstract class, which is the generalization of every element of an InteractionFlowModel.

DomainModel represents the business domain view of the application, i.e., the description of the content and behaviour that is dealt with (and referenced) within the InteractionFlowModel. The DomainModel comprises DomainElements, which are specialized as concepts, properties, behaviors and methods (DomainConcept, FeatureConcept, BehaviorConcept, and BehavioralFeatureConcept respectively).

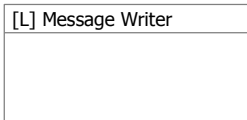NamedElement is an abstract class that specializes the Element class (the most general class in the model) denoting the elements that have a name. Besides IFMLModel, InteractionFlowModel, DomainModel, DomainElement and ViewPoint, NamedElement has other subclasses, which will be described in the contexts where they play a major role.

For any Element, Constraints and Comments can be specified. Constraints are an extension mechanism to the IFML, in the sense that they may constrain further, for a specific model, the existing IFML syntactical rules.

8.1.2   Interaction Flow Model
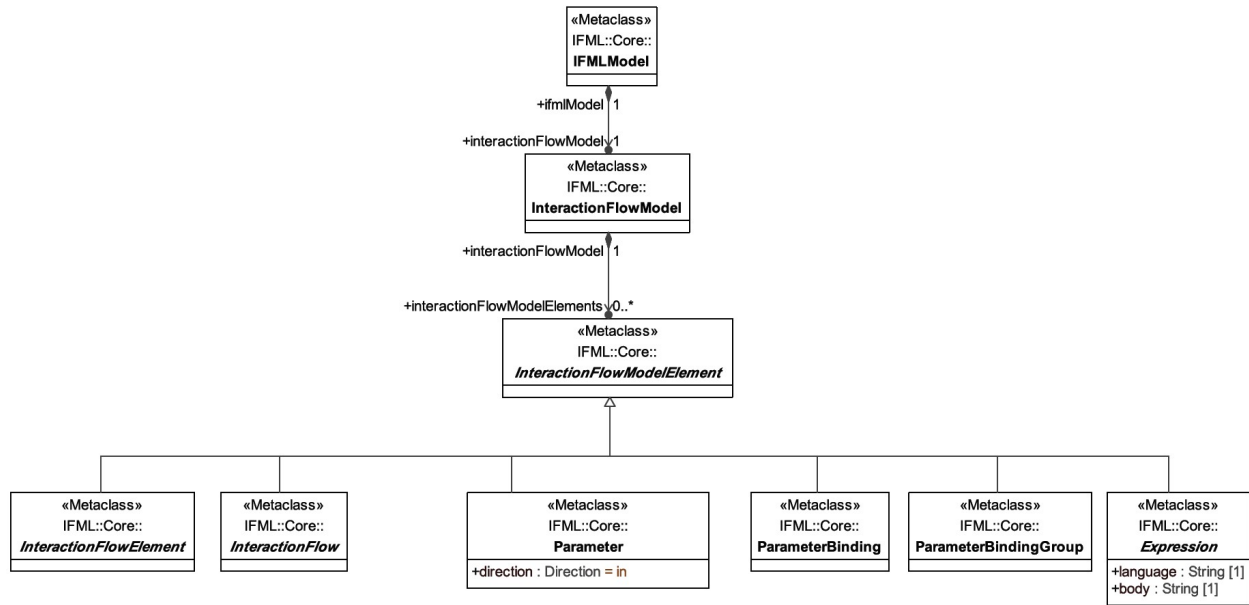
Figure 6: Interaction Flow Model

An InteractionFlowModel contains all the elements of the user view of the application represented by the InteractionFlowModelElement. InteractionFlowElement has seven direct subtypes: InteractionFlowElement, InteractionFlow, ParameterBindingGroup, ParameterBinding, Parameter, Expression and Module.

InteractionFlowElements are the building blocks of interactions. They represent the pieces of the system, which participate in interaction flows through InteractionFlow connections.

An InteractionFlow is a directed connection between two InteractionFlowElements. InteractionFlows may imply navigation along the user interface or only a transfer of information by carrying parameter values from one InteractionFlowElement to another.

A Parameter is a typed name, whose instances hold values. Parameters are held by InteractionFlowElements i.e. ViewElements, ViewComponentPart,s Ports and Actions. Parameters flow between InteractionFlowElements when Events are triggered. Considering the flow of a Parameter P from an InteractionFlowElement A to an InteractionFlowElement B, the Parameter P is considered as an output parameter of InteractionFlowElement A and as an input Parameter of InteractionFlowElement B.

ParameterBindings determine to which input Parameter of a target InteractionFlowElement an output Parameter of a source InteractionFlowElement is bound. ParameterBindings are in turn grouped into ParameterBindingGroups.

A Module is a fully functional collection of InteractionFlowModelElements, which may be reused for improving IFML maintainability. Modules may be replaced by other Modules or InteractionFlowElements with the same input and output parameters.

An Expression defines a statement that will evaluate in a given context to a single instance, a set of instances, or an empty result. An Expression is side effect free. Specific kinds of expression, such as boolean expressions, etc., are represented as specializations of Expression.

The interactions between all these elements will be described in the following subclauses.

8.1.3   Interaction Flow Elements
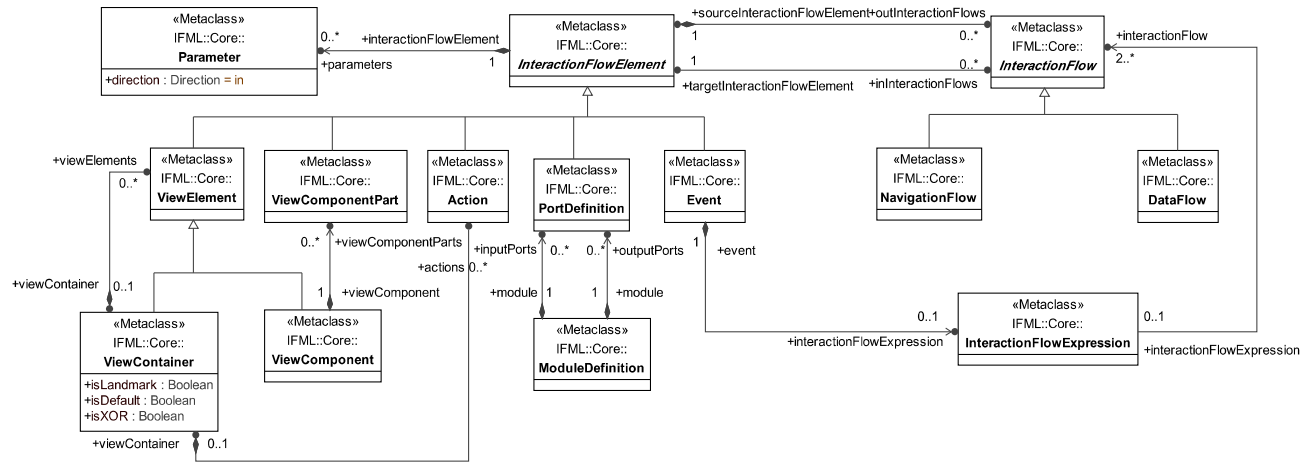
Figure 7: Interaction Flow Elements

The InteractionFlowElement is one of the key concepts of IFML. InteractionFlowElements represent pieces of the system, such as ViewElements, ViewComponentParts, Ports, Actions and Events, which participate in InteractionFlow connections. InteractionFlowElements contain Parameters, which usually flow between InteractionFlowElements as a consequence of ViewElementEvents (user events), ActionEvents or SystemEvents. InteractionFlowElements may have both incoming and outgoing interaction flows.

InteractionFlows are specialized into NavigationFlows and DataFlows. A NavigationFlow represents navigation or change of ViewElement focus, the triggering of an Action processing or a SystemEvent. NavigationFlows are followed when Events are triggered. NavigationFlows connect Events of ViewContainers, ViewComponents, ViewComponentParts or Actions with other InteractionFlowElements. When a NavigationFlow is followed Parameters may be passed from the source InteractionFlowElement to the target InteractionFlowElement through ParameterBindings. A DataFlow is a kind of InteractionFlow used for passing context information between InteractionFlowElements. DataFlows are triggered by NavigationFlows, causing Parameter passing but no navigation.

Events may be associated with an InteractionFlowExpression when they have more than one outgoing NavigationFlow. An InteractionFlowExpression is used to determine which of the InteractionFlows will be followed as a consequence of the occurrence of an Event. When an Event occurs and it has no InteractionFlowExpression, all the InteractionFlows associated with the event are followed.

ViewContainers can contain ViewElements (namely other ViewContainers or ViewComponents) or Actions.

8.1.4   View Elements
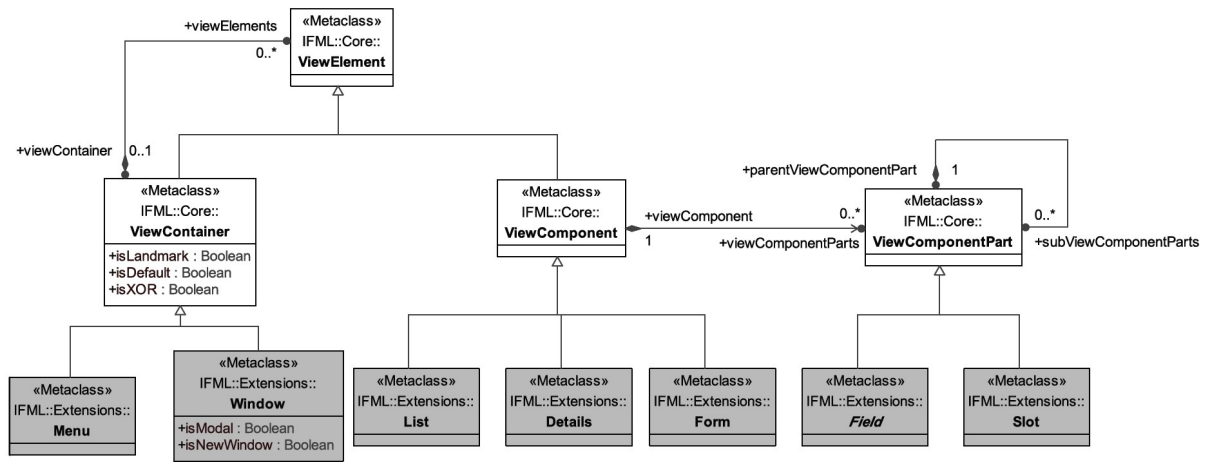
Figure 8: View Elements

The elements of an IFML model that are visible at the user interface level are called ViewElements, which are specialized in ViewContainers and ViewComponents. ViewContainers, like HTML pages or windows, are containers of other ViewContainers or ViewComponents, while ViewComponents are elements of the interface that display content or accept input from the user.

A ViewContainer may be landmark, XOR, and/or default. A landmark ViewContainer may be reached from any other ViewElement without the need of explicit InteractionFlows. ViewContainers that are not landmark may be reached only with an InteractionFlow.

In case a ViewContainer (the *enclosed* ViewContainer) is contained in another ViewContainer (the *enclosing* ViewContainer), like a frame in an HTML page, if it is marked as default, it will be presented to the user when its enclosing ViewContainer is accessed. Enclosing ViewContainers may be marked as XOR. In this case, the contained ViewElements of the current ViewContainer will be presented to the user only one at the time, as the user interacts with the system. A ViewContainer may be also opened as a new window. This new window may be"modal". Modal windows are meant to block any user interaction in all other previously active containers, until the new window is closed. Another special kind of ViewContainer is the Menu. Menus represent sets of interactive buttons or links that lead to some target container. Menus cannot contain subcontainers or ViewComponents.

ViewComponents exist only inside ViewContainers. A ViewComponent is an element of the interface that may have dynamic behavior, display content or accept input. It may correspond e.g. to a form, a data grid or an image gallery.

A ViewComponent may be build up from ViewComponentParts. A ViewComponentPart is a part of the ViewComponent that cannot live outside the context of a ViewComponent but may have Events and incoming and outgoing InteractionFlows. ViewComponentParts may hierarchically contain other ViewComponentParts. A ViewComponentPart may be visible or not at the level of the user interface depending on the kind of ViewComponentPart. For instance, a RichTextField is a ViewComponentPart that is visible to the user, may trigger events, and may receive values through parameter passing, while a Slot is a value placeholder that is not visible to the user.

The extension package includes concrete examples of ViewComponents such as List, Details, and Form and ViewComponentParts such as Fields and Slots. A List is for displaying, selecting and capturing lists of items of the same kind, Details is a component for displaying detailed information on a content element and a Form is for capturing user input through forms. All these elements will be described in detail in the following subclauses.

8.1.5　　　Parameters
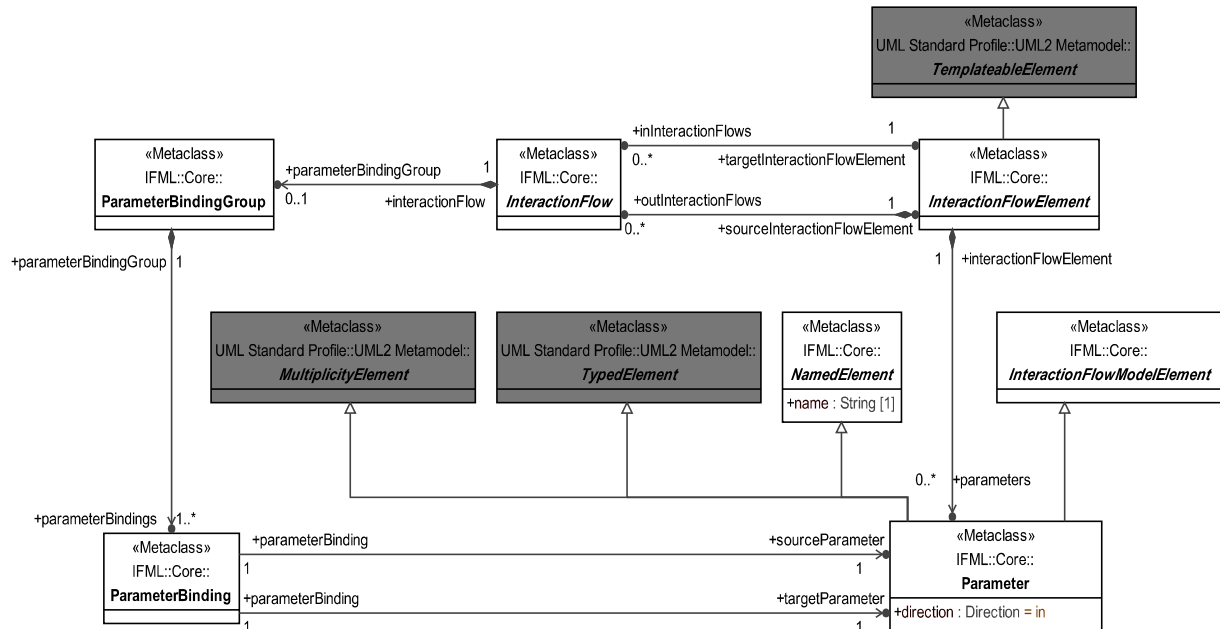
Figure 9: Parameters

A Parameter is a typed element with multiplicity, whose instances hold values. A Parameter may be of a primitive type or a complex type such as object or collection of objects. Parameters are held by InteractionFlowElements and flow between them when Events are triggered. Parameters may be mapped to a single element of the user interface i.e. ViewComponentPart or to a complex hierarchical set of ViewComponentParts.

Parameters have a direction property, which can be input (in), output (out) or input-output (inout). Default direction is input. An input Parameter allows an InteractionFlowElement to receive one or more values through an incoming NavigationFlow or DataFlow. An output Parameter allows an InteractionFlowElement to expose one or more values through an outgoing NavigationFlow or DataFlow. A input-output Parameter allows for both behaviours.

A ParameterBinding determines to which input Parameter of a target InteractionFlowElement an output Parameter of a source InteractionFlowElement is connected and thus how the parameter value will flow when an Event is triggered and the InteractionFlow is followed. ParameterBindings that flow together with an InteractionFlow are grouped by a ParameterBindingGroup, which in turn is related to the InteractionFlow.

8.1.6   Events
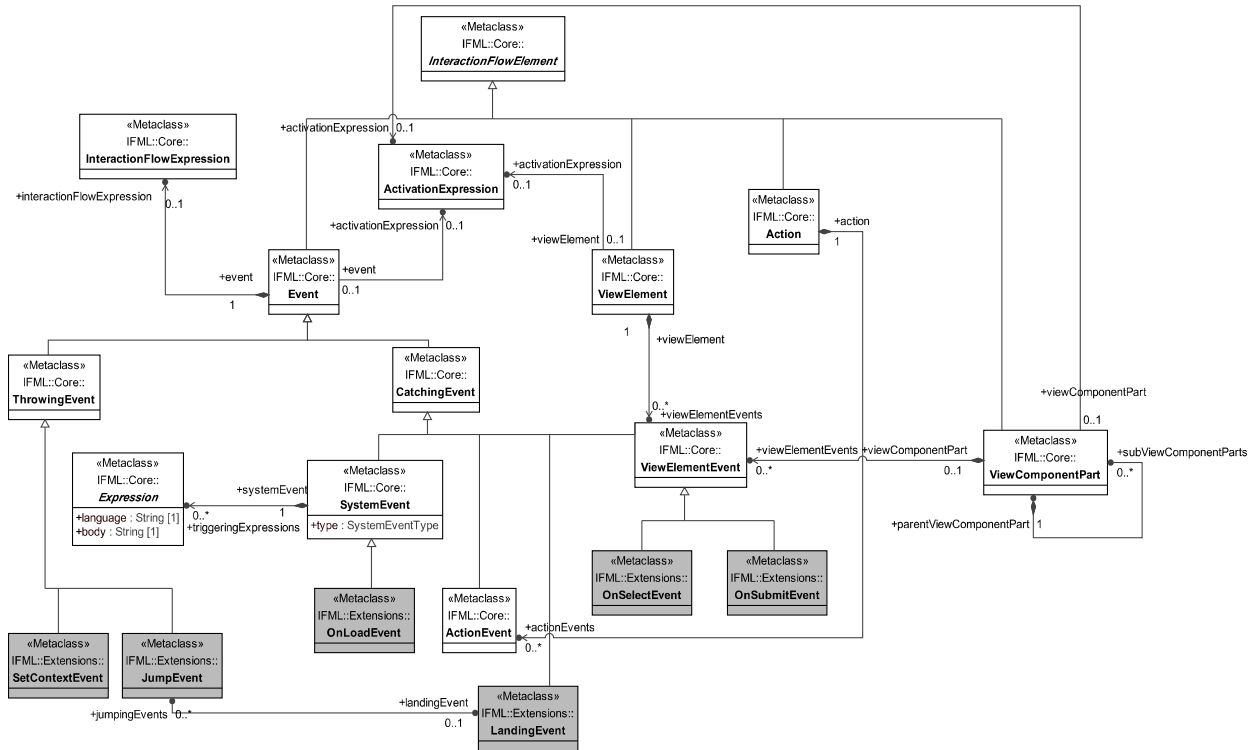
Figure 10: Events

Events are occurrences that can affect the state of the application, and they are a subtype of InteractionFlowElement. Events are classified in two main categories: CatchingEvents (events that are captured in the UI and that trigger a subsequent interface change) and ThrowingEvents (events that are generated by the UI). There are three types of CatchingEvents: ViewElementEvents, resulting from a user interaction (with specific subtypes OnSelectEvent and OnSubmitEvent), ActionEvents and SystemEvents (such as OnLoadEvent).

ViewElementEvents are owned by their related ViewElements. This means that ViewElements contain Events that allow a user to activate an interaction in the application, e.g., with the click on a hyperlink or on a button. ActionEvents are owned by their related Actions. An Action may trigger ActionEvents during its execution or when it terminates, normally or with an exception.

SystemEvents are stand-alone events, which are at the level of the InteractionFlowModel. SystemEvents result from an Action execution termination event or a triggeringExpression such as a specific moment in time, or special condition events such as a problem in the network connection.

CatchingEvents own a set of NavigationFlows. An InteractionFlowExpression is used to determine which of the NavigationFlows are followed as a consequence of the occurrence of an Event. When an Event occurs and it has no InteractionFlowExpression, all the NavigationFlows associated with the event are followed.

An Event may have an ActivationExpression that determines whether the Event is enabled or disabled. In practical terms, disabling a ViewElementEvent means, for example, that the UI element (e.g. a button) that triggers an InteractionFlow is disabled.

8.1.7   Expressions
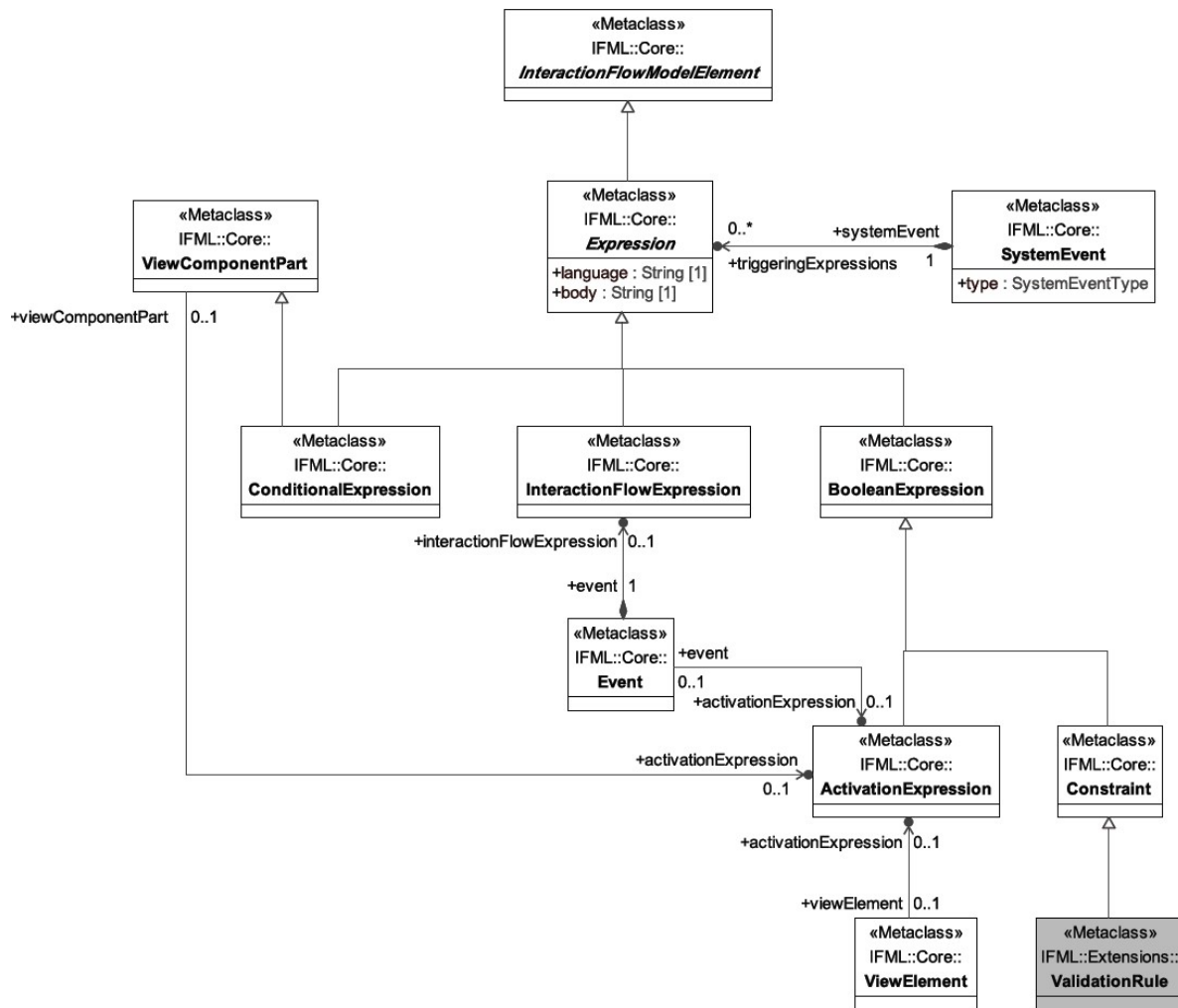
Figure 11: Expressions

An Expression defines a side-effect free statement that will evaluate in a given context to a single instance, a set of instances, or an empty result.

The subtypes of Expression are InteractionFlowExpression, BooleanExpression and ConditionalExpression.

An InteractionFlowExpression, as discussed in 8.1.3, determines which NavigationFlow should be followed, when more than one NavigationFlow comes out from an Event.

A ConditionalExpression is a ViewComponentPart representing predefined queries contained by DataBindings (see 8.1.8) that may be executed on them to obtain specific content information from the DomainModel. ConditionalExpressions can be defined only inside a DataBinding ViewComponentPart.

A BooleanExpression is an expression that evaluates to true or false. BooleanExpression has the specializations ActivationExpression and Constraint. An ActivationExpression determines if a ViewElement, ViewComponentPart or Event is enabled, and thus available to the user for interaction, while a Constraint restricts the behavior of any element.

 Thevalues used to evaluate the expressions are defined depending on the specific Expression type. For instance SystemEvent expressions may have as scope specific system condition values, the current date and time, etc., not modeled in IFML.

8.1.8   Content Binding
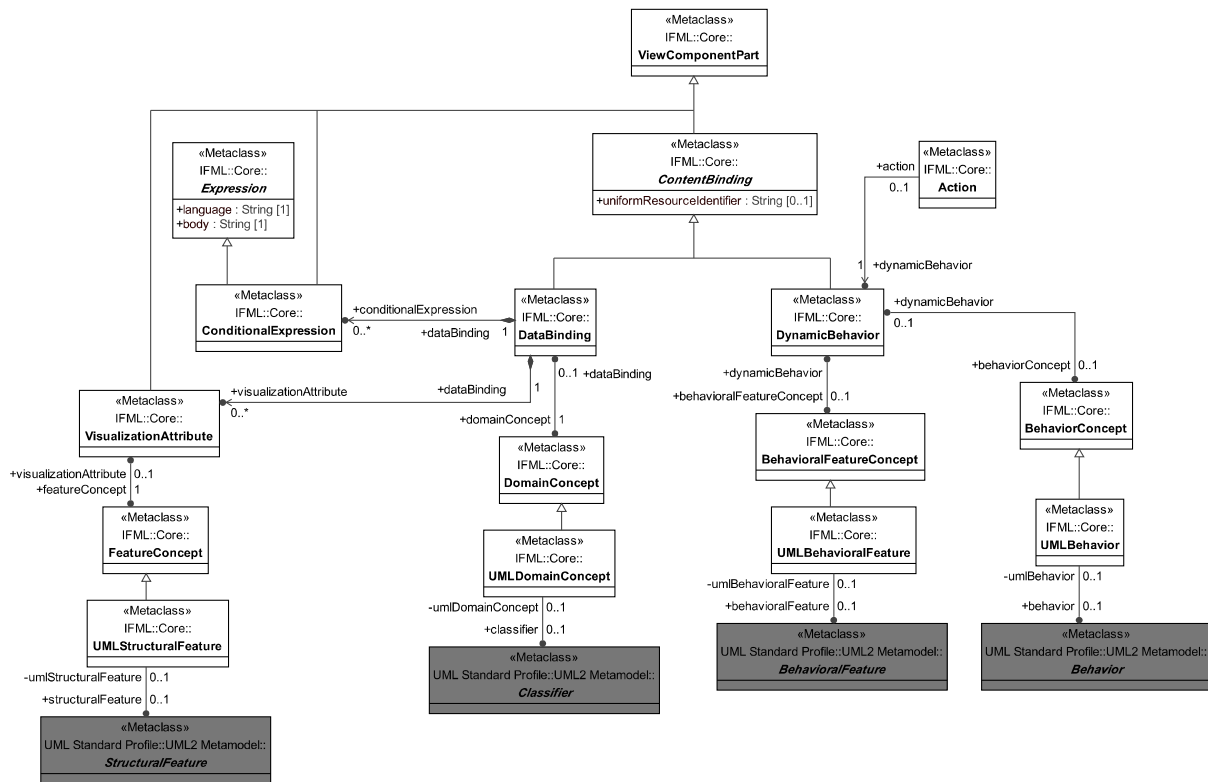
Figure 12: Content Binding

ViewComponents may retrieve content by means of the ContentBinding. ContentBinding represents any source of content. ContentBinding has as an optional attribute the URI of the resource from which the content may be obtained.

ContentBinding is specialized in two concepts, DataBinding and DynamicBehavior. A DataBinding references a DomainConcept (for instance, a Classifier in UML) that may represent an object, an XML file, a table in a database etc. A DataBinding is associated with a ConditionalExpression, which determines the specific content to be obtained from the content source. A DynamicBehavior represents a content access or business logic such as a service or method that returns a result after an invocation, as represented by a BehavioralFeature or Behavior in UML.

A DataBinding contains VisualizationAttributes used by ViewComponents to determine the features accessed from the DataBinding that may be shown to the user, such as a data base column or an XML element or attribute, as represented using UML StructuralFeatures.

8.1.9 Context
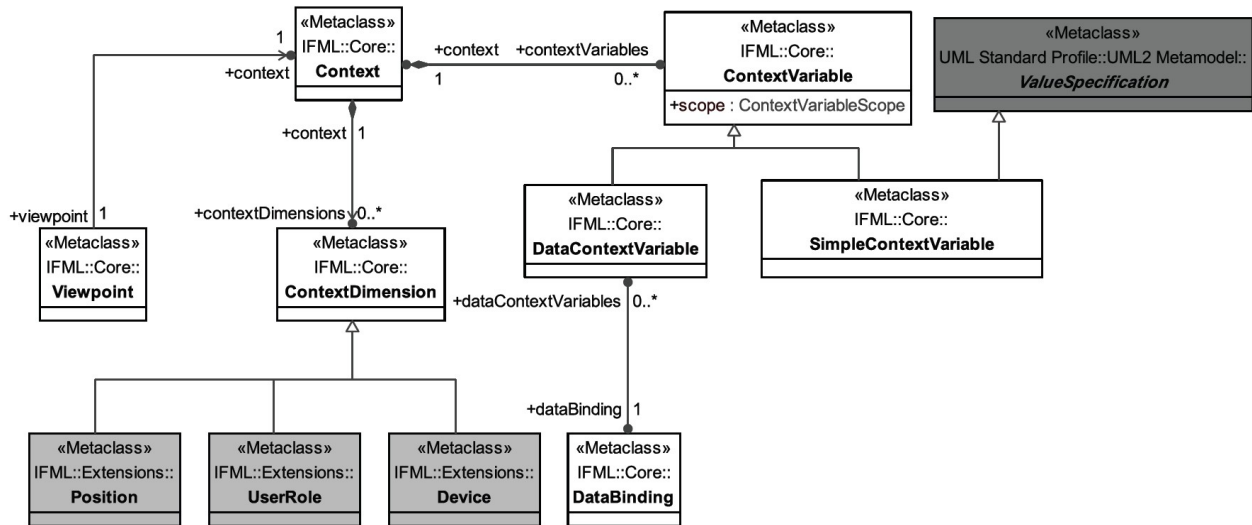
Figure 13: Context

The Context is a runtime aspect of the system that determines how the user interface should be configured and the content that it may display. The configuration and content of the user interface is determined by the ViewPoint, and thus Context is related to ViewPoint.

A Context has several dimension called ContextDimensions, which represent not only the user's id and preferences but also the interaction environment of the system. ContextDimension has the specializations UserRole, Device and Position. When the user context satisfies all the ContextDimensions, access is granted to the ViewElements of the ViewPoint and to the Events that may be triggered on them.

UserRole represents the profile that a user should have for satisfying the UserRole dimension.

A Device represent a specific kind of device for which the ViewPoint is configured. When a user accesses the application through such a device, the Device dimension is satisfied.

Position represents the location and orientation of the device for which a ViewPoint is configured. When the device the user uses for accessing the application reaches the given position or orientation, the Position dimension is satisfied.

ContextDimension may be specialized to represent other dimensions, such as user preferences, etc.

ContextVariables can be associated to the Context to store primitive values (SimpleContextVariable) or objects (DataContextVariable) that store the state of the system in the current context.

8.1.10   Specific ViewComponents
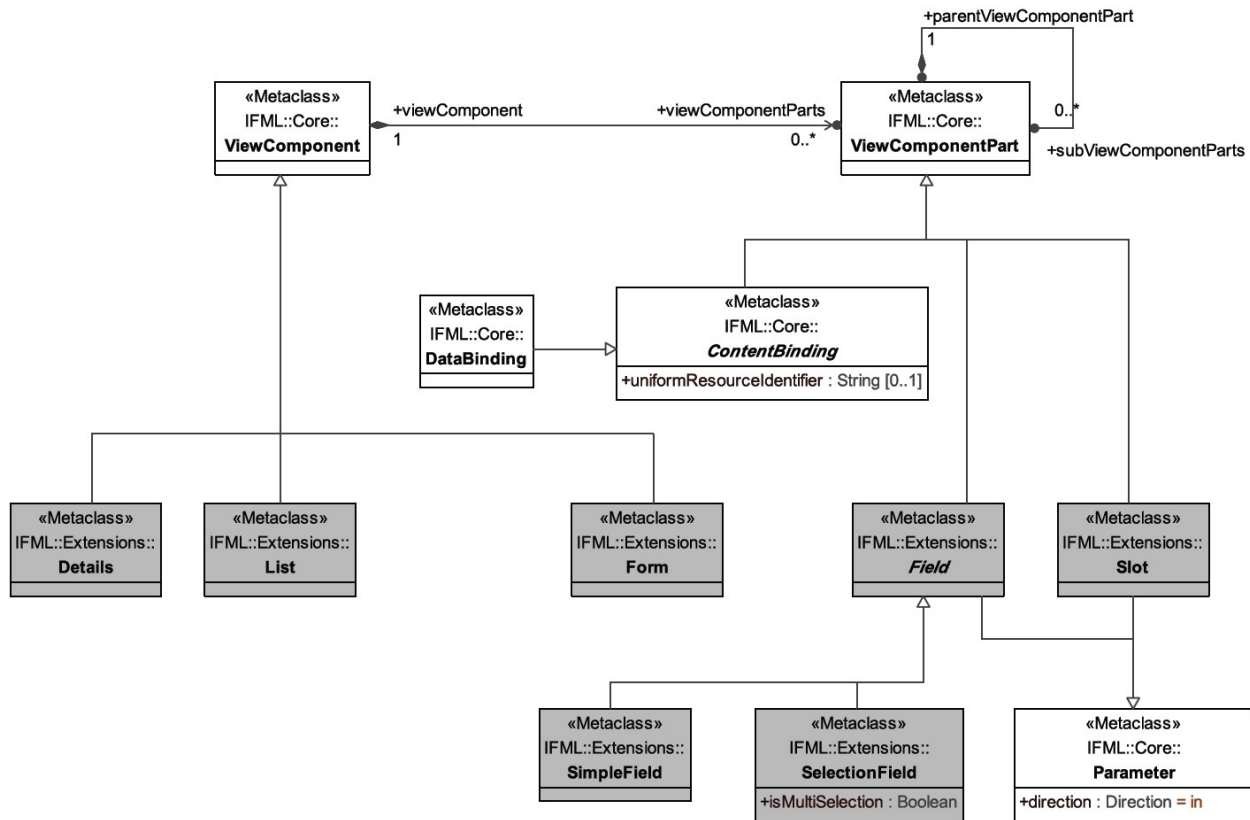
Figure 14: Specific ViewComponents

IFML includes a basic set of extensions to the core elements that exemplify how IFML may be extended.

List, Details and Form are specializations of ViewComponent (see 8.1.4). The List ViewComponent is used to display a list of DataBinding instances. When a List ViewComponent is associated with an Event, it means that each DataBinding instance displayed by the component may trigger that Event. The Event will in turn cause the passing of the parameter values mapped to the DataBinding instance to a target InteractionFlowElement. The Details ViewComponent is used to display detailed information of a DataBinding instance. When the Details ViewComponent is associated with an Event, the triggering of the Event will cause the passing of the Parameter values mapped to the DataBinding instance to a target InteractionFlowElement. The Form ViewComponent is used to display a form, which is composed of Fields that may display or capture content from the user. Fields have Slots that hold their value. When the Field is a SelectionField, its associated Slots contain the available selection options and the selected one. When the Field is a SimpleField, the Slot contains the Field value. A Slot value of a SimpleField and the Slots corresponding to the selected options of SelectionFields also behaves as Parameters in order to be passed to other ViewElements or Actions when an Event is triggered. Form ViewComponents have ValidationRules, which determine if a Field value is valid or not.

8.1.11   Modularization
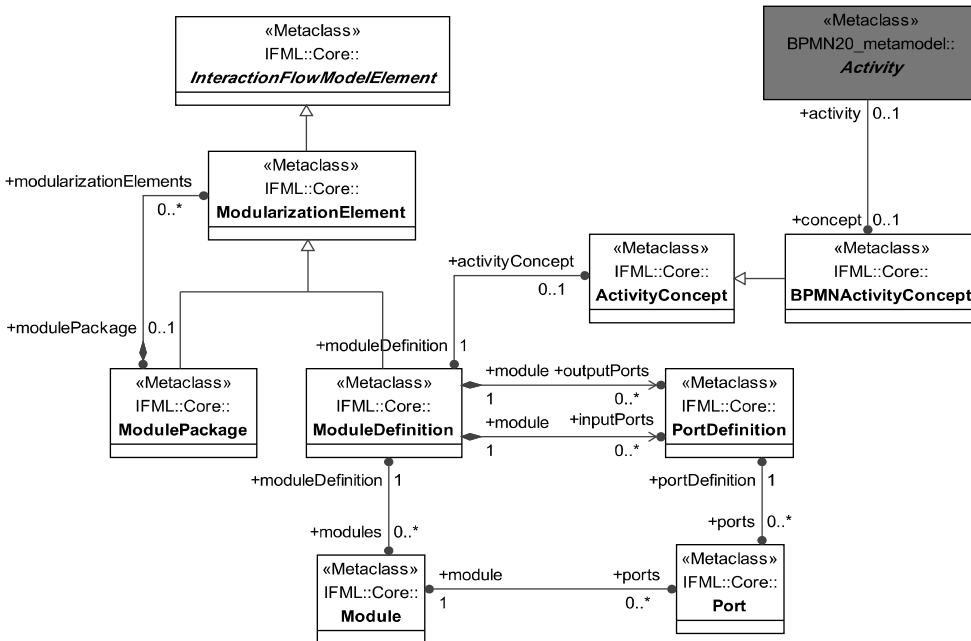
Figure 15: Modularization concepts in IFML

IFML includes a set of concepts that help improving reuse and modularization of models.

The main concept is ModuleDefinition, which allows the definition of an arbitrary piece of IFML model, which can be subsequently reused in models. ModuleDefinitions can be aggregated in a hierarchical structure of ModulePackages. ModuleDefinitions may comprise PortDefinitions.

PortDefinitions represent interaction points with a Module. PortDefinitions hold Parameters, for transferring values to and from the ModuleDefinition. An input PortDefinition has outgoing InteractionFlows to the inside of the Module. An output PortDefinition has incoming InteractionFlows from the inside of the Module.

Module is the concept that enables reuse of ModuleDefinitions. Module has a reference to the relevant ModuleDefinition and is associated with a set of Ports, which in turn reference the corresponding PortDefinitions. An input Port (i.e., a Port referencing an input PortDefinition) has incoming InteractionFlows from the outside of the Module, for receiving input Parameters. An output Port has outgoing InteractionFlows to the outside of the Module, for shipping output Parameters.

8.2   Package DataTypes

8.2.1   Enumeration Direction

Description

Enumeration specifying the different possible directions of parameters.

Literals

- in (input): An input Parameter allows an InteractionFlowElement to receive one or more values through an incoming NavigationFlow or DataFlow..

- inout (input-output): An output Parameter allows an InteractionFlowElement to expose one or more values through an outgoing NavigationFlow or DataFlow.

-

- out (output): **A** Parameter of kind input-output allows for both behaviours.

8.2.2   Enumeration ContextVariableScopeDescription

Description

Enumeration specifying the different scope levels for ContextVariables.

Literals

- ApplicationScope: Scope of the ContextVariable is the application

- SessionScope: Scope of the ContextVariable is the user session

- ViewContainerScope: Scope of the ContextVariable is the ViewContainer (for instance, the Web page or the window)

8.2.3   Enumeration SystemEventType

Description

Enumeration specifying the different system event types.

Literals

- actionCompletion: Kind of Event triggered by a business operation completion.

- specialCondition: System special condition event such as data base connection loss, network loss, etc.

- time: System event of time kind, such as absolute time event, periodic time event and time out event.

8.3   Package Core

8.3.1   Class Action

Abstract: No

Generalization:

- [InteractionFlowElement](#)
- NamedElement

Description

An Action is an InteractionFlowElement that represents a piece of business logic triggered by an Event. Actions may reference behavior models describing the actual business logic to be performed. Actions may trigger different Events called ActionEvents as the result of business logic computation termination or the occurrence of exceptions. Actions may reside on the server or on the client side. If no ActionEvent (and corresponding outgoing flow) is specified, IFML assumes as default an ActionEvent and NavigationFlow that lead back to the ViewComponent or

ViewContainer from which the navigation to the Action was launched.

Constraints

- actionsCannotCallActions
  self.actionEvent->forAll(e | e.navigationFlow->forAll(nf | not
  nf.targetInteractionFlowElement.oclIsTypeOf(IFML::Core::Action)))

Association Ends

- actionEvents[0..*]: ActionEvent - Events triggered by the Action.

- dynamicBehavior [1]: DynamicBehavior – The business logic to be carried out by the Action.

- viewContainer [0..1]: ViewContainer – The ViewContainer that contains the current Action.

8.3.2   Class ActionEvent

Abstract: No

Generalization:

- CatchingEvent

Description

An ActionEvent is an Event that may be triggered by an Action such as a normal termination event or exception event.

8.3.3   Class ActivationExpression

Abstract: No

Generalization:

- BooleanExpression

Description

ActivationExpressions are used by ViewElements, Events or ViewComponentParts to determine if they are enabled or not. An ActivationExpression is a BooleanExpression such that, if it evaluates to true, the element is active, otherwise the element is inactive. ActivationExpressions use Parameter values for the expression evaluation.

8.3.4   Class ActivityConcept

Abstract: No

Generalization:

- NamedElement

Description

ActivityConcepts are an abstract concept describing the idea of business action that can be implemented through an IFML ModuleDefinition. Typical examples could be BPMN Activities or Activities in UML Activity Diagrams.

Association Ends

- moduleDefinition [1]: ModuleDefinition – The ModuleDefinition that implements the business process ActivityConcept.

8.3.5   Class Annotation

Abstract: No

Description

An Annotation represents a comment, note, explanation, or other type of documentation that can be attached to any Element.

Attributes

- text: String - Annotation text.

8.3.6 Class BehaviorConcept

Abstract: No

Generalization:

- DomainElement

Description

BehaviorConcept represents the generic Behavior (for instance, a UML dynamic diagram) which can be referenced as DynamicBehavior in a ContentBinding.

Association Ends

- dynamicBehavior [0..1]: DynamicBehavior – Placeholder in a ContentBinding of the Behavior to be executed by the Action or ViewComponent.

8.3.7 Class BehavioralFeatureConcept

Abstract: No

Generalization:

- DomainElement

Description

BehavioralFeatureConcept represents the generic BehavioralFeature of a DomainModel element (for instance, a Class method) which can be referenced as DynamicBehavior in a ContentBinding.

Association Ends

- dynamicBehavior: DynamicBehavior – Placeholder in a ContentBinding of the BehavioralFeature to be executed by the Action or ViewComponent.

8.3.8 Class BooleanExpression

Abstract: No

Generalization:

- [Expression](#)

Description

A BooleanExpression is a kind of Expression that evaluates to true or false.

8.3.9 Class BPMNActivityConcept

Abstract: No

Generalization:

- ActivityConcept

Description

BPMNActivityConcepts are a specialization of ActivityConcept for representing the particular case of BPMN Activities.

Association Ends:

- Activity [0..1] : Activity – Reference to a BPMN Activity

8.3.10   Class CatchingEvent

Abstract: No

Generalization:

- Event

Description

A CatchingEvent is an occurrence that can affect the state of the application, by causing navigation and/or Parameter value passing between InteractionFlowElements. CatchingEvents may be produced by a user interaction (ViewElementEvent), by an action when it finishes its execution (ActionEvent), or by the system in the form of notifications (SystemEvent), or by a navigational jump (JumpEvent) in the model that reaches a LandingEvent.

8.3.11   Class ConditionalExpression

Abstract: No

Generalization:

- Expression
- ViewComponentPart

Description

A ConditionalExpression is a predefined query expression associated with a DataBinding. When a ConditionalExpression is present, the DataBinding is queried by applying the query in the ConditionalExpression for retrieving content.

8.3.12   Class Constraint

Abstract: No

Generalization:

- BooleanExpression

Description

A Constraint is a BooleanExpression that may be defined for any model Element in order to restrict its behavior.

8.3.13   Class ContentBinding

Abstract: Yes

Generalization:

- ViewComponentPart

Description

A ContentBinding allows the system to access a given source of content. A content source access may be done through a DataBinding or a DynamicBehavior of a DomainModel element.

Attributes

- uniformResourceIdentifier[0..1] : String - URI used to identify or locate the resource from which the

content may be obtained.

Constraints

- noViewElementEvent
    self.viewElementEvent -> isEmpty()

8.3.14   Class Context

Abstract: No

Generalization:

- Element

Description

The Context is a runtime aspect of the system that determines how the user interface should be configured. A Context has several dimensions that represent not only the user's identity and preferences, but also the interaction environment of the system. Context is composed of one or more ContextDimensions and may comprise ContextVariables.

Association Ends

- contextDimensions [0..*]: ContextDimension - ContextDimensions the user context must satisfy to have access to one or more Viewpoints.

- contextVariables [0..*]: ContextVariable – set of ContextVariables whose values are relevant for the current Context.

8.3.15   Class ContextDimension

Abstract: No

Generalization:

- NamedElement

Description

ContextDimensions are dimensions of the Context that represent not only the user's identity and preferences, but also the interaction environment of the system. ContextDimension has the specializations UserRole, Device, and Position. When the user context satisfies all required ContextDimensions, access is granted to the ViewElements of the Viewpoint and to the Events that may be triggered on them.

8.3.16   Class ContextVariable

Abstract: No

Generalization:

- NamedElement

Description

ContextVariable is a name-value pair that allows to store information associated to the current Context. It can be a SimpleContextVariable, storing a primitive type value, or a DataContextVariable, referencing a DataBinding.

Attributes

- scope : ContextVariableScope – scope of the ContextVariable.

Association Ends

•   context [1] : Context – Context within which the ContextVariable is relevant.

8.3.17   Class DataBinding

Abstract: No

Generalization:

•   [ContentBinding](#)

Description

DataBinding represents the binding of the system with an instance of an element of the DomainModel such as a table, an object, an XML file etc.

Association Ends

•   domainConcept [1]: domainConcept – A concept specifying the data structure to which the ViewComponent is bound, such as a UML class, a table in a relational data base or an XML file.

•   conditionalExpressions [0..*]: [ConditionalExpression](#) - ConditionalExpressions that determine how to access the content.

•   visualizationAttributes[0..*]: [VisualizationAttribute](#) - VisualizationAttributes that determine the StructuralFeatures that should be shown to the user, such as a data base column or an XML element or attribute.

•   dataContextVariables [0..*]: DataContextVariable – reference to the ContextVariable that makes use of the DataBinding.

8.3.18   Class DataContextVariable

Abstract: No

Generalization:

•   ContextVariable

Description

DataContextVariable allows to associate a DataBinding to the current Context.

Association Ends

•   dataBinding [1]: DataBinding - Reference to the DomainModel concept used as ContextVariable.

8.3.19   Class DataFlow

Abstract: No

Generalization:

•   [InteractionFlow](#)

Description

A DataFlow is a kind of InteractionFlow used for passing context information between InteractionFlowElements. DataFlows are triggered by NavigationFlows causing Parameter passing but no navigation. DataFlows are triggered any time a parameter is available in output from the source InteractionFlowElement and transfer the Parameter values to the target InteractionFlowElement. The target of a DataFlow cannot be an Event.

Constraints

- targetNotInstanceOfEvent
  not self.targetInteractionFlowElement.oclIsTypeOf(IFML::Core::Event)

8.3.20   Class DomainConcept

Abstract: No
Generalization:

- DomainElement

Description

The DomainConcept represents a generic concept, class, entity of the DomainModel, which can be referenced in a DataBinding. Its purpose is to allow extensibility in terms of concepts from different modeling languages representing the DomainModel.

Association Ends

- dataBinding [0..1]: DataBinding - Reference to the DataBinding in a ViewComponent that uses the current DomainConcept.

8.3.21   Class DomainElement

Abstract: No
Generalization:

- NamedElement

Description

The DomainElement represents a generic element of the DomainModel, which can describe a concept, a property, a behavior, or a behavioral feature. It is a generic representative for classes DomainConcept, FeatureConcept, BehavioralFeatureConcept, BehaviorConcept.

Association Ends

- domainModel [1]: DomainModel - Reference to the DomainModel comprising the DomainElement.

8.3.22   Class DomainModel

Abstract: No
Generalization:

- NamedElement

Description

The DomainModel is a model that contains content elements that ViewComponents may access to retrieve information, process it, and show it to the user. The DomainModel also stores information captured from the user. The DomainModel is presumed to be represented in UML and therefore consists of a set of UML model elements .

Association Ends

- elements[0..*]: DomainElement - References to the elements of the DomainModel.

8.3.23 Class DynamicBehavior

Abstract: No

Generalization:

- ContentBinding

Description

DynamicBehavior represents the binding of the system with a service or operation, which may be invoked in order to carry out business logic or return content.

Constraints

- eitherBehavioralFeatureOrBehavior
  self.behavioralFeature -> notEmpty() xor self.behavior -> notEmpty()

Association Ends

- behavioralFeatureConcept [0..1]: behavioralFeatureConcept - BehavioralFeatureConcept representing a procedure, method, function etc, that may be invoked by a ViewComponent to carry out business logic or obtain content.

- behaviorConcept [0..1]: BehaviorConcept - Representing a procedure, method, function etc, that may be invoked by a ViewComponent to carry out business logic or obtain content.

8.3.24 Class Element

Abstract: Yes

Description

Element is the base class for the representation of all model elements in an IFML model.

Attributes

- id: String [1] - String for unequivocally identifying a model element.

Association Ends

- annotations [0..*]: Annotation - Annotations, comments, tags, etc., owned by the Element.

- constraints [0..*]: Constraint - Constraints applied to the Element.

8.3.25 Class Event

Abstract: No

Generalization:

- InteractionFlowElement

Description

An Event is an occurrence that can affect the state of the application. Events can be ThrowingEvent (events that are thrown by the modeled interaction) or CatchingEvent (events that are captured by the modeled interaction and used as triggers causing navigation and/or Parameter value passing between InteractionFlowElements.

Constraints

- onlyOneInAndOutFlow
  self.outInteractionFlow -> size() <= 1 and self.inInteractionFlow -> size() <= 1

Association Ends

- activationExpression [0..1]: <u>ActivationExpression</u> - Reference to an ActivationExpression whose evaluation result determines if the Event is active or inactive. If no ActivationExpression is given, the default is that the Event is active.

- interactionFlowExpression [0..1]: <u>InteractionFlowExpression</u> - InteractionFlowExpression determining the InteractionFlows to be followed after the occurrence of the Event.

- navigationFlows [0..*]: <u>NavigationFlow</u> - NavigationFlows triggered by the Event.

8.3.26   Class Expression

Abstract: Yes

Generalization:

- <u>InteractionFlowModelElement</u>

Description

An Expression is an element that, evaluates to a single instance, a set of instances, or an empty result. An Expression must be side effect free. Specific expression types, such as BooleanExpression, etc., specialize this concept.

Attributes

- body[1]: String - Code of the Expression.

- language[1]: String - Language in which the Expression is written, e.g. OCL, Java, etc.

8.3.27   Class FeatureConcept

Abstract: No

Generalization:

- <u>Do</u>mainElement

Description

FeatureConcept represents the generic attribute or property of a DomainModel element, which can be referenced as VisualizationAttribute.

Association Ends

- visualizationAttribute [0..1]: VisualizationAttribute – Placeholder in a DataBinding of the FeatureConcept to be visualized.

8.3.28   Class IFMLModel

Abstract: No

Generalization:

- <u>NamedElement</u>

Description

An IFMLModel is the top-level container of all other elements in an IFML model. All model elements are grouped into two submodels, the InteractionFlowModel and the DomainModel. An IFMLModel may also contain a number of Viewpoints of the InteractionFlowModel.

Association Ends

- DomainModel [1]: <u>DomainModel</u> - Model describing the domain concepts and behaviors used in the

user interaction modeled by the IFML model.

- interactionFlowModel [1]: <u>InteractionFlowModel</u> - The complete model that describes the interaction of the user with the system.

- interactionFlowModelViewpoints [0..*]: <u>Viewpoint</u> - Viewpoints of the InteractionFlowModel.

8.3.29   Class InteractionFlow

Abstract: Yes

Generalization:

- <u>InteractionFlowModelElement</u>

Description

An InteractionFlow is a directed connection between ViewElements or ViewElements and Actions, which enables communication between them by means of Parameter passing. InteractionFlows are divided into NavigationFlows and DataFlows. NavigationFlows cause navigation or change of focus to the target element and Parameter passing, while DataFlows cause only Parameter passing to the target element.

Association Ends

- parameterBindingGroup [0..1]: <u>ParameterBindingGroup</u> - Group of parameters that are passed to the target interaction flow element by following the InteractionFlow.

- targetInteractionFlowElement [1]: <u>InteractionFlowElement</u> - Target InteractionFlowElement of the InteractionFlow.

- sourceInteractionFlowElement [1]: <u>InteractionFlowElement</u> - Source InteractionFlowElement of the InteractionFlow.

8.3.30   Class InteractionFlowElement

Abstract: Yes

Generalization:

- <u>InteractionFlowModelElement</u>

- <u>NamedElement</u>

- TemplateableElement

Description

InteractionFlowElements represent pieces of the system such as Actions, Events, ViewElements, and ViewComponentParts, which participate in user interaction flows through InteractionFlow connections. Usually there is a flow of Parameter values between InteractionFlowElements as a consequence of user, action, or system events.

Association Ends

- inInteractionFlows [0..*]: <u>InteractionFlow</u> - Incoming InteractionFlows.

- outInteractionFlows [0..*]: <u>InteractionFlow</u> - Outgoing InteractionFlows.

- parameters [0..*]: <u>Parameter</u> - Parameters contained by the InteractionFlowElement.

8.3.31   Class InteractionFlowExpression

Abstract: No

Generalization:

- [Expression](#)

Description

An InteractionFlowExpression is used to determine which of the InteractionFlows will be followed as a consequence of the occurrence of an Event. When an Event occurs and it has no InteractionFlowExpression, all the InteractionFlows associated with the event are followed. At least two InteractionFlows must be associated with an InteractionFlowExpression. An InteractionFlowExpression uses the ViewElement Parameter values and the InteractionFlows for the evaluation of the expression.

Association Ends

- interactionFlows [2..*]: [InteractionFlow](#) - InteractionsFlows for which the expression is evaluated.

8.3.32 Class InteractionFlowModel

Abstract: No

Generalization:

- [NamedElement](#)

Description

An InteractionFlowModel aggregates all the elements modeling interaction with the user.

Association Ends

- interactionFlowModelElements [0..*]: [InteractionFlowModelElement](#) - Elements of the InteractionFlowModel.

8.3.33 Class InteractionFlowModelElement

Abstract: Yes

Generalization:

- [Element](#)

Description

An InteractionFlowModelElement is the top-level class that generalizes all the elements that are part of an InteractionFlowModel.

8.3.34 Class ModularizationElement

Abstract: No

Generalization:

- [InteractionFlowModelElement](#)
- NamedElement

Description

A ModularizationElement is an abstract concept that represents both ModulePackages and ModuleDefinitions.

Association Ends

- modulePackage [0..1]: ModulePackage – The ModulePackage containing the ModularizationElement

8.3.35 Class Module

Abstract: No

Generalization:

- [InteractionFlowModelElement](#)
- NamedElement

Description

A Module is a named reference to a ModuleDefinition, which allows reuse of the model part specified in the ModuleDefinition. Module has a reference to the relevant ModuleDefinition and may be associated with a set of Ports, which in turn reference the corresponding PortDefinitions. For every PortDefinition in the ModuleDefinition there shall be 0 or 1 Ports in each corresponding Module. An input Port (i.e., a Port referencing an input PortDefinition) has incoming InteractionFlows from the outside of the Module, for receiving input Parameters. An output Port has outgoing InteractionFlows to the outside of the Module, for shipping output Parameters.

Constraints

- onlyOnePortPerPortDefinition
  self.moduleDefinition.portDefinitions -> forAll(pd | pd.ports -> select(p|p.module = self) -> size() = 1)

Association Ends

- ports[0..*]: [Port](#) - Ports that collect InteractionFlows and Parameters incoming or outgoing from the Module.

- ModuleDefinition [1]: ModuleDefinition – The ModuleDefinition that is instantiated by the current Module.

8.3.36   Class ModuleDefinition

Abstract: No

Generalization:

- [M](#)odularizationElement

Description

A ModuleDefinition is a fully functional collection of user InteractionFlowModelElements and their corresponding Actions, which may be reused for improving IFML model maintainability. ModuleDefinitions can be aggregated in a hierarchical structure of ModulePackages. ModuleDefinitions may comprise PortDefinitions. A ModuleDefinition receives Parameter values from outside and provides Parameter values to the outside. ModuleDefinitions exchange Parameters by mean of input and output PortDefinitions. InteractionFlowModelElements contained in a Module may not be shared or referenced by other Modules or by the main InteractionFlowModel. A ModuleDefinition may comprise a reference to a BPMN Activity (meaning that the Module implements that Activity). Reuse of ModuleDefinition is obtained by adding Modules referencing that ModuleDefinition in IFML models.

Association Ends

- inputPorts [0..*]: [Port](#)Definition - Ports that distributes InteractionFlows and Parameters coming into the Module.

- interactionFlowModelElements [1..*]: [InteractionFlowModelElement](#) - InteractionFlowModelElements contained by the Module.

- outputPorts [0..*]: [Port](#)Definition - Ports that collect the InteractionFlows and Parameters going out from the Module.

- modules [0..*]: Module – The set of Modules that are defined in the IFML model and reference the current ModuleDefinition.

- activityConcept [0..1]: ActivityConcept – Reference to a process activity (e.g., a BPMN Activity). If

present, the current module is describes the technical implementation of the process activity.

8.3.37 Class ModulePackage

Abstract: No

Generalization:

- ModularizationElement

Description

A ModulePackage is a container of ModuleDefinitions. ModulePackages can be nested in arbitrarily deep hierarchical structure.

Association Ends

- inputPorts [1..*]: Port - Ports that distributes InteractionFlows and Parameters coming into the Module.

- modularizationElements [0..*]: ModularizationElement – Set of ModularizationElements contained in the current ModulePackage

8.3.38 Class NamedElement

Abstract: Yes

Generalization:

- Element

Description

A NamedElement is an Element that requires a name for easy visual identification in diagrams or for being handled as a named variables in a concrete textual syntax.

Attributes

- name[1]: String - Element name.

8.3.39 Class NavigationFlow

Abstract: No

Generalization:

- InteractionFlow

Description

A NavigationFlow represents navigation or change of ViewElement focus, the triggering of Action processing, or a SystemEvent. NavigationFlows are followed when Events are triggered. NavigationFlows connect Events of ViewContainers, ViewComponents, ViewComponentParts, or Actions with other InteractionFlowElements. When a NavigationFlow is followed, Parameters may be passed from the container of the source Event to the target InteractionFlowElement through ParameterBindings. When a NavigationFlow is triggered, a corresponding set of DataFlows may be triggered, at the purpose of carrying further parameters to the target InteractionFlowElement. The DataFlows that are triggered are all the ones having some parameter values available as an effect of the last interface status change.

8.3.40 Class Parameter

Abstract: No

Generalization:

- InteractionFlowModelElement
- MultiplicityElement
- TypedElement
- NamedElement

 Description

A Parameter is a typed name, whose instances hold values. Parameters are held by InteractionFlowElements, i.e., ViewElements, ViewComponentParts, Ports, and Actions. Parameters flow between InteractionFlowElements when Events are triggered. Parameters may correspond to elements of the user interface (for instance, fields in a form), determining whether the element of the user interface is read-only or modifiable. Parameters have a direction property, which can be input (in), output (out) or input-output (inout). Default direction is input.

The scope of a Parameter (i.e., the model space where it can be used or referenced) is the InteractionFlowElement that holds the Parameter, plus the incoming and outgoing InteractionFlows. This means that: if the parameter is held by a ViewComponent, it can be referenced only within the ViewComponent itself and the contained ViewComponentParts (plus the incoming and outgoing InteractionFlows); if the parameter is held by a ViewContainer, it can be referenced within the ViewContainer itself, and within the contained ViewContainers, ViewComponents, and ViewComponentParts (plus the incoming and outgoing InteractionFlows).

A Parameter can have a default value.**Attributes**

- direction: Direction - Determines if the parameter direction is input, output or input-output.

- defaultValue: Expression – default value of the parameter, calculated through the specified expression.

8.3.41   Class ParameterBinding

Abstract: No

Generalization:

- InteractionFlowModelElement

Description

A ParameterBinding determines how data flow between components. A ParameterBinding connects an output Parameter of a source InteractionFlowElement with an input Parameter of a target InteractionFlowElement. When an Event is triggered, InteractionFlows are followed and Parameter values flow from source InteractionFlowElements to target InteractionFlowElements, according to how they have been bound.

Association Ends

- sourceParameter [1]: Parameter – Output Parameter of the source InteractionFlowElement that participates in the ParameterBinding.

- targetParameter [1]: Parameter – Input Parameter of the target InteractionFlowElement that participates in the ParameterBinding.

8.3.42   Class ParameterBindingGroup

Abstract: No

Generalization:

- InteractionFlowModelElement

Description

A ParameterBindingGroup aggregates all the ParameterBindings of an InteractionFlow.

Association Ends

*   parameterBindings [1..*]: ParameterBinding - The ParameterBindings composing the ParameterBindingGroup.

8.3.43   Class Port

Abstract: No

Generalization:

*   InteractionFlowElement

Description

A Port is an interaction point between a Module and the surrounding model within which it is defined.

Module is associated with a set of Ports, which in turn reference the corresponding PortDefinitions. An input Port (i.e., a Port referencing an input PortDefinition) has incoming InteractionFlows from the outside of the Module, for receiving input Parameters. An output Port has outgoing InteractionFlows to the outside of the Module, for shipping output Parameters.

Association Ends

*   PortDefinition [1]: PortDefinition – Reference to the PortDefinition that defines the interface of the current Port
*   module [1]: Module - Module that contains the current Port

8.3.44   Class PortDefinition

Abstract: No

Generalization:

*   InteractionFlowElement

Description

PortDefinitions represent interaction points with a ModuleDefinition. They are defined within a ModuleDefinition. They hold Parameters, for transferring values to and from the ModuleDefinition. An input PortDefinition has outgoing InteractionFlows to the inside of the Module. An output PortDefinition has incoming InteractionFlows from the inside of the Module. Modules that reference a ModuleDefinition may comprise Ports, which in turn reference the corresponding PortDefinitions.

Association Ends

*   ports [0..*]: Port – Set of Ports referencing the current PortDefinition in some Modules implementing the ModuleDefinition within which the current PortDefinition is defined.

8.3.45   Class SimpleContextVariable

Abstract: No

Generalization:

*   ValueSpecification

Description

SimpleContextVariable is a typed name-value pair that can be associated to the current Context. Allowed types are

the primitive ones.

8.3.46   Class SystemEvent

Abstract: No

Generalization:

- Catching[Event](#)

Description

A SystemEvent is an Event produced by the system, which triggers a computation reflected in the user interface. Examples of SystemEvents are time events, which are triggered after an elapsed frame of time, or system special conditions events, such as a database connection loss event.

Attributes

- type: SystemEventType - Determines the kind of SystemEvent.

Association Ends

- triggeringExpressions [0..*]: [Expression](#) - Expressions that determines when or under what conditions the SystemEvent should be triggered.

8.3.47   Class ThrowingEvent

Abstract: No

Generalization:

- Event

Description

A ThrowingEvent is an occurrence of event that is generated by the modeled application. Event occurences generated by ThrowingEvent can be captured by CatchingEvents.

8.3.48   Class UMLBehavior

Abstract: No

Generalization:

- BehaviorConcept

Description

UMLBehavior represents a Behavior specified in UML (that is, a UML dynamic diagram) which can be referenced as DynamicBehavior in a ContentBinding.

Association Ends

- behavior [0..1]: UML::Behavior – UML Behavior to be executed by the Action or ViewComponent.

8.3.49   Class UMLBehavioralFeature

Abstract: No

Generalization:

- BehavioralFeatureConcept

Description

UMLBehavioralFeature represents a BehavioralFeature specified in UML (typically, a UML method in a Class) which can be referenced as DynamicBehavior in a ContentBinding.

Association Ends

- behavioralFeature [0..1]: UML::BehavioralFeature – UML BehavioralFeature to be executed by the Action or ViewComponent.

8.3.50   Class UMLStructuralFeature

Abstract: No

Generalization:

- FeatureConcept

Description

The UMLStructuralFeature is a specific FeatureConcept referring to a UML StructuralFeature.

Association Ends

- structuralFeature [0..1]: UML::StructuralFeature - Reference to the UML element of the DomainModel.

8.3.51   Class UMLDomainConcept

Abstract: No

Generalization:

- DomainConcept

Description

The UMLDomainConcept is a specific DomainConcept referring to a UML Classifier.

Association Ends

- classifier [0..1]: UML::Classifier - Reference to the UML Classifier of the DomainModel that will be connected to a ViewElement through a DataBinding.

8.3.52   Class ViewComponent

Abstract: No

Generalization:

- ViewElement

Description

A ViewComponent is an element of the user interface that displays content or accepts input. A ViewComponent may be bound to a ContentBinding through its association with ViewComponentPart.

Association Ends

- viewComponentParts [0..*]: ViewComponentPart - Parts of the ViewComponent.

8.3.53   Class ViewComponentPart

Abstract: No

Generalization:

- InteractionFlowElement

Description

A ViewComponentPart is an InteractionFlowElement that may not live outside the context of a ViewComponent. A ViewComponentPart may trigger Events and have incoming and outgoing InteractionFlows.

Association Ends

- activationExpression [0..1]: <u>ActivationExpression</u> - Reference to an ActivationExpression whose evaluation result determines whether the ViewComponentPart is active or inactive. If no ActivationExpression is given, by default the ViewComponent is active.

- subViewComponentParts [0..*]: <u>ViewComponentPart</u> - Nested ViewComponentParts.

- viewElementEvents [0..*]: <u>ViewElementEvent</u> - Events that this ViewComponentPart may trigger.

- parentViewComponentPart [1]: <u>ViewComponentPart</u> - Parent ViewComponentPart.

8.3.54   Class ViewContainer

Abstract: No

Generalization:

- <u>ViewElement</u>

Description

A ViewContainer is an element of the interface that aggregates other ViewContainers and/or ViewElements displaying content.

Constraints:

- deafultMustHaveXorParent

self.viewContainer and self.viewContainer.isXor

- xorMustHaveADefaultParent

self.isXor implies self.viewElements -> select(c|c.oclTypeOf(ViewContainer) and c.asOclTypeOf(ViewContainer).isDefault) -> size()=1

Attributes

- isDefault: Boolean - If true, the ViewContainer will be presented to the user when its enclosing ViewContainer is accessed. This attribute is relevant when this ViewContainer shares the same parent ViewContainer with other ViewContainers, and the parent ViewContainer has property isXOR = true.

- isLandmark: Boolean - If true, the ViewContainer is directly reachable from any ViewElement from any ViewElement contained, directly or indirectly, in the same ViewContainer. It represents an implicit link between all the other ViewElements and the ViewContainer.

- isXOR: Boolean - If true, the contained ViewElements of thisViewContainer will be presented to the user only one at the time, as the user interacts with the system. One of the contained ViewContainers must have attribute isDefault = true.

Association Ends

- viewElements [0..*]: <u>ViewElement</u> - The ViewElements owned by the ViewContainer.

- actions [0..*]: Action – The Actions owned by the ViewContainer.

8.3.55 Class ViewElement

Abstract: No

Generalization:

- InteractionFlowElement

Description

ViewElements are elements of the user interface that display content. ViewElements are divided into ViewContainers and ViewComponents. ViewContainers are aggregations of other ViewContainers and/or ViewComponents.

Association Ends

- activationExpression [0..1]: ActivationExpression - Reference to an ActivationExpression whose evaluation result determines whether the ViewElement is active or inactive. If no ActivationExpression is given, by default the ViewElement is active.

- viewElementEvents [0..*]: ViewElementEvent - ViewElementEvents contained by the ViewElement.

- viewContainer [0..1]: ViewContainer - ViewContainer of the current ViewElement.

8.3.56 Class ViewElementEvent

Abstract: No

Generalization:

- CatchingEvent

Description

A ViewElementEvent represents a user interaction Event, which may be triggered by ViewElements (ViewContainers and ViewComponents).

Association Ends

- viewElement [1]: ViewElement – ViewElement owning the ViewElementEvent.

8.3.57 Class Viewpoint

Abstract: No

Generalization:

- NamedElement

Description

A Viewpoint is a reference to an interrelated set of InteractionFlowModelElements, which as a whole define a functional portion of the system. The purpose of a Viewpoint is to facilitate the comprehension of a complex system, to allow or disallow access to the system by a specific UserRole, or to adapt the system to a specific context change.

Association Ends

- interactionFlowModelElements [0..*]: InteractionFlowModelElement - InteractionFlowModelElements that build up this Viewpoint.

- context [1]: Context - Application context that determines the Viewpoint to be used.

8.3.58 Class VisualizationAttribute

Abstract: No

Generalization:

- [ViewComponentPart](#)

Description

The VisualizationAttributes used by a ViewComponent determine the features obtained from a DataBinding that may be shown to the user, such as a data base column or an XML element or attribute. A feature is represented using a UML::StructuralFeature.

Association Ends

- featureConcept [1]: FeatureConcept – A FeatureConcept of the DomainConcept bound to a DataBinding to be shown to the user, such as a data base column, an XML element or attribute, or a UML class attribute.

8.4 Package Extensions

8.4.1 Class Details

Abstract: No

Generalization:

- [ViewComponent](#)

Description

A Details ViewComponent is used to display the details of a DataBinding instance. When the Details ViewComponent is associated with an Event, it means that the DataBinding instance displayed by the component may trigger the Event. The Event will in turn cause the passing of the Parameter values mapped to the DataBinding instance to a target InteractionFlowElement.

Constraints

- mustHaveOneDataBinding
  self.viewComponentPart -> select(v | v.oclIsTypeOf(DataBinding)) -> size() = 1

8.4.2 Class Device

Abstract: No

Generalization:

- [ContextDimension](#)

Description

A Device is a ContextDimension that represents any device such as desktop, laptop, smart phone, tablet, or any other device from which the application may be accessed. A Device is associated with one or more Viewpoints (through the association from Viewpoint to Context). When the user context specifies the same device as the one specified by Device, the ContextDimension is satisfied and access is granted to the Viewpoint elements.

8.4.3 Class Field

Abstract: Yes

Generalization:

- [ViewComponentPart](#)
- Parameter

Description

A Field is a value-type pair whose value may be displayed to the user or serves as a mean for capturing input from the user. Fields also behave as Parameters for passing their values to and from other InteractionFlowElements. There are two kinds of fields, SimpleFields and SelectionFields.

8.4.4 Class Form

Abstract: No

Generalization:

- ViewComponent

Description

The Form ViewComponent represents input forms where user can submit information through Fields (SimpleFields or SelectionFields). It comprises at least one Field and typically at least one OnSubmitEvent.

Constraints

- mustHaveAtLeastOneField
  self.viewComponentPart -> select(v | v.oclIsTypeOf(Field)) -> notEmpty()

8.4.5 Class List

Abstract: No

Generalization:

- ViewComponent

Description

The List ViewComponent is used to display a list of DataBinding instances. When the List ViewComponent is associated with an Event, it means that each DataBinding instance displayed by the component may trigger the Event. The Event will in turn cause the passing of the Parameter values mapped to the DataBinding instance to a target InteractionFlowElement.

Constraints

- mustHaveOneDataBinding
  self.viewComponentPart -> select(v | v.oclIsTypeOf(DataBinding)) -> size() = 1

Association Ends

- selectEvent [0..*]: OnSelectEvent - Events that represent the selection of a DataBinding instance of the List ViewComponent and the passing of the value as a Parameter.

8.4.6 Class LandingEvent

Abstract: No

Generalization:

- CatchingEvent

Description

A LandingEvent is the destination of a JumpEvent.

Association Ends

- jumpingEvents [0..*]: JumpEvent - Reference to the JumpEvents targeting the current LandingEvent.

8.4.7   Class JumpEvent

Abstract: No

Generalization:

- ThrowingEvent

Description

- A JumpEvent is a ThrowingEvent that, when launched, redirects the NavigationFlow entering the event to a referenced LandingEvent.

Association Ends

- landingEvent [0..1]: LandingEvent - Reference to the LandingEvent targeted by the JumpEvent.

8.4.8   Class Menu

Abstract: No

Generalization:

- ViewContainer

Description:

A Menu is a special kind of ViewContainer used to model the concept of a menu of options in IFML. It cannot contain ViewComponents or sub-ViewContainers.

8.4.9   Class OnLoadEvent

Abstract: No

Generalization:

- SystemEvent

Description

An OnLoadEvent is triggered by the system when a ViewElement is completely computed and rendered.

8.4.10   Class OnSelectEvent

Abstract: No

Generalization:

- ViewElementEvent

Description

An OnSelectEvent is a kind of Event that, when triggered, results in a selected value being passed as a Parameter to the target InteractionFlowElement of its associated NavigationFlow.

8.4.11   Class OnSubmitEvent

Abstract: No

Generalization:

- ViewElementEvent

Description

An OnSubmitEvent triggers the Parameter passing of a ViewComponent to the target ViewElement or Action of its corresponding NavigationFlow. An OnSubmitEvent is typically found in Form ViewComponents.

8.4.12   Class Position

Abstract: No

Generalization:

- [ContextDimension](#)

Description

A Position is a ContextDimension representing the location and orientation of a device from which the application is accessed. A Position is associated with one or more ViewPoints (through the association between ViewPoint and Context). When the user context indicates having reached the location or orientation described by a Position, the ContextDimension is satisfied and access is granted to the ViewPoint elements and presented to the user.

8.4.13

8.4.14   Class SelectionField

Abstract: No

Generalization:

- [Field](#)

Description

A SelectionField is a kind of Field that enables the selection of one or more values from the predefined set of values given in its Slots.

Attributes

- isMultiSelection: Boolean - If true, the SelectionField allows the selection of multiple values.

8.4.15   Class SetContextEvent

Abstract: No

Generalization:

- ThrowingEvent

Description

A SetContextEvent is launched every time a ContextVariable is set or assigned a new value.

8.4.16   Class SimpleField

Abstract: No

Generalization:

- [Field](#)

Description

A SimpleField is a kind of Field that displays a value or captures a textual input from the user. A SimpleField also behaves as a Parameter,so that its value may be passed to other ViewElements or Actions.

8.4.17   Class Slot

Abstract: No

Generalization:

- [ViewComponentPart](#)
- [Parameter](#)

Description

A Slot is a value placeholder for a Field. When the Field is a SelectionField, its associated Slots contain the available selection options and the selected one. When the Field is a SimpleField, the Slot contains the Field value. A Slot value of a SimpleField and the Slots corresponding to the selected options of SelectionFields are copied to Parameters in order to be passed to other ViewElements or Actions when an Event is triggered.

8.4.18   Class UserRole

Abstract: No

Generalization:

- [ContextDimension](#)

Description

A UserRole is a ContextDimension that represents a role played by a human user or external system that accesses the application through its user interface. A UserRole is associated with one or more ViewPoints (trough the association between ViewPoint and Context). When the user context has the same user role as the one specified by the UserRole, the ContextDimension is satisfied and access is granted to the ViewPoint elements.

8.4.19   Class ValidationRule

Abstract: No

Generalization:

- [Constraint](#)

Description

A ValidationRule is a Constraint, which, when evaluated, determines if the content of a Field or group of Fields is valid or not.

8.4.20   Class Window

Abstract: No

Generalization:

- [ViewContainer](#)

Description:

A Window is a special kind of ViewContainer used to model the concept of a window in IFML.

Attributes

- isNewWindow: Boolean – If true, the container will be opened as a new window.
- isModal: Boolean – If true, the window will be rendered as a modal window.

9   IFML Execution Semantics

9.1   Introduction

This clause specifies the execution semantics of IFML. The purpose is to define when and how to compute the values to be shown to the user, based on an IFMLModel . A few aspects affect the execution semantics:

1. Computation of triggering events
2. Parameter propagation
3. Navigation history preservation

9.2   Relevant Aspects for IFML Execution Semantics

9.2.1   Triggering Events

The content of a ViewContainer must be (partially or completely) computed when the following *events* arise:

1. *Inter-container navigation flow traversal*: The container is entered through a NavigationFlow originated by an Event in another container.
2. *Intra-container navigation flow traversal:* The user produces an Event inside a container that triggers the navigation of a flow targeting an Element in the same top-level ViewContainer (e.g., Window). Firing the navigation may have side effects on the content of the currently visualized Elements (e.g., it may modify content currently shown to the user) and may invalidate (partially or totally) the information used to compute the container.

9.2.2   Parameter Propagation

A ViewContainer typically contains several pieces of related information. This corresponds to having several ViewComponents linked in a network topology through NavigationFlows and DataFlows. Information may be propagated from one ViewComponents to other ViewComponents through ParameterBindings. Actual propagation depends on the Events that trigger the flows.

Conflicts may arise in the propagation of Parameters. A *conflict* arises when a ViewComponent receives more than one input value for the same Parameter. This could happen due to multiple incoming flows in a ViewComponent or ViewContainer. A *conflict resolution strategy* (CRS) specifies which Parameter value is selected to compute the data content of the ViewComponent. A conforming tool shall use one of the following possible strategies:

1. *Non-deterministic choice*: One input parameter is chosen non-deterministically at run-time among the set of available inputs.
2. *With priorities*: Priorities are assigned at design-time to the incoming flows (for the ViewComponent or ViewContainer), and, in case of run-time conflict, the Parameter value transported by the flow with highest priority is chosen. Priorities define a total ordering on the incoming flows for the ViewComponent or ViewContainer.
3. *Mixed*: A partial order of prioritization is defined at design-time over the input flows, and, in case of run-time conflict, the Parameter values transported by the flow with highest priority is chosen. If the ViewContainer is accessed at run-time in such a way that multiple flows with highest priority are in conflict, a non-deterministic choice is taken.

9.2.3   Navigation History Preservation

When the user triggers an Event, the content of the destination ViewContainer is refreshed, in a way that may depend on the past history of the user interaction. The alternatives for re-computing a ViewContainer (or a part thereof) depends on the "degree of memory" used for computation. A conforming tool may use one of the following possible interaction history policies:

1. *Without history*: The contents of the ViewComponents are computed as if the ViewContainer was accessed for the first time. The computation without history policy may be used to "reset" and forget the choices done by the user in a container.
2. *With history*: The contents of the ViewComponents are computed based on the input history of the ViewComponents existing prior to the last navigation event.

9.3   ViewComponent Computation Process

In this section we provide a brief description of an algorithm for computing the content of a generic ViewContainer, with particular attention to containers of type Window.

The computation process is performed every time an Event arises. The process tries to determine the data content of all the ViewComponents of the ViewContainer, taking into account the semantic aspects discussed in 9.2. Intuitively, the process determines at each step the set of *computable* ViewComponents, i.e., the subset of ViewComponents that receive their input Parameters and therefore can be calculated.

A ViewComponent is computable if it has no incoming InteractionFlows or if it has incoming InteractionFlows and the following conditions are satisfied:

1. The ViewComponent has not been already computed (a ViewComponent cannot be computed more than once upon the same Event).
2. All the ViewComponents from which the ViewComponent may receive Parameters have been computed already.
3. All the input Parameters needed to compute the ViewComponent have a value.

If the computation semantics of the ViewContainer is without history, only current input parameters are considered in point 3. If the computation semantics is with history, components may draw their input values either from default input Parameter values or from the past Parameter values, existing prior to the last flow navigation.

The algorithm computes a the contents of the ViewComponents starting from the following input parameters: it must receive the ViewContainer to compute, the set of ViewComponents to be considered in the computation (initially all the ViewComponents of the ViewContainer), the conflict resolution strategy, the interaction history policy, the past Parameter values of all the ViewComponents prior to the last flow navigation, the destination ViewComponent of the fInteractionFlow whose navigation has produced the computation event together with the past Parameters transported by the flow. The following steps of the algorithm are then carried out:

1. *Component invalidation*: If the destination of the navigated flow is a ViewComponent, all its dependent ViewComponents are invalidated. (We say that ViewComponent u1 *depends* onViewComponent u2 if u1 can be reached through NavigatioFlows from u2.)
2. *Non-invalidated component computation:* One ViewComponent at a time is computed, until all possible components are considered. At each step, if there is at least one computable ViewComponent, one of them is selected and its content is computed, based on the conflict resolution strategy, the interaction history policy, and the past Parameter values. In particular:

2.1 If a ViewComponent does not depend on any other ViewComponent, i.e. it does not expect any input Parameter , it can always be computed.

2.2 If a ViewComponent is the destination ViewComponent of the InteractionFlow whose navigation has produced the computation event, then the past and new values of the flow Parameters are used for computing the component.

2.3 In all the other cases, the interaction history policy determines which input Parameters must be used. If the interaction history policy is "without history", one of the possible current input Parameters is chosen, according to the conflict resolution strategy. If the interaction history policy is "with history" the past values of the input Parameters are considered. If the past values are available and no newer value is available for that Parameter, the old value is used to instantiate the ViewComponent; if no past values of input Parameter for the component is available, one of the possible input Parameter values is chosen according to the conflict resolution strategy.

10 IFML Diagram Definition

10.1 Introduction

This clause specifies the metamodel for IFML Diagram Interchange (IFML DI). The IFML DI is meant to facilitate interchange of IFML diagrams between tools rather than being used for internal diagram representation by the tools.

The IFML DI metamodel, similarly to the IFML abstract syntax metamodel, is defined as a MOF-based metamodel. As such, its instances can be serialized and interchanged using XMI.

The IFML DI classes only define the visual properties used for depiction. All other properties that are required for the unambiguous depiction of IFML diagram elements are derived from the referenced IFML model elements.

Multiple depictions of a specific IFML Element in a single diagram are not allowed.

10.2 Conformance Criteria

As stated in the Diagram Definition (DD) specification, Modeling language DD enables a) **Diagram Information Interchange Conformance** and b) **Diagram Graphics Conformance.** Modeling language specifications can conform to DD in two levels by supporting either (a) only, or (a) and (b). The IFML Diagram Definition provides (a) and (b).

10.3 Architecture

The IFML language specification provides three normative artifacts at M2 (shown with shaded boxes in Figure 16): the abstract syntax model (IFML), the IFML diagram interchange model (IFML DI), and the mapping specification between the IFML DI and the graphics model (IFML Mapping Specification).
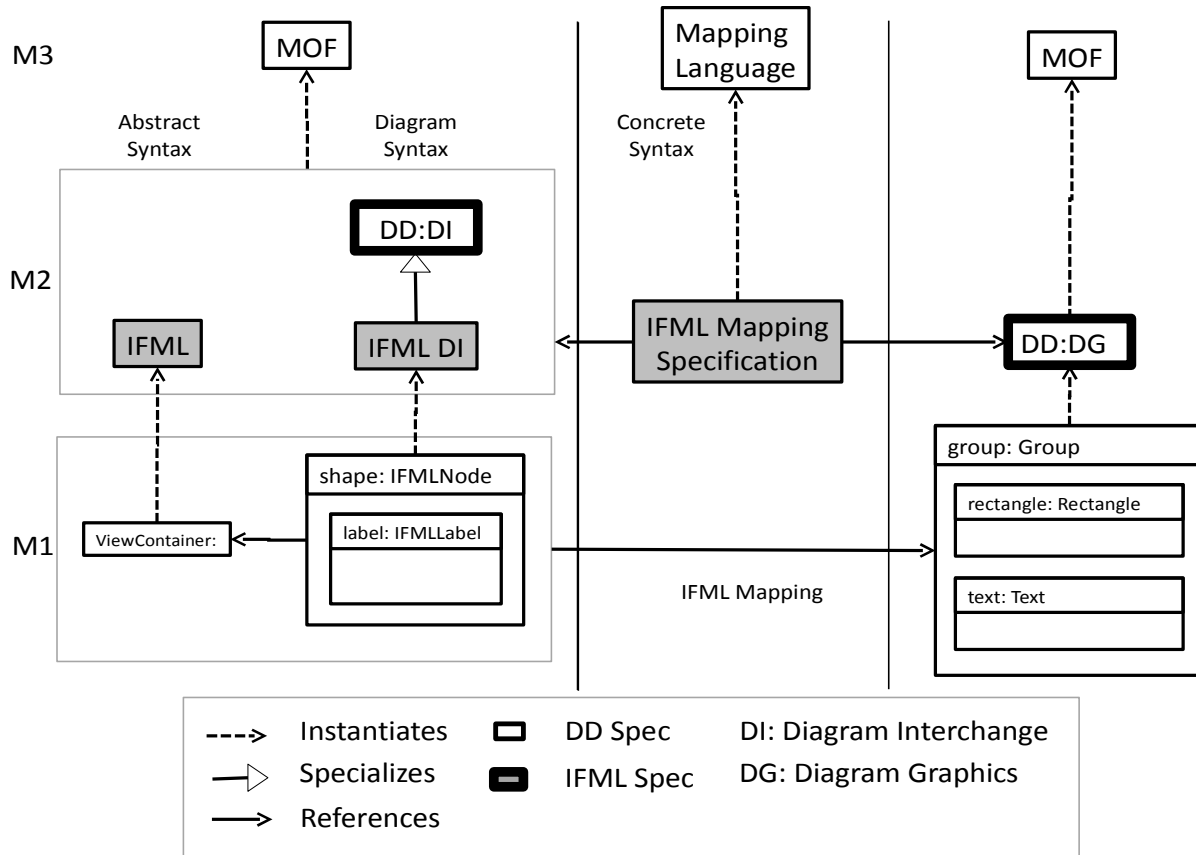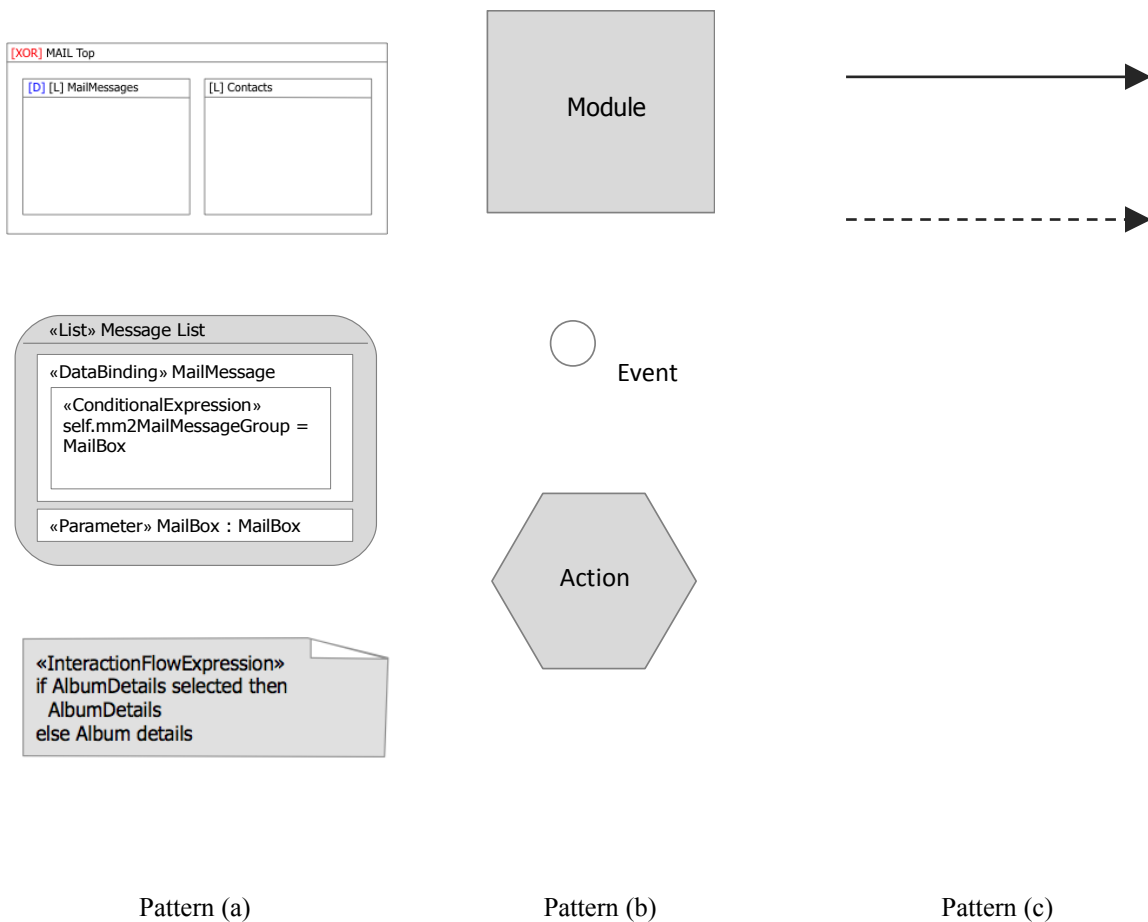
Figure 16: Diagram Definition Architecture for IFML

At M1 (left), Figure 16 shows an instance of IFML::Core::ViewContainer as a model element. Next to it, on the right, the figure shows an instance of IFMLDI::IFMLNode referencing the ViewContainer element, indicating that the ViewContainer is depicted as a node on the diagram. The node also contains an instance of IFMLDI::IFMLLabel representing the textual label of the ViewContainer on the diagram. On the right of M1, the figure shows an instance of DG::Group containing instances of DG::Rectangle and DG::Text

IFML DI specializes DD:DI, which specifies the graphics the user has control over, such as the position of nodes and line routing points. This information is what is captured for interchange between tools.

DD:DG represents the graphics that the user has no control over, such as shape and line styles, because they are the same in all languages conforming to the DD specification. DD:DG is derived by executing the mapping specification, in the middle, between IFML DI and DG.

10.4   IFML Diagram Interchange (DI) Meta-model

The IFML DI metamodel extends the DI metamodel, where appropriate. The class IFMLDiagram represents the diagram, which composes IFMLDiagramElements. An IFMLDiagram is an IFMLNode because it may be rendered as a figure and be connected to other figures. IFMLDiagramElements optionally reference elements of an IFML model, the latter denoted by the IFML:Core:Element class. IFMLDiagramElements that do not reference elements of an IFML model are purely notational diagram elements such as notes and the link that connects the note with the model element. IFMLDiagramElements may also be styled with instances of class IFMLStyle (e.g. font type and size).

Figure 17: IFML Diagram Interchange (DI) Meta-model

Classes are defined for interchanging shapes and edges of the interaction flow diagram and the content diagram, based on the following notational patterns (see Figure 18):

- Pattern (a): A shape that has a label and an optional list of compartments, each of which having an optional list of labels or other shapes (e.g., the ViewContainer box, ViewComponentPart box, Form ViewComponent rounded box or the classes of the DomainModel).

- Pattern (b): A shape that has a label only (e.g., the Event ball or Action hexagon notation).

- Pattern (c): An edge that may be dashed or solid (e.g., NavigationFlows and DataFlows).

| Pattern (a) | Pattern (b) | Pattern (c) |

Figure 18: Notational patterns

Based on these patterns, three shape classes (IFMLNode, IFMLLabel and IFMLCompartment) and one edge class (IFMLConnection) are defined and related to realize the patterns. These classes (except IFMLCompartment) are subclasses of IFMLDiagramElement to allow them to be styled independently and to reference their own IFML Element.

Some classes have properties to disambiguate the notation and a corresponding enumeration. For instance labels may be of different kinds such as Parameter, ViewContainer, etc., which will determine how the text decoration will be rendered.

The following subclause provides the detailed specification of the DI metamodel.

10.5   Package IFMLDI

10.5.1   Enumeration LabelKind

Description

Enumeration defining the kinds of labels, which will determine how to render the label decoration.

Literals

- action: Label of an Action.
- activationExpression: Label of an ActivationExpression.
- conditionalExpression: Label of a ConditionalExpression.
- dataBinding: Label of a DataBinding.
- form: Label of a Form.
- interactionFlowExpression: Label of an InteractionFlowExpression.
- list: Label of a List.
- namedElement: Label of any NamedElement without additional decoration.
- parameter: Label of a Parameter.
- parameterBindingGroup: Label of a ParameterBindingGroup.
- selectionField: Label of a SelectionField.
- simpleField: Label of a SimpleField.
- validationRule: Label of a ValidationRule.
- viewContainer: Label of a ViewContainer.

10.5.2   Class IFMLCompartment

Abstract: No

Generalization:

- DD::DI::Shape

Description

An IFMLCompartment is a section within an IFMLDiagramElement. An IFMLCompartment organizes the items in an IFMLDiagramElement so that it is easy to differentiate between them. IFMLCompartments may contain IFMLNodes or IFMLLabels.

Association Ends

- ownedLabels [0..*]: IFMLLabel - Composite association to the IFMLLabels owned by the compartment.

- ownedNodes [0..*]: IFMLNode - Composite association to the IFMLNodes owned by the compartment.

10.5.3   Class IFMLConnection

Abstract: No

Generalization:

- DD::DI::Edge
- IFMLDiagramElement

Description

An IFMLConnection represents a depiction of a connection between two (source and target) IFMLDiagramElements. It specializes DI::DD::Edge. IFMLConnections do not contain labels. All IFMLConnections are owned directly by an IFMLDiagram. The way-points of IFMLConnections are always relative to that diagrams's origin point and must be positive coordinates.

Association Ends

- sourceElement [1]: IFMLDiagramElement - Source IFMLDiagramElement of the connection.

- targetElement [1]: IFMLDiagramElement - Target IFMLDiagramElement of the connection.

10.5.4 Class IFMLDiagram

Abstract: No

Generalization:

- DD::DI::Diagram

- IFMLNode

Description

IFMLDiagram represents a depiction of all or part of an IFMLModel. It specializes DD::DI::Diagram and IFMLNode, since a diagram may be seen as a node as in the case of ViewPoint and Module.

Association Ends

- diagramElements [0..*]: IFMLDiagramElement – The diagram elements contained in this diagram.

10.5.5 Class IFMLDiagramElement

Abstract: No

Generalization:

- DD::DI::DiagramElement

Description

IFMLDiagramElement extends DD::DI::DiagramElement and is the supertype of all elements in diagrams, including diagrams themselves. When contained in a diagram, diagram elements are laid out relative to the diagram's origin.

An IFMLDiagramElement can be useful on its own (i.e., purely notational), or, more commonly, it is used as a depiction of another IFML Element from an IFMLModel. An IFMLDiagramElement can own other diagram elements in a graph-like hierarchy. IFMLDiagramElements can own and/or share IFMLStyle elements. Shared IFMLStyle elements are owned by other IFMLDiagramElements.

Association Ends

- localStyle [0..1]: IFMLStyle - Composite associations to IFMLStyles owned by the diagram element.

- sharedStyle [0..1]: IFMLStyle - Reference to IFMLStyles shared with other diagram elements.

- modelElement [0..1]: ifml::core::Element - Referenced Element from and IFML model.

- ownedElements [0..*]: IFMLDiagramElement - Composite association to the IFMLDiagramElements owned by the current IFMLDiagramElement.

10.5.6 Class IFMLLabel

Abstract: No

Generalization:

- DD::DI::Shape

- IFMLDiagramElement

Description

An IFMLLabel is a label that depicts textual information about an IFML Element. An IFMLLabel is always

contained (but not always rendered) in an IFMLNode directly or through an IFMLCompartment. In IFML, labels are not found in IFMLConnections. IFMLLabels may derive the textual information to be depicted from a referenced IFML model Element that contains the property with the label text.

Attributes

- kind: LabelKind - Determines to what kind of Element the IFMLLabel corresponds, e.g., label of a Parameter, a ViewContainer, an Action, etc.

10.5.7   Class IFMLNode

Abstract: No

Generalization:

- DD::DI::Shape

- IFMLDiagramElement

Description

An IFMLNode represents a figure with bounds that is laid out relative to the origin of the diagram. Note that the bounds' x and y coordinates are the position of the upper left corner of the node (relative to the upper left corner of the diagram). IFMLNodes may contain IFMLCompartments and other IFMLNodes and may be connected by IFMLConnections.

Association Ends

- ownedCompartment [0..*]: IFMLCompartment - Composite associations to the IFMLCompartments owned by the node.

- ownedLabel [0..1]: IFMLLabel - Composite association to the label owned by the node.

- ownedNodes [0..*]: IFMLNode - Nested nodes of the current node. This relation is only valid if the nested node is fixed to the parent node side.

10.5.8   Class IFMLStyle

Abstract: No

Generalization:

- DD::DI::Style

Description

An IFMLStyle represents appearance options for IFMLDiagramElements. One or more elements may reference the same IFMLStyle element, which must be owned by an IFMLDiagramElement.

Attributes

- fillColor: Color - Background color of the figure.

- fontName: String - Name of the font used by the styled IFMLDiagramElement

- fontSize: Real - Size of the font used by the styled IFMLDiagramElement

10.6   IFML DI to DG Mapping Specification

The DD architecture expects language specifications to define mappings between interchanged and non-interchanged graphical information, but does not restrict how it is done. The IFML DI to DG mapping is shown in Figure 16 by a shaded box labeled "IFML Mapping Specification" in the middle section and is accomplished in this specification by means of the following QVT mapping.

```
1      transformation IFMLDItoDG(in ifmldi: IFMLDI, in ifml: IFML, out DG)
```

```
2
3    main() {
4      ifmldi.objectsOfType(IFMLDiagram)->map toGraphics();
5    }
6
7    mapping IFMLDiagram::toGraphics(): Canvas {
8      member += self.diagramElements->map toGraphics();
9    }
10
11   mapping IFMLDiagramElement::toGraphics(): Group {
12     localStyle  := copyStyle(self.localStyle);
13     sharedStyle := copyStyle(self.sharedStyle);
14   }
15
16   mapping IFMLNode::toGraphics(): Group inherits IFMLDiagramElement::toGraphics() {
17     member += self.modelElement.map toGraphics(self);
18     member += self.ownedCompartments->map toGraphics();
19     member += self.ownedLabel.map toGraphics();
20   }
21
22   mapping IFMLLabel::toGraphics(): Text inherits IFMLDiagramElement::toGraphics() {
23     var e := self.modelElement;
24     bounds := self.bounds;
25     data := switch {
26       case (self.kind = LabelKind::NAMED_ELEMENT)
27         e.name;
28       case (self.kind = LabelKind::VIEW_CONTAINER)
29         e.oclAsType(ViewContainer).getLabelText();
30       case (self.kind = LabelKind::ACTION)
31         e.oclAsType(Action).getLabelText();
32       case (self.kind = LabelKind::PARAMETER)
33         "«Parameter» " + e.name + ": " + e.type.name;
34       case (self.kind = LabelKind::FORM)
35         "«Form» " + e.name;
36       case (self.kind = LabelKind::LIST)
37         "«List» " + e.name;
38       case (self.kind = LabelKind::SIMPLE_FIELD)
39         "«SimpleField» " + e.name;
40       case (self.kind = LabelKind::SELECTION_FIELD)
41         "«SelectionField» " + e.name;
42       case (self.kind = LabelKind::PARAMETER_BINDING_GROUP)
43         "«ParameterBindingGroup»";
44       case (self.kind = LabelKind::ACTIVATION_EXPRESSION)
45         "«ActivationExpression»";
46       case (self.kind = LabelKind::INTERACTION_FLOW_EXPRESSION)
47         "«InteractionFlowExpression»";
48       case (self.kind = LabelKind::VALIDATION_RULE)
49         "«ValidationRule»";
50       case (self.kind = LabelKind::CONDITIONAL_EXPRESSION)
51         "«ConditionalExpression»";
52       case (self.kind = LabelKind::DATA_BINDING)
53         "«DataBinding»";
54       default
55         "";
56     };
57   }
58
59
60   query ViewContainer::getLabelText(): String {
61    var text := if self.oclTypeOf(Window) and self.isModal then "«Modal»" endif;
62    var text := if self.oclTypeOf(Window) and not self.isModal then "«Window»" endif;
63
64     text += if self.isXOR then "[XOR] " endif;
65     text += if self.isLandmark then "[L] " endif;
66     text += if self.isDefault then "[D] " endif;
67
```

```
70       return text + self.name;
71
72   }
73
74   mapping Element::toGraphics(n: IFMLNode): GraphicalElement
75     disjuncts ViewContainer::toRectangle, ViewComponent::toRectangle,
76               Module::toRectangle, ViewComponentPart::toRectangle, Event::toCircle,
77               Action::toPolygon, ViewPoint::toPolygon, ModuleDefinition::toRectangle,
78   Port::toRectangle, PortDefinition::toRectangle, ModulePackage::toRectangle {
79   }
80
81   mapping ViewContainer::toRectangle(n: IFMLNode): Rectangle {
82     bounds := n.bounds;
83   }
84
85   mapping ViewComponent::toRectangle(n: IFMLNode): Rectangle {
86     bounds := n.bounds;
87     cornerRadius := 15;
88   }
89
90   mapping Module::toRectangle(n: IFMLNode): Rectangle {
91     bounds := n.bounds;
92   }
93
94   mapping ModuleDefinition::toRectangle(n: IFMLNode): Rectangle {
95     bounds := n.bounds;
96   }
97
98   mapping PortDefinition::toRectangle(n: IFMLNode): Rectangle {
99     bounds := n.bounds;
100  }
101
102  mapping Port::toRectangle(n: IFMLNode): Rectangle {
103    bounds := n.bounds;
104  }
105
106  mapping ModulePackage::toRectangle(n: IFMLNode): Rectangle {
107    bounds := n.bounds;
108  }
109
110  mapping ViewComponentPart::toRectangle(n: IFMLNode): Rectangle {
111    bounds := n.bounds;
112  }
113
114  mapping Event::toCircle(n: IFMLNode): Circle {
115    var b := n.bounds;
116    center := object Point{b.x + b.width / 2; b.y + b.height / 2};
117    radius := if b.width < b.height then
118                  b.width / 2
119                else
120                  b.height / 2
121                endif;
122  }
123
124  mapping Action::toPolygon(n: IFMLNode): Polygon {
125    var b := n.bounds;
126    point += object Point {b.width * (1/4); y := 0};
127    point += object Point {b.width * (3/4); y := 0};
128    point += object Point {b.width; b.height * (1/4)};
129    point += object Point {b.width; b.height * (3/4)};
130    point += object Point {b.width * (3/4); b.height};
131    point += object Point {b.width * (1/4); b.height};
132    point += object Point {0; b.height * (3/4)};
133    point += object Point {0; b.height * (1/4)};
134  }
135
136  mapping ViewPoint::toPolygon(n: IFMLNode): Polygon {
137    var b := n.bounds;
```

```
138    point += object Point {b.width * (1/2); y := 0};
139    point += object Point {b.width; b.height};
140    point += object Point {0; b.height};
141  }
142
143  mapping ParameterBindingGroup::toPolygon(n: IFMLNode): Polygon {
144    var b := n.bounds;
145    point += object Point {x:=0,y:=0};
146    point += object Point {b.width*3/4,y:=0};
147    point += object Point {b.width,b.height};
148    point += object Point {b.width*1/4,b.height};
149  }
150
151  mapping IFMLCompartment::toGraphics(): Group {
152    member += object Rectangle {bounds:= self.bounds};
153    member += self.ownedNodes.map toGraphics();
154    member += self.ownedLabels.map toGraphics();
155  }
156
157  mapping IFMLConnection::toGraphics(): Group inherits
158  IFMLDiagramElement::toGraphics() {
159    member += self.modelElement.map toGraphics(self);
160  }
161
162  mapping Element::toGraphics(c: IFMLConnection): GraphicalElement
163    disjuncts NavigationFlow::toPolyline, DataFlow::toPolyline {
164  }
165
166
167  mapping NavigationFlow::toPolyline(c: IFMLConnection): Polyline {
168    point := c.waypoint;
169    sharedStyle := solidStyleProp;
170    endMarker := arrowMarkerProp;
171  }
172
173  property solidStyleProp = object DG::Style {
174    strokeDashLength := Sequence{};
175  }
176
177  property arrowMarkerProp = object Marker {
178    size := object Dimension {width := 2; height := 2};
179    reference := object Point {x := 2; y := 1};
180    member += object Polygon {
181      point += object Point {x := 0; y := 0};
182      point += object Point {x := 2; y := 1};
183      point += object Point {x := 0; y := 2};
184    }
185  }
186
187  mapping DataFlow::toPolyline(c: IFMLConnection): Polyline {
188    point := c.waypoint;
189    sharedStyle := dashedStyleProp;
190    endMarker := arrowMarkerProp;
191  }
192
193  property dashedStyleProp = object DG::Style {
194    strokeDashLength := Sequence{2, 2};
195  }
196
197  helper copyStyle(s: IFMLStyle): DG::Style {
198    fontName := s.fontName;
199    fontSize := s.fontSize;
200    fillColor := s.fillColor;
201  }
202
203
204
205
```

11   UML Profile for IFML

11.1   Overview

The UML Profile for IFML enables the use of UML for representing IFML models. The purpose of the profile is to extend the UML metamodel by customizing it with specific IFML constructs.

The UML Profile for IFML is based on the use of UML components (both basic components and packaging components), classes and other concepts.

Components may form hierarchical structures (a packaging component that owns other components) and they may be connected with dependencies, either through explicit interfaces or directly.

Components may be shown in a structural UML diagram, such as a component diagram.

Their dynamic behavior may be shown in interaction diagrams, such as a communication diagram. The behavior of components may also be described in a statechart diagram or in an activity diagram. Examples of these diagrams are not shown here.

11.2 The IFML Profile of UML

The UML Profile for IFML consists of the stereotypes defined in this subclause. These stereotypes are shown in a set of UML diagrams below, along with a table for each diagram giving the specification of the depicted stereotypes.
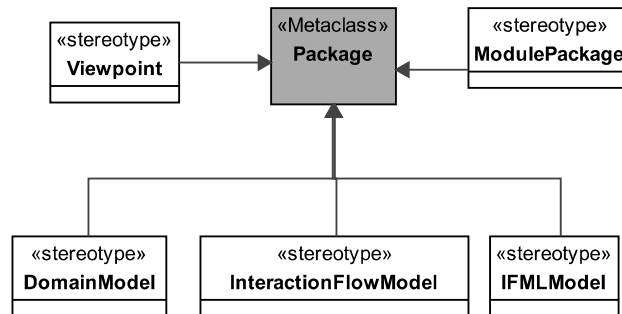
Figure 19: Package stereotypes

Table 3: Package stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«DomainModel»	UML::Kernel::Package			
«IFMLModel»	UML::Kernel::Package			
«InteractionFlowModel»	UML::Kernel::Package			
«ModulePackage»	UML::Kernel::Package			
«Viewpoint»	UML::Kernel::Package			
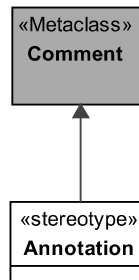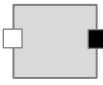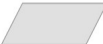
Figure 20: Annotation stereotype

Table 4: Annotation stereotype

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«Annotation»	UML::Kernel::Comment			

Figure 21: ViewContainer, ParameterBindingGroup, and ModuleDefinition stereotypes

Table 5: ViewContainer, ParameterBindingGroup, and ModuleDefinition stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«ModuleDefinition»	UML::Components::BasicComponents::Component		It must contain at least 1 InteractionFlow-ModelElement	
«ParameterBindingGroup»	UML::Components::BasicComponents::Component			
«ViewContainer»	UML::Components::BasicComponents::Component	isLandMark: Boolean isDefault: Boolean isXor: Boolean		

Table 6: ViewContainer and ModuleDefinition stereotypes (extension)

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«Menu»	UML::Components::BasicComponents::Component			
«Window»	UML::Components::BasicComponents::Component	isNewWindow: Boolean isModal: Boolean		
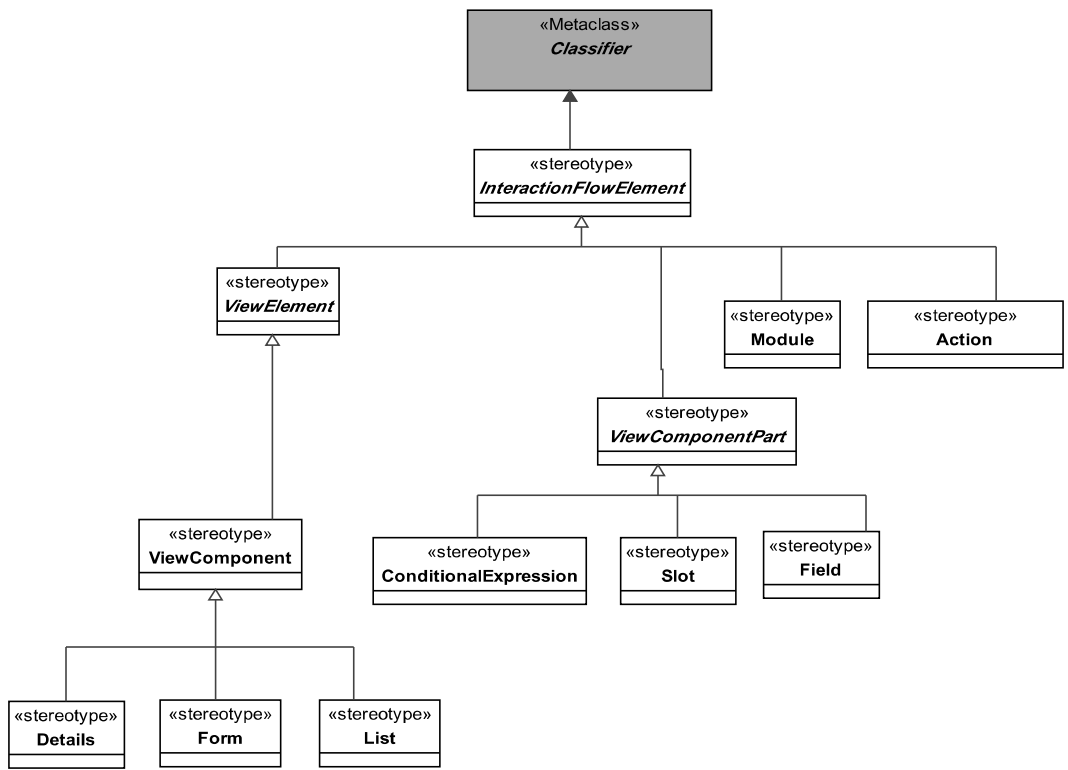
Figure 22: InteractionFlowElements stereotypes (except events)

Table 7: InteractionFlowElements stereotypes (except events)

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«ViewElement»	UML::Kernel:: Classifier			
«InteractionFlowElement»	UML::Kernel:: Classifier			
«Module»	UML::Kernel:: Classifier			
«ViewComponent»	UML::Kernel:: Classifier			
«ViewComponentPart»	UML::Kernel::Classifier			

Table 8: InteractionFlowElements stereotypes (except events) (extensions)

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«List»	UML::Kernel:: Classifier			
«Form»	UML::Kernel:: Classifier			
«Details»	UML::Kernel:: Classifier			
«Field»	UML::Kernel:: Classifier			
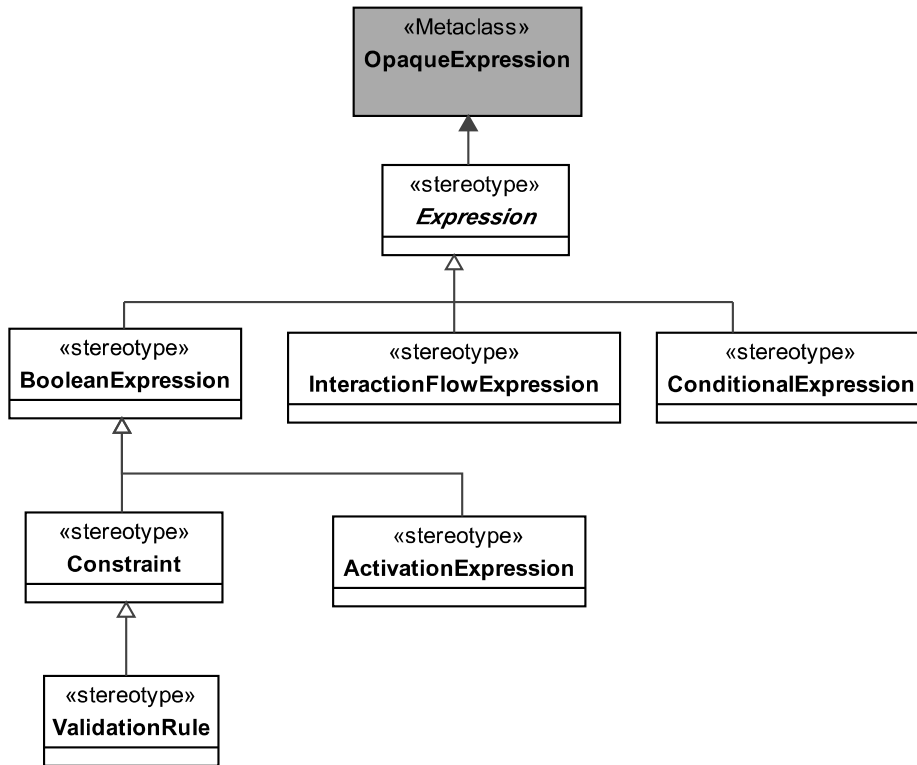
Figure 23: Expressions stereotypes

Table 9: Expressions stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«ActivationExpression»	UML::Kernel:: OpaqueExpression			
«BooleanExpression»	UML::Kernel:: OpaqueExpression			
«ConditionalExpression»	UML::Kernel:: OpaqueExpression			
	UML::Kernel:: Classifier			
«Constraint»	UML::Kernel:: OpaqueExpression			
«Expression»	UML::Kernel:: OpaqueExpression			
«InteractionFlowExpression»	UML::Kernel:: OpaqueExpression			
«ValidationRule»	UML::Kernel:: OpaqueExpression			
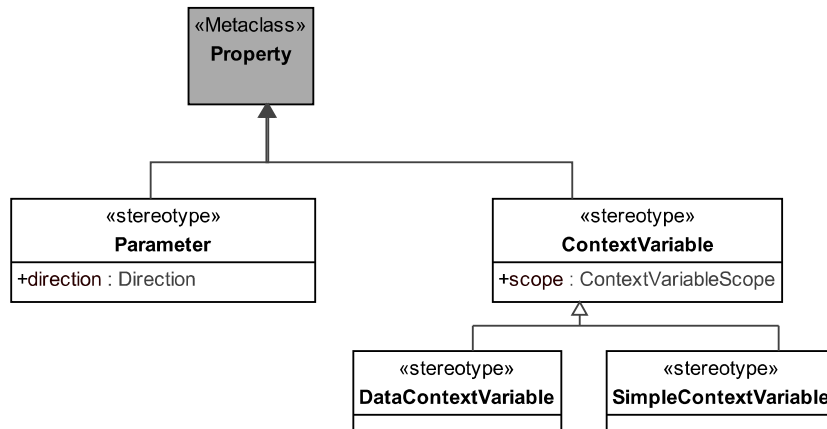
Figure 24: Parameter and context variable stereotypes

Table 10: Parameter and context variable stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«Parameter»	UML::Kernel::Property	direction: Direction		
«ContextVariable»	UML::Kernel::Property	scope:ContextVariableScope		
«SimpleContextVariable»	UML::Kernel::Property			
«DataContextVariable»	UML::Kernel::Property			
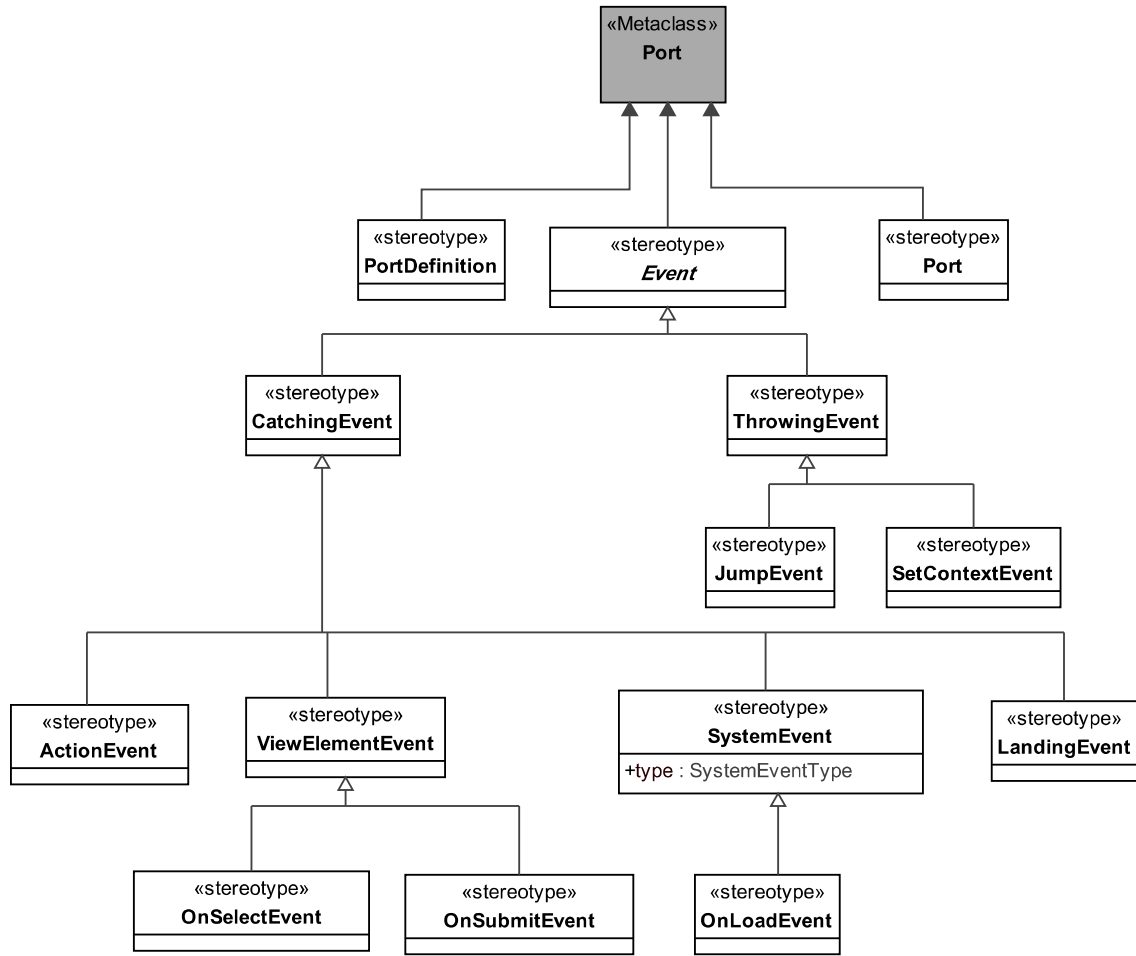
Figure 25: Port and Events stereotypes

Table 11: Ports and Events stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«ActionEvent»	UML::CompositeStructures:: Ports::Port			○
«Event»	UML::CompositeStructures:: Ports::Port			○
«SystemEvent»	UML::CompositeStructures:: Ports::Port	type: SystemEventType		○
«CatchingEvent»	UML::CompositeStructures:: Ports::Port			○
«ThrowingEvent»	UML::CompositeStructures:: Ports::Port			●
«JumpEvent»	UML::CompositeStructures:: Ports::Port			⬤➡
«LandingEvent»	UML::CompositeStructures:: Ports::Port			⬤➡
«ViewElementEvent»	UML::CompositeStructures:: Ports::Port			○
«Port»	UML::CompositeStructures:: Ports::Port			□
«PortDefinition»	UML::CompositeStructures:: Ports::Port			□

Table 12: Ports and Events stereotypes (extensions)

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«OnSelectEvent»	UML::CompositeStructures:: Ports::Port			
«OnSubmitEvent»	UML::CompositeStructures:: Ports::Port			
«SetContextEvent»	UML::CompositeStructures:: Ports::Port			
«OnLoadEvent»	UML::CompositeStructures:: Ports::Port			
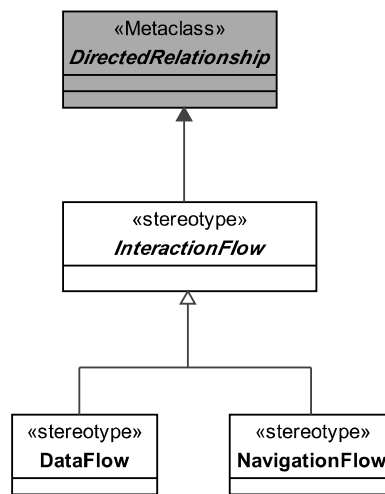
Figure 26: InteractionFlow stereotypes

Table 13: InteractionFlow stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«DataFlow»	UML::Kernel::DirectedRelationShip		Must be associated with a ParameterBinding	
«InteractionFlow»	UML::Kernel::DirectedRelationShip			
«NavigationFlow»	UML::Kernel::DirectedRelationShip			
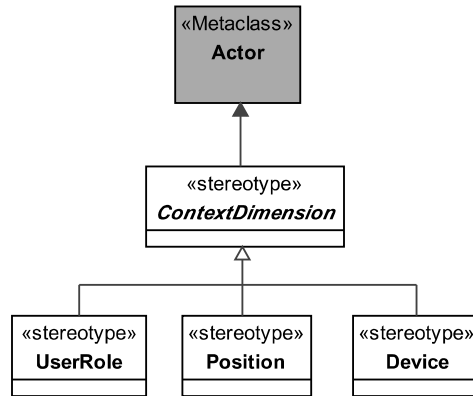
Figure 27: ContextDimensions
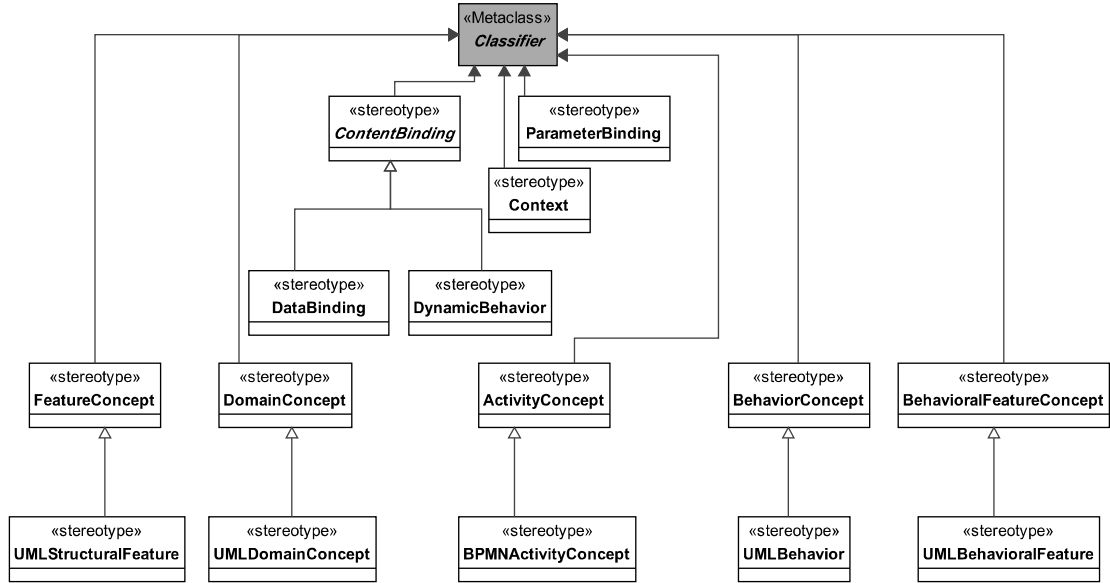
Table 14: ContextDimensions stereotypes

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«ContextDimension»	UML::UseCases::Actor			

Table 15: ContextDimensions stereotypes (extension)

Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«Device»	UML::UseCases::Actor			
«UserRole»	UML::UseCases::Actor			

| «Position»	UML::UseCases::Actor			

Figure 28: ContentBindings, Context, and ParameterBindings stereotypes

Table 16: ContentBindings, Context, and ParameterBindings stereotypes.

| Stereotype	UML Metaclass	Tagged Values	Constraints	Icon
«ContentBinding»	UML::Kernel::Classifier			
«Context»	UML::Kernel::Classifier			
«DataBinding»	UML::Kernel::Classifier			
«DynamicBehavior»	UML::Kernel::Classifier			
«ParameterBinding»	UML::Kernel::Classifier			
«ActivityConcept»	UML::Kernel::Classifier			
«FeatureConcept»	UML::Kernel::Classifier			
«DomainConcept»	UML::Kernel::Classifier			
«BehaviorConcept»	UML::Kernel::Classifier			
«BehavioralFeatureConcept»	UML::Kernel::Classifier			
«BPMNActivityConcept»	UML::Kernel::Classifier			
«UMLStructuralFeature»	UML::Kernel::Classifier			
«UMLDomainConcept»	UML::Kernel::Classifier			
«UMLBehavior»	UML::Kernel::Classifier			
«UMLBehavioralFeature»	UML::Kernel::Classifier			

11.3   Using IFML Stereotypes

IFML stereotypes can be user with different abstraction levels, i.e. using stereotypes that correspond to the IFML Core package, the IFML Extension package, or user provided platform-specific stereotypes.
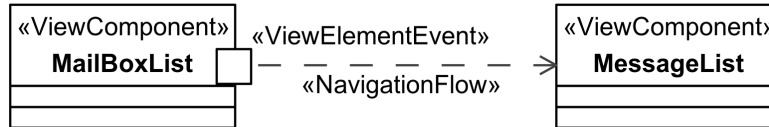
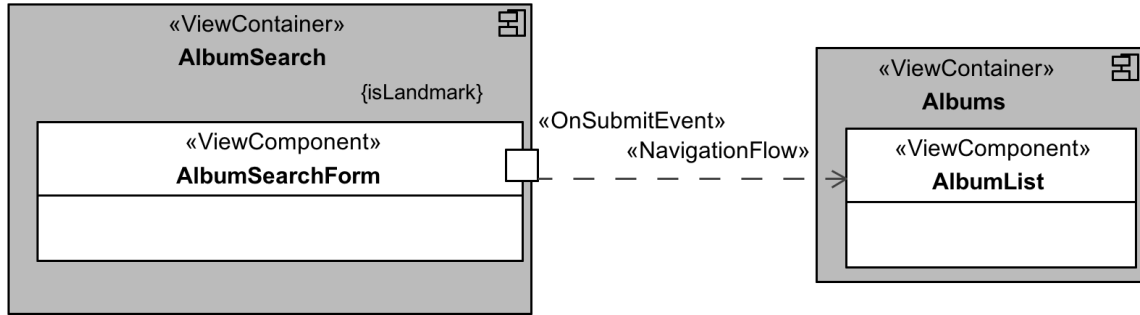**Figure 29: Stereotyped UML diagram with IFML Core**

For instance, for stereotyping with IFML Core concepts, classes, components and ports may be stereotyped with ViewContainer, ViewComponent, ViewComponentPart, Event, and Action concepts, and directed relationships with NavigationFlow and DataFlow concepts as shown in Figure 29.
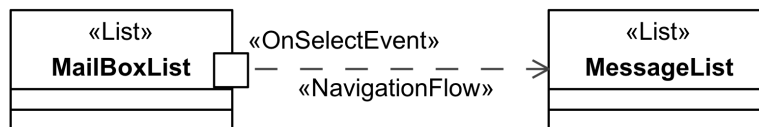
Figure 30: Stereotyped UML diagram with IFML Extensions

For stereotyping with IFML Extension concepts, classes may be stereotyped as List, Details and Form concepts, and ports with events like OnSelectEvent and OnSubmitEvent as shown in Figure 30.

Components own (or import) classes. In Figure 31, ViewContainers are shown as components and contain classes stereotyped as ViewComponents.

Figure 31: Stereotyped UML diagram with ViewContainers containing ViewComponents

Parameters are defined as properties of the ViewComponents, with a tagged value represeting their direction. ParameterBindingGroups are asociated to NavigationFlows and DataFlows and contain classes stereotyped as ParameterBindings.

Figure 32 shows the representation of Parameters, ParameterBindingGroups and ParameterBindings on the previous example.
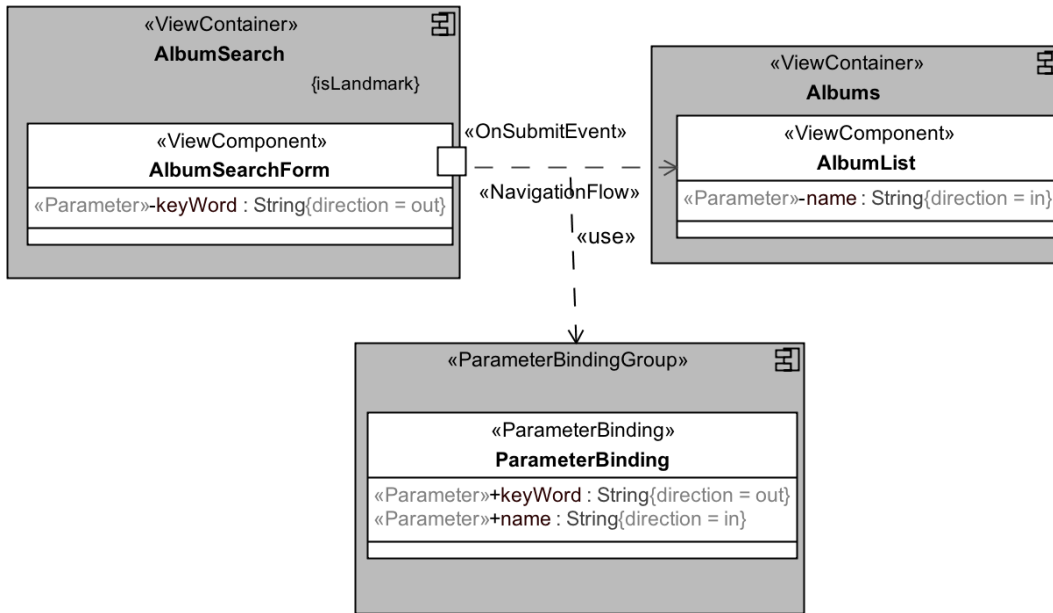
Figure 32: Stereotyped UML diagram including Parameters, ParameterBindings and ParameterBindingGroups

11.4   Profile Metamodel Mapping

Table 17 Shows, for each metaclass from the IFML metamodel in Clause 8, the mapping to the respective stereotype of the IFML UML profile.

Table 17: Profile metamodel mapping

IFML Metaclass	Stereotype
IFML::Core::Action	«Action»
IFML::Core::ActionEvent	«ActionEvent»
IFML::Core:: ActivationExpression	«ActivationExpression»
IFML::Core:: ActivityConcept	«ActivityConcept»
IFML::Core::Annotation	«Annotation»
IFML::Core:: BehavioralConcept	«BehavioralConcept»
IFML::Core:: BehavioralFeatureConcept	«BehavioralFeatureConcept»
IFML::Core::BooleanExpression	«BooleanExpression»
IFML::Core::BPMNActivityConcept	«BPMNActivityConcept»
IFML::Core::CatchingEvent	«CatchingEvent»
IFML::Core::ConditionalExpression	«ConditionalExpression»
IFML::Core::Constraint	«Constraint»
IFML::Core::ContentBinding	«ContentBinding»
IFML::Core::DomainModel	«DomainModel»
IFML::Core::Context	«Context»
IFML::Core::ContextDimension	«ContextDimension»
IFML::Core::ContextVariable	«ContextVariable»
IFML::Core::DataBinding	«DataBinding»
IFML::Core::DataContextVariable	«DataContextVariable»
IFML::Core::DataFlow	«DataFlow»
IFML::Core::DomainConcept	«DomainConcept»
IFML::Core::DomainModel	«DomainModel»
IFML::Core::DynamicBehavior	«DynamicBehavior»
IFML::Core::Element	«Element»
IFML::Core::Event	«Event»
IFML::Core::Expression	«Expression»
IFML::Core::FeatureConcept	«FeatureConcept»
IFML::Core::IFMLModel	«IFMLModel»
IFML::Core::InteractionFlow	«InteractionFlow»
IFML::Core::InteractionFlowElement	«InteractionFlowElement»
IFML::Core::InteractionFlowExpression	«InteractionFlowExpression»
IFML::Core::InteractionFlowModel	«InteractionFlowModel»
IFML::Core::InteractionFlowModelElement	«InteractionFlowModelElement»
IFML::Core::Module	«Module»

Table 17: Profile metamodel mapping

IFML Metaclass	Stereotype
IFML::Core::ModuleDefinition	«ModuleDefinition»
IFML::Core::ModulePackage	«ModulePackage»
IFML::Core::NamedElement	«NamedElement»
IFML::Core::NavigationFlow	«NavigationFlow»
IFML::Core::Parameter	«Parameter»
IFML::Core::ParameterBinding	«ParameterBinding»
IFML::Core::ParameterBindingGroup	«ParameterBindingGroup»
IFML::Core::Port	«Port»
IFML::Core::PortDefinition	«PortDefinition»
IFML::Core::SimpleContextVariable	«SimpleContextVariable»
IFML::Core::SystemEvent	«SystemEvent»
IFML::Core::ThrowingEvent	«ThrowingEvent»
IFML::Core::UMLBehavior	«UMLBehavior»
IFML::Core::UMLBehavioralFeature	«UMLBehavioralFeature»
IFML::Core::UMLDomainConcept	«UMLDomainConcept»
IFML::Core::UMLStructuralFeature	«UMLStructural»
IFML::Core::ViewComponent	«ViewComponent»
IFML::Core::ViewComponentPart	«ViewComponentPart»
IFML::Core::ViewContainer	«ViewContainer»
IFML::Core::ViewElement	«ViewElement»
IFML::Core::ViewElementEvent	«ViewElementEvent»
IFML::Core::ViewPoint	«ViewPoint»
IFML::DataTypes::ParameterKind	«ParameterKind»
IFML::DataTypes::SystemEventTypeEnum	«SystemEventTypeEnum»
IFML::Extensions::Device	«Device»
IFML::Extensions::Form	«Form»
IFML::Extensions::Field	«Field»
IFML::Extensions::List	«List»
IFML::Extensions::LandingEvent	«LandingEvent»
IFML::Extensions::JumpEvent	«JumpEvent»
IFML::Extensions::Menu	«Menu»
IFML::Extensions::Details	«Details»
IFML::Extensions::Window	«Window»
IFML::Extensions::Position	«Position»

Table 17: Profile metamodel mapping

IFML Metaclass	Stereotype
IFML::Extensions::OnLoadEvent	«OnLoadEvent»
IFML::Extensions::OnSelectEvent	«OnSelectEvent»
IFML::Extensions::SetContextEvent	«SetContextEvent»
IFML::Extensions::Slot	«Slot»
IFML::Extensions::OnSubmitEvent	«OnSubmitEvent»
IFML::Extensions::UserRole	«UserRole»
IFML::Extensions::ValidationRule	«ValidationRule»

Annex A   IFML by Example: Modeling an Email (informative)

A.1 Introduction

This annex exemplifies the modeling construct and the expressive power of IFML by modeling a popular Internet Application specialized on email service.

A.2 The Domain Model

The email application manages mail messages and contacts of users.

An **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**.
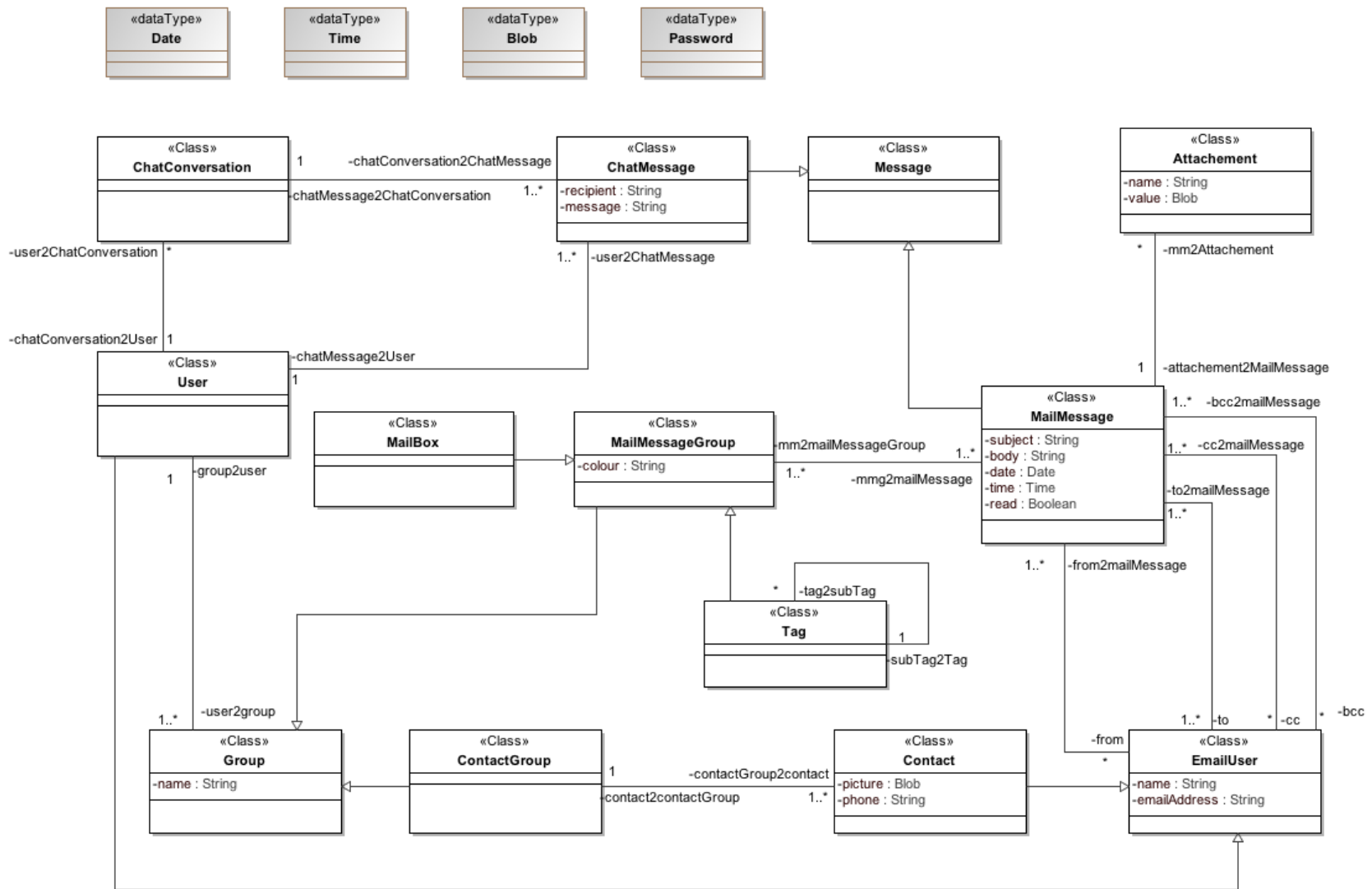
Figure 33: the Domain Model of the online mailing application.

A.3 Model of the Interface

The email application 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*.
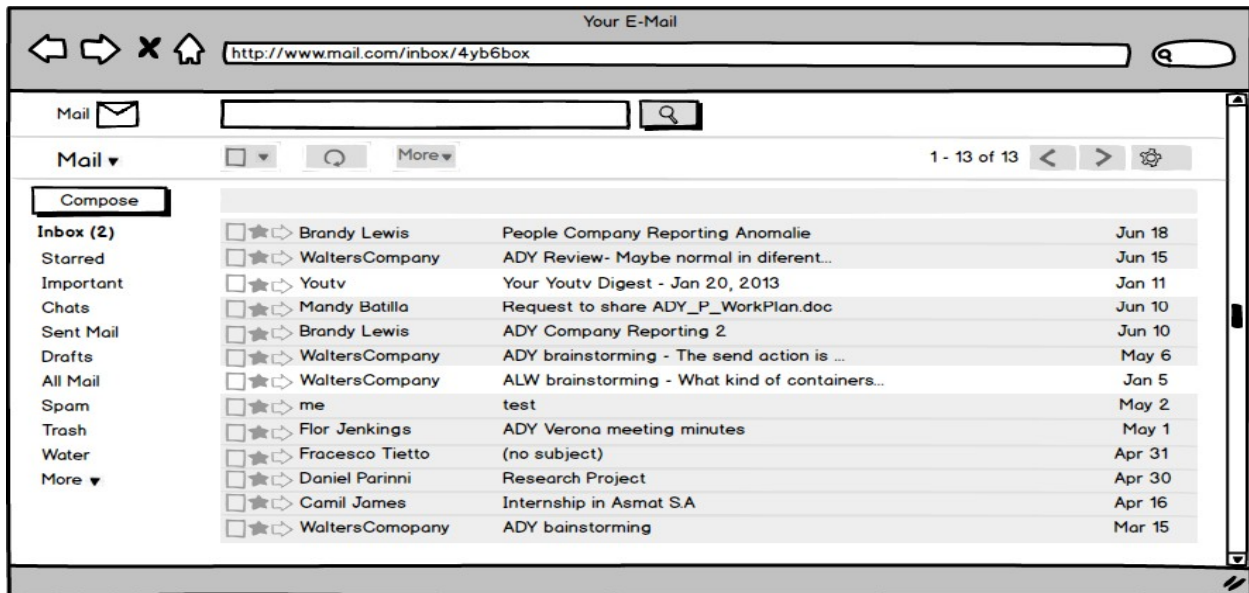

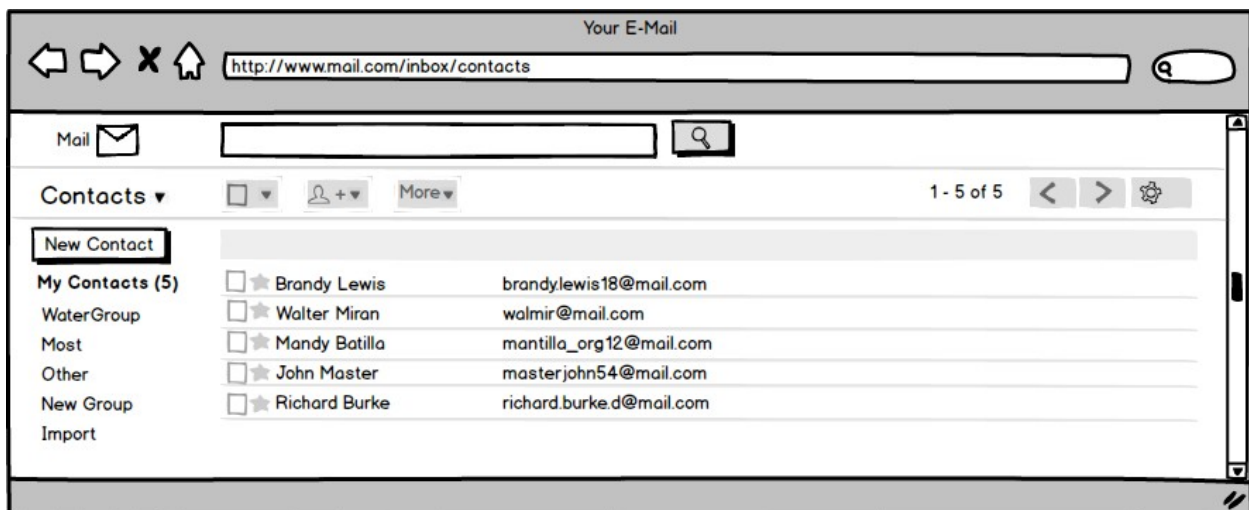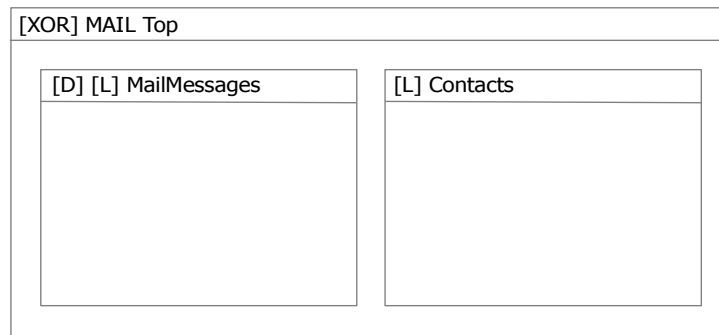**Figure 34: The email application view container for MailMessages**

Figure 35: email application view container for Contacts

By default, when the application 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 36.

Figure 36: A menu allows one to switch from the MailMessages to the Contacts view components

The model of the top level container of the application is shown in Figure 37

Figure 37: IFML model of the Top Container of the email application.

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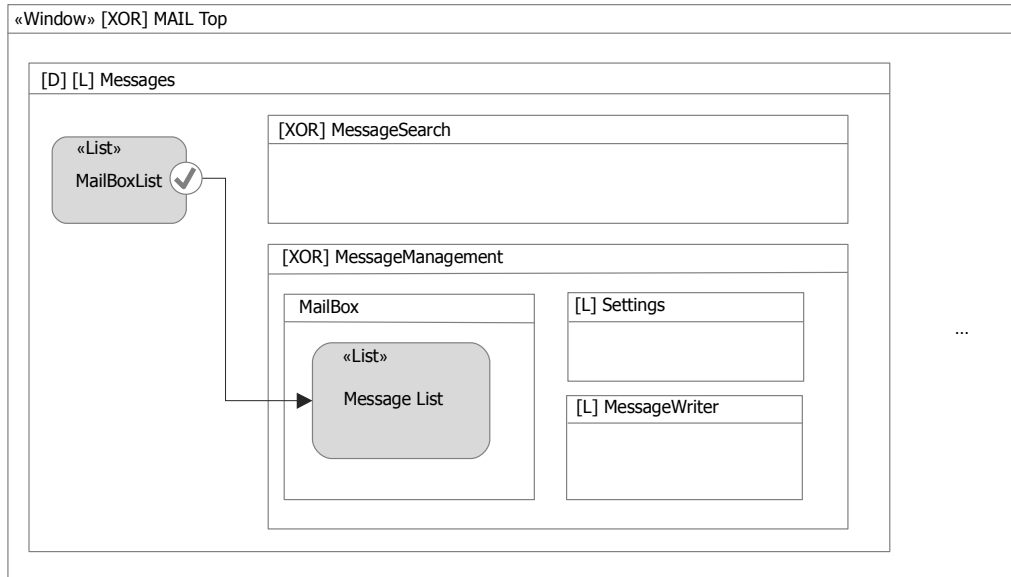
The *MailMessages* view container comprises five main nested elements:

- 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 the email application.

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 38.

Figure 38: Model of the *MailMessages* view container: a navigation event and parameter passing flow between the *MailBoxList* 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 *MailBoxList* 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 38 can be refined to show the parameter binding that binds the selection of a *MailBox* in the *MailBoxList* component and the display of the messages *of that MailBox* in the *MessageList* view component.
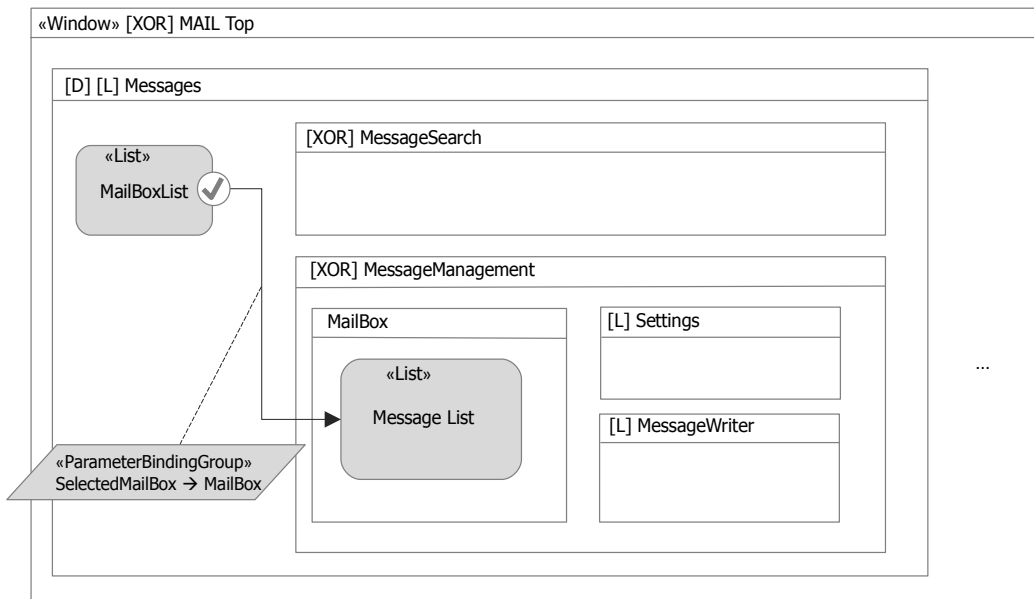
Figure 39: Notations to express (or infer) parameter dependencies between view components.
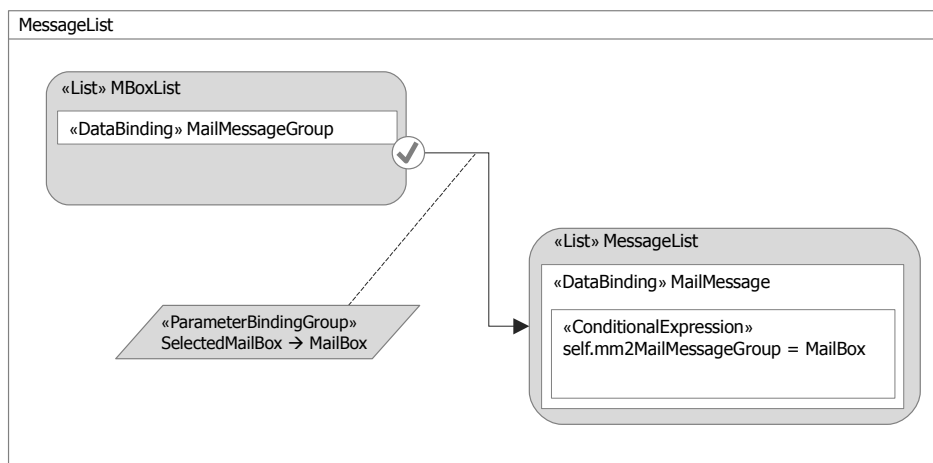
Figure 40: Notations to express (or infer) parameter dependencies between view components with extension mechanism.

Language extension and notation

1. In the upper part of Figure 40, 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 40, 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[4]. The component refers a content binding of the DomainModel where the objects of

---

[4]   IFML has an extension mechanism whereby generic view and business components can be extended to introduce domain-specific view and business logic. Object publishing and CRUD operations on objects are typical examples of extended

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 the example DomainModel) 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 *MailboxList* 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 41).


Figure 41: The MarkAllAsRead user-generated event marks all messages in the current mail box as "read"

As shown in Figure 42, 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 41);
2. an event for selecting/deselecting one or more messages (Figure 42);
3. an event for selecting an individual message and opening it for reading.
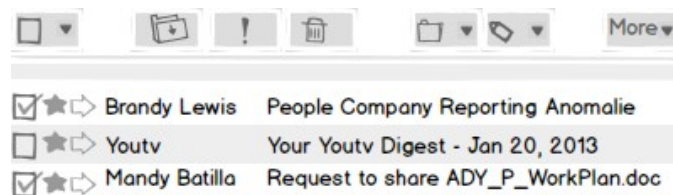

Figure 42: When one or more messages are selected in the *MessageList* component, the *MessageToolbar* view container is displayed, which allow the user to perform several actions of the selected set of messages. If all messages are deselected, such view container is no longer displayed

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 43). 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.
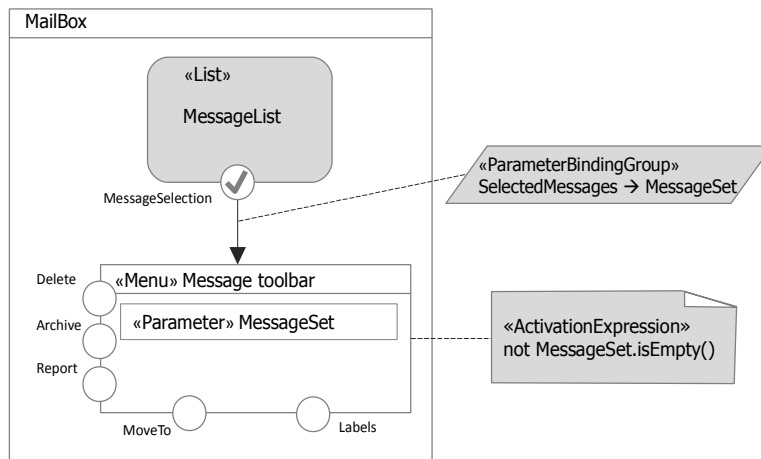
components.
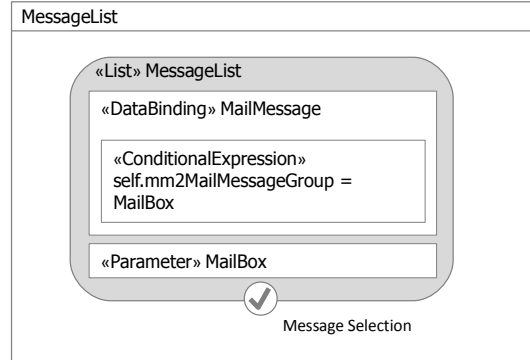
Figure 43: The «Multi-choice List» view component extends the «List» view component to enable more types of interaction events with the element of the list

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 44.

Figure 44: 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 43 and in Figure 44. 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 45. An interaction flow arrow connects the event responsible of triggering the action to the action itself, supporting the

specification of parameter bindings.
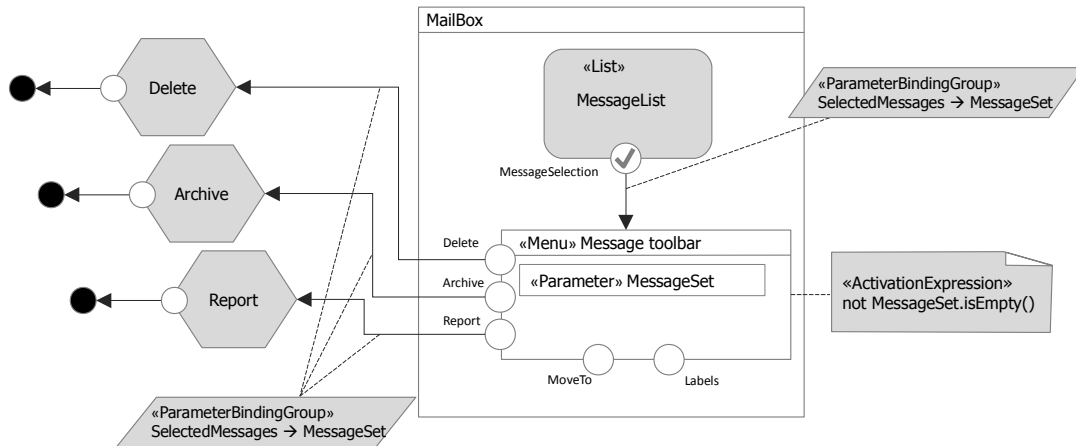
Figure 45: The *MessageList* view component and the *MessageToolbar* view container are associated with events that trigger actions on messages. Actions are represented as components placed outside the view containers, with input and output parameters

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 throwingEvent notification, denoted as a black circle reached by the outgoing InteractionFlow from the Action box. Such a notification throwing event is matched by a catching event, which triggers the display of a *MessageNotification* view component, shown in Figure 46.
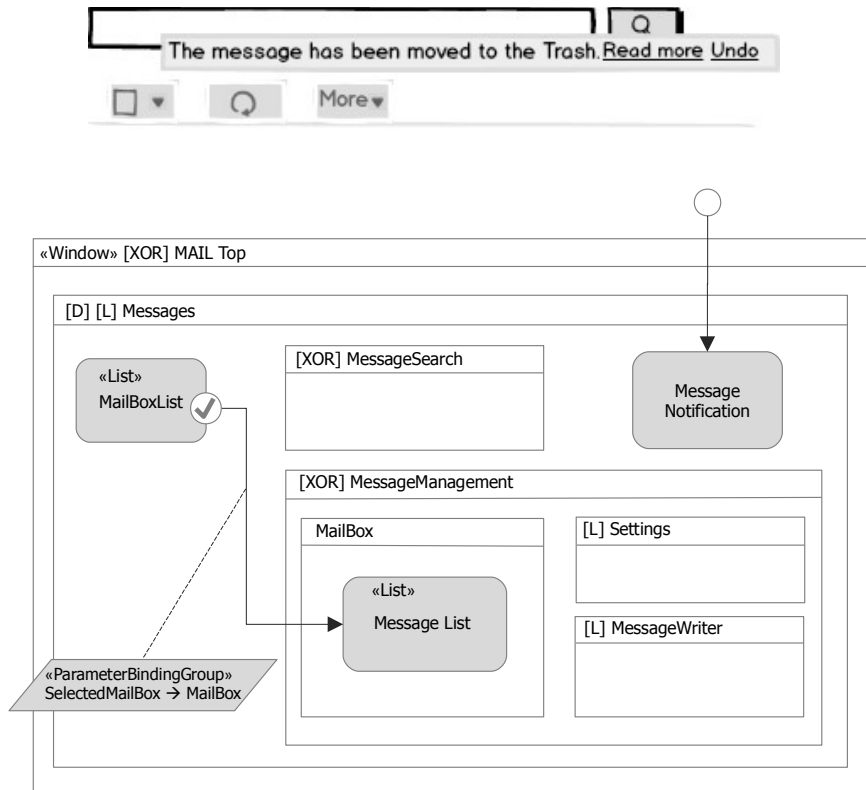
Figure 46: The *Messages* view container comprises a message notification component, which displays notifications of executed actions on *Messages* (illustrated above)

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[5] (not modeled for brevity).

Some actions on mail messages require a more elaborate interaction flow: *Move to folder* and *Associate with tag* (see Figure 50). 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 47) and then selecting from such list the destination *MailBox* or *Tag*.

---

[5]   Modeling the undo also requires discriminating the action to undo, which can be simply modeled, e.g., with an additional parameter denoting the type of action (e.g., delete) set by each action when creating an instance of the notification sending event.
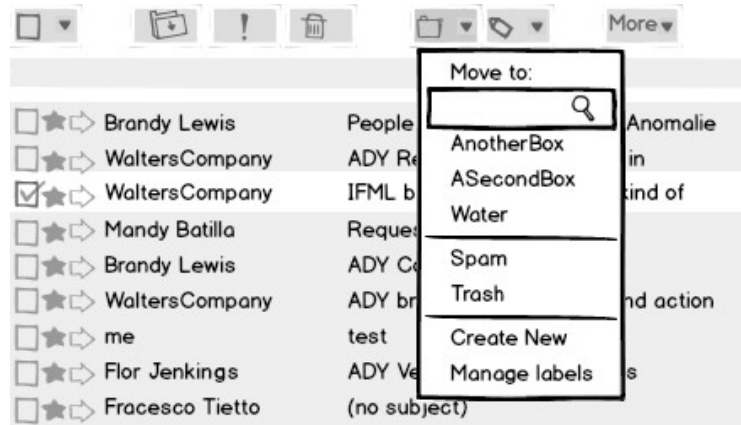
Figure 47: The *MoveTo* action is activated by first accessing a modal view container with the list of the available *MailBoxes* and *Tags* and then selecting the target one. The view container comprising the list of *MailBoxes* and *Tags* is also associated with navigation events for creating new tags 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 48).

Figure 48: 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 49. The view container stereotypes «Modal» and «Modeless» annotate the view containers to specify that they open in a new window and are modal or modeless.
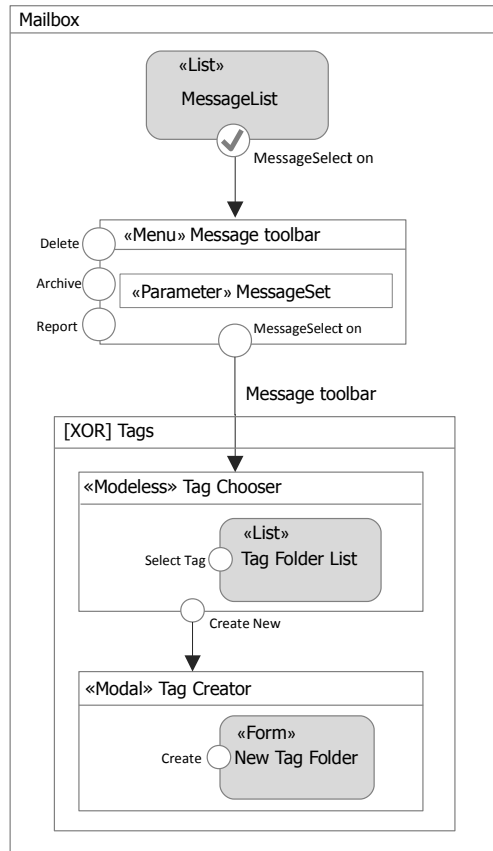
Figure 49: The model of the interaction flow for moving a message to an existing or newly created tag. The view container *TagChooser* is a modeless view container (which hides when clicking outside of it) and the *TagCreator* is a modal view container.

Archiving, reporting, and associating messages to existing/new tags imply the invocation of business logic components, as shown in Figure 50.
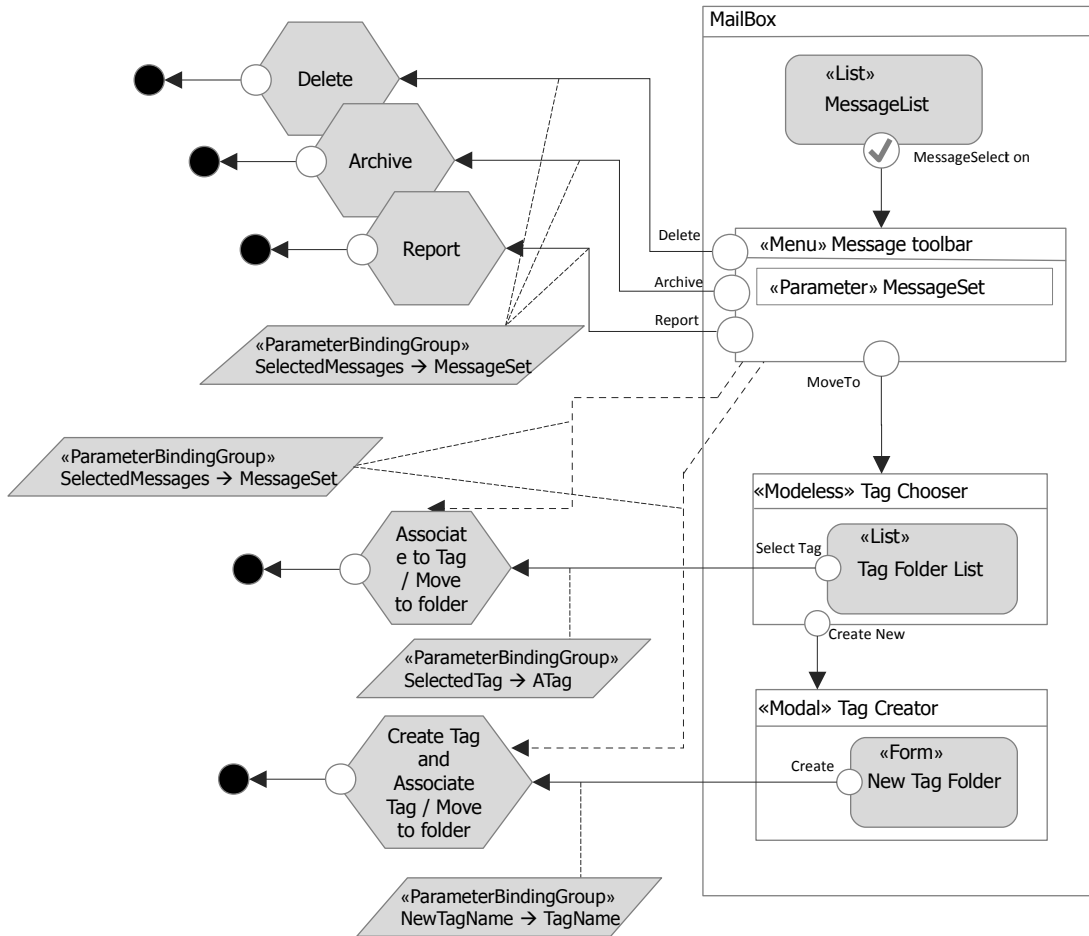
Figure 50: The model of the interaction flow for moving a message to an existing or newly created tag

In Figure 50 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[6]. 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 51.

---

[6]   For simplicity, which only model the AddToTag functionality; the MoveToFolder command is similar.
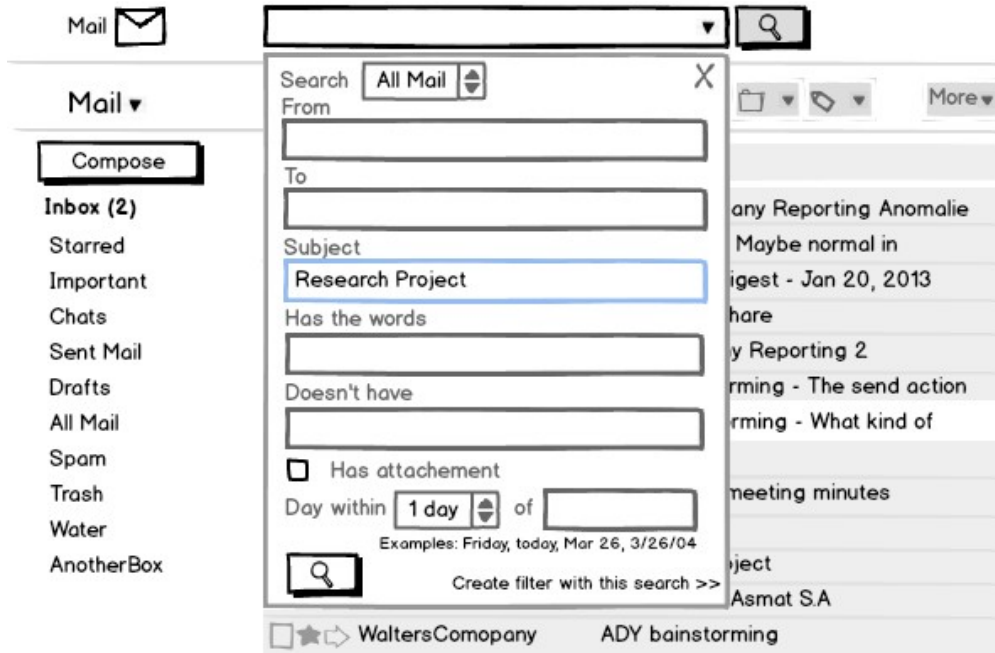
Figure 51: The message search functionality (full search modal view container)

The IFML model of the search functionality (shown in Figure 52) 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 Form 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 52 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 «Form»), composed of a set of typed fields (e.g., denoted as nested view components of type «SimpleField»); a «Form» 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 52). 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.
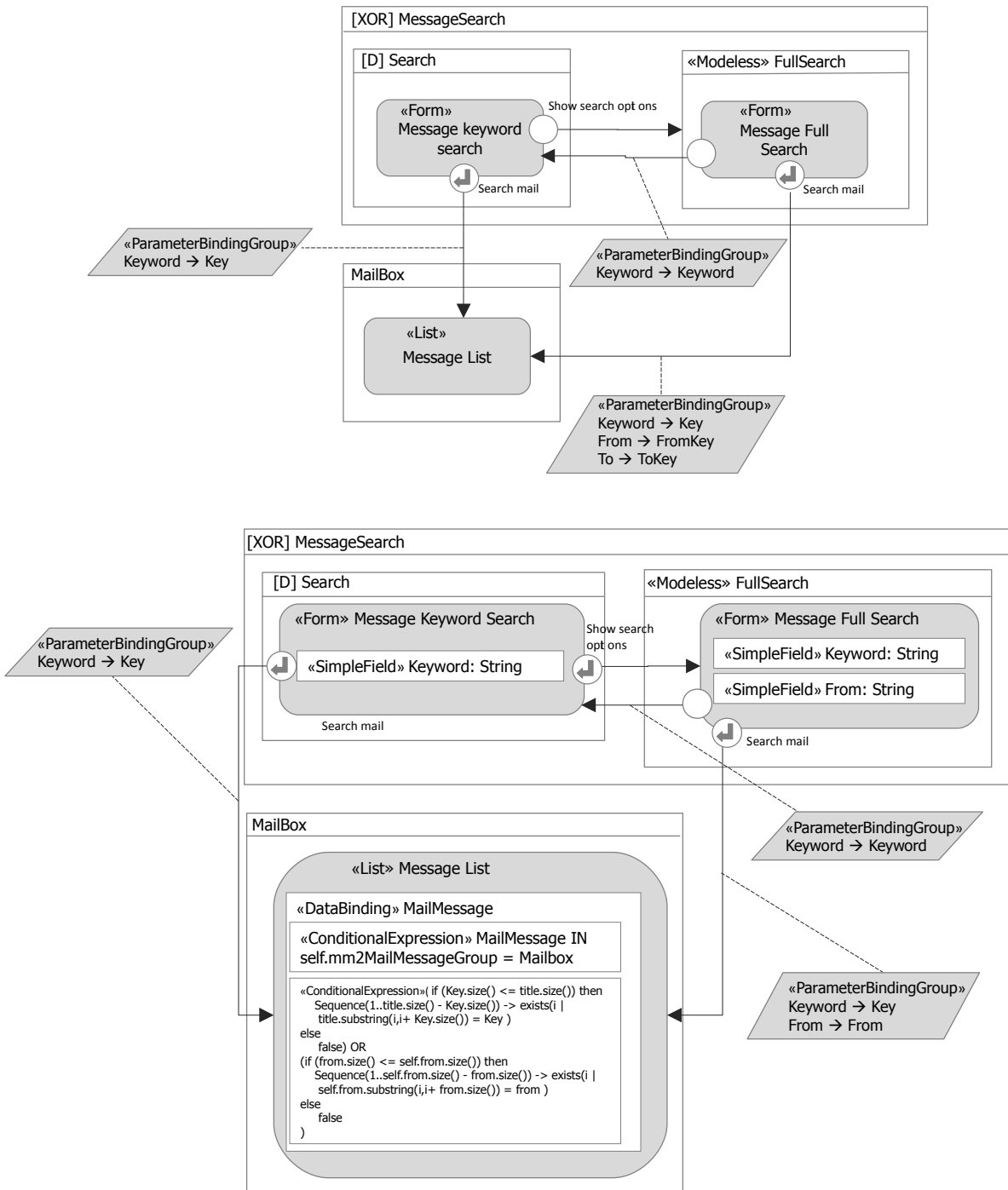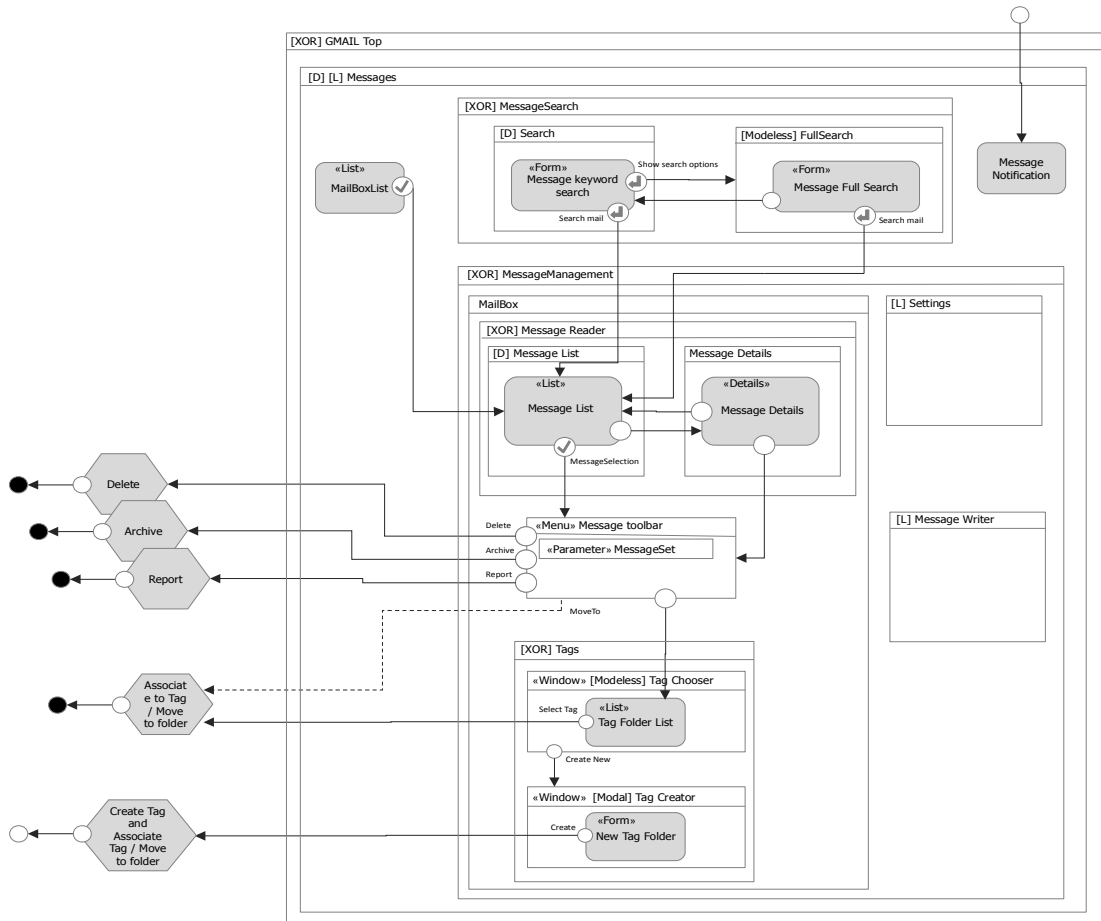
Figure 52: The model of the message search functionality (top). The same model refined with the use of the extended view components «Form» and «List» (bottom)

As shown in Figure 53, 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. The XOR *MessageReader* ViewContainer enables alternative visualization of the MessageDetails and MessageList ViewComponents. .

Figure 53: The *MessageList* and the *MessageDetails* view components are shown in alternative

The example continues with the model of the message composer functionality. This can be activated in two ways: 1) from any view containers inside the *Messages* top view container as denoted by the landmark icon of the *MessageWriter* view component; 2) from the *MessageDetails* view component, by activating the *Reply*, *ReplyToAll*, or *Forward* command, as denoted by the three event and interaction flows from the *MessageDetails* view component (shown in Figure 54).
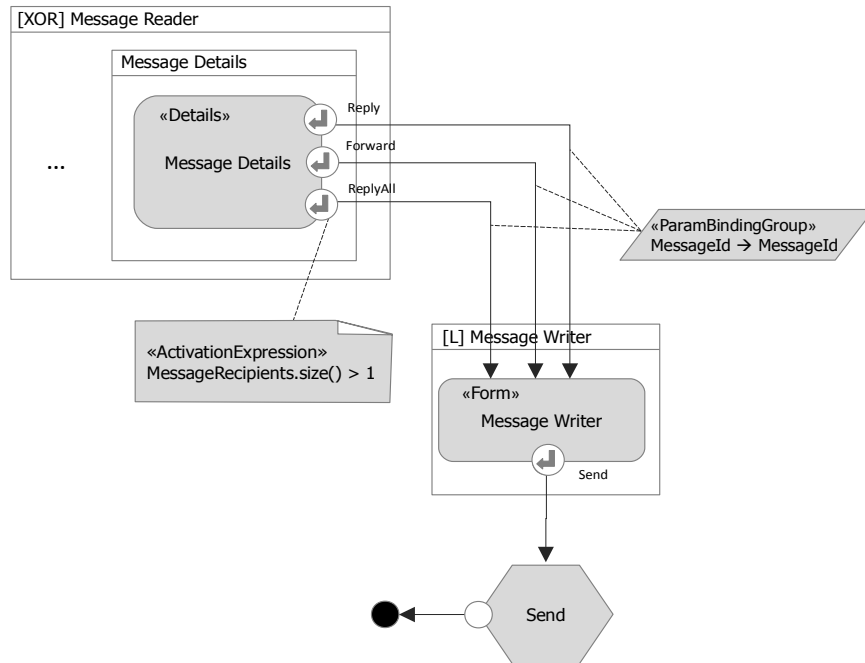
Figure 54: The different ways to access the *MessageWriter* view component

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 54). The *MessageWriter* view component has an internal structure, shown in Figure 55.

Figure 55: 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*.
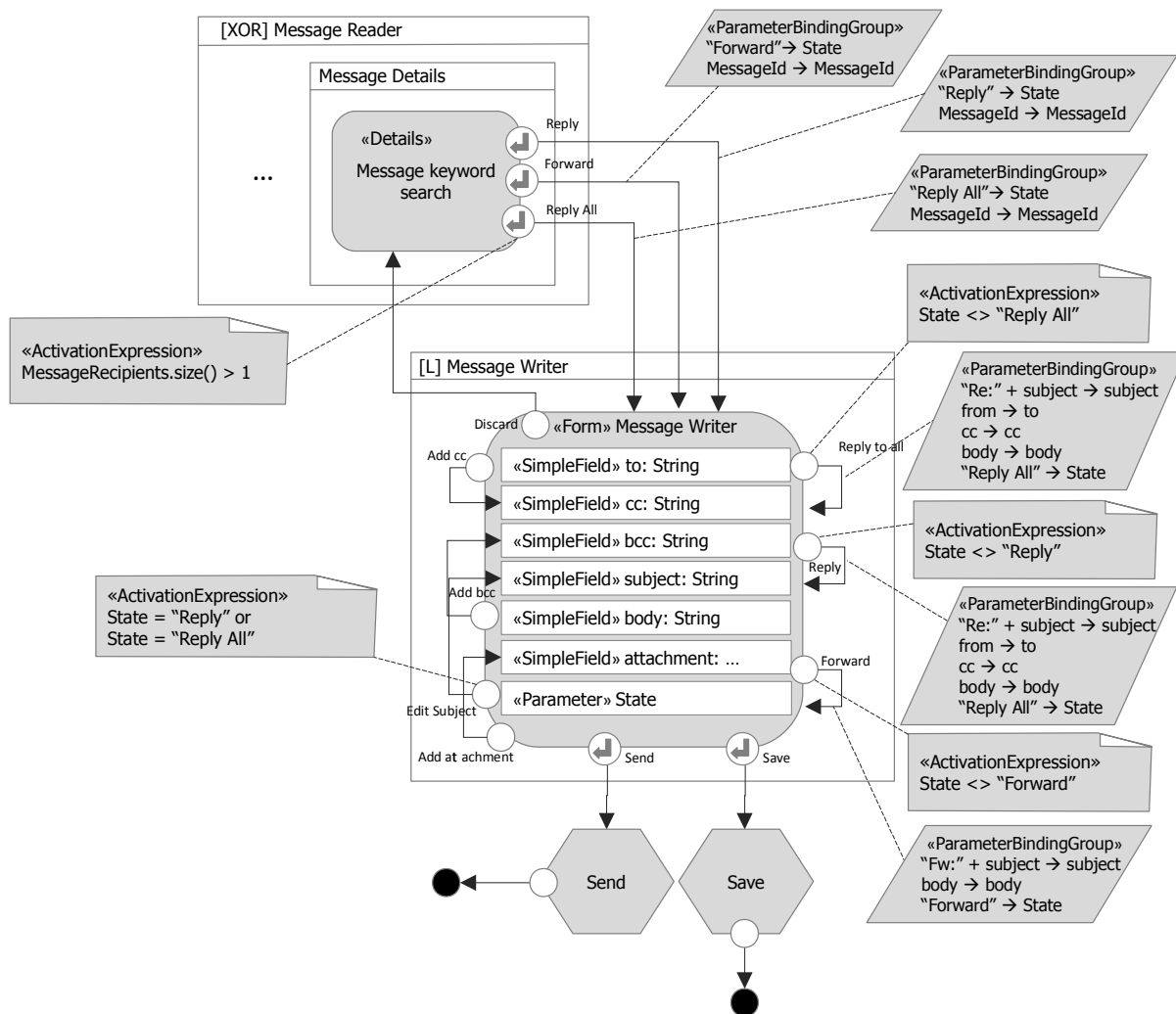
Figure 56: The IFML model of the internal structure of the *MessageWriter* view component, with the names of the event displayed for clarity

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 «Form» 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 55 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 56 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.
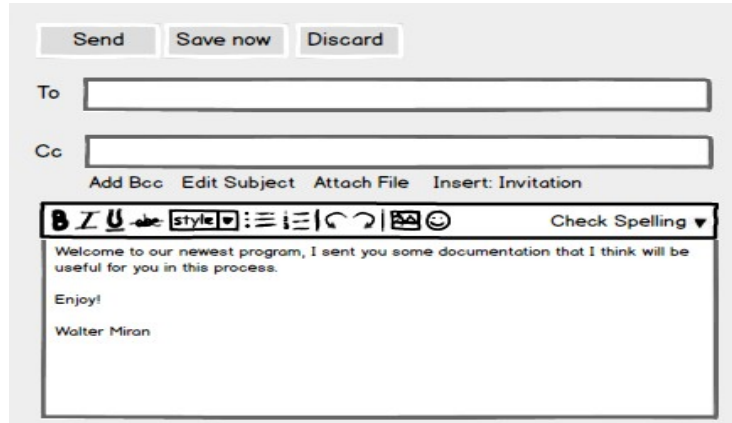
Figure 57: The rich text editing toolbar in the Body input field of the MessageWriter view component

Figure 57 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 55.

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.
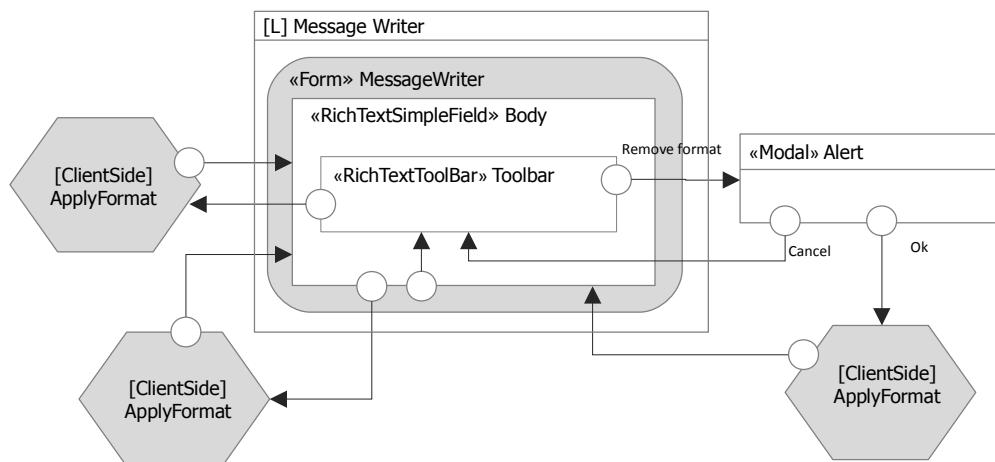
Figure 58: The rich text editing toolbar in the *Body* input field of the *MessageWriter* view component

Figure 58 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 57) permits one to remove the formatting and go back to the plain text mode; before firing the action, tough, 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*.

Annex B   IFML by Example: Modeling an Online Bookstore (Informative)

This annex exemplifies the versatility and adaptability of IFML by modeling the most common features available in a simple UI for a point of sale (POS) management, specifically targeted to a bookstore environment.

B.1 Domain Model

During the session, an **User** is assigned a **Shopping cart** that at the beginning is empty. As the user browses through the page and gets information about the products available, adds products to the shopping cart. The list of products selected at the moment by the user, can be consulted at any time, offering the option of pay the current order, empty the cart or continue browsing in order to add more products.
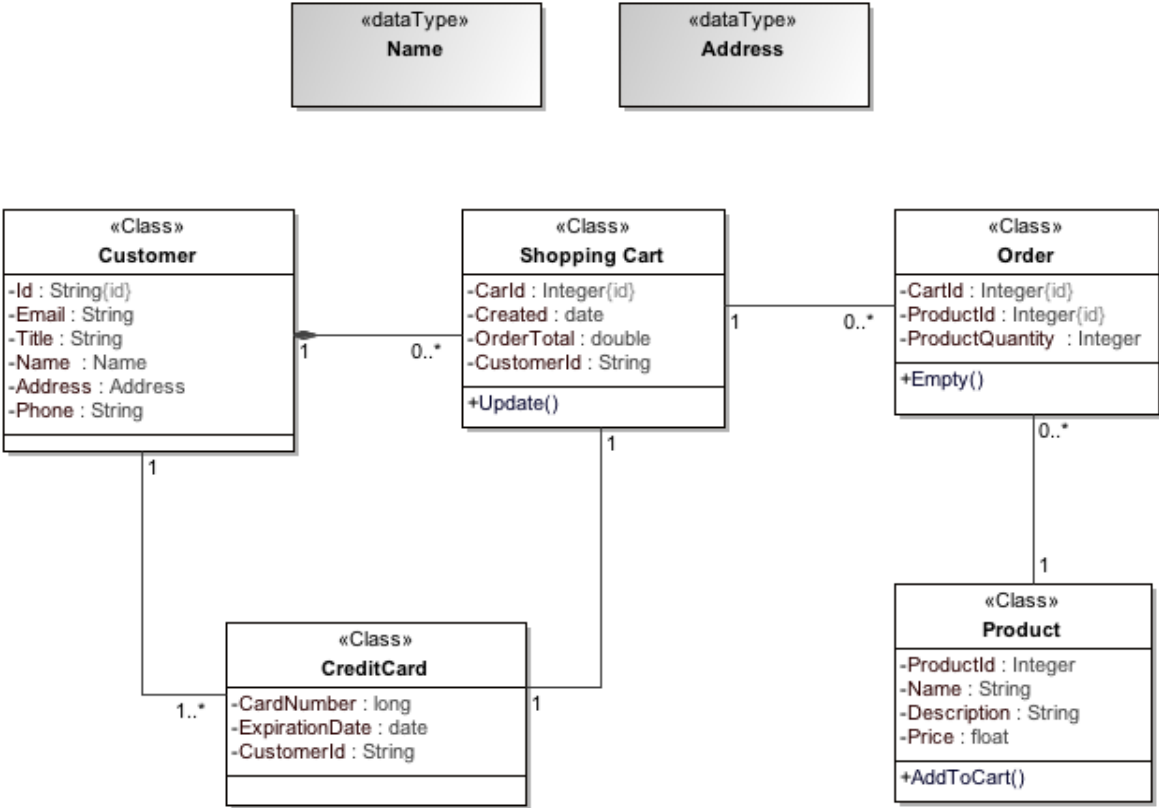
Figure 59: Domain Model of the Online Bookstore

B.2 Process Model

When the user enters into the website, starts exploring the products available. Once he finds a product of interest, selects it, and the item goes to the shopping cart. The user can either keep exploring products in order to add more items to his order,   or continue to manage the shopping cart by deleting all the products, or updating quantities of the selected ones. Once the user is ready to proceed with the payment, performs the checkout.

In order to authorize the payment, it's necessary to send the customer information to the bank entity, and wait for the confirmation. This procedure is illustrated in the Figure 60.
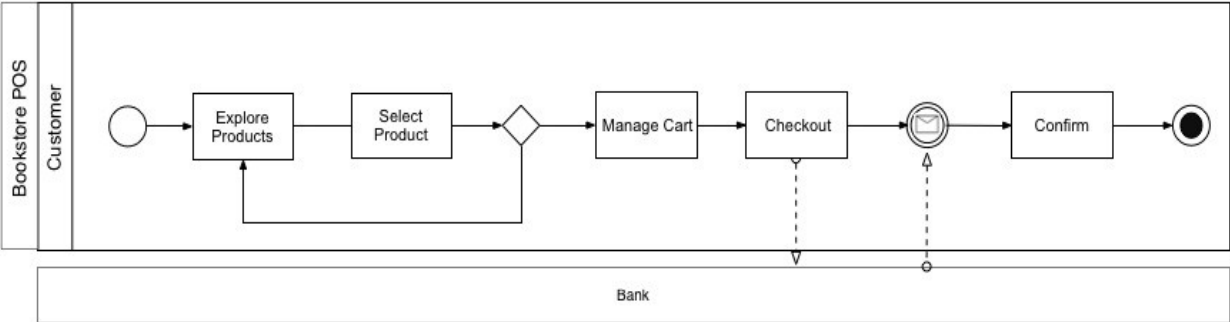
Figure 60: Process Model of the Online Bookstore

B.3 Model of the User Interaction Flow

Figure 61 shows the home page of the online Bookstore. In this section, the user can select one of the product categories, or go directly to the shopping cart.
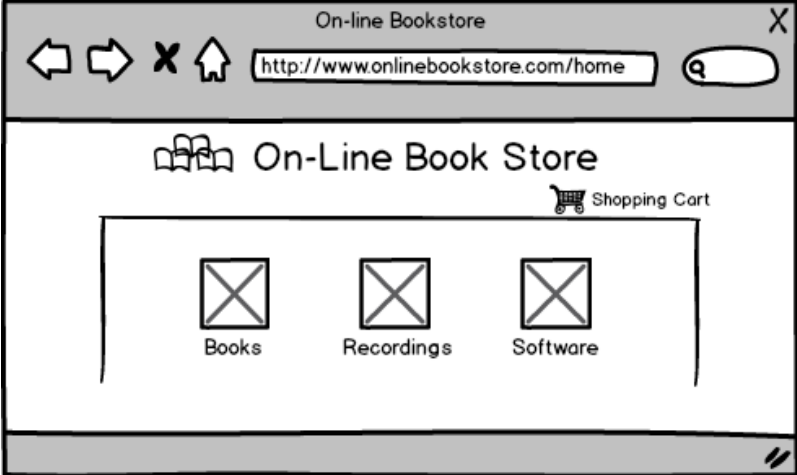
Figure 61: Online Bookstore Home Page

After selecting a category, a list of products is displayed. For instance, Figure 62 shows all the products belonging to the books category.
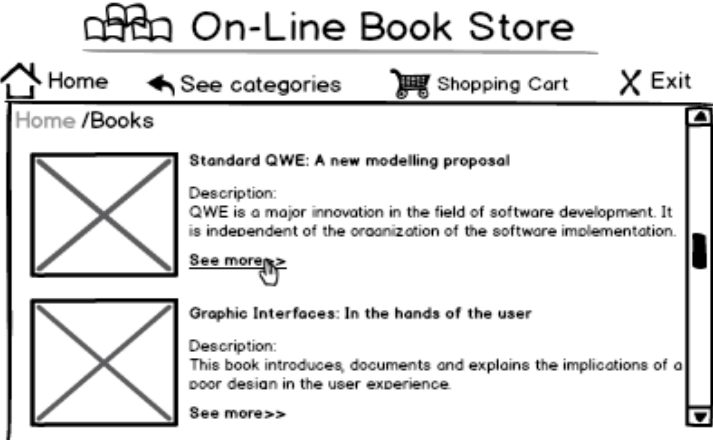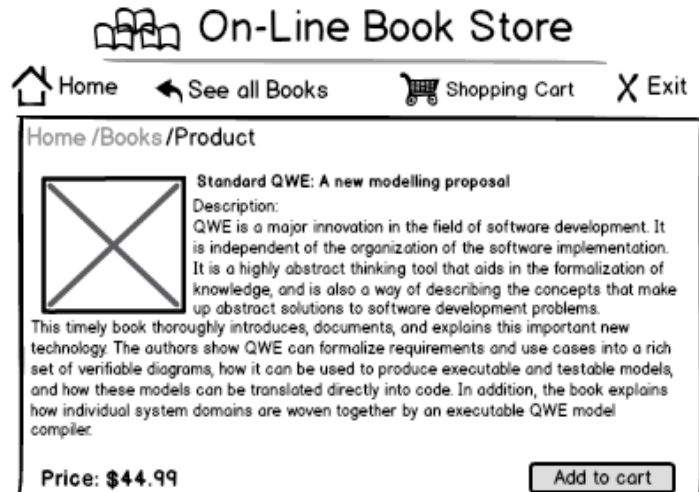

Figure 62: List of products belonging to the books category

When the user selects a product obtains the details of the selected item (such as full description and price) along with the option to add the product to the cart, as shown in the Figure 63.

Figure 63: Details of the Selected Product

The procedure described in the figures 61,62, and 63 is represented in IFML as shown in the Figure 64. Once the user selects a category from *CategoryList* a navigation event is produced, and as a result, the products corresponding to the *SelectedCategory* are displayed. Similarly, when the user selects a product from *ProductList*, the details of the *SelectedProduct* are displayed.
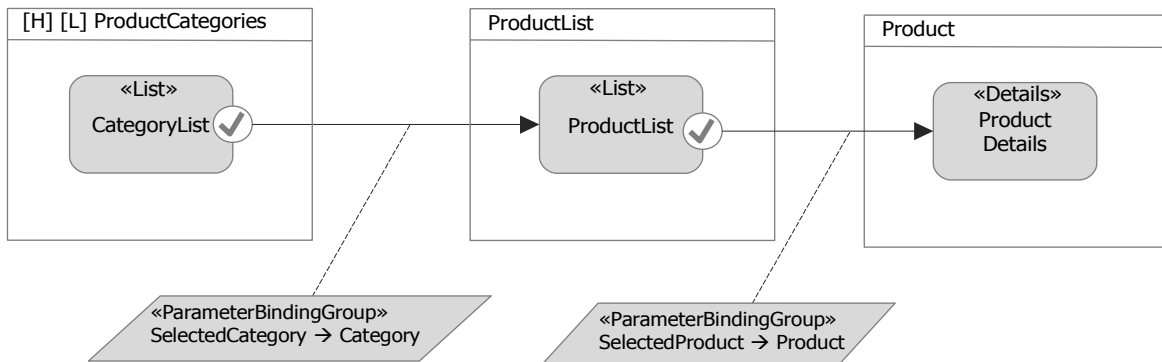
Figure 64: IFML model corresponding to the exploration of products

When the user decides to buy the product and add it to the cart, causes a modal view container to be displayed, where the user must provide the quantity of items of the desired product (see Figure 65). After accepting the quantity, the article is added to the cart, and a confirmation window is displayed as shown in the Figure 66.
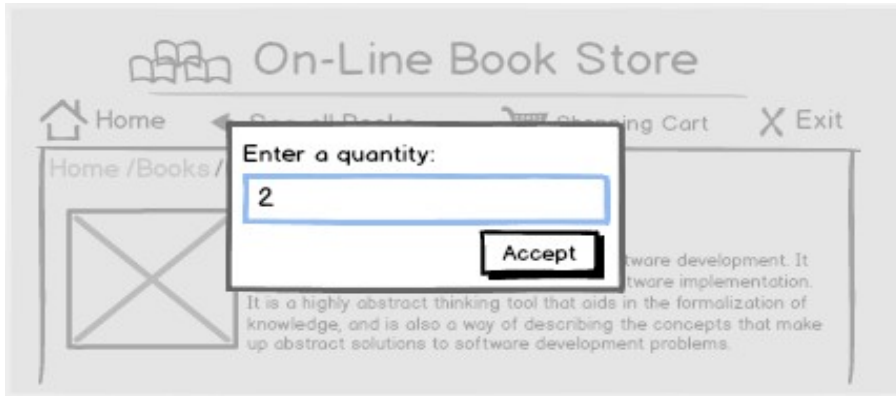
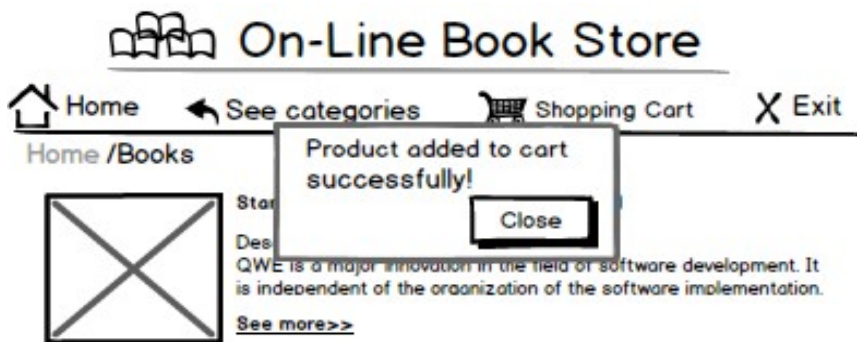**Figure 65: Figure 7.Window displayed in order to catch the number of items desired by the user**

Figure 66: Confirmation window for the action add to cart

Figure 67 shows the model fragment that adds a product to the cart: once the user press the add to cart button, a modal window appears asking for the quantity of items desired. This value, along with the *SelectedProduct* are submitted as parameters and represent the input of the add to cart action triggered. Once the action is performed, a confirmation window is displayed.
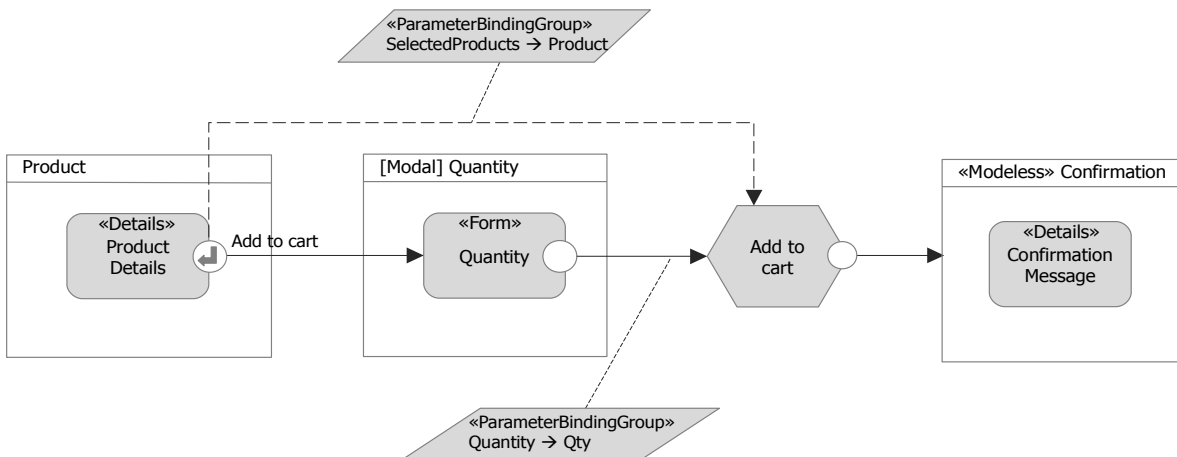
Figure 67: IFML model corresponding to the add to cart event

The shopping cart is the list of products previously selected by the user. In this section are shown the quantities and

the order details. The user is able to *update the cart* by changing the quantities, e*mpty the cart* by deleting all the products of the current order, and start the payment process by clicking in the *checkout* button (see Figure 68).

When the user chooses to update the cart, the total amount is recalculated.

When the user empties the cart is redirected to a confirmation page as shown in the next Figure 69.
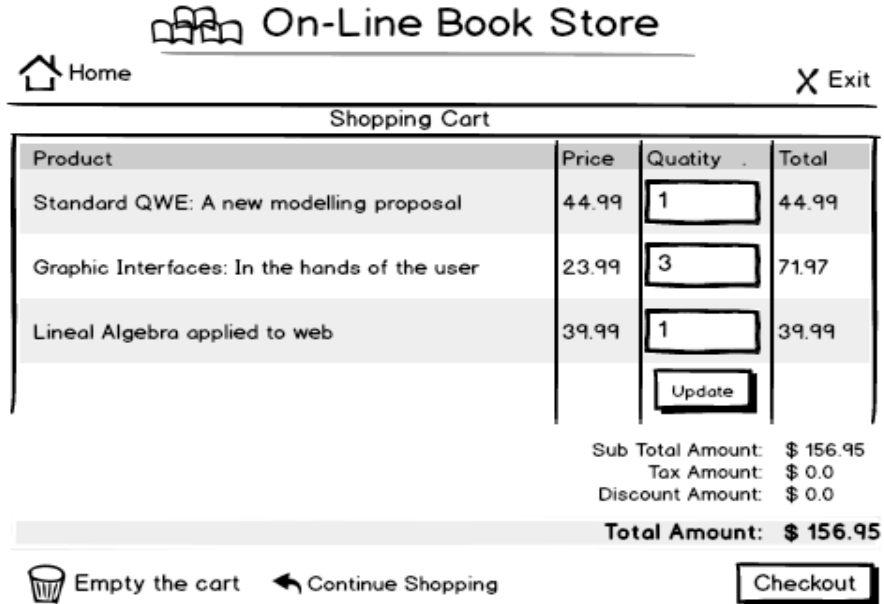

Figure 68: Interface of the Shopping Cart
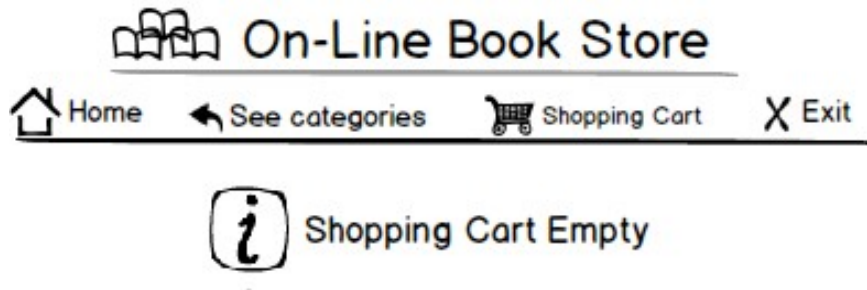
Figure 69: Confirmation page for the Empty Shopping Cart Event

As illustrated in the IFML model of the Figure 70, when the user decide to delete all the items from the current order, the action *Empty the car*t is triggered, and after its execution, a confirmation message is displayed.

In the *Update* event, the user modify the values of the quantities and submits them by clicking in the button *Update*; this event causes an *Update* action to be triggered after which the shopping cart is redisplayed (see Figure 70).
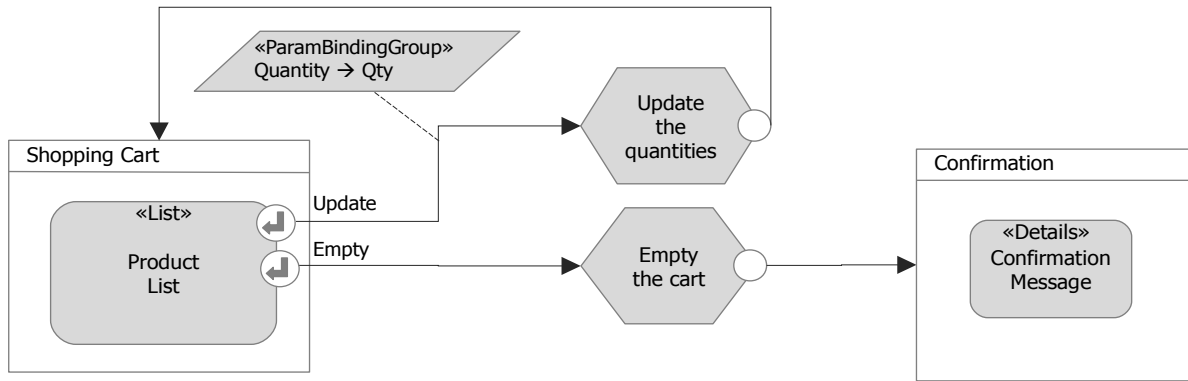
Figure 70: IFML model corresponding to the events Update and Empty of the Shopping cart

Once the user has decided to perform the payment, he must provide his personal information and press "Next" (see Figure 71).

Figure 71: The user must provide its personal information and continue

After providing his personal information, the user must provide his bank account information and confirm the payment in order to proceed with the transaction (see Figure 72). After performing the transaction, a confirmation page appears showing the details of the payment as shown in the Figure 73.

Figure 72: The user must enter the bank account information and confirm the payment

Figure 73: Payment confirmation

When the user chooses the Checkout option, the container *Customer Information* is displayed. The user must provide his personal information by filling out the form within this container.

After the user submits his personal information, the container *Payment Information* is displayed. In this container the

user must provide his bank account details. The name of the user (sent previously as the parameter: Name), is forwarded along with the credit card number (CC) and the total amount of the offer (previously sent by the shopping cart container) to the payment action (Execute the payment).
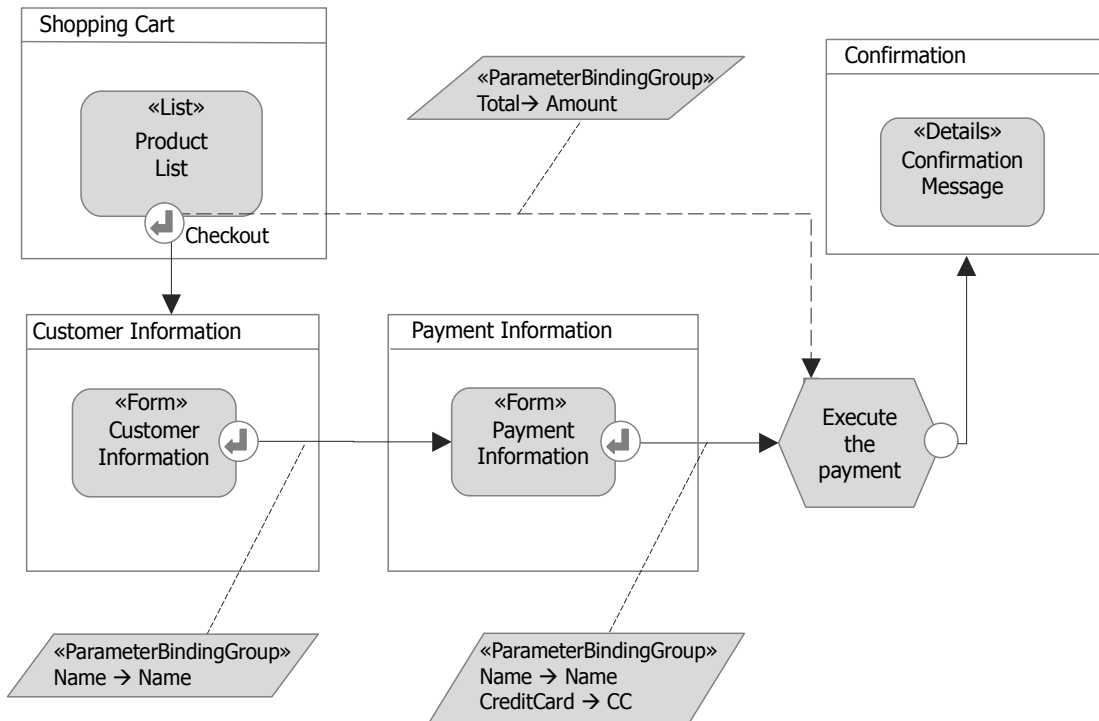
Figure 74: IFML model corresponding to the event Checkout

After the payment execution, a confirmation message is displayed with the transaction details. The IFML representation of this procedure is shown in the Figure 74.

To increase reusability and modularization in the models, designers may decide to cluster homogeneous parts of the model into Modules. For instance, the part of the model that deals with the payment management can be packaged into a specific module. This would simplify the model of the application, which would appear as in Figure 75.

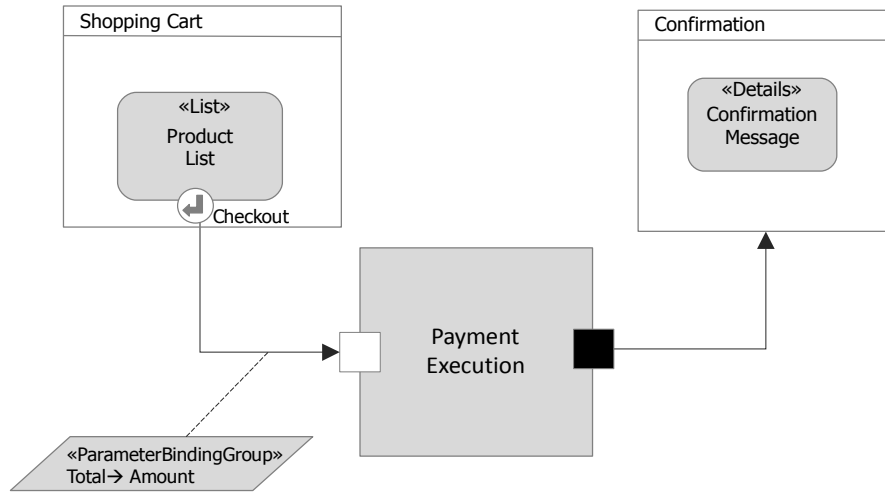The definition of the corresponding module is shown in Figure 76.

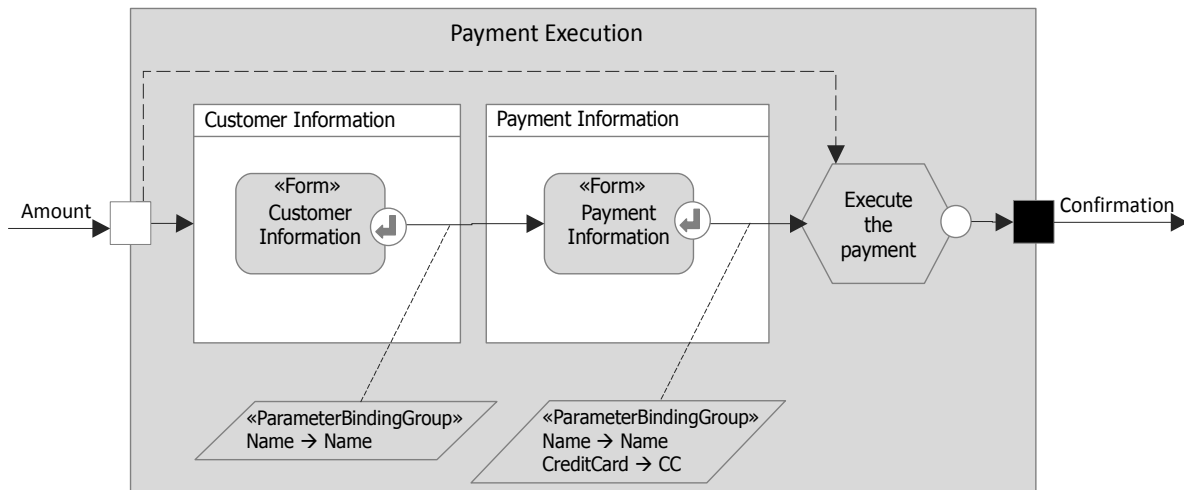**Figure 75: IFML Module usage upon the Checkout Event**

Figure 76: Module Definition of Payment Execution

B.4 System Modeling

IFML can be suitably used together with UML models and other OMG standards (e.g., BPM models).

For instance, UML sequence diagrams complement IFML models at the purpose of highlighting sequences of activation of client- and server-side components depending on user interaction events.

In the example, when the customer chooses the option update, the *Browser* sends a message to the *WebServer* with the id of the product and the new quantity, then the *WebServer* updates the shopping cart and returns a confirmation message.

If the user decides to delete all the products previously selected, he clicks the empty cart button, sending the message to the *Browser*. The *Browser* sends a message to the *WebServer* who is in charge of executing the deleting action and return a confirmation message.

When the user is ready to proceed with the payment, notifies the *Browser* who asks to the *WebServer* for the customer information form. After the *WebServer* returns the form, the *Browser* displays it. The next step to continue with the payment is wait for the user to fill out the form with his personal and bank information. When the user submits his information, this is sent to the *WebServer* who asks the *DataServer* to return the customer information in order to verify it. After verifying the customer information, the *WebServer* sends it to a *ExternalBankService* who is in charge of authorize the payment. Finally, after the *WebServer* receives the confirmation from the *ExternalBankService*, sends a confirmation message to the *Browser*
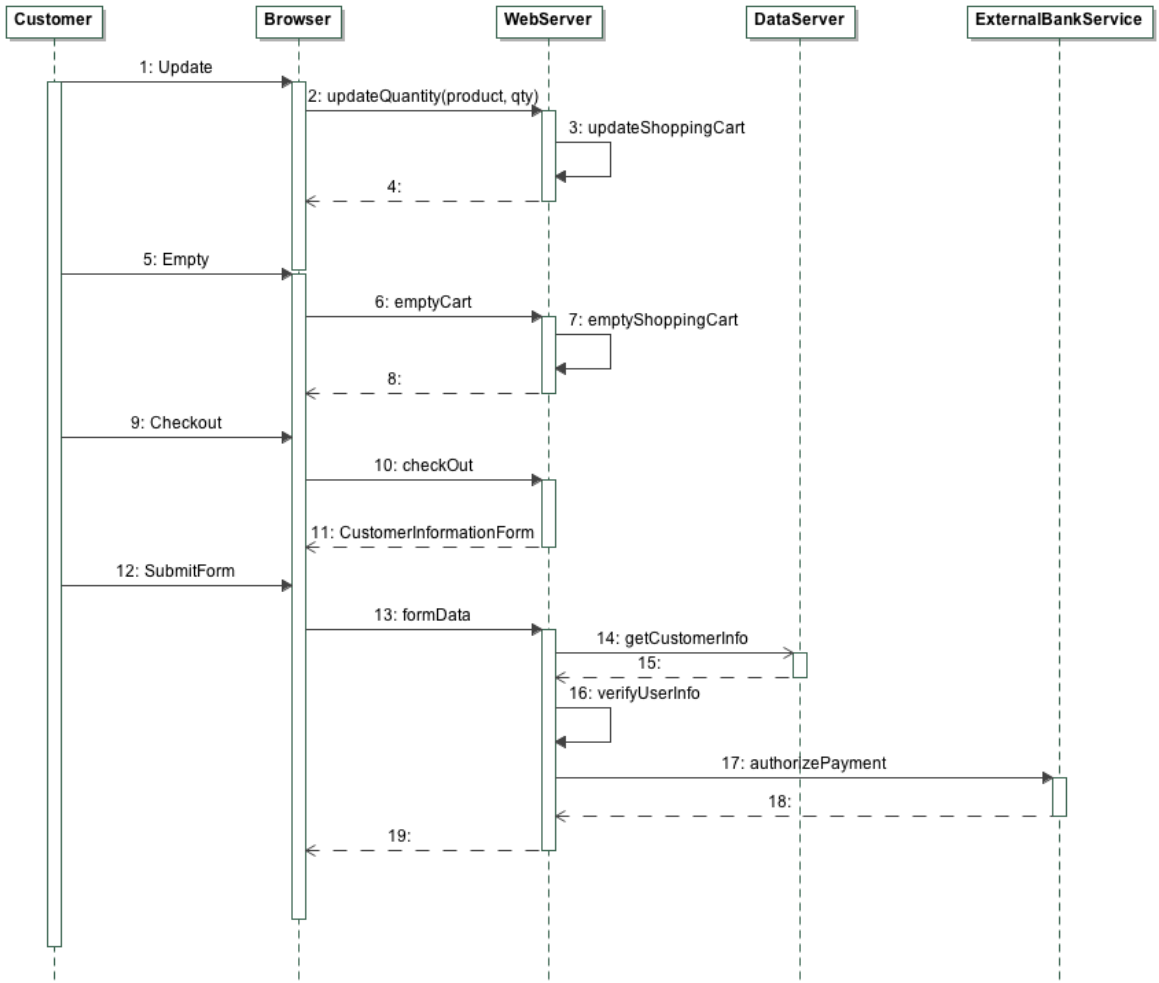
Figure 77: Sequence Diagram of the Online Bookstore

Additional diagrams can be used to describe the deployment of the components and other aspects, as shown in Figure 78.
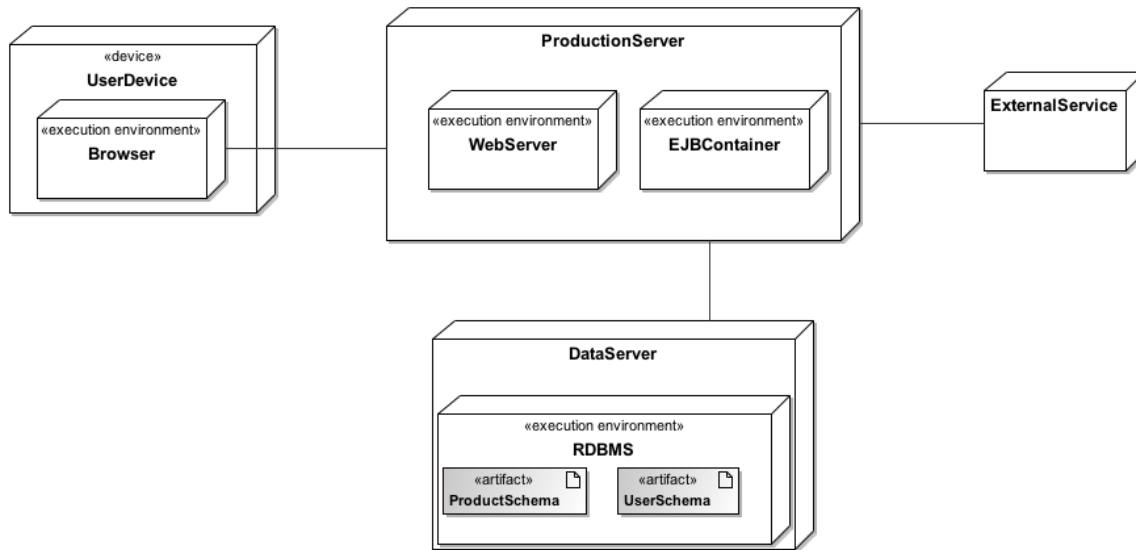
Figure 78: Deployment Diagram of the Online Bookstore

Annex C   Mapping to the Windows Presentation Framework (Informative)

C.1 Introduction

This annex describes an example of mapping from IFML to a platform specific language. In particular, this maps the main IFML concepts to the .Net Windows Presentation Framework (WFP).

C.2 The WPF meta-model

Windows Presentation Framework (WPF) is a part of .NET Framework by Microsoft that is meant to be the substitute of the old WinForms UI interface. It brings separation of concerns between interface and code-behind. This is made possible by detaching presentation defined using the XAML language from business logic written in C#.
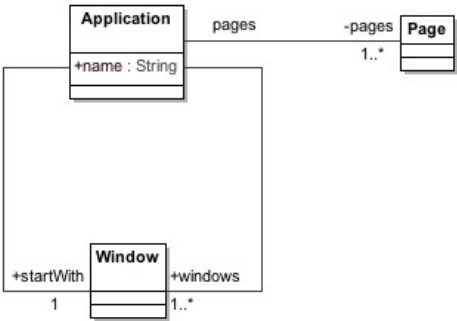
Figure 79: WPF metamodel, the Application element

In WPF the interface building blocks are nested. This generates a visual tree that is rendered by the framework.

The target application is modeled by the **Application** class which is the main container of all the elements of the model. It has a start window which is the first one to be opened at startup.

Figure 80: WPF metamodel, the DependencyObject element

All the visual objects inherit from **DependencyObject**, a class that allows the attachment of **DependencyProperty**. This lets define properties that may be shared among all the objects of the framework and used as target for bindings.

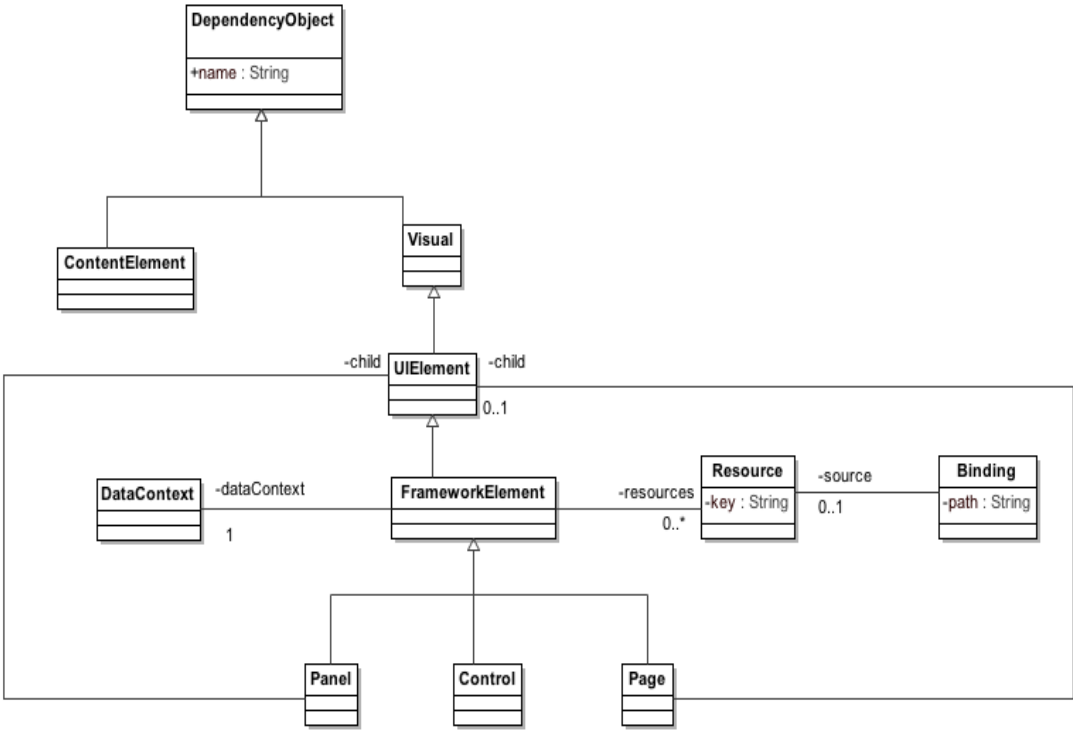**DependencyObject** can be split in two classes, **Visual** and **ContentElement**. Visuals elements are actually rendered by the framework, while **ContentElement**s are used to better define the layout of **Visual**s.

The main subclass of **Visual** is **UIElement** which is used as common superclass to define nesting among elements of the UI.

The main subclass of **UIElement** is **FrameworkElement** which is the one that allows to define **Resource**s and the **DataContext**. **Resource**s are objects related to the **FrameworkElement** organized as a dictionary; they are used by the framework to enhance and better define layout and behavior of the interface. **DataContext** can be associated through a **Binding** to another object to define the source of all the contained **Binding**s, not otherwise specified.
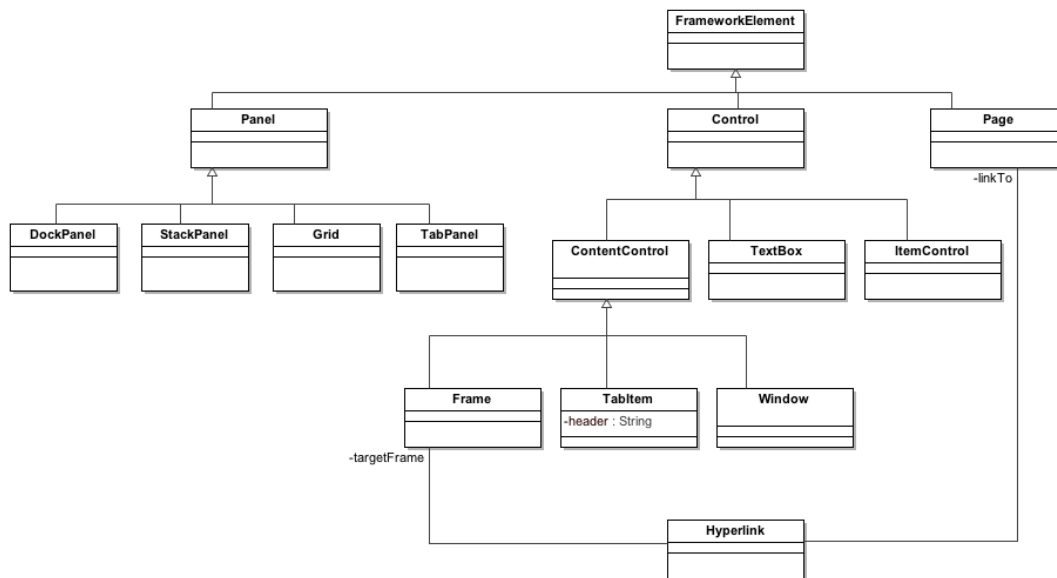
Figure 81: WPF metamodel, the FrameworkElement element

FrameworkElements can be divided in **Panel**s, **Page**s and **Control**s.

Panels are UI elements which can contain more than one child. They are classified by behavior:

- **DockPanel**: this container tries to minimize space wasting by expanding all the children to fit all the available space.
- **TabPanel**: it defines a XOR behavior (one by one), allowing to select the child to display through a tabbed header.
- **StackPanel**: it put all the children in a stack, queuing them one after another.
- **Grid:** it features a m by n grid in which all the children are placed. The coordinates of the cell in which the child resides is defined by the attached properties **Grid_Column** and **Grid_Row**.

Pages are one-child containers that allow navigation in a **Frame**.

Controls include **TextBox**es, **ContentControl**s and **ItemsControl**s.

ContentControls are **Window**s, **UserControl**s, **TabItem**s and **Frame**s.

- **Window**s are the outer containers of all **UIElement**s and have at most one child.
- **TabItem**s are one-child containers that allow to define the header used by a **TabPanel**.
- **Frame**s are controls that can dynamically navigate through **Page**s using **Hyperlink**s or explicit navigation.

ItemsControls are meant to dynamically define their children applying a template to items to be retrieved by an **ItemsSource**.

C.3 Model to Model Transformation

The IFML model is mapped to a WPF application as one window (the startup one) that contains a frame in which it's possible to navigate within pages.

All the first level ViewContainers are mapped to pages; to bypass the limitation related to the one-child nature of pages in WPF, ViewContainers with one child are mapped directly, while the ones with more children are mapped to pages with a grid as a child.

If there is at least one first level landmark ViewContainer, the main window does not contain directly the frame, but a grid with two children: the frame and a StackPanel that contains Hyperlinks to all the landmarked pages.

All the sub-ViewContainers are mapped to grids; otherwise, if they are XOR, they are mapped to TabPanels whose children are surrounded by TabItems.

All the ViewElementsEvents of type OnSelectEvent that reference a ViewContainer are mapped to a StackPanel containing Hyperlinks to all the pages linked by outgoing NavigationFlows.

List ViewComponents are mapped to ListBoxes: if they have a ViewElementEvent of type OnSelectEvent with an outgoing NavigationFlow that links to another ViewComponent, they are also mapped to a ViewSource bound to a ObjectObservableCollection and to a grid which DataContext is bound to the ViewSource current item.

Forms are mapped to grids; their fields are mapped to TextBox (SimpleField) or ComboBox (SelectField).

Finally since the WPF metamodel is a direct mapping of the entities that compose the .Net framework for desktop applications, a simple model to text transformation is needed for generating a working application.

Annex D   Mapping to Java Swing (Informative)

D.1 Introduction

This annex describes an example of mapping from IFML to Java Swing in order to model very simple Java-based desktop application.

Java Swing is a Model-View-Controller GUI framework for Java application. Thus it allows to develop desktop application in Java decoupling the data viewed from the interface from the user interface controls through which it is viewed.
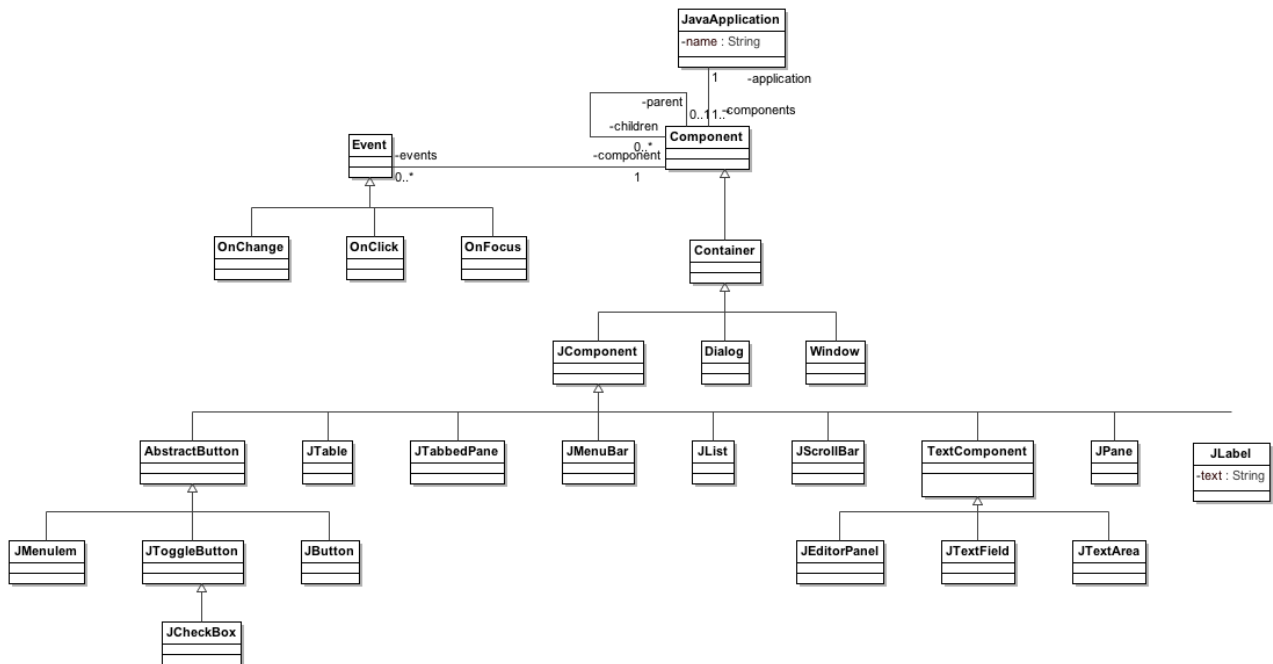
D.2 The Java Swing meta-model

Figure 82: The Java Swing metamodel

The desktop application is described by the **JavaApplication** element, which contains all the **Component**s.

The **Component** element is the abstract description of the element of a graphical user interface. In particular a **Component** can have a set of child element and a set of **Event** used to enable the user's interaction. Furthermore an **Event** can be associated to a set of **Action**s

Every **Component** is a **Container.** In particular there are the **Window, Dialog**, JComponent elements. The first two are pure container while the last comprehends a set of elements that can contain other element or just show data.

The **JComponent** element is then specialized by a set of class that represent the actual GUI elements, for example there are: **AbstractButton,** that model the general button that is more specialized by the class **JToggleButton, JButton, JmenuItem; JTable,** that model a table, **JPane**, **JTabbedPane**, **JScrollBar**, **Jlist**, **Jlabel** and **TextComponent**, that represent the general component to edit text, which is further specialized by the class **JTextField**, **JTextArea** and JeditorPanel.

D.3 Model to Model Transformation

The IFML model is mapped to a **JavaApplication** element.

Each IFML::Window element is mapped to a **Window** element (in case of a modal window a **Dialog** is created instead).

Each not XOR sub-ViewContainer is mapped as a **JPane** (while a XOR container is mapped as a **JTabbedPane** with each of its child ViewContainer mapped as **JPane** element).

Forms are mapped as **JPane** elements, their fields are then mapped as **JTextField** (in case of SimpleField) or **JCheckBox** in case of multi selection field).

List are mapped as **JList** elements.

Details are mapped as **JTable** showing at each row an attribute of the DataBinding considered.

If events were defined, the corresponding **Event** is created and associated to the correct **Component.** In particular, in case of Select and Submit a **JButton** is created in order to trigger the event. If an Action was defined, a element of type **Action** will be created.

If one or more ViewContainer marked as "landmark" exist, a **JMenuBar** element will be created in each **Window**, containing all the **JMenuItem** element linking to the landmark ViewContainers.

Annex E   Mapping to HTML (Informative)

E.1 Introduction

This annex describes an example of mapping from IFML to HTML in order to model a very simple web application.

E.2 The HTML meta-model

The web application is modeled by the **WebSite** class, which is the main container of all the other elements. In particular a WebSite is composed by a set of **Pages**. Then the metamodel describes in details the structure of each element.
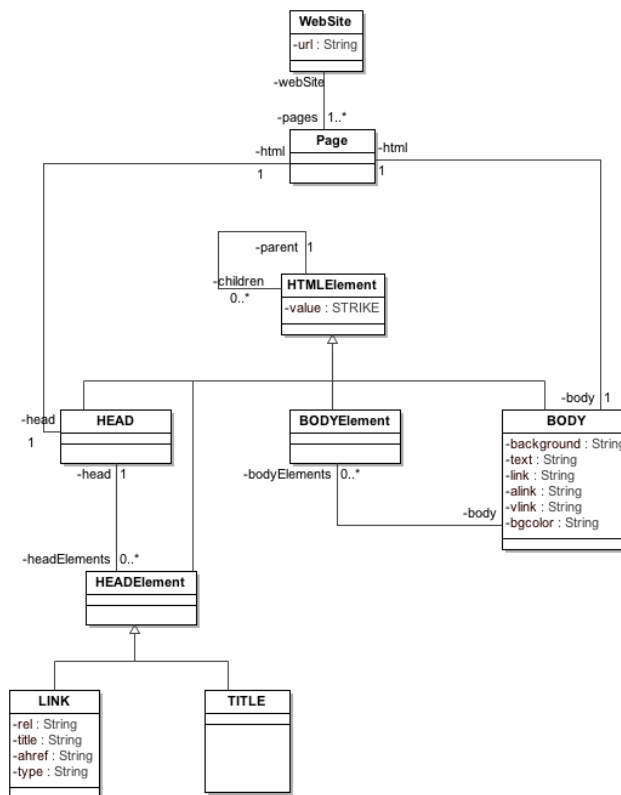
Figure 83: HTML metamodel, the Page and Head element

A Page is composed by a **HEAD** and a **BODY** (represeniting the <head> and <body> tags), the HEAD contains a set of **HEADElement** while the **BODY** a set of **BODYElement**, both of them inhertis from the general class **HTMLElement** and are abstraction of the concrete html tag.

The **HEADElement** comprehends the **TITLE** and **LINK** tags, while the **BODYElement** comprehend all the html tags used for creating web pages (**P**, **TABLE**, **FORM, DIV, A** etc..).

In order to allow the nesting of tags, the **HTMLElement** class has a reference to a set of children **HTMLElement.**
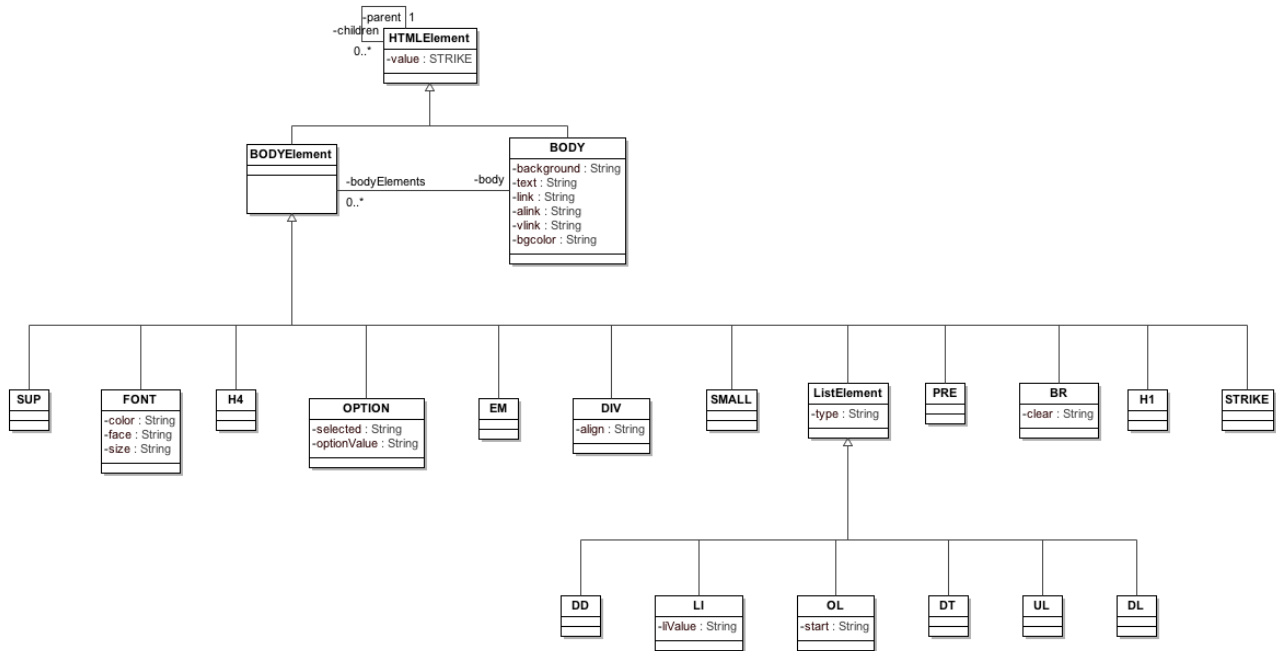
Figure 84: HTML metamodel, a fragment of the BODY element

E.3 Model to Model Transformation

The IFML model is mapped to a **WebSite** element.

Every first level ViewContainer is mapped to a **Page** element, in particular the one marked as "home" will be named "index".

Each sub-ViewContainer will be mapped to a **DIV** element.

Each NavigationFlow not associated to a SystemEvent is mapped to a **A** element. If an Action is present, its name will be appended at the end of the link.

Forms are mapped into **FORM** element and their fields are mapped to corresponding **INPUT** elements.

Details are mapped into a **UL – LI** elements, in which each list item is a attribute of the data binding considered.

Lists are mapped into **TABLE**, in which the first row is composed by the field of the corresponding data binding. If an OnSelectEvent is associated to the component, then a last column is added which contains a **A** element.

If one or more ViewContainer marked as "landmark" exist, a **DIV** element containing all the **A** element linking to the landmark ViewContainers will be created in each **Page**.