Traceability Management through Use Cases when Developing Distributed Object Applications

Nelly Bencomo[1],Alfredo Matteo[2]

[1] Computing Department, Lancaster University,
Bailrigg, Lancaster, LA1 4YR, UK
nelly@acm.org
[2] Laboratorio TOOLS, Escuela de Computación, Facultad de Ciencias, UCV,
amatteo@kuaimare.ciens.ucv.ve

Abstract. The software life cycle of Distributed Object applications encompasses many activities, which go from requirements specification and leads to design and implementation taking into account aspects related to architectural issues. In such a life cycle, activities related to communication and integration mechanisms defined in Distributed Objects Technologies have to be executed. On the other hand, the support for software traceability has been established as an important task in the development life cycle of software systems. As the design is refined to a concrete implementation, it is important that concepts in analysis and design have a clear correspondence to implementation artifacts. This article describes activities and artifacts associated with Analysis, Design, Implementation and Deployment models when developing Distributed Object applications. In this sense, this work proposes a clear traceability from the Use Case model through Analysis, Design, Implementation and Deployment models. An example of the traceability is presented by means of a case study involving web access to Bank accounts. **Keywords**: use cases, distributed objects, traceability, and UML notation.

1. Introduction

Distributed Object Technologies, such as CORBA and RMI, became popular and evolved quickly over the past decade. Some of the most important software systems in the world have been developed using these technologies. Distributed object software can be found in software-intensive distributed domains such as telecommunications, health care, aerospace, and online financial applications. While the standards have matured considerably we face a lack of methods and notations for distributed object development and maintenance. Traditional non-distributed Object-Oriented design and programming differ from the IDL (Interface Description Language) and Java Remote Interfaces design and programming to be used in the construction of distributed objects; these differences must be taken into account by the software development process.
On the other hand, software traceability - that is the ability to relate artifacts which are created during the development of a software system (e.g., requirements, design and code artifacts) with each other - has been recognized as a significant capability in the software development and maintenance process, and as an important factor for the quality of the final product [18]. Traceability information can be used to support the analysis of the changes requested in the system development process; the maintenance, evolution and documentation of software systems, the reuse of software systems and their components, and testing. The development of software systems can be complex, and this complexity is even greater for distributed systems; consequently, software traceability management for distributed systems is crucial.
In this paper we describe how to establish a clear traceability among Analysis, Design, Deployment and Implementation models when developing Distributed Object applications based on the use cases approach. In this sense, control objects of the analysis have a direct correspondence with distributed components in the implementation and deployment models. Our approach emphasizes *thematic* use cases; use cases that are related to distribution concerns. We use UML [14] notation in the specification of models, and define and use some extensions of the language.
This paper is organized as follows: Section 2 provides an overview of the approach; Section 3 gives the description of activities and artifacts related to the design of the models; Section 4 presents the case study, Section 5 discusses related works; and finally Section 6 talks about future work and draws some conclusions.

2. The Approach

The essence of our approach can be synthesized in three key phrases – *architecture, use case driven and traceability*.

2.1. Architecture

The architecture embodies the major static and dynamic aspects of a system. It is a view of the whole system highlighting the important characteristics and ignoring unnecessary details. In the context of our approach, architecture is primarily specified in terms of views of five models; the Use-Case model, Analysis Model, Design Model, Deployment model and Implementation model. These views show the "architecturally significant" elements of those models. The models have the following specific characteristics in our approach:

- The Use-Case model shows the thematic use cases related to functionality associated with distribution.
- The Analysis model illustrates how boundary, control and entity classes are associated with the thematic use cases identified in the Analysis. Remote Communication Control classes shown in this model are specializations of Control classes and represent the abstraction of components that deal with remote communication and distribution using CORBA.
- The Design model shows the design classes that trace the specialized Remote Communication Control classes in analysis. Special attention is given to the interfaces provided by these design classes. We show how some of these are represented by IDL interfaces.
- The Implementation model describes how elements in the design model are implemented in term of components.
- Finally, the Deployment model explains how CORBA-based components are assigned to nodes.

2.2. Use Case Driven

In the early steps of the life cycle, use-cases are mainly used to specify the functional requirements of the system. Later on, and based on the use-case model, developers create the models that realize the use cases. The developers review each successive model for conformance to the use-case model [7]. Our approach emphasizes *thematic* use cases. In general, the *theme* varies depending on the nature of the project. In our case, a use case is thematic if it is related to distribution. Once thematic use cases are specified identifying the Remote Communication Control, they are designed and implemented using the corresponding design classes and their IDL interfaces.

2.3. Traceability

An important part of traceability is that the final implementation is consistent with the design and analysis. As the design is refined to a concrete implementation, it is important that concepts have a clear correspondence to implementation artifacts – even if the mapping is not one-to-one [13][15]. In our approach, specialized control objects in analysis –that are associated with thematic use cases and are called Remote Communication Control Objects– are the abstractions of components in charge of remote communication in implementation. In between we define the design classes, specified by their IDL and UML interfaces.

3. Models

This section presents a short description of the Analysis, Design, Implementation and Deployment models.

3.1. Analysis Model

Control classes responsible for remote communication and that can potentially be mapped onto different nodes in the distributed system are identified. To do this, we define a Class Diagram (Architectural Description – Analysis View) that comprises boundary, control and entity classes of the thematic use cases. Initially, we have a control class for each use case. The generic class diagram proposed is shown in Figure 1. Control classes address the messages exchanged among boundary and entity classes to fulfill a specific functionality. Changes to identity or boundary classes are locally solved without changing their counterpart.

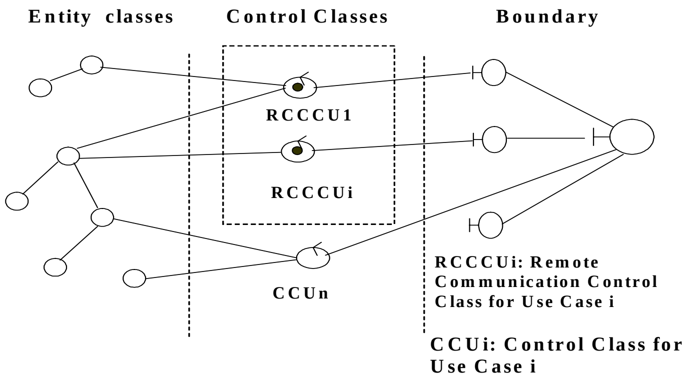
Entity classes
Control Classes
Boundary
RCCCU1
RCCCUi
CCUn
RCCCUi: Remote Communication Control Class for Use Case i
CCUi: Control Class for Use Case i

Figure 1: Class diagram objects related to thematic use cases

Because we are focusing on thematic use cases –related to distribution concerns- entity and boundary classes might be related to functionality associated with distribution. Entity and boundary classes are then abstractions of components deployed on different nodes. In these cases, the intermediary control class has to deal with remote communication and distribution. These intermediary control classes are specializations or adaptations of UML control classes in the Use Case Model. We adapted and stereotyped them to get the Remote-Communication Control Class (RCCC). As shown in Figure 1, RCCCs are graphically represented as a common control in UML with a filled circle inside. These RCCCs are the first link in a chain of artifacts that evolve from Analysis through all the process until reaching the CORBA distributed objects in Implementation.

In some cases, the nature of the application could dictate specific conditions of component distribution in the implementation. For example, two different sets of analysis objects might be required to represent implementation components deployed on different nodes. We propose to use a variation of the Analysis Class Diagram explained above. In these cases, the control classes identified are intermediaries that allow the communication among components deployed on different nodes. Figure 2 shows an example of a class diagram associated with the communication between different nodes. As in Figure 1, RCCCs have to deal with remote communication and distribution but this time entity classes are the only abstractions of components, boundary classes related with actors of the system are not included.

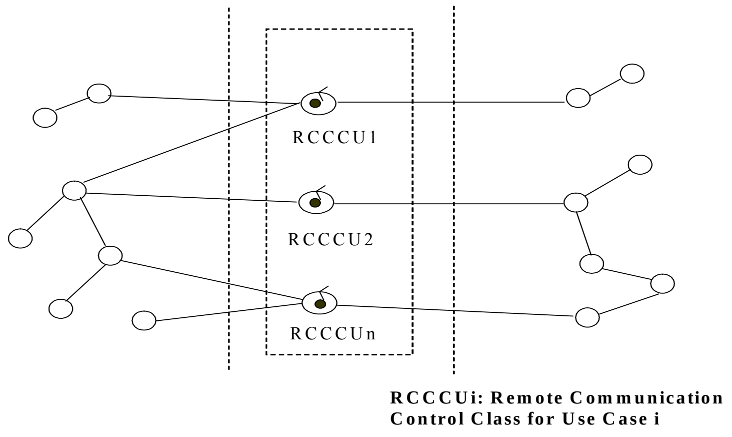

Figure 2: Class diagram associated with the communication between different nodes

In both Figure 1 and Figure 2, the dashed areas depict how abstractions related to the distribution concern are modularized in what we have called thematic use cases.

Example of Legacy Applications: applications that use a Legacy system through remote interfaces

In this case the system to be developed involves the integration of existing applications. The Legacy subsystem is modeled using analysis packages. Control classes in analysis are the abstractions of the wrappers for the Legacy subsystem and are modularized in a Distribution Management Analysis Package. See Figure 3.

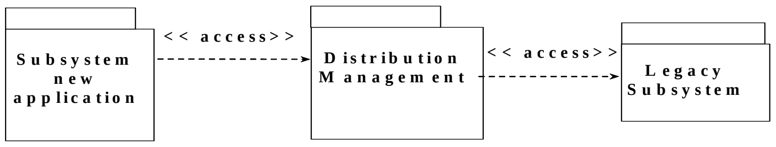

Figure 3: Analysis Packages when integrating existing applications

3.2. Design Model

In Design, there are two important activities to be performed: architecture definition and the specification of design classes. We study the use case realizations in analysis and define the corresponding design classes and their sequence diagrams.

Some design classes can be initially sketched from analysis classes; this is the case of design classes that deal with remote communication. A RCCC associated with the *use case i* in analysis will correspond to a pair of design classes. In Figure 4, the *trace* relationship between a RCCC and its two corresponding design classes is shown.

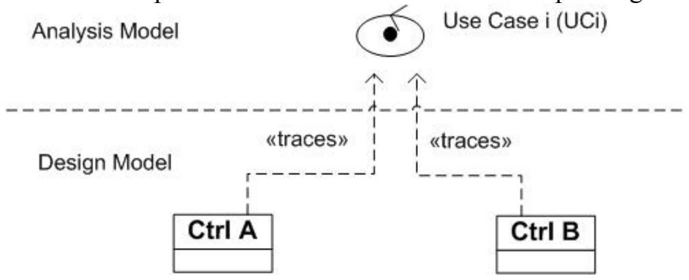

Figure 4: Correspondence between remote-communication control class in analysis and control classes in design

Basically, design classes expose two kinds of interfaces. One interface has the common UML semantics and the other is an IDL interface. IDL interfaces let CORBA objects communicate and send/receive the messages that components are receiving/sending. Methods of these interfaces are specified from the interaction diagrams. A concrete example of these interaction diagrams is shown in the case study of Section 4.

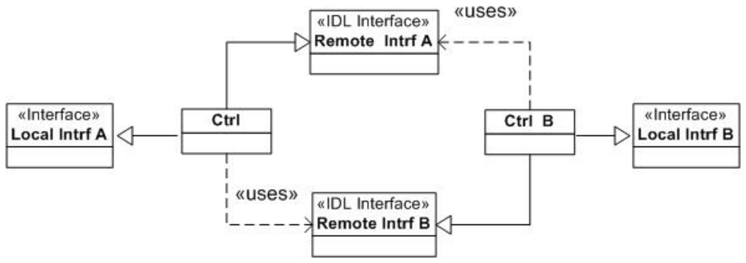
«IDL Interface» Remote Intrf A
«uses»
«Interface» Local Intrf A
Ctrl
Ctrl B
«Interface» Local Intrf B
«uses»
«IDL Interface» Remote Intrf B

Figure 5: Control classes and their interfaces

The graphic notation used in Figure 5 has an alternative where IDL interfaces are represented by T-connectors, see Figure 6. The T-connector notation is based on [16].

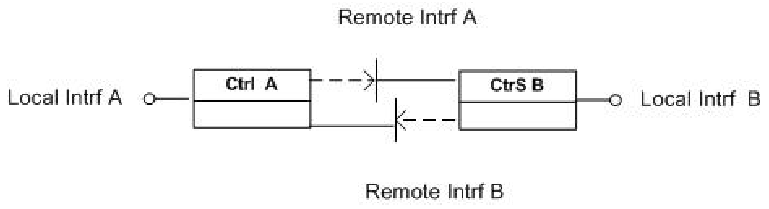

Figure 6: Another notation for IDL interfaces

3.3. Implementation and Deployment Models

In implementation, we have to program the code associated with CORBA objects and components based on the IDL interfaces in design. The Deployment model shows the mapping of CORBA components onto nodes.

Each design class traces to a CORBA Component in the implementation. Each CORBA component is a fundamental part of the system architecture. The graphic notation adopted to identify a CORBA component is based on [16]. The small ellipses and arrows in the top left corner represent remote interfaces and local (non remote) interfaces respectively.

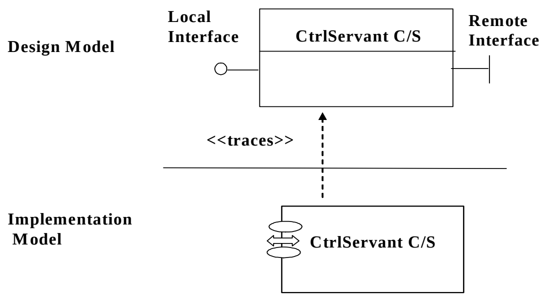

Figure 7: Correspondence between a control design class and a CORBA component in implementation

To describe the functionality and interactions among components we define a diagram that includes and modularizes only CORBA components and their interfaces. This Diagram is used to define the Deployment Model**.**

3.4. Traceability

Figure 8 shows the traceability among the different artifacts in Analysis, Design, Implementation, and Deployment models. Note that the Remote Class Control RCCUCi is related to the use case i.

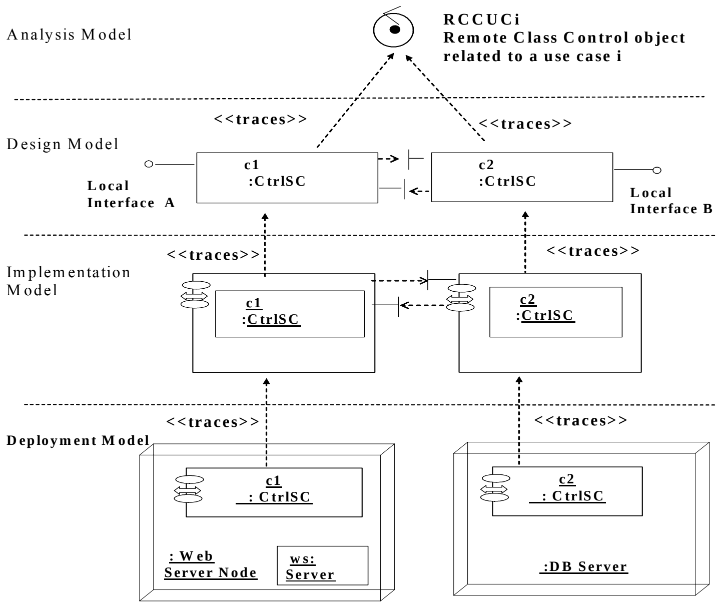

Figure 8: Traceability among the Analysis, Design, Implementation, and Deployment models related to the use case UCi

We have aspectized distribution from the very early stages of the development, isolating the business logic from the confines of system architecture. We start from a use case i and its RCCUCi. This object control is represented by two design classes in design that communicate using their IDL interfaces. These design classes are implemented by two CORBA components (Implementation Model) that are finally deployed onto different nodes.

4. Case Study: Banking System using ATMs and the Internet

We have a Banking software system that includes client services through ATMs and the Internet. A client uses the system to *withdraw*, *deposit*, *transfer* and *view* the balance of her/his accounts. Clients can use these services using ATMs or the Internet, see Figure 9.

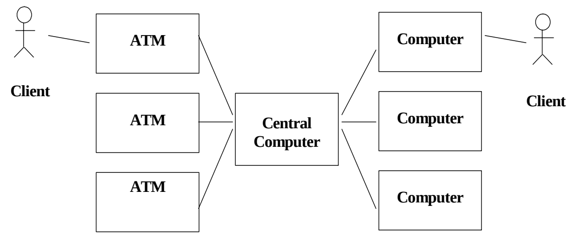

Figure 9: Banking system

4.1. Use Cases

Figure 10 shows the use cases of this system. The functionality is offered through ATMs (*withdraw*, *deposit*, *view balances, and transfer*) or through the Internet (*view balances and transfer*). For both cases, ATM and the Internet, we consider the use cases to *login* into the system.

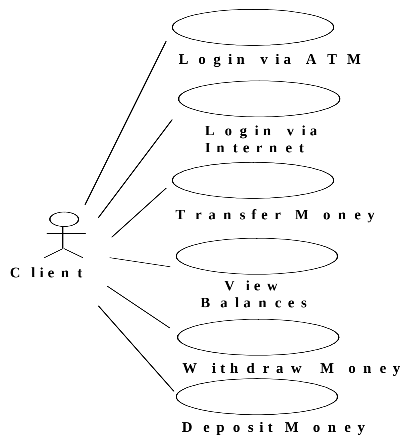

Figure 10: Use Cases of the system

4.2. Analysis: Class Diagrams and Packages

The class diagram related to use cases *Login via Internet, View Balances, and Transfer Money* when the user is using the Internet is shown in Figure 11. All these use cases are thematic as we can see that boundary and entity objects are abstractions of components deployed on different nodes. The intermediary control classes involved have to deal with the communication among boundary and entity objects and are specialized as Remote-Communication Control Classes.

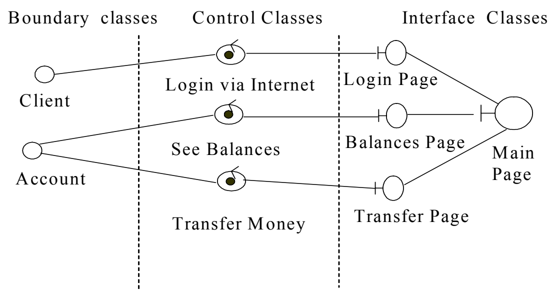

Figure 11: Class Diagram of the Case Study Banking System

It was convenient in terms of modularization of the system to group the analysis classes in three kinds of analysis package; an analysis package that contains classes related to boundary classes of the Graphical User Interface (GUI), an analysis package that contains the entity classes, and an analysis package that contains the control classes in charge of the logic of the remote communication between the boundary package and the application domain. In Figure 12 we have two analysis packages associated with the GUI, one related to the ATM and the other related to the GUI via Internet.

Figure 12 shows how abstractions related to the distribution concern are modularized in the Distribution Management Analysis Package.

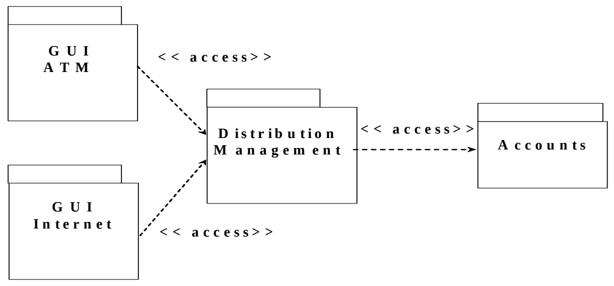

Figure 12: Analysis Packages

4.3. Design

We illustrate our approach in Design using the use case *Login via Internet*. This use case presents the following sequence diagram:

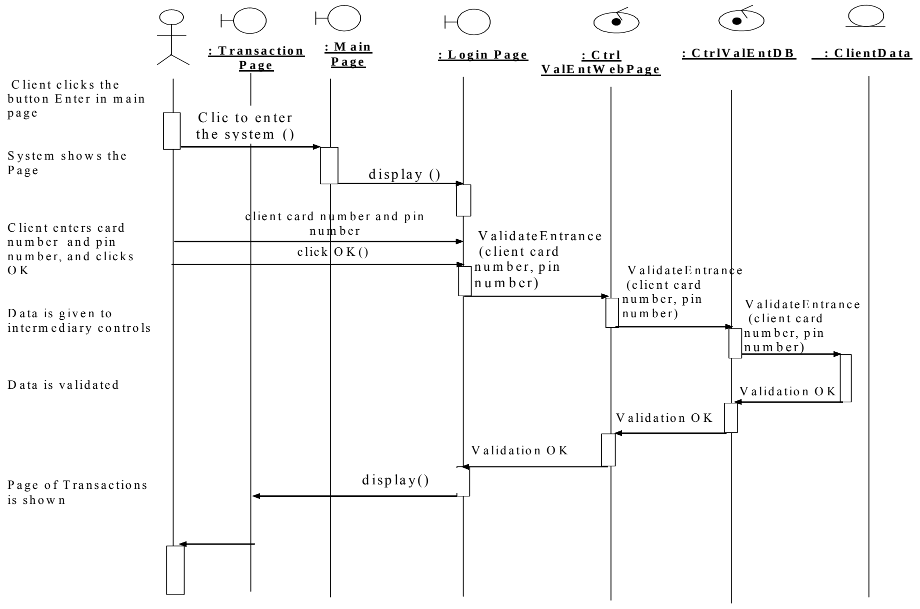
: Transaction Page
: Main Page
: Login Page
: Ctrl ValEntWebPage
: CtrlValEntDB
: ClientData
Client clicks the button Enter in main page
Clic to enter the system ()
System shows the Page
display ()
Client enters card number and pin number, and clicks OK
client card number and pin number
click OK()
ValidateEntrance (client card number, pin number)
ValidateEntrance (client card number, pin number)
ValidateEntrance (client card number, pin number)
Data is given to intermediary controls
Data is validated
Validation OK
Validation OK
Validation OK
Page of Transactions is shown
display()

Figure 13: : Sequence diagram for the use case Login via Internet

Messages *ValidateEntrance* and *ValidationOK* in the sequence diagram are candidates to be operations in the IDL interfaces of the control design classes *CtrltValEntDB and CtrlValEntWebPage*. The given name is related to the services provided on each side, Web Page side and Client Data side.

4.4. Remote Interfaces (IDL) and Local Interfaces (UML)

We have two control classes in the design; *CtrltValEntWebPag* and *CtrlValEntBD*. Figure 14 shows the IDL interfaces designed from the sequence diagram. Note that interface *CtrltValEntWebPag* contains the operation *validationOK()* and the interface *CtrlValEntBD* contains the operation validateEntrance(), operations that were identified from the sequence diagram.

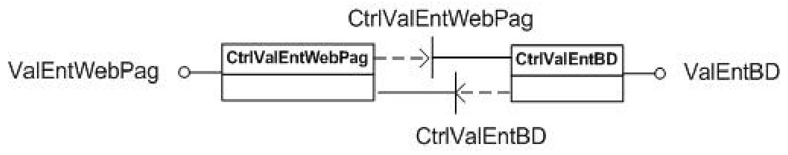

Figure 14: **Control classes in design, CtrlValEntWebPag and CtrlValEntBD**

CORBA offers the notion of IDL modules. Modules are used to encapsulate IDL interfaces. Example specifications of IDL interfaces are given in the IDL Module as follows:

```
//IDL
// Module:  Control of data verification when entering the system via Internet
Module CtrlValEntWebPage {
// Operations on the Web page side
interface CtrlValEntWebPage  {
            void validationOK();
            void validationError();
};
// Operations DB side
interface CtrlValEntBD {
void validateEntrance(
            in long client_card,
            in long pin_number);
};
}
```

4.5. Implementation

Figure 15 shows the component diagram related to the use case *Login via Internet*. A complete diagram of all CORBA components in a system is given by the union of all CORBA component diagrams associated with all the use cases.

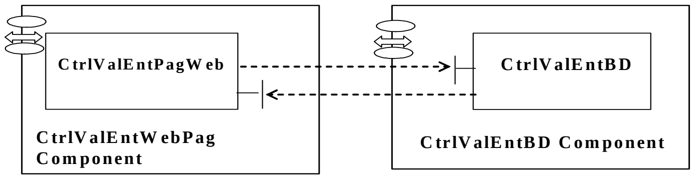

Figure 15: Component diagram: CORBA control components associated to the use case Login via Internet

4.6. Distribution

Figure 16 shows the deployment of CORBA components on nodes of the system. Specifically they describe the components related to the Use Case *Login via Internet*. Intermediary control CORBA components CtrlValEntWebPage and *CtrlValEntBD* are c1 and *c2* respectively. *c1* is on the Bank Web Server side node and *c2* is on the DB Server side (data of Bank clients).

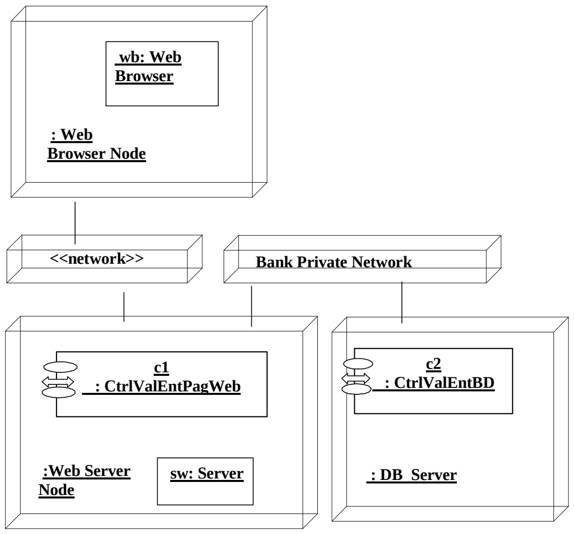

Figure 16: Distribution model: mapping of CORBA components onto nodes of the system

5. Related Work

The concept of traceability in [11] is about tracing relations among all elements, so that associations can be tracked among any given two objects, at any time. This assumption is very difficult to bring into effect when talking about distributed systems because of the number of relationships. The work presented in [18] is related to the use of natural language processing (NLP) techniques in requirements engineering and requirements traceability what is a very different approach from ours. An interesting research is shown in [19], authors present a lightweight approach to support generation of bi-directional traceability relations between organizational requirements modeled in i* and UML use cases and class diagrams. Their approach is based on the use of XML-based traceability rules to identify the relations. Finally, article [6] demonstrates how a use case is adapted to form the change case, to identify and articulate anticipated system changes. The article introduces the concept of a change case as a way to describe potential system functionality, and demonstrated how it can be used to capture potential changes and design systems that are robust to the changes identified. Authors in [6] claim that by dealing with change early in the development process, it should be possible to both reduce future maintenance costs, and extend the system's effective life span. The concept of *change case* can be compared with the notion of our t*hematic use cases* given the fact that both are special kind of use cases associated with the concerns of the authors.
The approaches cited above are general and are not tailored to satisfy the software traceability for distributed applications. In particular [11], [18], and [19] tackle the problem of software traceability from the point of view of requirements engineering, what is different from our approach. What makes our design different in comparison with other works in this area is the fact that we explicitly deal with distribution concerns.

6. Conclusions and Future Work

This paper proposed and illustrated how to specify a clear traceability from the Use Case model through Analysis, Design, Deployment and Implementation models. In this sense, specialized control objects in analysis, called Remote Communication Control Objects, are the abstractions of components in charge of remote communication aspects in implementation, in other words, control objects of the analysis have a direct correspondence with distributed components in the implementation and deployment models. We have used use cases and their subsequent realizations through all the lifecycle models to encapsulate and trace the distribution concern in separate modules

promoting localization and reutilization. These modules and localizations are reflected in the architecture of the system. The article is based on practical experience [2,3,4].
One of the issues we are currently working on is the definition of an approach for modeling distribution using a combination of Aspect Oriented Software Development (AOSD) and use cases. One of the aims is to bridge the gap between the handling of crosscutting concerns during the early and later phases of the lifecycle when developing distributed applications [1]. We are also interested in the problem of decoupling the development of distributed applications from specific middleware technologies and how the ideas expressed in this article can be applied in the definition of an abstract platform that allows clients to be developed independent of middleware implementations. Unfortunately, the large number of middleware technologies conspires against this purpose – the development and maintenance of distributed systems have become coupled to constant evolution of middleware technologies. We think that the Model Driven Architecture (MDA) and reflection together gives the basis to tackle this problem [5].

Acknowledgement
The authors gratefully acknowledge the comments and critical feedback of Pete Sawyer (Lancaster University).

6. References

[1]Bencomo N., Matteo A., and Sawyer P.: Tracing the Distribution Concern: Bridging the Gap, Submitted to Early Aspects 2004
[2]Bencomo N.,: CORBAdapted-UP: Una adaptación del Proceso Unificado para la Construcción de Aplicaciones CORBA, Promotion Project to get the Cathegory of Assistant Lecturer, Escuela de Computación, UCV, Venezuela, 2002
[3]Bencomo N. Matteo A.: Correspondencia entre Modelos en el Desarrollo de Aplicaciones CORBA, Chapter in book on Avances en tecnologías de la Información, Universidad de los Andes, Venezuela 2003
[4]Bencomo N., et all.: An experience using CORBA and OOSE in the construction of a Graphical multi-user Interface based on Distributed Objects, Proceeding of Practical Experience Segment of Objetos Distribuidos 2000, Sao Paolo, Brazil, 2000
[5]Bencomo N., and Blair G.: Middleware unaware software development and interoperability using MDA, accepted for the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations, September 7th-8th 2004, Canterbury, England
[6] Ecklund E., Delcambre L., Freiling M. Change Cases: Use Cases that Identify Future Requirements, OOPSLA 96
[7]Hunt J.: The Unified Process for Practitioners Object Oriented Design, UML and Java, Springer, 2000
[8]Jacobson I.: Use Cases and Aspects – Working Seamlessly Together, in Journal of Object Technology, vol. 2, no. 4, July-August 2003, pp. 7-28.
[9]Jacobson, I., Booch G., Rumbaugh J.: The Unified Software Development Process, Addison-Wesley , 1999
[10]Jacobson, I., Magnus C., Patrik J., Gunnar O.: Object-Oriented Software Engineering: A Use case driven Approach, Addison-Wesley, 1993
[11]Kowalczykiewicz K., Weiss D.: Traceability: Taming uncontrolled change in software development, IV Krajowa Konferencja Inżynierii Oprogramowania, Poznań 2002
[12]Farooqui K., Logrippo L., Meer J.: The ISO Reference Model for Open Distributed Processing- An Introduction, 1996
[13]Ovlinger J.: From Aspect-Oriented to Implementation, Position Paper for AOM, 2003
[14]OMG Unified Modeling Language Specification, Version 1.4, OMG (2001)
[15]Simmonds D., Ghosh S., and France R..: An Aspect-Oriented Model Driven Architectural Framework for Middlware Transparency, AOSD Workshop on Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, March, 2003.
[16]Slama D., Garbis J., Russelm P.: Enterprise CORBA, Prentice Hall, 1999
[17]Soares S., and Borba P.: PaDA: A pattern for distribution aspects. Proceedings of: Second Latin American Conference on Pattern Languages Programming - SugarLoafPLoP. 2002 Brasil. ICMC - Revista da Universidade de São Paulo, páginas 87-99.
[18]Zisman A., Spanoudakis G., Perez-Miñana E., and Krause P.: Tracing Software Requirements Artifacts. Proceedings of the 2003 International Conference on Software Engineering Research and Practice, SERP '03.
[19]Zisman A., Spanoudakis G., Cysneiros G.: A Traceability Approach for i* and UML Models, Proceedings of 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems - ICSE 2003, May 2003