Reusing functional aspects : from composition to parameterization

Alexis Muller
Laboratoire d'Informatique Fondamentale de Lille
UMR CNRS 8022
Université des Sciences et Technologies de Lille
F-59655 Villeneuve d'Ascq cedex, France
Alexis.Muller@lifl.fr

ABSTRACT

We are considering in this paper the difficult task of the design of information systems requiring to take into account a great number of entities and concerns functional or not. Several approaches have been proposed in order to structure and thus simplify the design of such systems.

Considering the concepts of views and components, we proposed a model of design allowing the reuse of functional aspects. This paper summarize it and give some criticisms about it and other composition based approaches. Other approaches, based on a model of parameter setting, are presented. Finally, this paper concludes with the advantages that parameterized based approaches can brings to the reuse of functional aspects.

Keywords

functional aspects, composition, parameterization

1. INTRODUCTION

The design of information systems remains at the present time a difficult task which requires that great number of entities and concerns be taken into account whether they are functional or not. Several approaches have been proposed in order to structure and thus simplify the design of such systems.

Quote first the objects oriented approaches and the components approach[14, 10], which allow a system to be structured relative to its entities. If these approaches allowed a first and significant simplification for the design of information systems, they did not entirely solve the problem. Indeed, the complexity of these systems is such that each entity must take into account a large number of concerns spread over a great number of entities.

Consequently, approaches supporting a decomposition of systems according to their functional dimensions were thus proposed at the programming level with AOP (Aspect-Oriented Programming) [9] but also at the design level with the SOD (Subject-Oriented Design) [5] or with views approaches [6].

In this paper we are interested here in a design for the reuse of components for information systems adaptable in their functional or "business" dimensions (aspect).

The problem of the reuse of these functional aspects arises now. Indeed, this reuse must make it possible to improve the productivity and reliability in the field of information systems design. Various approaches propose the reuse of functional aspects in various forms, like the design of reusable frameworks[7] or in the form of UML templates [4].

Considering the concepts of views and components, we also proposed a model of design allowing the reuse of functional aspects [11]. This model that is summarized in section 2 allows the design of what we called "view components". By connecting these components we are able to create new views for a system.

Finally, in the last section we will give some criticisms of our model and other composition based approaches. Then we will present some techniques, used in other approaches, based on a model of parameter setting. And we will conclude with the advantages that parameterized based approaches can brings to the reuse of functional aspects.

2. THE VIEW COMPONENT MODEL

The idea of the software components [17] is strongly inspired by what exists in the other industrial fields (mechanical, electronic...). The goal is to be able to create a software as one assembles a television or a car, by assembling existing components and limiting the specific developments to the minimum. The object oriented programming addresses this quest of reuse and makes it possible to obtain libraries for structures of rich data (lists, piles...) or for "standard" functionalities (like the services common objects of CORBA[13]) or in the field of the Human-machine interfaces. At higher level of granularity, the design by software components must allow a more complex reuse of structures by making them available as soon as possible in the lifecycle. This type of components, sufficiently complex to contain some "know to make", is called a business component. These are the ones we consider in our approach.

We propose to consider components approaches and views approaches in the same model : the first approach bringing the principle of reuse and configuration, and the second one, bringing a method of coherent structuring, richer than the traditional object oriented approach. In our model views are full right components. Giving a statute of components to views makes it possible to preserve the structuring in functional aspects until the exploitation. This presents multiples benefits : a perfect traceability of functions, a better ability to the reuse these components. By comparison, the approaches previously quoted are based upon a fusion process of the various functions at programming level and do not offer such benefits.

To illustrate the benefits of views and promoting them as components, let us consider a system for managing a university library. Such system must offer functionalities to seek a document, to manage the resources of the library (addition, transfer, suppression) as well as the borrowing of documents. This system is represented by Figure 1 with the traditional object oriented approach. We can observe that functions of the system are all "intertwined" into the classes.

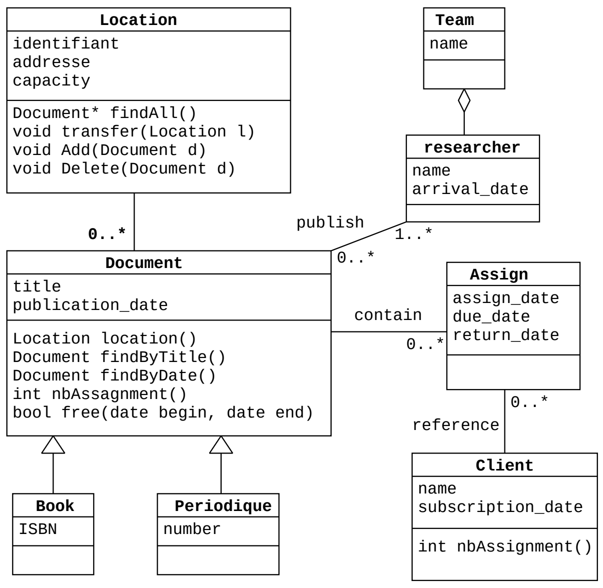

Figure 1: Object oriented approach

As illustrated in Figure 2 views approaches adds functional decomposition to structural decomposition by classes. The various functions, stock management (at the top), document search (on the left) and borrowing management (at the bottom) are separated from the base (in the middle). Each function describes only the elements which it adds. The common elements are represented in the base. The added elements can be attributes (*capacity* in the management function), operations (*transfer*), or new classes (*Assign* in the borrowing function). The matching of entities between functions is determined here by using the same name.

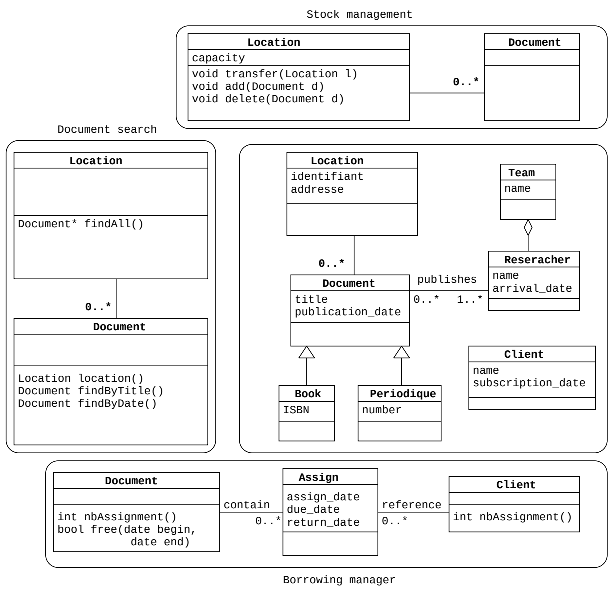

Figure 2: Views approach

These functional views are related to a base. To make these views generic, we propose to disconnect them from any base and make them components by identifying the necessary elements (cf Figure 3). In our abstract model, each view (here *SotckManager*, *Search* and *Assignment*) correspond to a `Component` (view component) made up of `ViewClass`. The necessary attributes and associations are specified by `ViewAttribute` and `ViewAssociation`. Each component then adds the elements it bring (class, attribute, association...). This approach gives the ability to specify a required diagram for a view component thus disconnecting it from any base. It is this mechanism which makes it generic. This way, the component could be applied to any base presenting this diagram. The views elements of the *Search* component, indicate that it can be applied to any base diagram containing a class (playing the role of *Location*) owning at least two attributes to materialize the `ViewAttribute` *name* and *address*, and another class (playing the role of *Resource*) with attributes materializing *key* and *date*. These two classes must be linked by an association to materialize the `ViewAssociation`. The formal definition of views elements (constituent parts of a view component) as well as the constraints they are subjected to are detailed in the section 3.

Figure 4 illustrates the reuse of views components upon an another base. The system we want to obtain is a car hire system. The configuration of components for this base is done by a mechanism of connection. The components are connected to the base and each `ViewClass` is connected to the corresponding class (this connection is represented here by the dotted arrows). The `ViewAttribute` and the `ViewAssociation` are also connected to the base elements to which they correspond. By using this mechanism, we bind the views elements to the "real" base elements. A certain number of constraints must be respected to ensure the coherence of these connections (see section 3). In our approach, the matches between base elements and view com-

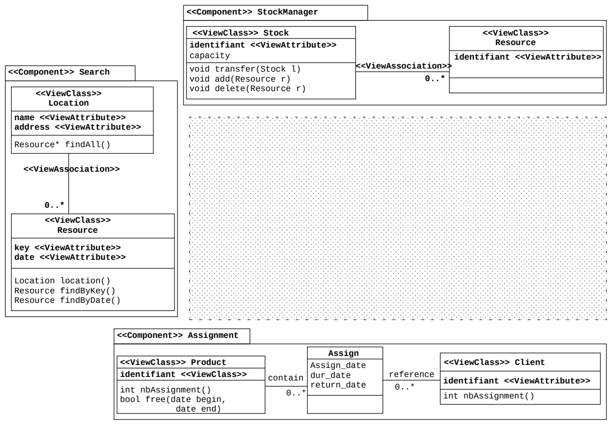

Figure 3: Dconnexion of the functions

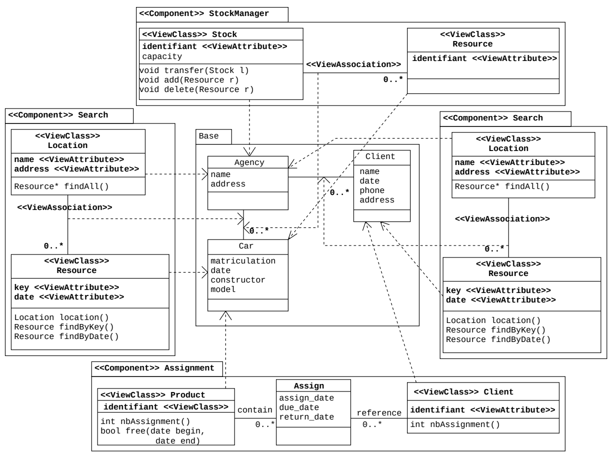

Figure 4: Reconnexion at another base

ponents are not done by naming anymore, but explicitly by the establishing connections between them. A mechanism of constraints checking makes it possible to determine if the system resulting from this assembly is coherent.

This example illustrates the genericity of views components applied to two different bases. The components identified by designing of the university library system are reused the design of a car hire system. It also illustrates the capacity of using the same component several times in the same base. This capacity is shown by the *Search* component used between *Agency* and *Car* to find the agency a car, and between *Agency* and *Client* to know in which agency a customer is recorded.

3. VIEWS COMPONENTS METAMODEL

We propose a meta-model of view components, formulated as an extension of UML 1.4 meta-model. Figure 5 shows the concepts of this model. The elements displayed on the grey background correspond to UML meta-model elements. This meta-model introduces the concept of connection between an element and a base, thanks to the `viewOf` and `root` associations. The semantics of these two types of relation are different. A `viewOf` association indicates that the source element (of type `Component` or `ViewClass`) is an extension of the target element (respectively of type `Package` or `Class`). On the other hand, a `root` association means that the source element (of type `ViewAttribute` or `ViewAssociation`) is an imported part of the target element (respectively of type `Attribute` or `Association`).

In this meta-model, a component is a collection of `ViewClass` and `Class` which can be applied to a `Class` collection (`Package`). Thanks to the connection mechanism, a component can be modeled independently of the `Package` to which it will be applied. This meta-model is provided with two types of constraints expressed in OCL :

- Design constraints to check coherence at the modeling phase.
- Connection constraints to check that the system obtained by the assembly of packages is coherent.

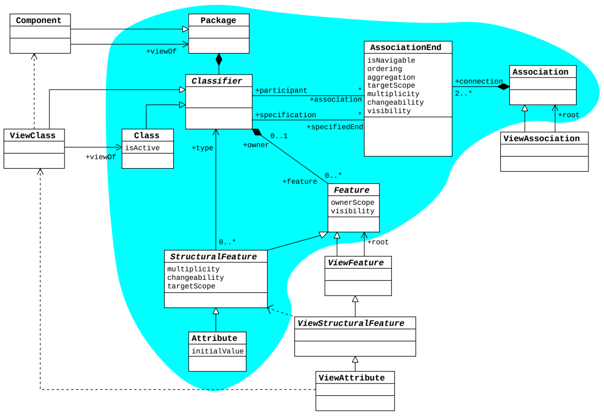

Figure 5: Le mtamodle

3.1 Description of Meta-model Elements

A **Component** is a specialization of the `Package` UML meta-class. Its role is to contain all the `ViewClass`, `Class` as any other UML element necessary to the modeling of the view component.

A **ViewClass** is a specialization of the `Classifier` UML meta-class UML. An instance of **ViewClass** can contain the same elements as another `Classifier` as well as `View-Attribute`. The `viewOf` association (between a `ViewClass` and a `Class`) indicates in which base `Class` the root `Attribute` of the `ViewAttribute` of the `ViewClass` must be in.

A **ViewFeature** is an abstract element constituent of `Classifier` like `StructuralFeature` or `BehavioralFeature`. It indicates, by the `root` association, a base `Feature`. It is **ViewFeature** elements that are used to specify the required information. The `ViewStructuralFeature` element specializes `ViewFeature` by imposing a `StructuralFeature` as root.

A **ViewAttribute** is a `ViewStructuralFeature` which must have its root association linked to an `Attribute`. A `View-`

`Attribute` must belong to a `ViewClass`. It is the only element of `ViewFeature` type which is not abstract, therefore the only one being able to appear in a model.

A **ViewAssociation** is a specialization of the `Association` UML meta-class. An instance of this one makes it possible to specify and import with connection an `Association` of the `Package` to which is applied (by the `viewOf` bond) the `Component`.

The use of these concepts, is illustrated by Figure 4. Only the concept of **ViewFeature** does not appear in this example since it is abstract.

3.2 Constraints

We present here the constraints defined for our meta-model. They are classified in two categories : design constraints to check properties on the structure of the view components, and connection constraints to check that the assembly is correct and corresponds to a coherent model.

Design constraints

Design constraints guarantee that view elements are indeed contained by an element intended to contain them. I.e. that a *ViewClass* does belong well to one *Component* (constraint 1), and similarly for the *ViewAssociation* (constraint 3). Also, *ViewAttribute* does belong well to a *ViewClass* (constraint 2).

```
[1] A ViewClass must be in a Component.
context ViewClass inv :
  self.package.oclIsKindOf(Component)

[2] A ViewAttribute must be in a ViewClass.
context ViewAttribute inv :
  self.owner.oclIsKindOf(ViewClass)

[3]A ViewAssociation must be in a Component.
context ViewAssociation inv :
  self.namespace.oclIsKindOf(Component)
```

Connexion constraints

These constraints check conformity properties between type of an *Attribute* and type of a *ViewAttribute* (constraint 7), as well as their meta-types (constraints 4), or to check that the structure formed by the view elements corresponds to the one formed by the elements to which they are connected to (constraints 5, 6, 8, 9, 10).

```
[4] The root of a ViewStructuralFeature must be a
     StructuralFeature.
context ViewStructuralFeature inv :
  self.root.oclIsKindOf(StructuralFeature)

[5] For all ViewAttribute of a ViewClass, the root
     owner must be the viewOf of the ViewClass.
context ViewClass inv :
self.allFeatures->select( f |
          f.oclIsKindOf(ViewAttribute) )
             ->forAll ( f : ViewAttribute |
                    f.root.owner = self.viewOf )

[6] The root owner of a ViewAttribute must be a Class.
context ViewAttribute inv :
  self.root.owner.oclIsKindOf(Class)

[7] A ViewAttribute and his root must have the same type.
context ViewAttribute inv :
  self.root.oclIsType(self.type)

[8] For each ViewAssociation of a Component, if there is
     a ViewClass in which his viewOf participates in the
     ViewAssociation.root, then this ViewClass must
     participates in the ViewAssociation.
context ViewAssociation inv :
self.namespace.allContents->select( v |
   v.oclIsKindOf(ViewClass) )
     ->forAll( v : ViewClass |
        self.root.allConnections->collect(type)
               ->includes(v.viewOf)
        implies self.allConnections->collect( type )
               ->includes(v))

[9] Every ViewClass of a Component must have the same
     viewOf Package as the viewOf Component.
context Component inv :
self.allReferencedElements->select( v |
   v.oclIsKindOf(ViewClass) )
     ->forAll ( v : ViewClass |
        v.viewOf.package = self.viewOf )

[10] There mustnot be two ViewClasses of the same Class
     on a Component.
context Component inv :
self.allReferencedElements->select( v |
   v.oclIsKindOf(ViewClass) )
     ->forAll ( v1, v2 : ViewClass |
        not v1.viewOf = v2.viewOf )
```

3.3 Implementation

We implemented this meta-model in the form of a profile for the UML Objecteering tools. This profile makes it possible to check if a given UML diagram is in conformity with our abstract model, by the checking of the design and connection constraints (translated into J, the proprietary language of Objecteering). It also supports automatic generation of IDL3 specifications for developing view components in the Corba Component platform. We also implemented this model on the Fractal platform [2]. Another realization was also implemented for the EJB platform leading to the generation of an executable code, based on our view pattern [12].

We obtain a model of view components that is usable and exploitable. However, this realization raised some weaknesses. We present them in the following section and we also present the approaches which we are studying in order to improve our model.

4. COMPOSITION OR PARAMETERIZATION

Approaches allowing the decomposition of a system following its functional or technical dimensions aim to simplify the design of information systems. However to form the global system all the dimensions must be assembled. Various approaches exist to express this assembly. For our model we used a technique of connexions composition.

However, our approach suffers from several weaknesses. It requires the addition of new meta-classes to UML metamodel to allow the expression of necessary elements and their connection, as well as a complex set of constraints to check the connections coherence. Currently, it only offer the connection of a view component to a base and not to another view component. This prevent sharing entities between several view components except those identified by the base, or the functional enrichment of elements of other view components. Finally, the connection mode we propose is not easy to use. Indeed, it requires connecting elements of view components and base one by one, which results in a great number of dependencies harming the legibility of models. Thus, the connection of a component to the base is not made by only one relation, but by all these dependencies. Consequently, We are currently studying various approaches to improve our model on these various points. In the following we first review approaches that use, composition to express the systems assembly, like our model. Next we discuss approaches that express this assembly by parameterization.

4.1 Composition

The Subject-Oriented Design approach [15] proposes the design of an independent model for each concern of the system. These models are called *Subject* and take the form of a standard UML Package. A new type of relation (*CompositionRelationship*) is proposed to compose subjects and express the composition of their elements. Two strategies of integrations are proposed, *Override* and *Merge*. *Override* aims to replace an element by another whereas *Merge* allows the merging of two elements into one. Moreover, one specialization of this relation (*CompositeComposition*) makes it possible to express the composition of composite elements which are *Classifiers*, *Collaborations* and *Subjects*. A criterion of correspondence can be attached with this relation to indicate whether elements of the same name constituting the composites represents the same entity or not.

This approach offers the expression of various dimensions of the system using standard UML elements and proposes an elegant composition relation of these various dimensions. Nevertheless, the ease of use of this approach is based on the fact that the composed subjects come from the analysis of the same system. That is the reason why, most of the elements denoting same concepts have the same name. The compositions relations are thus simplified and need to be "detailed" only when this property is not respected.

The Catalysis approach [7] proposes to decompose the design of systems in horizontal and vertical slices. Vertical slices correspond to a functional decomposition of the system from the points of view of various categories of users. Horizontal slices give a decomposition according to the technical concerns of the system. In this approach, packages are also used to represent the various slices of the system. Integration of these slices is based on *join* relations between packages. This *join* relation corresponds mainly to the *import* relation of UML, but specifies that elements having the same name must be merged by default.

Within our context, which is the reuse of functional aspects, various dimensions are not conceived by analysis of a particular system, but by analysis of a function designed to be reused in different systems. Since entities of the same function in various fields being named differently, we cannot use elements nomenclature to simplify our connection model. Moreover, in order to design and validate functional dimensions independently of any system, it is necessary to clearly identify necessary elements and supplyed elements. The first ones corresponding to elements which must play a part in the function but which are not owned by it; the second ones being specific elements to this function. For these reasons, these techniques are not adapted to the definition and the use of reusable functional aspects.

4.2 Parameterization

The Catalysis approach proposes other constructions in order to design reusable packages : model frameworks. Those are represented using template packages which are abstract packages containing some elements that must to be replaced to be concretized and used. In the Catalysis approach, template elements which must be substituted are identified using the symbols '<' and '>'.

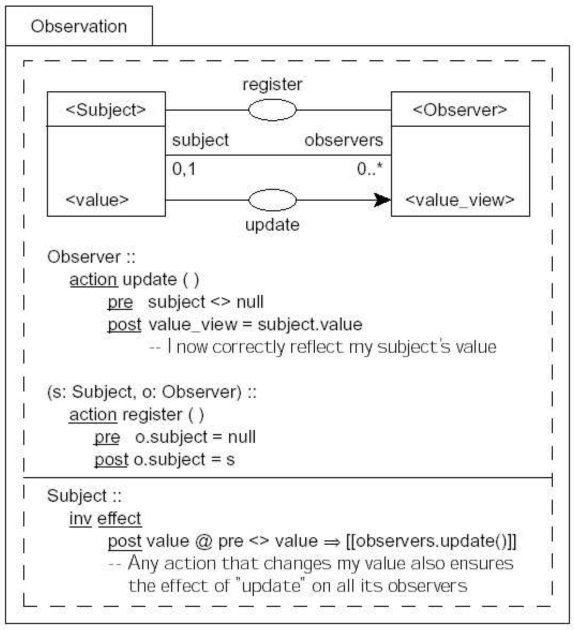
Observation
register
<Subject>
<Observer>
subject
observers
0,1
0..*
<value>
<value_view>
update
Observer ::
action update ( )
pre subject <> null
post value_view = subject.value
-- I now correctly reflect my subject's value
(s: Subject, o: Observer) ::
action register ( )
pre o.subject = null
post o.subject = s
Subject ::
inv effect
post value @ pre <> value ⇒ [[observers.update()]]
-- Any action that changes my value also ensures
the effect of "update" on all its observers

Figure 6: Template package of the owner to observe according to the Catalysis notation

Figure 6 shows this notation to represent the observer pattern [8]. The required elements to realize this template package are *Subject*, *Observer*, *value* and *value_view*.

The Theme approach [16] proposes an analysis method of the system (Theme/Doc) allowing identification of relations between various functionalities, and a notation (Theme/UML) allowing modeling of these various functionalities in the form of parametrized packages called *Theme*. A relation (named *bind*) is used to express the parameterized composition of two Themes. The checking of assembly is done by adding informations of compositions to the analysis graph (Theme/Doc). Figure 7 shows the Theme/UML notation to express a "Theme" for the observe pattern. Figure 8 illustrates the use of the *bind* relation to apply the previous theme to a package.

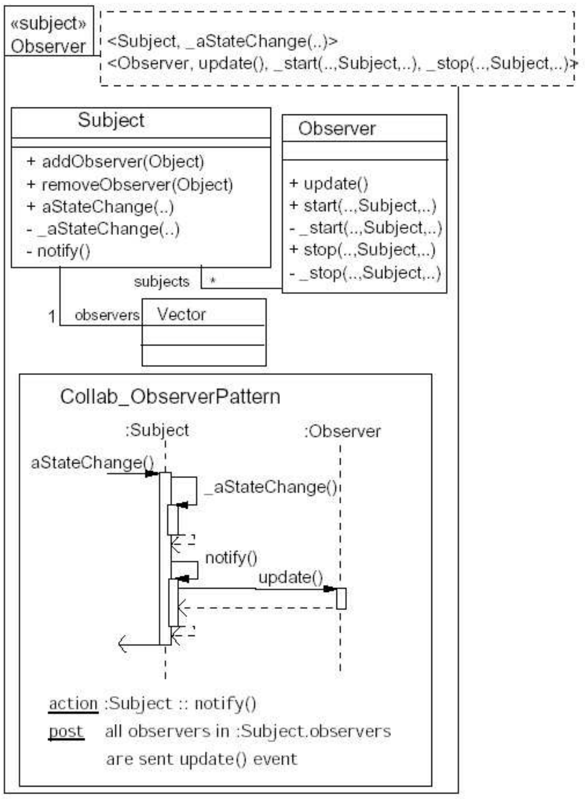

Figure 7: Template package of the owner to observe according to the notation Theme/UML

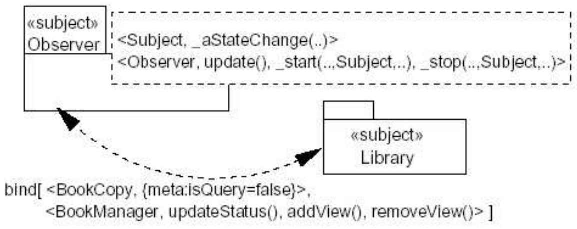
«subject» Observer
<Subject, _aStateChange(..)>
<Observer, update(), _start(..,Subject,..), _stop(..,Subject,..)>
«subject» Library
bind[<BookCopy, {meta:isQuery=false}>,
<BookManager, updateStatus(), addView(), removeView()>]

Figure 8: The *bind* relationship of Theme/UML

Lastly, UML notation [1] also offer the possibility of defining template packages. Those take the form of "standard" package that list elements having to be provided to construct the package in a dotted rectangle located at the top right hand corner. The concretization of a template is expressed using a *bind* relation which indicates how the resulting package was actually built from the template [3].

5. CONCLUSION

In this paper we presented our model of views components, which allow us to specify at a model level of the generic units of design for the functional enrichment of information systems. We described it by formulating an extension of UML meta-model and a set of constraints. We also experimented this approach on various platforms and we entirely automated the generation of technical code for the EJB platform. Finally we presented some weaknesses of our model and studied other approaches proposing the use of parameterization.

These approaches based on parameterized compositions appear adapted to our views components. The parameterization could express the adaptation of a generic view component on a particular system, and the composition could express the assembly of the various components to describe the global system. We are thus studying a new expression mode of our view components based upon the UML templates in order to bring us closer to standard notation. Lastly, the use of parameterized composition approach makes it possible to reduce the connections checking problem, requiring a great number of constraints, in a parameters type checking problem.

6. REFERENCES

[1] U.M.L. Home Page, http://www.omg.org/technology/uml, 2001.

[2] O. Barais, A. Muller, and N. Pessemier. Extension de Fractal pour le Support des Vues au sein d'une Architecture Logicielle. In *Objets Composants et Modèles dans l'ingénierie des SI (OCM-SI 04)*, Biarritz, France, 2004. http://inforsid2004.univ-pau.fr/AtelierOCMv1.htm.

[3] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. Formulation of UML 2 Template Binding in OCL. In *UML'2004:7th International Conference on UML Modeling Languages and Applications*, October 2004.

[4] S. Clarke. Extending standard uml with model composition semantics. In *Science of Computer Programming, Elsevier Science*, volume 44, 2002.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Objecteed-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Denver, November 1999.

[6] L. Debrauwer. *Des vues aux contextes pour la structuration fonctionnelle de bases de donnes objets en CROME.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, dcembre 1998.

[7] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Westley Professional Computing, USA, 1995.

[9] G. Kiczales and al. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag LNCS 1241.

[10] L. Michiel. *Enterprise Java Beans Specification v2.1 - Proposed Final Draft http://java.sun.com/products/ejb/docs.html.* Sun Microsystems, August 2002.

[11] A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. Réutilisation d'aspects fonctionnels : des vues aux composants. In *Langages et Modèles à Objets (lmo'03)*, pages 241–255, Vannes, France, January 2003. Hermès Sciences.

[12] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. A Framework for Supporting Views in Component Oriented Information Systems. In *OOIS*, volume 2817 of *Lecture Notes in Computer Science*, pages 164–178. Springer, Sept. 2003.

[13] OMG. *CORBA 3 Specification*. Object Management Group, July 2002. http://www.omg.org/cgi-bin/doc?formal/02-06-33.

[14] OMG. *CORBA Components*. Object Management Group, June 2002. http://www.omg.org/cgi-bin/doc?formal/02-06-65.

[15] Siobhán Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, Dublin City University, Jan. 2001.

[16] Siobhán Clarke and Robert J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In *23rd International Conference on Software Engineering (ICSE)*, May 2001.

[17] G. T. Heineman and W. T. Councill. *Component-based software engineering : Putting the pieces together*. AddisonWesley, 2001.