MULTIVIEWS COMPONENTS FOR INFORMATION SYSTEM DEVELOPMENT

Bouchra El Asri*, Mahmoud Nassar*,**,***, Bernard Coulette** , Abdelaziz Kriouile*
*Laboratoire de Génie Informatique
ENSIAS, BP 713 Agdal, Rabat, Maroc

** Laboratoire GRIMM – IRIT
Université de Toulouse le Mirail, Département de Mathématiques- Informatique
5, allées A. Machado 31058 Toulouse cédex, France

*** ENSAM, B.P.4024 Béni M'Hamed – Meknès, Maroc

Keywords: Information System Modelling, UML, View, Viewpoint, VUML, Multiviews component

Abstract: *Component based software* intends to meet the need of reusability and productivity. *View concept* allows software flexibility and maintainability. This work addresses the integration of these two concepts. Our team has developed a view-centred approach based on an extension of UML called VUML (View based Unified Modelling Language). VUML provides the notion of multiviews class that can be used to store and deliver information according to users viewpoints. Recently, we have integrated into VUML multiviews component as a unit of software which can be accessed through different viewpoints. A multiviews component has multiviews interfaces that consist of a base interface (shared interface) and a set of view interfaces, corresponding to different viewpoints. VUML allows dynamic changing of viewpoint and offers mechanisms to manage consistency among dependent views. In this paper, we focus on the static architecture of the VUML component model. We illustrate our study with a distant learning system case study.

1 INTRODUCTION

With the popularity of the Internet and web-based access to information, software development must face up to heterogeneous environments and changing client's needs. In this context, reusability and interoperability are key criteria. Component based software construction intends to meet these needs. The basic idea is to allow developers to reuse simple units of software called *components* to build up more complex applications.

Moreover, to be efficient, software access must be given to any user with respect to his culture, rights, education, etc. A lot of web-based information systems are now available in such fields as e-learning, tourism, environment, health, transport, etc. Some of them try to adapt themselves to users' profile and behaviour at execution time, especially to give them a rapid access to information. But so far, development and maintenance of those systems are not guided by users' profile (viewpoints) and thus such systems are very difficult to adapt and maintain.

We need methodologies that explicitly support the concept of *viewpoint* in a *component* based software development.

We already investigated the notions of *view* and *viewpoint* (Coulette et al., 1996) and elaborated a view-based analysis and design method called *VBOOM*, but this method is not compatible with OMG standards and thus practically unusable.

UML (OMG, 2001) provides development views (use case, logical, deployment...) to structure a system at several levels of abstraction. However, UML views are not sufficient to model system architecture according to users' viewpoints. We need a fine grained mechanism to support both functional and non functional views.

To meet these requirements, we have defined a UML profile called VUML (View based Unified Modelling Language) (Nassar et al., 2003) implemented into the Objecteering case tool (Objecteering, 2004). VUML provides new

El Asri B., Nassar M., Coulette B. and Kriouile A. (2005).
MULTIVIEWS COMPONENTS FOR INFORMATION SYSTEM DEVELOPMENT.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 217-225
DOI: 10.5220/0002538702170225
Copyright © SciTePress

modelling elements derived from the UML meta-model through extension mechanisms (mainly stereotypes and constraints in OCL): multiviews class, base, view, viewpoint, extension relationship, dependency relationship, etc. In this paper we present the integration into VUML of the notion of *multiviews component* as a unit of software which can be accessed through different viewpoints. A first introduction of this concept was done in (Nassar et al., 2004). Due to size constraints, we focus here on two main characteristics of such components: *static structure* and *composition*. We illustrate these features with a distant learning system case study.

The rest of this paper is structured as follows: section 2 gives a brief overview of component based software; section 3 introduces the UML distant learning modelling, especially the use cases and components diagrams models. Section 4 describes the concept of multiviews component and associated mechanisms. In the section 5, we present some related works, and in the last section we give a conclusion and perspectives to our work.

2 COMPONENT: DEFINITIONS, MODEL AND COMPOSITION

The field of component-based software engineering (CBSE) is in a phase of rapid growth and change. Standards industry components are shipped as “plug-ins” into existing run-time architecture. Components are increasingly used to create complex and distributed systems and applications.

Software component merges two distinct perspectives: component as an implementation, and component as an architectural abstraction. Viewed as implementations, components can be deployed and assembled into larger systems. Viewed as architectural abstractions, components express design rules that impose a standard coordination model on all components. These design rules take the form of a component model, or a set of standards and conventions to which components must conform.

In this section, we first give some component concepts definitions. Then, we present the UML component model and composition elements.

2.1 Definitions

C. Szyperski defines a component *as a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition* (Szyperski, 2002). This definition is closed to that of B. Meyer who considers a component as *an oriented client software unit* (Meyer, 2000). In general, a component is a unit of program which comprises at least two parts: a specification part of its interfaces and behaviours, and an implementation part that carries out its services. An interface is *a collection of operations that are used to specify a service of a component* (Kruchten, 1999).

2.2 UML component Model

A component model specifies the standards and conventions imposed on developers of components. It specifies the design rules that must be satisfied by components. The UML 2.0 language (OMG, 2003) allows the definition of component specification and architecture. It defines a component as *a modular part of system that encapsulates its contents and whose manifestation is replaceable within its environment.* A component defines its behaviours in terms of provided and required ports. A port is a point for conducting interaction between the component internals and his environment. Ports are typed by interfaces. An interface includes a set of services and constraints. Ports can be provided or required. Connection between provided and required ports is made by connectors. Two types of connectors exist: the *delegation connector* and the *assembly* one. A delegation connector is a connector that links the external contract of a component to its internal realisation. The assembly connector is a connector between two components. One of the components provides the service that the other one requires.

2.3 Composition of components

Composition is the term used in component-based development to refer to systems assembling. Components are composed so that they may interact.

Contract is the concept which shifts the focus on components interactions, and the mutual obligations of participants in these interactions. There are two senses of contract that are necessary to CBSE: *component contracts* and *interaction contracts*. Component contracts describe patterns of interaction that are rooted on a component. Interaction contracts describe abstract patterns of interaction among roles that are filled by components. Systems are assembled from components through a process of filling roles with components. (Beugnard et al., 1999) categorise contracts in four levels: Syntactic;

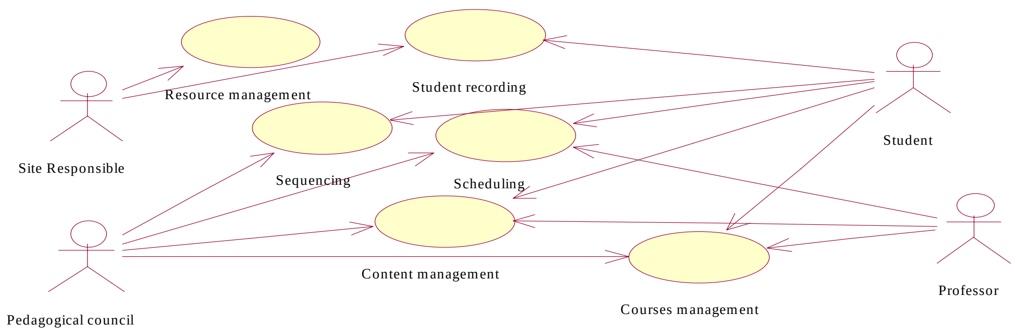

Figure 1: Use Case model of the Distance Learning System (extract)

Behavioural; Synchronisation contracts and Quality of Service.

3 CASE STUDY: THE DLS

All along this paper, we illustrate our approach through a Distance Learning System (called DLS in the following). The main goal of that system is to allow distant students to apply for courses, access to related documentation (slides, web pages, text, etc.), make exercises, communicate with teachers, and take exams. The DLS provides for students runtime sequences. It can be distributed over several sites, and is managed by a responsible whose job consists in : Student recording; Resource management (creating, updating and removing resources, their availability and their interactions) and Content management (creating, updating, organising and publishing information resources).

Each site has a pedagogical council whose responsibilities are: Course management (creating, updating and deleting units of learning); Scheduling (allocation and deallocation of resources against time slots) and Sequencing (organising sequences for learning units).

Professors propose and update their own courses; plan learning experiences and units of work for delivery on or off line and record student assessments. They are in charge of writing exam subjects. Tutors are in charge of a group of students. They mark student exams and answer students' questions and report progress to inform students.

As we can easily understand, such DLS system development requires a number of people working simultaneously. The DLS should be divided into small modules to minimize risk. Component technology allows developing complex applications by mixing and matching specialized modules. Each component is developed independently from the others so that any developer may focus on a single component.

So, we have divided the DLS into components that are described below. First, we consider a classical UML modelling of this system. We present the use case diagram, a general components diagram and give details of some components that are composed eventually to build a Training course scheduling application.

3.1 Use case diagram of DLS

Figure 1 below shows a simplified Use Case diagram of DLS in UML. Actors of the DLS are *students*, *teachers*, *site responsible* and *pedagogical council*. Only a subset of identified use cases is considered in figure 1: student recording, courses management, scheduling, resource management, content management, and sequencing.

3.2 UML component diagrams

Distant learning applications are built from several components. The most important of them are shown below in figure 2. The *Student record* component provides services for applying available course modules and provides information about prices, contents and sequencing. The *Content management* component provides services for publishing, retrieval, description, and organisations of information resources. The *Sequencing* component provides services about sequenced learning objects. The *Course management* component provides services that allow access and management of learning units. The *Resource management* component provides services to create, update and remove available resources and their kind of use. The *Group management* component provides services to manage information about groups. A group aims to get students together for learning, exams and other activities.

Every component of the figure 2 may be a composite one. For instance, Scheduling, Resource management and Course management are composite components. Figure 3 shows the *Resource management* component. Resources comprise three major parts: immobile resource such as classrooms, etc; equipments such as projectors, computers, video players, and human resources such as teachers, group of students, etc. Each part provides services of creating, updating and revoking the resource and special services. For example, Human resource provides information about profiles, units to teach (for teacher) or to pass (for group of students). Equipment provides information about the way of using, etc.

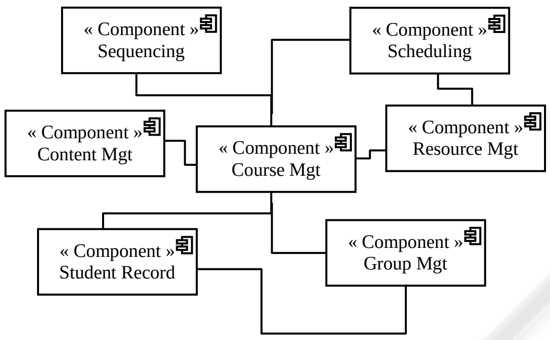

Figure 2: Component model of the DLS

While a class interface is a single collection of provided operations, a component interface is a subset of operations smoothly gathered for a specific service and a further connection. As an example, the *ResourceMgt's* services cited above have been gathered into provided interfaces called *ImInf* (providing information about immobile resources), *EquitInf* (for equipments), *ProfInf* (for professors) *GroupInf* (for groups), *ImMgt* (for creating, updating and removing immobile resources) and so on for the resources management interfaces. The *RsrceConst (resource constraints)* described in figure 6 provides facilities to mark fixed resources closures or unavailability dates determined by fixed commitments or public holidays, and to mark other schedule constraints such as equipment breakdown, immobile fitting, teacher absence, etc.

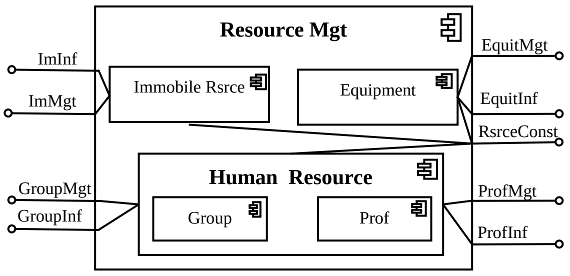

Figure 3: The Resource Management Component

For the *Course management* component (cf. figure 4 below), we give only the required and provided interfaces needed to illustrate the assembling functionality.

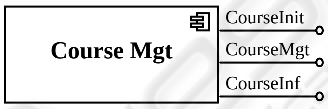

Figure 4: The Course Management Component

Figure 5 shows the *Scheduling component* made of two sub-components. To timetable courses, we need to manage different types of resources together: immobile resources where learning units will take place, equipments necessary for a given unit, professor who gives the unit, group of students who will attend the unit, the learning unit and the slot of time which brings the involved resources together. We name *Schedule* the sub-component which allows to determine whether a particular resource is available or not. It requires the *ResourceConst (RsrceConst)* interface and provides the *ResourceAvailability* interface (cf. figure 5). That interface enables users to reserve resources for particular dates and is responsible for setting resource priorities. The *Allocation* sub-component finds out for each required resource what is its availability, and then selects dates that suit for all the resources involved. The *AllocationMgt* interface (cf. figure 6) sets information about all resources required and the suitable period for a unit learning scheduling.

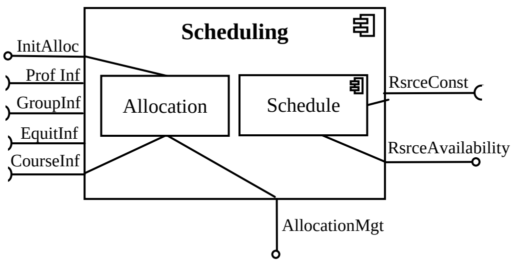

Figure 5: The Scheduling component

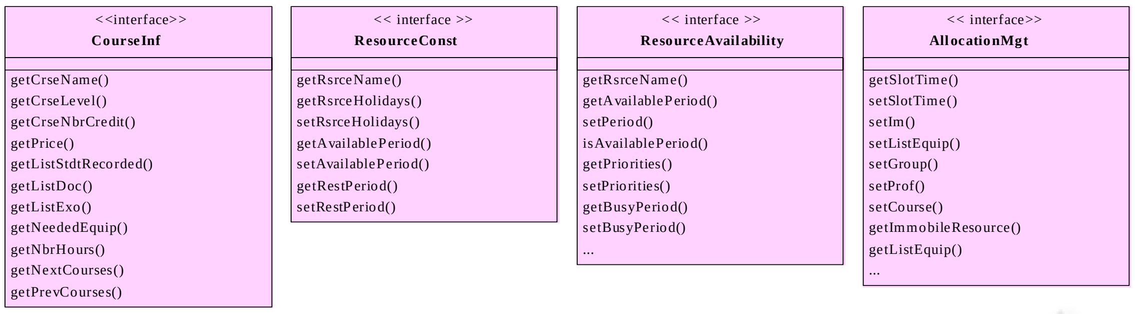

Figure 6: Description of major interfaces in UML

3.3 Assembling components for a Scheduling training course application

As mentioned in the previous section, the component assembly is based on contracts (basic, behavioural, synchronization and quality of service). In this paper we discuss only basic contract.

The basic contract checks syntactic conformance between required and provided interfaces of the components to compose. So, for each assembly connector, originated in a required port and delivered to a provider port, we need to check the interfaces compatibility. This leads to check if provided and required interfaces define compatible services.

ResourceMgt, *CourseMgt* and *Scheduling* components may be connected to compose a *Scheduling Training course* (see the resulting component model figure 7). The connection between components is assured by provided and required interfaces conformance.

3.4 Discussion

In UML 2.0, according to the component diagram of figure 7, all the actors of the system have potentially the same access rights to information and services encapsulated in components. As an example, students can access to all the allocation related to courses and to all the resource constraints. This is not acceptable because part of the allocation should be hidden to students, and some reservations should be accessible by professors only. In UML, access rights control cannot be captured in component diagrams but only in dynamic diagrams, and hence must be programmed in component implementation. Any use of a component must be carried out under the constraints of a control view. For example, the schedule component must define student control view to restrict access to available services. Component-based development has several advantages over traditional approaches to software development: reusability and productivity; but, it does not provide any mechanism for defining control view and access right at design-time.

Our goal is to describe such information access rights at a high abstraction level. Indeed, we believe that one can gain a lot from taking into account such information as early as possible, that is during the analysis phase. To achieve this goal, we decided to introduce a new type of component, the *multiviews component* which allows defining views associated to actor's profiles. The challenge is then to put information (attributes, methods, constraints) into the right view interface of a given component.

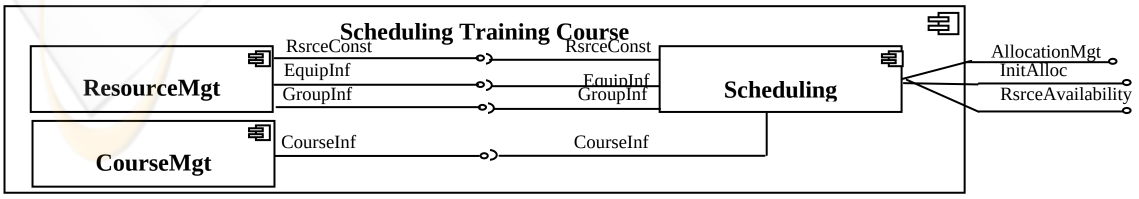

Figure 7: Component model of the Scheduling Training Course

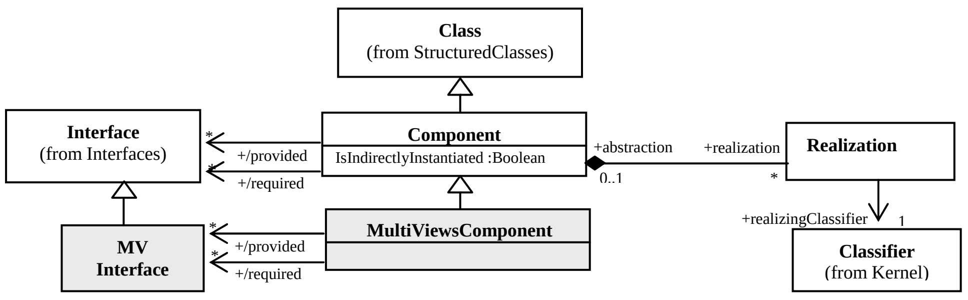

Figure 8: Static structure of a multiviews component

4 MULTIVIEWS COMPONENT

In this section, we first give definitions related to VUML component concepts. Then we describe some details about the structure of a multiviews component and related mechanisms.

4.1 Multiviews Component

An **actor** is a logical or physical person who interacts with the system. A **multiviews component** (MV-C) is a unit of abstraction and encapsulation (Nassar et al., 2004). It is an extension of the UML component concept. An MV-C (cf. figure 8) provides interfaces whose access and behaviour change according to the actor view. Such interfaces are called **multiviews interfaces**. These new concepts have been added into the VUML meta-model whose extract is show on figure 8.

4.2 Multiviews Interface

A new type of interface is defined for MV-C, to express its run-time behaviour change, called *Multiviews Interface*. Figure 9 illustrates the static structure of a MV-Interface. Such an interface is composed of a base interface (*baseInterface* stereotype), and views (*viewInterface* stereotype) that are related to the base through a *viewExtension* relation.

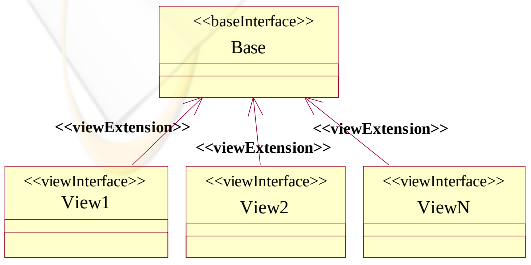

Figure 9: Static structure of a MV-Interface

The activation of a view (linkage to the current user's viewpoint) is done at execution-time. The base is a shared interface. The **viewExtension relation** is a dependency relation. A view interface depends on the base interface in the sense where attributes and methods of the base interface are implicitly shared by all views. At design-time, a MV-Component has a set of multiviews interfaces. At run-time, MV-Component behaves as a regular component with interfaces whose definitions, at a given time, are the combination of features of the base and the active viw interfaces. To complete this run-time MV-component's behaviour, we have conceived an implementation pattern for the MV-Component deployment. This pattern is inspired from *Role* and *Strategy* patterns, and implements a *setview()* method to dynamically change the active view.

4.3 The Multiviews DLS case study

Figure 10 below shows the Multiviews component model of the *Scheduling Training Course*. One can notice that each component of the UML component model (see figure 7) has become a MV-C, since it should be accessed from several viewpoints. Figure 11 gives details about the *MV-CourseInf* and *MV-ResourceConst* interfaces. In this simplified example, we only highlight views associated to the actors Professor, Student, Site responsible and Pedagogical council. Compared to the UML component diagram (see figure 6 above), services distribution into interfaces has been changed. For the *CourseInf* interface, we have defined a base interface which contains basic services for providing the *name*, the *level* and *number of credits* of a course. Other services are dispatched into view Interfaces. As an example, the method *getPrice()* has been put into the two view interfaces *VRespSiteCourseInf* and *VStudentCourseInf* associated respectively to the actors Site responsible and Student, because other actors do not need this information. The *setAvailablePeriod()* method of

MV-ResourceConst has been hidden to the Student and Professor because these latter must not update resource availability.

The MVC *Scheduling, CourseMgt* and *ResourceMgt* are composed by connecting provided and required MV-interfaces. Syntactic contracts must be checked for each viewpoint. This means to check required and provided interfaces conformance for base and view interfaces (for every viewpoint). At run-time, a view is activated thanks to the *setView* implicit interface. The component interfaces are then specified according to that view. For example, if the Pedagogical Council view is the active one, the *MV-CourseInf* interface connecting *Scheduling* and *CourseMgt* MVC comprises operations specified in the *CourseInf base* and those belonging to the *VPCCourseInf* (the view interface of CourseInf corresponding to the Pedagogical Council viewpoint.) An operation of the *base* interface - for instance *getRsrceHolidays()* - may be redefined in *view* interfaces (see the MV-interface *ResourceConst* in figure 11).

View interfaces of a multiviews interface may be dependent, so it is necessary to maintain the internal coherence of a multiviews interface. Our approach ensures that changes done into a view at execution time are reflected into dependent views. As an example, in the DLS system, one may assure that resources reserved by a professor will be transmitted to the site responsible; if a course price is changed into the Site responsible view, the price to pay must be changed into the Student view. Obviously, management of these repercussions is done at the implementation level; but such functional dependencies are very important information for system designers, so we decided to express those dependencies in VUML. We use UML notes and OCL (Object Constraint Language) to specify such constraints. The description of those dependencies is not in the scope of this paper.

5 RELATED WORKS

Researches in software modelling and development have spawned various concepts related to view and viewpoint concepts (El Asri et al., 2004). The *view* concept was first introduced by Shilling and Sweeny (Shilling et al., 1989) as a filter on global interface of a class. This concept has been then largely investigated in the field of databases (Abiteboul et al. 1991, Debrauwer 1998), Software Engineering (Finkelstein et al., 1990), Requirement Engineering (Charrel, 2002) and in Object-Oriented Development.

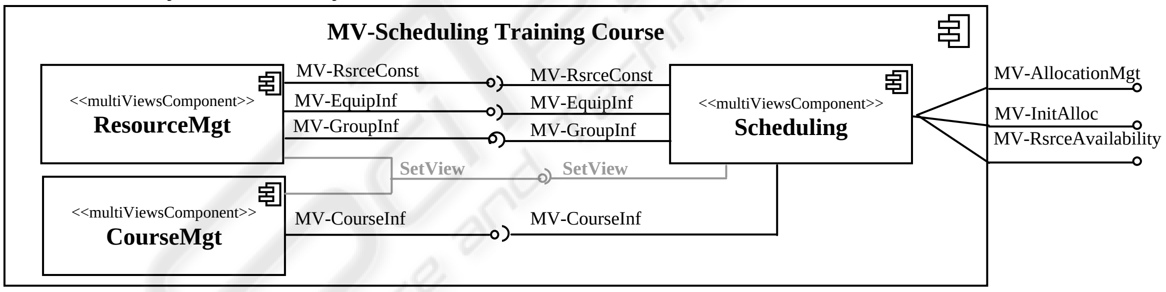

Figure 10: Multiviews component model of the Scheduling Training Course.

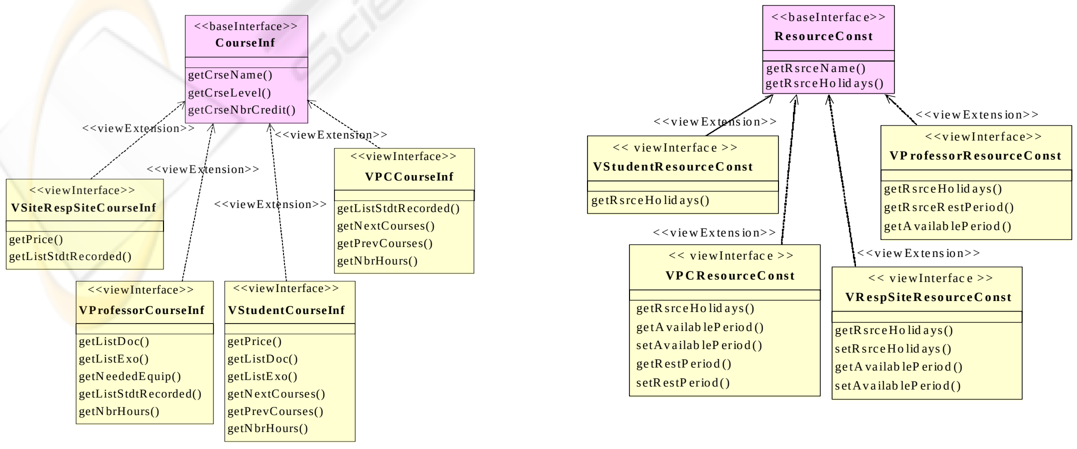

Figure 11: VUML model of the CourseInf and ResourceConst Multiviews Interfaces

Moreover, a number of concepts have been proposed to describe notions close to views such as role (Anderson et al., 1992), subject (Harrison et al., 1993) aspect (Kiczales et al., 1997) and more recently multidimentional separation of concerns (Osher et al., 2001).

Our team has been working on view-based object-oriented methodologies since 1993. Thus, we defined a view-based extension of Eiffel called VBOOL (Marcaillou et al., 1994) and a view-based analysis and design method called VBOOM (Kriouile, 1995). We are working now on VUML (Nassar et al., 2003), a UML profile that provides the concept of *multiviews class* whose goal is to store and deliver information according to user viewpoints.

For components, several models have been proposed : UML (OMG, 2003), ODP (RM/ODP, 1996), JAC AOP (Pawlak et al 2004), ACCORD (Florin et al., 2003), FRACTAL (Bruneton et al., 2004), etc.

In UML, the view notion is a way of structuring system designs according to different aspects of development: use cases, logical, components, deployment. So UML views are development views. ODP defines a set of viewpoints with associated viewpoint languages defining the concepts of each viewpoint. The RM-ODP viewpoints provide a useful abstraction for reasoning about distributed systems but it concerns the development process only whereas VUML views are actor views (covering development and execution).

The JAC AOP addresses dynamic and distributed Aspect Oriented Programming with Java Aspect Component. It allows dynamic add and remove of aspects to existing components using wrapping methods. JAC is a very interesting framework for adding non-functional concerns as persistence, integrity, load-balancing, etc. But it does not address add and remove of functional concerns. On the other hand, while JAC alters the basic component when adding new concerns, VUML approach lets basic interfaces of the MV-Component unchanged when adding functional concerns for a new viewpoint.

The fractal component model makes separation between functional and non-functional concerns. A *functional* interface is an interface that corresponds to a provided or required functionality of a component, while a *control* interface is a *server* interface that corresponds to a "non functional aspect". Internally, a Fractal component is formed out of two parts: a *controller*, and a *content*. The content of a component is composed of other components, called *sub-components*, which are under the control of the controller of the enclosing component. A component may be *shared* by several distinct enclosing components. The fractal extension (Caron et al. 2003, Barais et al. 2004), supports views by splitting component into basic ones (shared by all views) and views component. This approach is closed to ours in the sense where it considers basic and view services. But whereas the fractal system split the multiviews component into several ones (each one with its own identity), VUML considers a global component whose interfaces change depending on the system active view.

6 CONCLUSION

Undoubtedly, combining component and viewpoint concepts help decentralised development, enhance reusability, improve information accuracy and consistency, facilitate and reduce production time of software.

In this paper, we have first presented components and related concepts. Through the Distant Learning System case study, we have shown how components are enable to provide accurate services and controlled access to different clients. In the continuation of our works about VUML, we propose the concept of *multiviews component* which interacts with the environment through multiviews interfaces. Statically, a multiviews interface is composed of a *base interface* and a set of *view interfaces* extending this base. At any time of the execution, a component behaves according to the active view. That active view is automatically propagated to every component linked to the previous one. Management of views (add, suppress, lock, unlock) is done dynamically through an implicit interface called *view management*. Consistency among dependent views is managed thanks to firstly an explicit declaration of dependencies (in OCL), and secondly programming at the implementation level.

We have focused so far on the static aspect of the multiviews component and on syntactic contract for component assembling. Our objective is now to specify other types of contracts (behavioural, synchronisation and QoS) and to generate patterns for multiviews component deployment.

REFERENCES

Abiteboul S., Bonner A., 1991. Objects and Views. *Proc. of ACM SIGMOD*, pp. 238-24.

Andersen E., Reenskaug P., 1992. System Design by Composing Structures of Interacting Objects. *Proc. of the 6th ECOOP'92*, LNCS, Vol. 615. pp. 133-152, Utrecht, The Netherlands. Springer-Verlag.

Barais O., Muller A.,. Pessemier N., 2004. Extension de Fractal pour le Support des Vues au sein d'une Architecture Logicielle. *Objets, Composants et Modèles dans l'ingénierie des SI*. Biarritz, Mai 2004.

Beugnard A., Jézéquel J.M., Plouzeau N. and Watkins D., 1999. Making components contract aware. *IEEE Computer,32(7):38_45*, July 1999.

Bruneton, E. Coupaye T., Stefani J.B., 2004. The Fractal Component Model, version 2.0-3, *http://fractal.objectweb.org/specification/*, Feb. 2004.

Caron O., Carré B., Muller A., Vanwormhoudt G., 2003.. A Framework for Supporting Views in Component Oriented Information Systems. *In OOIS, LNCS, volume 2817, pages 164.178. Springer*, September 2003.

Charrel P.J., 2002. The Viewpoint Paradigm: a semiotic based Approach for the Intelligibility of a Cooperative Designing Process. *Australian Journal of Information Systems*, Vol. 10, n° 1. pp. 3-19.

Coulette B., Kriouile A., Marcaillou S., 1996. L'approche par points de vue dans le développement orienté objet de systèmes complexes. *Revue l'Objet vol. 2, n°4*, pp. 13-20.

El Asri, B., Nassar M., Coulette B., Kriouile A., 2004. Views, Subjects, Roles and Aspects: a comparison along Software lifecycle. *6th International Conference on Enterprise Information Systems (ICEIS'04)*, April (14-17) in Porto/Portugal.

Debrauwer L., 1998. Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME. *Thèse de Doctorat, LIFL, Lille.*

Finkelstein A., Kramer J., Goedicke M., 1990. Viewpoint Oriented Software Development. *Proc. of Software Engineering and Applications Conference*, pp. 337-351, Toulouse.

Florin G., Legond-Aubry F., Enselme D., 2003. Modèle abstrait d'assemblage de composants par contrats. *Rapport technique Livrable 1.4, Projet RNTL Accord*, juin 2003.

Harrison W., Ossher H., 1993. Subject-oriented programming : a critique of pure objects. *Proc. of OOPSLA'93*, Washington D.C., pp. 411-428.

Kiczales G., Lampng J., Mendhekar A., Maeda C., Lopes C. V., 1997. Aspect-Oriented Programming. *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*. Finland. Springer-Verlag LNCS 1241.

Kriouile A, 1995. VBOOM, une méthode orientée objet d'analyse et de conception par points de vue. *Thèse d'Etat*. Université Mohammed V de Rabat.

Kruchten P. 1999. Modelling Component Systems with the Unified Modelling Language. *Rational Software Corp*.

Marcaillou S., Coulette B., Kriouile A., 1994. Visibility : A new relationship for complex system modelling. *In TOOLS USA'94*. TOOLS13, Prentice Hall.

Meyer B., 2000. What to compose. Software Development, mars 2000. *Online: Software development columns : http://www.sdmagazine.com/articles/2000/0003/*

Nassar M., 2003., VUML : a Viewpoint oriented UML Extension. *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'2003)*, Doctoral symposium, Montreal, Canada.

Nassar M., Coulette B., Crégut X., Ebsersold S.., Kriouile A., 2003. Towards a View based Unified Modeling Language. *Proc. of 5th International Conference on Enterprise Information Systems (ICEIS'2003)*, Angers, France.

Nassar M., El Asri B., Coulette B. et Kriouile A., 2004. Une approche UML de composants multivues. *Workshop Objets-Composants-Modèles dans les Systèmes d'Informatio*n. Biarritz, France. 25 mai 2004.

Objecteering 2004. Objecteering software http://www.objecteering.com.

OMG 2003. UML 2.0 Superstructure Final Adopted specification, Document - ptc/03-08-02, 2003, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

OMG, 2001. Unified Modeling Language, version 1.4; http://www.omg.org/cgi-bin/doc?formal/01-09-67

Ossher H., Tarr P., 2001. Using multidimentional separation of concern to (re)shape evolving software. *Communication of the ACM October 2001/Vol. 44, No. 10 pp43-50.*

Pawlak R., Duchien, L., Seinturier L., Legond-Aubry F., Florin G., Martelli L., – JAC : An Aspect-based Distributed Dynamic Framework – *Journal Software Practice and Experience,* 2004.

RM/ODP 1996 ISO/IEC, "ISO/IEC 10746-1 Information technology - Basic reference model of Open Distributed Processing - Part 1: Overview," *ISO ITU-T X.901 - ISO/IEC DIS 10746-1*, 1996.

Shilling J., Sweeny P., 1989. Three Steps to Views, *Proc. of OOPSLA'89,* New Orleans, LA, pp. 353-361.

Szyperski C., 2002.: Component Software - Beyond Object-Oriented Programming. Addison-Wesley, 2nd edition, november 2002.