

Component based method for enterprise application design

Emmanuel Renaux
Trigone Laboratory
University of Lille, France
emmanuel.renaux@univ-lille1.fr

Eric Lefebvre
École de technologie supérieure
Montréal, Canada
lefebvre@ele.etsmtl.ca

Abstract

Component support has actually been enhanced with version 2.0 of the Unified Modeling Language and component appears as the best reusable unit of software, whereas more and more pre-built components are made available. However, reuse of components to build system remains a difficult task. Components are mostly identified in the late phases of the system development cycle without considering the end-users' requirements specified in the early phases. The effort required to develop or re-use components which satisfy the requirements is still significant, so that a lot of developers generally prefer to develop a system from scratch, while being largely influenced by technological concerns.

This article presents a Model-Driven Engineering method based on the early identification of business components. Setting up the component identification during use case modeling, transforming requirements into logical components, enriches analysis using UML diagrams. Business archetype concept and component paradigm are jointly used to structure the component-based process. This innovative method demonstrates that 1) components must be identified right from the use case model at the requirement stage of the development cycle and 2) a mechanism must be set up to ensure their traceability along the next stages. Building Platform Independent Models from the use case model, using a set of four Business Archetypes, maintains the consistency between components and requirements, ensures their traceability and facilitates their transformation into Platform-Specific Models, and then into code. In the whole, the proposed method should bring another significant progress to Model-Driven Engineering.

Keywords: Component, traceability, requirements, information systems, engineering process.

1. Introduction

The component idea is today omnipresent in software engineering mainly for development concerns. Support of component during analysis has been enhanced with version 2 of the Unified Modeling Language (UML) [25] but its use is not guided by a clear method. Component appears as the best reusable unit of software, whereas more and more pre-built binary components are made available. Component-Based Development (CBD) [6] has lately been extended to integrate the concepts of service and of Service-Oriented Architecture and then to allow a wide exchange of pre-built off-the-shelf components. In the mean time, Model-Driven Engineering (MDE), following the Model-Driven Architecture (MDA) specified by the Object Management Group (OMG), makes modeling an activity of software production instead of documentation. MDE clearly separates the business models independent of any technology from the system models dependent of them. Both CBD and MDE follow a convergent way, since MDE needs CBD to facilitate reusability during Platform Independent Model (PIM) building, whereas CBD needs MDE to facilitate the interoperability between technological platforms. However CBD and MDE are today not well integrated, so that reuse of components to build a system remains a difficult task. Moreover, decomposing a system into really reusable components is still non-trivial.

The main idea presented in this paper is the design of a system with a better integration of the component paradigm and model driven mechanism in a Unified Process-like [2] approach. We will particularly demonstrate that, because of this lack of integration, the traceability between requirements and components designed during deployment task is far from obvious in current software development processes.

In this paper, we propose to enrich the UML notation with the logical component concept by extending its meta-model and we describe the method to ensure traceability. The extension mainly consists in adding

the logical component and business archetype concepts [23]. Thus, it allows to map business processes as described by the use cases with logical components of the analysis model. We finally discuss the benefits of our approach, according to our current works. A simple case study, the “hotel room management” system illustrates our proposal.

2. The component dimension in current engineering processes

Based on the use case model, the analysis task consists in finding and defining system entities. Design and implementation tasks deal with technological concerns (see Figure 1). The use case approach allows to specify user requirements and to provide an artefact that is understood by each stakeholder. Nowadays, software engineers master analysis of information systems by applying approaches like the Unified Process (UP) [2]. The UML [1] is commonly used to build most of the models. Each model corresponds to a view on the whole system specifications, which is the responsibility of a particular competency [3]. However, working with engineers on actual projects in insurance, health care and banking companies, we observed that there is no structural guide to map use cases with deployable components. Moreover, designers and developers are guided by technology concerns, and therefore often change decisions previously made by business analysts about final users’ requirements. Different stakeholders in a project team guided by a current engineering process build several distinct models using UML. Each stakeholder has his own knowledge and responsibilities. Thus, existing tools provide an appropriate view of the system according to stakeholders’ different concerns.

The use case model contains the system requirements grouped in coarse-grained functions, i.e. the use cases. Sets of scenarios detail use cases. Dynamic UML sequence diagrams realize use cases and model each scenario of use. A sequence diagram shows interactions between architecture elements to process and realize requirements. Static UML class diagrams, an artefact of the analysis task, define the types of the architecture elements and their relationships. Package diagrams regroup classes to reduce system complexity and dependencies. The design model is the transformation of the analysis model taking into account technological concerns, applying design patterns, proposing new decomposition depending on these new concerns, and so on. A main goal of the design task is to discover UML deployment components and model them in a component diagram. This diagram is a representation of the binary components coded during the develop-

ment task. Then, deployed components are tested to finally check that all requirements have been realized.

The goal of this simplified description of a UP-like process is not to be exhaustive but to demonstrate that traceability between each of these models is not implicitly supported and therefore not guaranteed. Different models provide mismatched system decompositions. Use case model and sequence diagrams are organized by use cases. In analysis classes diagram, software engineers take care of object-oriented concerns, and of technological interests in design model and component diagrams. Testers come back to the use case paradigm to check that requirements have been completed. An experienced architect or a competent project team leader is the only one with a global perspective of the work. He is responsible to check the consistency between several views, using a matrix mechanism linking requirements with architecture elements of all views. This reduces quality and readability of the system.

Since the technological targets are component based, we claim that component must be the master piece of software and be present all along the development process. According to these issues and context, we list, in the next section, the fundamental features to be identified in a component-based system. The concept of component first appeared in research on middleware [8] [9] to deploy part of software that can be reused in another technological context. Interoperability was the initial purpose of component research. Some formalisms and notations have been elaborated to describe component-based systems. However, design approaches did not appear immediately. Thus, engineers adapted object oriented methods to deal with finding and defining system components. Currently, there exist component-based approaches [4] [6]. They are an adaptation of the UP [2]. Generally, they use and adapt UML and apply common development life cycle that drives developers from use cases to UML deployment and component diagrams. However these models remain relatively independent. Finally, these approaches do not cover a complete MDA process and do not obtain its benefits. UML2 only enriches the class concept with some component concepts like Provided interfaces that represent and categorize operations or services provided by a component and Required interfaces that represent operations and services requested by a component to successfully complete its functions.

The aim of a component-based method is to define the component boundaries, their interfaces and finally their connections to realize all the user requirements. According to these features, we claim that UML diagrams must be enriched with the concept of *logical component*. In the next part, we present in an iterative

way, our method to implement this proposal. We demonstrate that it should be the best unit of software at a conceptual level.

3. Method to transform requirements into logical components

The Information System (IS) engineering major issue is to define the best way to decompose a system, and thus to reduce and to manage its growing complexity. As the complexity of IS due to the large business domains and to the mismatched decompositions depending on the different views in the project life cycle, component based processes require a huge effort to maintain traceability. Logical component concept aims to reduce this effort by partitioning system models and by explicitly representing dependencies between its parts. Our proposal is to add the logical component concept in each view, by encapsulating a group of view constituents, as a membrane. Since each view addresses a specific concern, constituents are of different types, and their traceability is difficult to maintain. As the logical component semantic is the same in the different views, a logical component acts as a pivot to support traceability and should not be transformed.

Figure 1 is the use case diagram of our case study "Hotel room management". The system allows to reserve rooms, to check-in, and check-out. The accountancy concerns are processed. When a customer leaves the hotel and checks out, he pays to an employee who validates the payment.

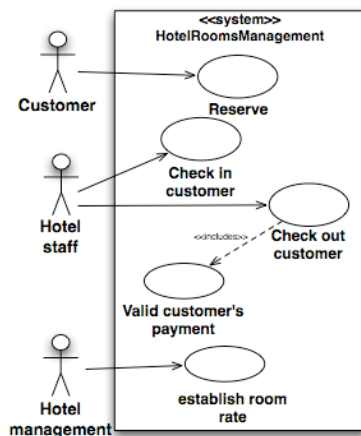


Figure 1 - Use case diagram of the "Hotel room management" system

We named our proposal CUP as *Component Unified Process* [24] because, it adapts the Unified Process by extending it with the logical component concept. We propose to split the system model in four views. The different views of our process are presented with ex-

amples from the case study. The CUP meta-model of the method links these four views. Any constituent depends on each other. The four views are the framework of the method and structure its presentation.

One of the first activities in a design process is the identification and the specification of the system functions. The habits are to use a UML use case diagram to do this. Anyone can understand this diagram, without any technical knowledge. It plays the role of a boundary object between each stakeholder. CUP introduces in the **use case view** the concept of logical component. It is an innovation because it allows an early identification of the component of the system. **Component design view** is a white box view representing the internal static structure of the logical component, its provided and required interfaces. The **interaction view** shows dynamically the interactions between each internal object of a component. This view is close of a UML sequence diagram. Moreover, CUP introduces external object interactions which specify that an object inside a component needs a service outside of it. This view specifies the dynamic of a component in an independent way. It shows interactions between parts and through interfaces of a component to complete use-case scenarios. The whole system is represented in the **assembly view** as an assembly of connected logical components. This black box view only shows provided and required interfaces. It does not consider their realization.

CUP is an iterative process. Thus, we present two iterations. The first one proposes a straight application of the method. It is a quasi-systematic way to proceed. The resulting model consists in decomposing the system into primitive logical components. The second iteration enriches the first one by detailing the features. It proposes a simple mechanism to support analysis. The system is represented by composite logical components which are a merge of the primitive ones.

3.1 Primitive logical components design

Use case view

A logical component is a membrane, which contains a subset of model elements. Requirements are modeled by a set of use cases. Our proposal is to identify logical components when building use cases. With the use case view of the system, we first propose to encapsulate each use case in a logical component. In that sense, a use case will be realized by elements enclosed within this logical component in each view of the system. We name this concept a *Primitive component*.

Component design view

Object oriented analysis consists in providing an abstraction of the real world. Object idea is twofold. Firstly, it has properties which represent its state. Secondly, it has operations which implement its behavior. A basic way to represent the real world is to group entities which have common state and behavior within an object. This is a wrong way to do, because objects are too small to be re-used and are too dependent of their context. We propose to use archetypes [23] to define a better way to create a more re-usable piece of software. Archetypes contribute to solve the issue of what is the best reusable module in the software. We propose the use of four business archetypes [23], one business archetype, which represents the dynamics of the business process whereas the three others represent the statics of the business entities involved in the process.

The Party, Place, or Thing (PPT) archetype depends on the subjacent entity. It allows to characterize an object as an entity which has properties and operations referring to business data and to business behaviour. It is used in one or several business processes. The archetype *description* models a record of data attached to a *PPT*, as, for instance, the Category of a room (see Figure 2).

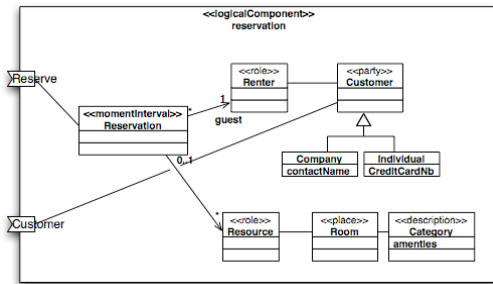


Figure 2 - Static diagram of Reservation and Check-in use cases

The *moment-interval* archetype refers to a business process. It is a session-long life object. It allows to model the purpose of a business process, or a use case as the Reservation, in our example.

The *role* archetype allows to link a moment-interval which has a short life with a specific purpose to the business entities represented by a *PPT* archetype. With a *role* archetype, we can represent the way a *PPT* is involved in one or several contexts of use. There is a strong link between the use case definition and the business process one. As we know that different companies share similar business processes and that a business process can be modelled by the four archetypes, we can link requirements with analysis artefacts in a readable way. Links are established by identifying roles of objects, from a really simple and straightforward

ward object oriented analysis. We then maintain traceability, with the *moment-interval* archetype.

Interaction view

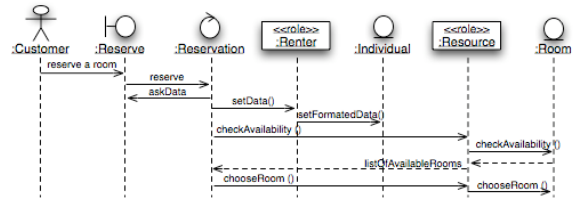


Figure 3 - Interaction view of the primitive

The interaction view (see Figure 3) partially models collaboration. Handling this view to find collaborating objects should help to discover components and to establish their boundaries. The view uses graphical UML analysis stereotypes [12] to represent instances of archetypes and traditional classes. **Entity**: *PPT* archetypes can be represented by an *entity* analysis stereotype. It is a business object containing business data and behaviour. **Control**: a *moment-interval* represents behaviour and data contained in an object which exists only during a transaction, we use a *control* analysis stereotype to represent it. **Boundary**: port is an interaction with another component, as an actor link is a relationship with a human or a computer system. We therefore use a *boundary* analysis stereotype to represent a port or an actor interaction.

Assembly view

An assembly view gives a black box point of view of the logical component (see Figure 4) and connections with other ones.

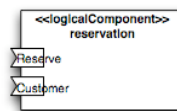


Figure 4 - Assembly view of the primitive logical component "Reservation"

Identification of primitive logical components in the use case view corresponds to Jacobson proposal of the use case module. Jacobson defines a use case module [10] to deal with that issue. "If we could keep use case and its realization separate, and maintain that separation, we would get a system really simpler to understand, to change, and to maintain". Unfortunately, he demonstrated that tangling (a component contains code that realizes several use cases) and scattering (a use case is realized by the code of several connected com-

ponents) phenomena do not allow splitting a system in use case modules only.

3.2 Composite Logical Components discovery

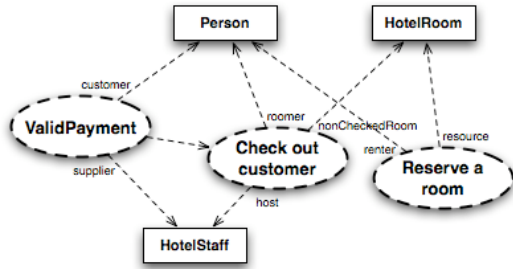


Figure 5 - UML collaborations

As we saw before, the use case is not the best module of software reuse. Dependence between objects is dramatically hard to manage. If we keep separate objects in different logical components, the complexity remains identical. UML 1.4 [1] specifications define a collaboration as a set of roles played by objects and their interactions [26] (see Figure 5). Each interaction of an object enrolled in a collaboration is represented by a link, which is characteristic of its role in this interaction. For instance, a *Person* object is enrolled in the *ValidPayment* collaboration, and has *customer* role within it. The same object *Person* is enrolled in the *Check-out customer* collaboration, and has a *host* role with other responsibilities. As the use case concept, collaboration has a useful goal. Thus, a use case can be transformed into a set of one or several collaborations which share the same role. This is a crucial logical link between the use case model and the analysis model which the process must save.

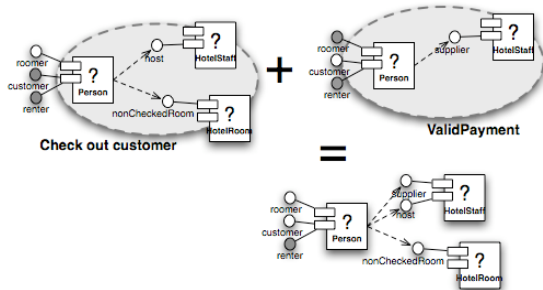


Figure 6 - Design frameworks composition

Logical interfaces, i.e. responsibilities, are added to an object, for each collaboration within which it is involved. To specify and then implement the whole object, it is mandatory to know all its responsibilities [4]. Named design framework in Catalysis method, a com-

position mechanism allows to define a complete specification of objects (see Figure 6). This mechanism achieves to explicit precisely how a use case is realized in the analysis task. It shows why it is difficult to keep traceability manually, because of the many choices to do.

We apply this composition mechanism to propose a heuristic to discover logical component boundaries. In our *Hotel room management* example, we group use cases as follows. Use cases are realized by collaborations which enroll objects of the system. The goal is to minimize dependencies between components. To achieve it, we do not authorize the enrollment of an object in collaborations which realize a use case encapsulated in another logical component. Sometimes, it is not possible and we add required interfaces to define interactions between components as actors interactions. The use case view (see Figure 7) shows two logical components that encapsulate sets of use cases. The interactions represented in are crossing logical component boundaries through interfaces between components. The second iteration of the process illustrates and explains the composition mechanism that creates composite logical components, in each view.

Use case view

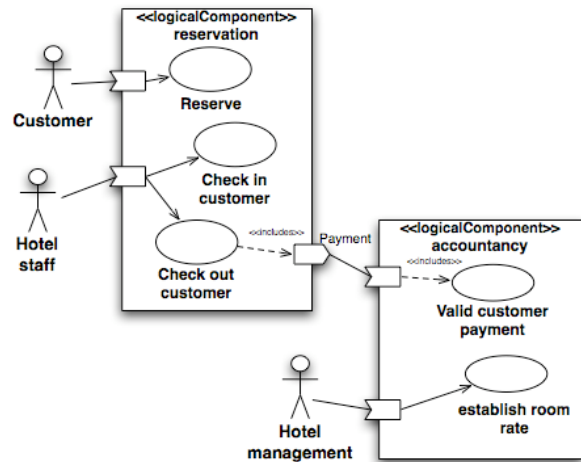


Figure 7 - Use case view of "Hotel room management" system

Regarding subjacent object collaborations realizing uses cases, some of the use cases can be grouped in the same component (see Figure 7). This kind of logical component is called *composite*. We choose this configuration containing two components, one for reservation and occupation of the room and one for accountancy concerns. Designers are free to choose the best selection, the one that reduces interactions and dependencies through component boundary, or simply the one that takes into account SI constraints. Figure 7 results

from some choice made by the designer. Boundaries of primitive logical components are not commonly kept in order to enhance the model readability.

Component design view

The goal of a UML collaboration [1] between a set of objects is to identify the best reusable piece of software, not the objects themselves. The use of archetypes can answer the issue: how to design collaborations? We propose to use archetypes to represent in a class diagram the associations between objects in one collaboration represented by a *moment-interval* (see Figure 8). A collaboration is typically designed with sequence diagrams. But sequence diagrams, which represent use case scenarios, detail interactions between objects. Moreover, the class diagram is no longer linked with the sequence diagrams that allow to define roles of objects.

A composite logical component applies the composition mechanism. As in a use case view, primitive logical components boundaries are not kept. But in this view, the designer has to choose the *PPT* objects. In the example (see Figure 8), the designer chooses to group the Reservation, Check-in and Checkout use cases. As views have to be consistent, the designer in charge of the component design view, groups the corresponding *moment-intervals*. The three *moment-intervals* define the roles of the *PPT* objects. As in the composition of frameworks, each role must be the responsibility of one entity object. In this view, the designer has also to define the interfaces of the *PPT* objects. Another decision has been to keep the Room entity outside of the component to enhance its reusability. This decision has been made in the assembly view presented in the next section.

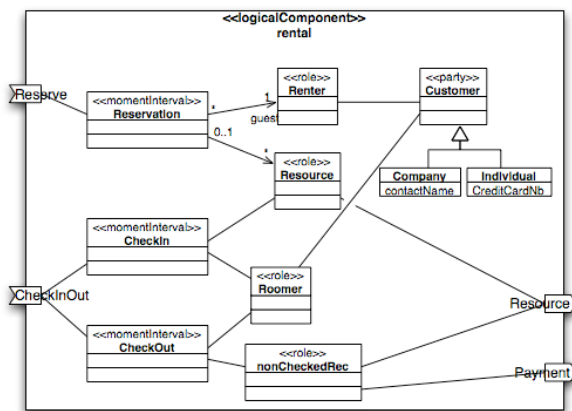


Figure 8 - Component Design view of the "Rental" logical component

In a collaboration, a set of objects interacts for one goal, as specified in the requirements. It is really diffi-

cult to reuse a part of a collaboration, i.e., a subset of the participating objects. Because they are linked by the collaboration, they have responsibilities corresponding to the role they have in this collaboration. Then, the analyst's goal is to define the best boundary. Archetypes provide a way to explicit collaboration in an object-oriented vision. They explicit interactions between objects enrolled in one or more collaborations.

Thanks to the archetypes, whatever the design choices are, traceability is maintained. The *moment-interval* archetypes always represent use cases and their related business process.

Assembly view

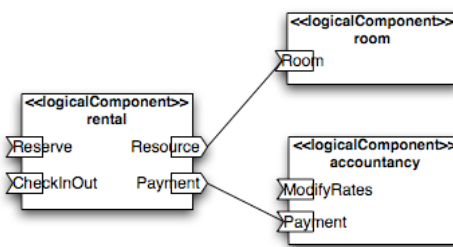


Figure 9 – Assembly view

In this view, an architect decides to isolate room entities. Then the Rental component becomes independent of what is rented and is reusable in another context. Actually, the main concern of this view is the reusability of the logical components.

Interaction view

As we saw before (see Figure 7), a use case view allows to decompose a system in logical components, grouping a set of use cases. Standard relationships between use cases, *extends* and *includes* are kept. The difference with a traditional use case diagram is that these relationships can cross component boundaries through interfaces of the logical components. They explicitly show the functional dependencies between components. It is obvious that there are interfaces in other views, which drive the designer to discover these new interfaces.

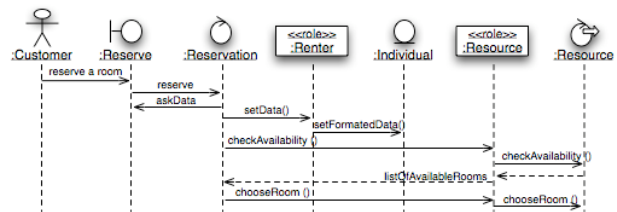


Figure 10 - Interaction view

In the interaction view, we add the externalControl stereotype for the *Resource* object. This analysis stereotype represents the component required interfaces. It shows the dependencies of the component with its context, while keeping autonomous the component.

4. Implementation and evaluation

4.1 Method implementation using MDE mechanisms

CUP is specified by an extension of the UML meta-model divided in four parts (see Figure 11). The concept of logical component has been added, with its required and provided interfaces, and the externalControl role. As the notation used in the method is described in a meta-model, its implementation can be facilitated by a modeling tool such as ModX [27], or by the IBM Eclipse Modeling Framework [28].

Object constraint language rules [1] have been defined to constrain and guide the design. They define consistency relationships between each model element. Once the PIM has been built (→1,2), these tools can generate code according to some generation rules (→3,4). As the properties of our component model are independent of any technological aspect, this enables the translation of a CUP component into several target platforms. In order to define mapping rules, we just need to associate a CUP concept (described by a meta class) with a platform specific concept. Secondly, at each model level, the design can induce modifications at another model level. The tool will inform the designer to carry out these modifications in order to restore the design consistency of the complete model.

The CUP approach has been experimented in actual projects in large business domains, i.e. insurance, banking, health care. It has been proven that applying such division early in the development process, and maintaining consistency with a tool is more efficient. Then traceability is more maintainable. The impact of an evolution is more quickly detected. The efforts are more easily evaluated. Some components have been reused in different projects. It is a practical and efficient way to validate the method and its benefits about requirements traceability.

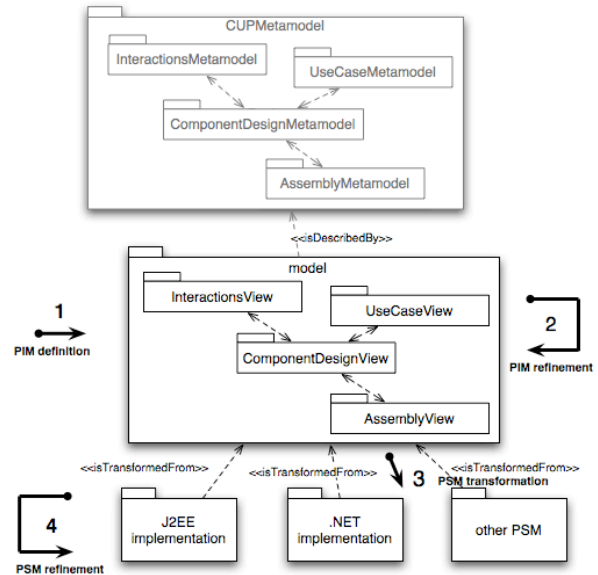


Figure 11 - Model Driven Engineering process

4.2 Related works

Components are often introduced too late in the process, according to technological concerns as in the popular UP process [12]. We propose with CUP to deal with component boundaries very in the process, during the requirements phase to ensure better traceability. Then, all stakeholders can participate in the component identification of their information systems. Benefits of a model-driven approach, such as CUP, are that, with design methods, analysts and designers focus on functional concerns, leaving aside the technological ones. Then the produced PIM is more re-usable, and could be more easily transformed into a Platform Specific Model, then into code.

Existing component-based methods often come with a complexity of use. Catalysis [4] is one of the most complete method, but is really difficult to implement. CUP is formalized by a meta-model, and its related constraints, and then MDE tools [27] can guide designers with more flexible processes and tools.

Finally, the concept of logical component guarantees a more focused and more efficient work, and as it is omnipresent, it guarantees consistency between views, from requirements to deployment. Regarding UML 2.0 [25] components, that are enriched classes, CUP ones, are more as a framework, i.e. a set of classes as [5]. Then reusing these components provides a better return on investment, because it includes the reuse of all the models of all the component views and

allows to connect these components without any technical concern.

5. Conclusion

In this paper, we propose an innovative method to ensure requirements traceability in the project development cycle. First, a use case based solution is used to express requirements. CUP allows an early identification of logical components in the use case view to be decided by each stakeholder of the project. The introduction of archetypes increases the quality of the system model, by checking the functional division of the system. Then, collaborations that realize use cases are explicitly designed with *moment-interval* archetypes in the component design view. Analysis based on archetypes, helps to find and to consolidate boundaries of logical components. Finding the required and provided interfaces completes the PIM building. Traceability has been ensured by the subjacent meta-model which links the different views. Then generative tools [27] can exploit the results thanks to MDE technology.

Dealing with component identification early in the process makes easier the component re-use. For instance, an existing component answering one or more use cases can be early detected. The re-use of business components will be favored by our approach and allows to build more efficiently future information systems.

References

- [1] OMG. UML1.4 – Unified Modeling Language. Object Management Group, September 2001.
- [2] Grady BOOCH, Ivar JACOBSON, and James RUMBAUGH. RUP Software Engineering. 1997.
- [3] Kruchten P., « Architectural BluePrints -- The "4+1" view Model of Software Architecture », IEEE Software 12, pages 42-50,1995.
- [4] Desmond Francis D'SOUZA and Alan Cameron WILLS. Objects, Components, and Frameworks with UML - The Catalysis Approach. Addison-Wesley, 1998.
- [5] John CHEESMAN and John DANIELS. UML Components - A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2001.
- [6] Peter HERZUM and Oliver SIMS. Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise. Wiley Computer Pub., 2000.
- [7] Hassine, I., Rieu, D., Bounaas, F., and Seghrnouchni, O. Symphony: a Conceptual Model based on Business Component. Revue ISI 7, 4, HermSs, 2002.
- [8] OMG, « CORBA 3.0 New Components Chapters », OMG ptc/2001-11-03, Object Management Group, Novembre, 2001.
- [9] Sun, E.J.B. Home Page, 2001.
- [10] Jacobson I., « Basic Use Case Modeling », ROAD 1, 1994.
- [11] Jacobson I., « Use Cases and Aspects - Working Seamlessly Together », Journal of Object Technology, vol. 2, num. 4, ETH Zurich, pages 7-28, July-August 2003.
- [12] Jacobson I., Booch G., Rumbaugh J., 97, « The Unified Software Development Process », Addison-Wesley, 1997.
- [13] Mellor, Stephen and Balcer, Marc, Executable UML, Addison-Wesley, 2002
- [14] Kruchten, Philippe, The Rational Unified Process- An Introduction, 2nd edition, Addison-Wesley, 2000.
- [15] Gamma, Eric et al, Design Patterns, Addison-Wesley, 1995.
- [16] IBM Corporation, Business Systems Planning, Information Systems Planning Guide, Publication No. GE20-0527.
- [17] Kerner, David, Business Information Characterization Study, Data Base 10, No.4, pp.10-17, Spring 1979.
- [18] Carlson, W.M., Business Information Analysis and Integration Technique (BIAIT)-The new horizon, Data Base Vol.10, No.4, pp. 3-9 Spring 1979.
- [19] Lefebvre, Éric, Améliorer les méthodes de planification informatique: une approche pluraliste, Thèse de doctorat, Université Grenoble II, avril 1996.
- [20] Fowler, Martin, Analysis Patterns, Addison-Wesley, 1996.
- [21] Taylor, David, Business Engineering with Object Technology, Wiley, 1995.
- [22] Jacobson, I., et al., Object-oriented Software Engineering: a Use-Case Driven Approach, Addison-Wesley, 1992.
- [23] Coad, Peter, Lefebvre, Eric and De Luca Jeff, Java Modeling in Color with UML, Prentice Hall, 1999.
- [24] Renaux E., Caron O., Geib J.M., SEA - IASTED, Marina del Rey - Los Angeles Nov. 2003, The CUP Project - Component Unified Process
- [25] UML 2.0 Infrastructure : OMG doc. ad/00-09-01
- [26] Cariou E., Beugnard A., Jézéquel J.M., « An Architecture and a Process for Implementing Distributed Collaborations, EDOC'2002, 2002.
- [27] Le Pallec X., Renaux E., Olavo Moura c., ModX - a graphical tool for MOF metamodels, ECMDA-FA'2005, Open Source and Academic Tools, November 7-10th, Nuremberg, Germany
- [28] IBM, Eclipse Modeling Framework (EMF), <http://www.eclipse.org/emf>