# Refactoring a Legacy System Using Components*

Eda Marchetti, Francesca Martelli, Andrea Polini

*Istituto di Scienza e Tecnologie dell'Informazione - "Alessandro Faedo"*
*Area della Ricerca del CNR di Pisa, Italy*
*phone: +39 050 315 3463 fax: +39 050 315 2924*
{eda.marchetti, francesca.martelli, andrea.polini}@isti.cnr.it

## Abstract

*We present our experience in reorganizing an "inherited" monolithic piece of software in a component-based manner. We follow the guidelines of the* UML Components *in order to obtain a meaningful component architecture, which is then used for modifying and revising the inherited code. In this paper we describe both the defined component-based architecture and the stepwise process that we adopted for reorganizing the source code.*

## 1. Introduction

In recent years, Component Based Software Development (CBSD) has attracted considerable interest from both the research and industry areas, and it is fast becoming the principal paradigm in Software Engineering. However, although today some commercial frameworks (e.g. EJB [12], .NET [13], CCM [11]) permit the actual realization of component-based applications, the research is far from complete, and further studies are necessary especially regarding component specifications, development methodologies and development tools [7].

The reason for the widespread diffusion of the CB paradigm is due to the increasing complexity of current developed systems. They are often characterized by distributed features, remote accessibility and high evolvability, i.e., the system must be easily expanded with new functionalities or improved by modifying the existing code without affecting the services already provided. To comply with the strict delivery times imposed by the market, the developers of complex systems use experiences coming from various research and industry fields such as *software architecture* (SA), *component-based software standard* and *middleware* (the latter refers to software that simplifies the construction of distributed heterogeneous systems by providing high-level interaction services on top of, e.g., a transport protocol). In particular, they mature the common best practice of specifying a software architecture as a first step during development. It is worth noting that the concept of SA is not yet well-established and several definitions have been provided to classify it [5]. Informally, the SA expresses the logical structure of the software to be developed, in terms of components and their interactions (connectors). The SA will then be implemented during the software development by substituting the architectural components with real ones which must conform to the logical structure. In this step, the real components can be retrieved either from the market or from inside the organization, or implemented from scratch.

One of the emerging trends in CBSD is the use of UML [10] for system specification, which is the *de facto* standard notation for the analysis and design of Object Oriented (OO) systems. Even if UML was not conceived with a component-based paradigm in mind, by its extensions mechanisms, it proves to be flexible enough to be adaptable to different contexts. Thanks to this characteristic, Cheesman and Daniels in [2] propose an innovative methodology, called *UML Components*, which focuses both on the representation of components and on the development process applicable for this purpose.

Therefore the advent of the CB paradigm has opened a new and promising approach for the refactoring of legacy systems [6, 8]. In this paper we apply the *UML Component* methodology for designing and re-implementing an additional layer over protocol GSM-MAP, used in the telecommunication context for mobility management and other services. Our objective is to define a suitable system architecture which can be used for re-organizing some of the already implemented functionalities in a component-based manner, by reducing modification and code rewriting as much as possible. In our case, in fact, the available pieces of software, each related to a different functionality, were developed during the course of several Master's theses, assigned and conducted in an uncoordinated and

[0]This work has been supported by Ericsson Lab Italy in the context of the PISATel initiative (www.iei.pi.cnr.it/ERI/)

independent manner, without referring to a common architecture. In particular, the UML diagrams provided by the students were related only to the code organization and specification, and thus are not easily usable for implementing further functionalities or improving the existing one. Moreover, available OO code was structured in a monolithic way with many inter-object references. In order to reduce the time and effort needed for system implementation, it is our intention to first define a suitable system architecture and to reuse the existing code as much as possible, by restructuring and extending it when necessary on the basis of the defined architecture, and completely realizing only the missing functionalities.

Considering the above situation, for us the methodology proposed by Cheesman and Daniels is a defined development process to follow in order to obtain a well-structured and working system, which facilitates UML specification, code restructuring and testing. Of course, other development processes could be adopted for the same purpose, but are beyond the scope of this paper either deeper investigation of this topic or providing a comparison between them. We adopted *UML Components* because it is an innovative methodology for defining a complete UML architecture specification, which can be used as an input to some existing tools, such as Cow_Suite [1], in order to reduce the effort due for testing phase development.

Thus, in this paper we report our experience by describing the steps of the process adopted, by pointing out the difficulties encountered and the problems faced in applying the *UML Components*, even if due to space limitations we present only the main results. During system architecture specification it was not always easy to follow this methodology faithfully, because our case study is quite different from that presented in [2]. In their book Cheesman and Daniels use as a case study a data management system, whose organization is quite easy and different from the development of a layer in a protocol stack. Therefore as an orthogonal objective, this paper attempts to be an integrative documentation for the people interested in *UML Components* applications in different contexts.

In Section 2 we introduce the main features of the case-study used and then, in Section 3, we briefly illustrate the main steps foreseen by the approach proposed by Cheesman and Daniels [2]. In Section 4 we describe how *UML Components* has been used to provide a meaningful software architecture for the system, also in order to revise the inherited code. Finally, we end the paper in Section 5 with some conclusions.

## 2. The JAIN MAP API case study

The aim of JAIN (Java APIs for Integrated Network) [6] initiative is to integrate wireline (PSTN), wireless (PLMN) and packet based (IP and ATM) networks, for allowing the simple creation and rapid development of a large variety of services. The novelty of JAIN basically resides in two fundamental aspects: first, the service portability, obtained by specifying standard java APIs over different protocols, and second, the network convergence, i.e. applications can be built only on the base of the service logic, irrespective of the underlying technologies (e.g. GSM stack or internet stack). In Figure 1 the JAIN level approach is depicted.

In this paper, we focus our attention on the JAIN MAP API. MAP (Mobile Application Part) is a protocol in the GSM stack, concerning mobility management and other services, such as the well-known SMS service. JAIN MAP API provides an abstraction level over the complicated MAP interface: in this way, a MAP service developer does not need to be aware of the specific MAP implementation features. At present, four MAP capabilities are specified for the JAIN MAP API: *transaction* (the SMS service), *session* (Unstructured Supplementary Service Data USSD), *position* (MAP location service, *information*. In this paper we only describe the implementation of the session capability, which allows information exchange between a mobile station and a GSM network application. The USSD is faster than the SMS service, because it is not a store-and-forward service. In particular, it is different from the SMS service for its session-oriented nature, which requires establishing a session each time a customer (a mobile station) or a network application approaches a USSD service. A USSD session can be one of two types: *network initiated* or *mobile initiated*: in the first case, the session is started by a network application (such as the residual credit notification after a call); in the second one, the session is started by a mobile station (such the request of notification of the residual credit). The session remains open during the entire messages exchange and is released at the end of the communication. Beyond the classes for the creation of the stack, for the USSD section capability, the JAIN MAP API specifications include:

- the classes of the interfaces between JAIN MAP application (for short JAIN Client) and JAIN MAP API layer: the `MessageSessionProvider` interface implemented by an object named MessageSessionProviderImpl and referred as JAIN Provider, and the `MessageSessionListener` interface implemented by the JAIN Client;
- the classes identifying the primitives that can be invoked both by the JAIN Client, when data flow from the application to the mobile user (`SessionOpenReqEvent`, `SessionDataReqEvent`, `SessionClo-seReqEvent`) and by the JAIN MAP API protocol, when data flow from the mobile user to the JAIN Client (`SessionOpenIndEvent`, `SessionDataIndEvent`).

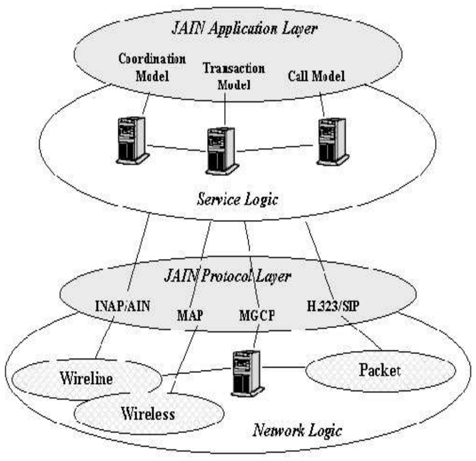


**Figure 1. JAIN levels approach**

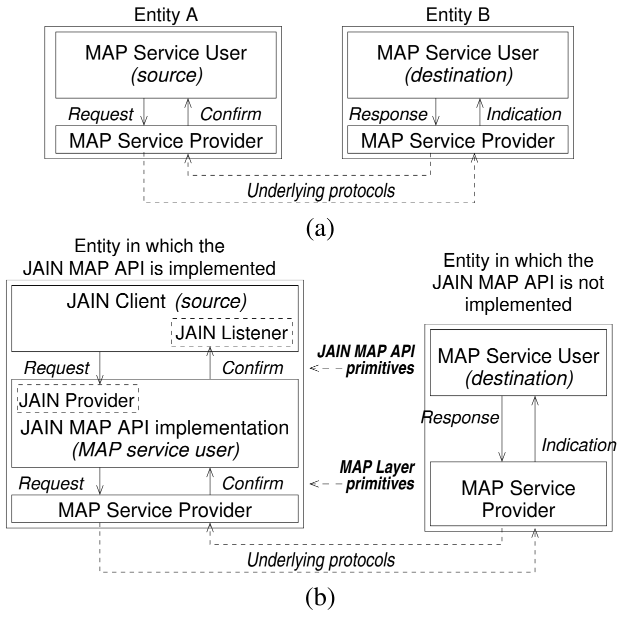


**Figure 2. MAP (a) and JAIN MAP (b) service models**

The JAIN Provider provides as many methods as the primitives going from the JAIN Client towards the network are; the JAIN Client implements the interface MessageSessionListener in order to receive events coming from the network. The JAIN Provider is responsible for the creation of a connection and for mapping between the JAIN primitives and the MAP primitives. The JAIN MAP API primitives reflect the model of the MAP primitives as depicted in Figure 2(a): when the JAIN Client starts a session it invokes a method in the JAIN Provider and it waits for confirmation on the MessageSessionListener (the JAIN Listener element in Figure 2(b)). Analogously, if the GSM user starts a session, the MAP layer invokes a method of the JAIN Provider in order to deliver the request of opening a new connection. In the previous implementations, the JAIN Providers used many objects in order to perform their tasks. Our interest was to split the task into logical capabilities and then to identify these in specific components. This goal allows also a simpler and faster implementation of the future JAIN MAP capabilities because starting from the component architecture it is sufficient to specify it in order to develop the new required functionality. For example, the opening of a connection between the JAIN Client and the MAP Layer is a feature to be implemented for each capability. Thus, it is natural to realize a component Connection Factory whose goal is to create an instance of a proper Connection object for the requests coming from the different JAIN Providers.

## 3. UML Components

In this section we briefly present the methodology proposed by Cheesman and Daniels [2], called the UML Component, which focuses on both the representation of the components, and the process development applicable for this purpose. In particular we report the main details of the specification process adopted in [2], divided into interacting workflows as suggested by RUP [4].

The first one is the **requirement workflow**, which produces the *business concept model* and *the use case model* [10]. The former is a conceptual model, which specifies the key concepts, their relations and a common vocabulary useful for avoiding misunderstanding and ambiguities. It is represented by a class model, but the classes involved, as well as their associations, are only conceptual and not related to the specification. Instead, the use case model represents the interaction of the system with the external users. It is represented by a Use Case Diagram, in which each Use Case is related to a different requirement or functionality. The system behavior and the main exceptions are represented for each Use Case in the associated scenario, following the textual structure of Cockburn's Use Cases [3].

The **specification workflow** is subdivided into three phases:

i identification of the components: starting from the requirements, an initial system architecture is produced;

ii interactions among the components, which identify the system's operations and responsibilities;

iii specification of the components, which specifies the operations and interfaces of the components themselves.

A *business type model*, represented by a class diagram, is used for modelling the entities of the business concept model actually perceived by the system. The involved classes are defined at the specification level, with no relation to a specific language. The notation used for the component interfaces differs from that defined in the standard UML, in which the interfaces represent implementation constructs typical of the OO languages and do not require attributes or associations. In the *UML Components*, an interface specification consists of: the type, the information model (the attributes, the interface roles in the association and their types), the specification of the operation (prototypes, pre- and post-conditions), and the invariants. All this information is grouped together in a package representing an interface specification, which can also import information from other packages. In *UML Components*, the concept of a component is also quite different than in the standard UML, because it is completely independent from the implementation. To differentiate the specification of a component from its implementation or the installed component, a new stereotype `<<comp spec>>` which has a set of interface types is introduced. Finally, the ways in which the components interact via the interfaces are described using collaboration or sequence diagrams.

Considering the **provisioning workflow**, it is aimed at ensuring that the released software is consistent with the given specification of the components. For this purpose the components can be implemented, bought, readapted or derived from the integration of existing software. Finally the **integration workflow** connects together the various components, the user interface, the application logic and the existing software to obtain an efficient application.

```
a) Open Session
CHARACTERISTIC INFORMATION
Goal in Context: The application requires the opening
            of a session
Scope: JAIN-MAP
Preconditions: true
Success End Condition: The connection is established
Failed End Condition: The connection cannot be
                established
Primary Actor: The application
Trigger: an opening request come in
-----------------------------------------------------
MAIN SUCCESS SCENARIO
1. The application requires opening a session with MAP
2. JAIN creates a MAP connection
3. JAIN returns the connection ID to the application
4. JAIN sends the data for the mobile to MAP, through
   the connection
5. JAIN receives the confirmation from MAP
6. JAIN receives data from MAP
7. JAIN sends data to the application
-----------------------------------------------------
EXTENSIONS
2. The parameters provided are not correct
  2.1 failure
4. JAIN sends a notification to MAP by the connection
  4.1 JAIN receives the confirmation
  4.2 JAIN sends the confirmation to the application
  4.3 stop
5. JAIN receives an abort from MAP
  5.1 JAIN sends the abort to the application

b) Process Open
CHARACTERISTIC INFORMATION
Goal in Context: MAP requires the opening of a session
Scope: JAIN-MAP
Preconditions: true
Success End Condition: The connection is established
Failed End Condition: The connection cannot be
                established
Primary Actor: MAP
Trigger: an opening request comes in
-----------------------------------------------------
MAIN SUCCESS SCENARIO
1. The MAP layer requires to open a session with the
   application
2. JAIN creates a MAP connection
3. JAIN sends the data to the application, by the
   connection
-----------------------------------------------------
EXTENSIONS
2. The parameters provided are not correct
  2.1 failure
```

**Figure 3. The textual description of the Use Cases Open Section and Process Open**

## 4. Application of the "UML Components" Methodology

In this section we present the application of the UML Component methodology (Section 3) to the above case study for deriving the JAIN MAP API architecture. Due to space limitations, in the following subsections we only describe the main details of each workflow, faithfully following the procedure described in [2].

### 4.1. Requirement Workflow

As a first step we analyzed the on-line documentation available for the considered application [9] consequently deriving a conceptual representation of the system, by highlighting the key concepts and their relations. Then, as suggested in [2], we represented them in a class model, the "Business Concept Model", which contains the following classes: **JAIN Client** (the user of the JAIN MAP API); **JAIN Factories** (factories required by the JAIN specifications to instantiate a protocol stack); **JAIN Provider** (it provides the standard JAIN MAP API specifications [9]); **Reference Tab** (it stores references and information concerning the opened connections); **Connections Factory** (factory of connection); **Connection Manager** (responsible for the communications between the JAIN Client and the MAP Layer); **MAP** (the inherited component which implements the MAP Layer). The second target of the methodology described in [2] is the description of the interactions of the JAIN MAP API system with the external users in a Use Case Diagram, the "Use Case Model". For this, we first identified the actors (JAIN Client and MAP Layer), and we then associated a different Use Case (UC in the following) with each JAIN primitive. Figures 5, 6 respectively detail the part of the Use Case Model relative to the JAIN Client and MAP Layer interaction, where the meaning of the collaboration diagrams associated to use cases will be explained in Section 4.2. In particular, the identified interactions are:

- *Actor JAIN Client* (Fig. 5): **JAIN creation** - a JAIN Client requires a JAIN stack instance; **Provider creation** - a JAIN Client requires a JAIN Provider instance; **Add Listener** - a JAIN Client supplies to a JAIN Provider a "MessageSessionListener" reference; **Remove Listener** - a JAIN Client requires to a JAIN Provider the deletion of a "MessageSessionListener"; **Open Session** - a JAIN Client requires the opening of a connection with the MAP Layer; **Data Session** - a JAIN Client requires the exchange of data with the MAP Layer; **Close connection** - a JAIN Client requires the closing of an open connection.
- *Actor MAP Layer* (Fig. 6): **Process open** - the MAP Layer requires the opening of a communication with a JAIN Client; **Process request** - the MAP layer requires to exchange data with a JAIN Client; **Abort** - the MAP layer requires the abort of an open connection as a consequence of network trouble.

  Finally we detailed the behavior of each primitive (Use Case) and the main system exceptions using the Cockburn textual description [3]. As an example, in Figure 3 we report the description of the two Use Cases, **Open Session** and **Process open**.

## 4.2. Specification Workflow

During the specification phase as first step, starting from the Business Concept Model described in Section 4.1, we deduced the "Business Type Model" characterizing the entities involved in our case study and specifying the types, the attributes and the associations for each of them. For instance, considering the "Connection Manager" we define a specific class (Figure 4(a)) and the attributes "sessionID: string" and "communNum: integer" which indicate the session identifier used by the JAIN Provider, and the communication reference number used by MAP Layer, respectively. We defined also the associations between the "Connection Manager" and the other classes and their navigability: for example, the "JAIN Client" can open sessions with different "Connection Managers", while "Connection Manager" is associated with only one "JAIN Client". Then we identified the `<<interface type>>` associated with the business type model classes, which in our case study are: "MessageSessionProvider", "IMAP-UpCallMgt", "IMAPListenerMgt" (Figure 4(a)). In particular considering the Use Case diagrams developed during the requirement workflow, we defined the correspondence between the Use Cases and the system interfaces by considering the textual description of each Use Case and distributing the responsibilities of operations execution between the established interfaces. For instance, considering the "Open Session" Use Case of Figure 3 the first three operations of the main success scenario are assigned to the "MessageSessionProvider" interfaces.

Finally, as shown in Figure 4(b), by using the information collected so far, we produced an initial system architecture that will be used as reference during the code restructuring and implementation of the system. Considering the case study, here we limit ourselves to the description of "Connection Manager" which is the most critical component for correct system behavior. This component must implement two different interfaces ("IConnectionMgt" and "IMapListener" see Figure 7) for providing services towards the JAIN Provider component and the MAP component respectively.

In particular, we refine the correspondence between Use Cases and the interfaces redistributing accurately the operation. For instance, considering the "Open Session" Use Case of Figure 3 and the architecture of Figure 4(b), the first operation is assigned to the "MessageSessionProvider" interface of the "JainProviderMgr" component, the second one to the "IConnectionMgt" interface of the "Connection Manager" component, the third one to the "MessageSessionListener" interface of the "JAIN Client" component.

Finally, using the textual description of the Use Cases and the established association between the operations and the interfaces, we specified the interaction between the architecture components by using Collaboration Diagrams. Specifically, we developed a different Collaboration Diagram for both each Main Success Scenario and all possible Extensions described in the Use Case textual descriptions. Considering our case study, for the Use Case "Open Session" Figure 5 shows the defined[1] Collaboration diagrams, while Figure 6 depicts those relative to the MAP Layer interaction. In particular Figures 8 and 9 respectively show in detail Collaboration Diagrams describing the component interaction in case of the Main success scenario of the Use Cases "Open Session" and "Process Open" of Figure 3. Thus, following the numbers assigned to the methods invocation in the Collaboration Diagrams it is possible to know how the system implements both the successful creation of a new connection requested by a JAIN Client (Fig. 8), and the start of a communication made by the MAP Layer (Fig. 9).

At the end of the specification workflow a complete description of the architecture of the system in terms of components and interfaces interaction is established. We used this structure for the system implementation, as will be described in the next section.

[1]Due to space limitations we only show the Collaboration of this Use Case

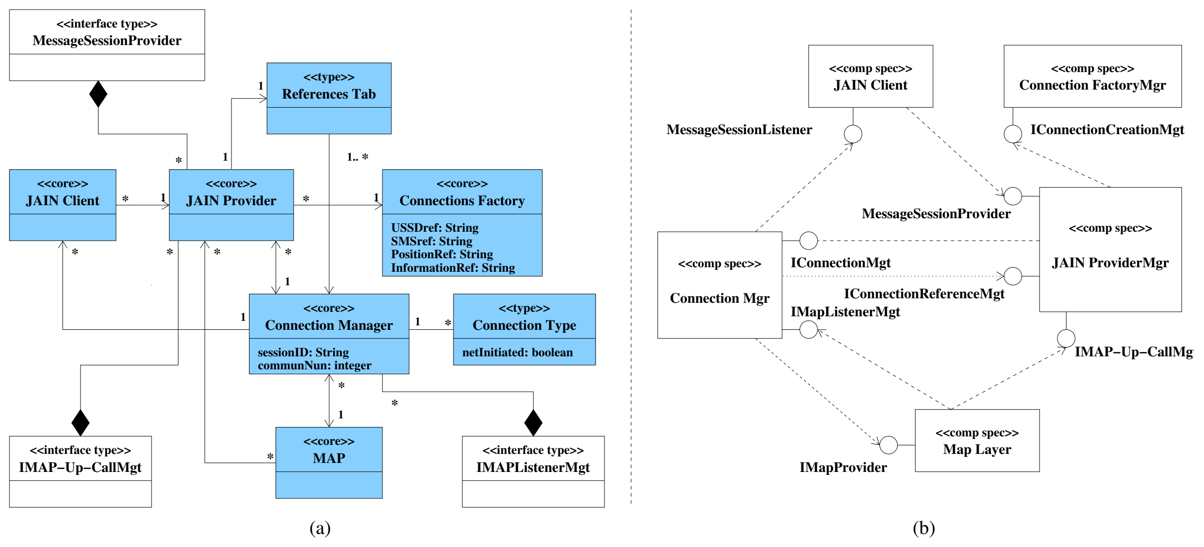
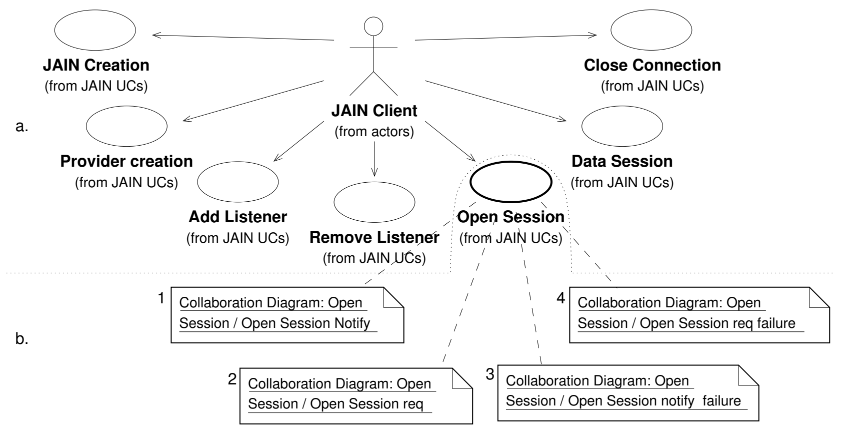
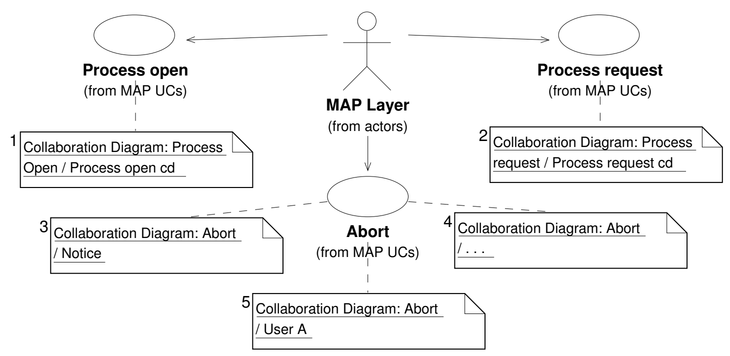


**Figure 4. The business type model (a) and the system component architecture (b)**



a.
JAIN Creation (from JAIN UCs)
JAIN Client (from actors)
Close Connection (from JAIN UCs)
Provider creation (from JAIN UCs)
Data Session (from JAIN UCs)
Add Listener (from JAIN UCs)
Remove Listener (from JAIN UCs)
Open Session (from JAIN UCs)
b.
1 Collaboration Diagram: Open Session / Open Session Notify
4 Collaboration Diagram: Open Session / Open Session req failure
2 Collaboration Diagram: Open Session / Open Session req
3 Collaboration Diagram: Open Session / Open Session notify failure

**Figure 5. The interaction between the JAIN Client and the system**



Process open (from MAP UCs)
MAP Layer (from actors)
Process request (from MAP UCs)
1 Collaboration Diagram: Process Open / Process open cd
2 Collaboration Diagram: Process request / Process request cd
3 Collaboration Diagram: Abort / Notice
Abort (from MAP UCs)
4 Collaboration Diagram: Abort / . . .
5 Collaboration Diagram: Abort / User A

**Figure 6. The interaction between the MAP Layer and the system**

## 4.3. Provisioning and Integration Workflows

The "canonical" form of a component based process certainly foresees, at the end of the specification phase, a provisioning phase in which suitable components are sought. However, in our case the purpose was to restructure, in a component based manner, the previously produced Object Oriented code, trying to reduce modifications and code rewriting. In the previous sections we described the process adopted for overcoming the nonexistence of a well-defined system architecture, here we concentrate the attention on how the system has been implemented. The main problem, transforming the OO code in a CB one, was the scarce use (no-use), in the inherited code, of the construct `interface` provided by the Java language, which consequently produced a web of direct "inter-class" references and a set of strongly coupled classes. Thus on the basis of the system architecture we modified the inherited code following the steps described below:

1. the inherited Java classes have been grouped according to the functionalities of the components foreseen in the architecture, i.e. for each class we analyze the tasks performed and we then identify the proper set (architectural component) in which to insert the class;
2. all the interfaces foreseen in the components architecture are codified using the construct `interface` provided by Java language;
3. for each class in a set we identified every "inter-set" dependence[2], and we removed them by using the suitable interfaces;
4. the sets are then handled in order to transform them into their logical components. We revised the code also developing, in each set, all the lacking functionalities. As consequence we extended the code of each set implementing the java interfaces foreseen by the logical components;
5. using Jar file, we packaged the elements forming a logical component, also enclosing an XML file that describes how the foreseen functionalities are provided. In this way we defined a raw component model (we will further clarify this point below);
6. for setting up a functioning system at run time, we defined a further XML file which reports the components to use and their relationships. The interpreter of this XML file will use the reflection mechanisms to instantiate the components from the Jar files defined in step 5.

In our component-based implementation we did not use any of the standard component model. However both to be compliant to the definition of "component", which requires that a component must be deployed independently from other components [7], and to obtain a component-based system implementation strictly conforms to the developed system software architecture, we needed to define a framework and a raw component model. In the JAIN MAP case study the independence among the components it has been obtained using, when necessary, the type defined by the interface construct instead of the type defined

```
public interface IConnectionMgt {
void setOpenConnection(IMapProvider provider,
     MessageSessionListener listener,
     IConnectionAbort connectionMgr,
     long sessionID, SessionOpenIndEvent event);
void setOpenConnection (IMapProvider provider,
     MessageSessionListener listener,
     IConnectionAbort connectionMgr, long sessionID);
void sendData(SessionDataReqEvent event);
void closeConnection(SessionCloseReqEvent even); }

public interface IMapListener {
void ManageMapEvent(MapOpenCnf event);
void ManageMapEvent(MapNoticeIndevent);
void ManageMapEvent(MapUAbortInd event);
void ManageMapEvent(MapPAbortInd event);
void ManageMapEvent(MapCloseInd event);
void ManageMapEvent(MapDelimiterInd event);
void ManageMapEvent(MapUSSDRequestCnf event);
void ManageMapEvent(MapUSSDNotifyCnfevent);
void ManageMapEvent(MapProcessUSSDRequestInd event); }
```

**Figure 7. Interfaces which must be implemented by the component "Connection Manager"**

by the class construct (see point 3 above) and removing all the occurrences of the "`new`" operator referring to classes that do not belong to the same set (component), and then implementing a raw naming service. In fact, we developed a simple framework which requires that the specification of the correspondences between the architectural components and the Jar files, containing the real implementation of a component, is reported in a XML file. A class that needs a particular service can obtain a reference to an instance of a component providing the specified service, invoking particular methods on the objects constituting the framework. This objects can read and instantiate, using also the Java reflection mechanisms, classes inside the Jar file (component). Therefore using this "naming" mechanism we can easily improve or extend the system implementation, modifying or adding new correspondences in the XML file. At the same time those mechanisms facilitate the black-box reuse of the defined components. This latter feature is particularly important in order to implement the other functionalities foreseen by the JAIN MAP API specification.

## 5. Conclusions

In this paper we presented our experience in deriving a suitable component architecture for the JAIN MAP API case study, by following the UML Components methodology described in [2]. In particular, in order to reduce the effort and the time needed for the system implementation, we defined a well-structured component architecture model used both for restructuring existing "inherited" monolithic code in a component-based way, and for eas-

[2]to say invocations of constructors or references to instances of real classes belonging to different sets

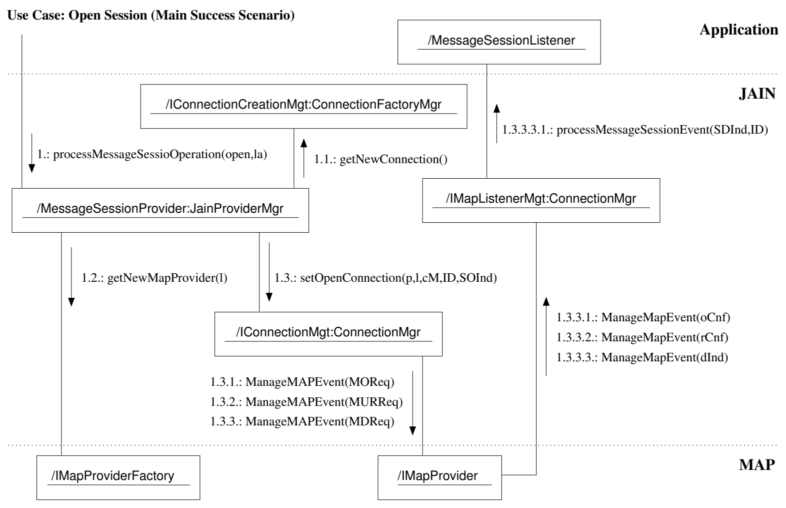

Use Case: Open Session (Main Success Scenario)
/MessageSessionListener
Application
JAIN
/IConnectionCreationMgt:ConnectionFactoryMgr
1.3.3.1.: processMessageSessionEvent(SDInd,ID)
1.: processMessageSessioOperation(open,la)
1.1.: getNewConnection()
/IMapListenerMgt:ConnectionMgr
/MessageSessionProvider:JainProviderMgr
1.2.: getNewMapProvider(l)
1.3.: setOpenConnection(p,l,cM,ID,SOInd)
1.3.3.1.: ManageMapEvent(oCnf)
1.3.3.2.: ManageMapEvent(rCnf)
1.3.3.3.: ManageMapEvent(dInd)
/IConnectionMgt:ConnectionMgr
1.3.1.: ManageMAPEvent(MOReq)
1.3.2.: ManageMAPEvent(MURReq)
1.3.3.: ManageMAPEvent(MDReq)
/IMapProviderFactory
/IMapProvider
MAP

**Figure 8. Collaboration Diagram specified for the Use Case Open Session**

ily extending it on the basis of the obtained architecture. In this context, UML Components methodology has been a valid reference for identifying the system architecture, isolating the necessary components, and establishing the relationships between them. With our refactoring, the software obtained is more flexible, testable and manageable, and it reduces the effort needed for further implementation of new JAIN capabilities.

In future studies we intend to extend the refactoring to the code available for the other JAIN MAP capabilities and to complete the implementation of the whole system. Moreover, we will test the obtained software for both discovering the problems of components integration, and verifying the correct system behavior. For this, the presence of a well structured and described architecture will simplify both the validation of the system against the JAIN specifications, and the test case derivation for testing the components in isolation, by allowing the use of available tools for automatic test generation and execution.

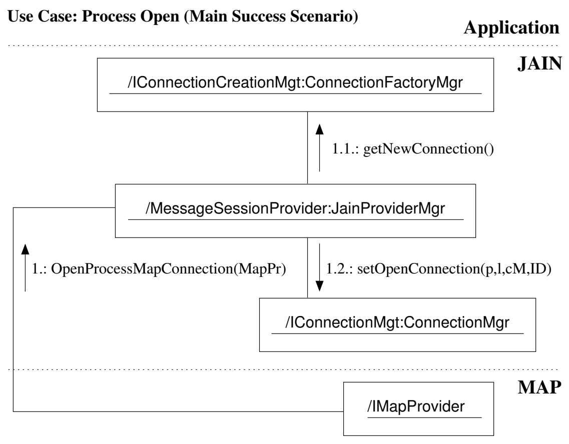

Use Case: Process Open (Main Success Scenario)
Application
JAIN
/IConnectionCreationMgt:ConnectionFactoryMgr
1.1.: getNewConnection()
/MessageSessionProvider:JainProviderMgr
1.: OpenProcessMapConnection(MapPr)
1.2.: setOpenConnection(p,l,cM,ID)
/IConnectionMgt:ConnectionMgr
MAP
/IMapProvider

**Figure 9. Collaboration Diagram specified for the Use Case Process Open**

## 6. References

1. F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite approach to planning and deriving test suites in UML projects. In *Proceedings of ≪UML≫ 2002, LNCS 2460*, pages 383–397, Sept. 30th-Oct. 4th 2002.
2. J. Cheesman and J. Daniels. "UML Components: A Simple Process for Specifying Component-Based Software". Addison-Wesley, 2000.
3. A. Cockburn. "Writing Effective Use Cases". Addison-Wesley, 2001.
4. P. Kruchten. "The Rational Unified Process - An Introduction". Addison-Wesley, 1999.
5. SEI. How do you define software architecture? On-line at: http://www.sei.cmu.edu/architecture/definitions.html.
6. H. Sneed. Recycling software components extracted from legacy programs. In *Proc. of 4th Int. Workshop on Principles of Software Evolution (ICSE 2001)*, pages 43–51.
7. C. Szyperski. "Component Software - Beyond Object-Oriented Programming". Addison-Wesley, 2002.
8. H. Washizaki and Y. Fukazawa. Automated extract component refactoring. In *Proc. of XP2003 conference (LNCS 2675)*, pages 328–330, 2003.
9. The JAIN MAP API specifications. Available at: http://java.sun.com/products/jain/api_specs.htm.
10. UML Documentation version 1.5 Web Site. On-line at: http://www.omg.org/technology/documents/formal/
11. CORBA Component Model specifications. Available at: http://www.omg.org/technology/documents/formal/components.htm.
12. Enterprise Java Bean Technology. Available at: http://java.sun.com/products/ejb/.
13. .Net resources available at: http://www.microsoft.com/net/.