Developing Object Oriented Designs from Component and Connector Architectures

Hyoung-iel Park[1], Sungwon Kang[2], Yoonsuk Choi[2] and Danhyung Lee[2]
[1]*Tmax Soft, 18F Glass Tower, 946-1*
Daechi-Dong,Kangnam-Gu, Seoul, Korea
hipark@icu.ac.kr
[2]*Information and Communications University*
Dogok-Dong, Kangnam-Gu, Seoul, Korea
{kangsw, yschoi, danlee}@icu.ac.kr

Abstract

In this paper, a systematic approach of developing detail OO designs from Component and Connector Architectures (CCAs) is proposed. In this approach, an intermediate model between the architecture model and the detail design model specified with class diagrams or sequence diagrams is introduced to narrow the wide gap between the two abstraction levels. Once a CCA is designed, candidate classes and their relationships are identified per each architectural element. In order to show the efficacy of this approach, we apply it to an industry software development project and verify that quality attributes supported by the CCA are equally maintained by the detail design.

1. Introduction

Some practitioners use Object-Oriented Design (OOD)[1] itself as the architecture description of software systems instead of using the emerging architectural concepts and notations. Although some OOD concepts can be used to address architectural design issues, and doing so is popular among software developers, there are significant differences in capabilities and benefits between them. The level of abstraction that the OOD description provides does not cover all the aspects required for architectural design.

In this paper, we propose a systematic method of developing OOD from an architectural design that is consistent with the Component and Connector Architecture (CCA) [1]. In this method, Intermediate Model (IM) is introduced between the architecture model and OOD to narrow the gap between the two widely different abstraction levels. We call this the Intermediate Model Introduction (IMI) approach. We applied the IMI approach to an industry project to demonstrate the method and to show its efficacy.

The rest of this paper is organized as follows. In Section 2, we introduce an industry project Metadirectory system development and describe what kinds of challenges we met during the initial attempt of architecture design. Section3 discusses related work and explains what candidate approaches exist for overcoming those problems but why they are inadequate as they are. Section 4 describes the IMI approach in detail. In Section 5, we apply it to the Metadirectory project and show its efficacy. Lastly, Section 6 is the conclusion.

2. The Metadirectory Project

The Metadirectory system integrates identity data from heterogeneous repository systems while keeping consistency between duplicated data for single resource. Through the system an application or a user that wants to access the identity data can see the unified view of the data distributed over various repositories.

A team of five software engineers developed the Metadirectory system for one year period. Every member participated in all phases of development from requirements specification through designing to implementing. For the development process, the Unified Process (UP) was adopted and tailored to the team size and the project period and also improved to embrace architectural design.

[1] In this paper, we use the terms Object-Oriented Design and Object-Oriented Detail Design interchangeably.

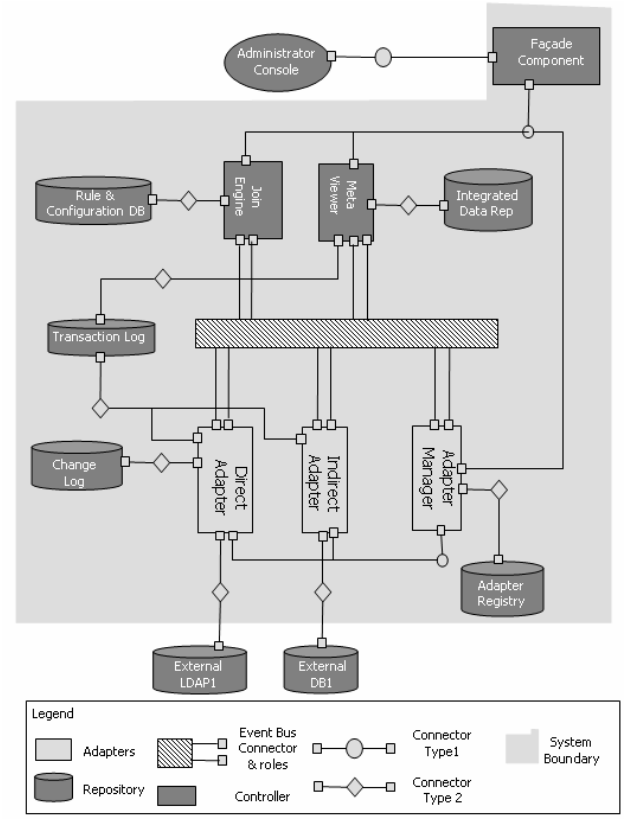

Figure 1. CCA Description of the Metadirectory System

For architectural design, ACME[2] [2] was selected as an architecture building tool which helps build a cartoon view of the architecture and add constraints on the structure and data flow. ACME allowed us to draw CCA in which components and connectors are described in Figure 1.

The team members and the clients considered this architecture as the most important artifact to be obtained in the inception and elaboration phase in that it can show the developers and the stake holders what the product will be like and how it will function from the whole system's viewpoint. Thus the team members hoped that everything the CCA implies to be maintained to other subsequent artifacts without distortion as the development process goes. In every development step, the developers referred to the CCA lest they got lost. But most of the members experienced significant problems when trying to design OO design models from the CCA.

We encountered several problems during the project as follows:

(P1) CCA is not harmonized with UP. In UP, most tasks have their input and output. Although we modified UP by putting the architectural design activity ahead of the OOD activities in order to make the ACME-based architecture artifacts merged into the process, it was not effective because of the different properties between the architecture based artifacts and OOD based artifacts.. The different properties were apparent because artifacts except that of CCA linked to each other in OO process defined in UP.

(P2) Consistent implementation of CCA was difficult. Since how to realize the elements of CCA was not explicitly defined, implementations by different members did not conform to each other. For instance, a port listening for message incoming was implemented with an independent thread but another port that is supposed to be implemented in the same way was coded using the Observer pattern[3].

(P3) There is no concrete process for implementing architectural styles. Another thing to consider was the difference between of the concept of the design pattern and the concept of architectural style that are to be used in OOD like class diagram and the CCA, respectively. As the techniques of software architecture construction matures, they would need to more and more utilize architecture patterns such as "pipe and filter," "blackboard," "publish and subscribe," and so on [3]. Since CCA is the basis for later detail design process, these kinds of styles also needed to be embodied into the subsequent OOD. But the concrete way of transition from the architectural style to the OOD was not clear. Some members of the development team used one design pattern to realize an architectural style while others used another design pattern to do the same style.

(P4) Requirements realization is not ensured through the overall design process. CCA reflects functional requirements as well as non-functional requirements. It has components, connectors, constraints and styles based on the requirements. Although the CCA was frozen by agreeing with the client, it was not explicitly determined how components or connectors could be implemented. Thus how to maintain the functional and non-functional requirements in the OO design was not clear.

3. Related Work

There are researches attempting to connect the architectural design world and the OO design world. Most of such researches tend to be lopsided to one world trying to represent architectural concepts with the OOD tools or vice versa but are not sufficient for overcoming the problems that we described in Section 2. In the following, such works are examined and criticized.

[2] ACME is an architecture interchange language, which is intended to support mapping of architectural specification from one ADL to another.

[3] It is an OO design pattern.

Mapping ADL to UML

Various Architecture Description Languages (ADL) have been proposed such as Wright[4], ACME and so on. Regardless of the kinds of ADL, the types of information on which ADL focuses are common characteristics of an application domain, a style of system composition, or a specific set of properties. To implement these common characteristics, attempts to map ADL to UML detail design have been made [5]. Even though UML 2.0 provides some useful notations for architectural elements such as connector and port [6], it does not give any directions on how to transform architectural notations into the more detail ones such as class diagram.

Component based infrastructure

Component-based infrastructures such as COM, CORBA[8], Enterprise Java Beans (EJB), and Web Services provide sophisticated services such as naming, transaction, and distribution for component-based applications [9]. These infrastructures have provided uniform, standard, high-level interfaces to the application developers and integrators, so that applications can be easily composed, reused, ported, and made to interoperate They also supply a set of common services to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications Many software engineers have tried describe overall architectural structure with these interfaces in order to hide complex protocols.

However, component infrastructures such as CORBA or COM don't include mechanism for explicitly describing software architecture. They are usually directly dependent upon the characteristics of the involved middleware and provide little, if any, guidance as to how a similar outcome can be achieved with a different set of middleware platforms [10]. Even though Web Services provides architecture level abstraction by allowing an architect to design overall structural design, at the detail design level it is very dependent on so called Web Services platform with several standards such as SOAP, WSDL or BPEL.

Archjava

In order to integrate software architecture specification smoothly into Java implementation code, Archjava, which is a sort of java extension, was invented [11]. Archjava adds new language constructs to support components, connectors, and ports so that programmers can descry software architecture. The code of Archjava consists of abstract classes for each architectural element. These abstract components and ports allow an architect to specify and type-check the Archjava architecture before beginning program implementation.

However, from the practical point of view, if we design a compiler component, for example, an important consideration is how to design the component which is composed of classes and their relationships. In Archjava, there are no such constraints for the programmer to refer to the class level design. If a programmer wants to design a native class and make inheritance relationships between the classes, he/she must consider class level design options regardless of abstract entities for architectural elements. Even though Archjava transfers the architectural concepts to the java language with its extension, it fails to bring them into the OO world.

4. The Intermediate Model Introduction (IMI) Approach

Before getting into the IMI approach, we need to note that there is no direct connection between OO design pattern and architectural style [12]. It is because there are no primitive types or conventions in OO language specifically devised for architectural elements like components or connectors [13].

With this fact, it is possible for the designer to separate the structure of architecture that describes how components are connected with connectors and OO design that define what the classes and their relationships such as associations, aggregations, compositions and inheritances are like.

Table 1. Input and output artifacts of CCA and OOD

	Input	Output
CCA	Functional requirements, Non-functional requirements	Components, Connectors, Ports, Constraints, Architectural Patterns, Refined Quality attributes
OOD (Class diagram)	Objects, Relations, Functional requirements, Non-functional requirements	Objects (classes), Static Relations (association, aggregation, composition, inheritance)

This separation can be inferred from the artifacts as in Table 1. The artifacts of an architectural design and sequence or class design are inherently different with each other because the former considers the overall structure and system properties and the latter focuses on detail design which can is implemented by OO Language.

In building an OO design such as developing class diagrams from use-cases, it is often required to generalize the use-cases, extracting objects or defining the rough relationship between the objects to keep consistency from abstract model to more concrete one. Similar to this, through introducing an IM between architecture and detail design, developers can accomplish the transition smoothly because it can ameliorate the gap between the two artifacts. But strictly speaking, it is a little different from the modeling from use-cases in that the objects residing in a CCA are tightly categorized by the CCA elements and refined under the architectural constraints. Thus a more restrictive way is required.

The IMI approach is composed of the three major steps as shown in Figure 2.

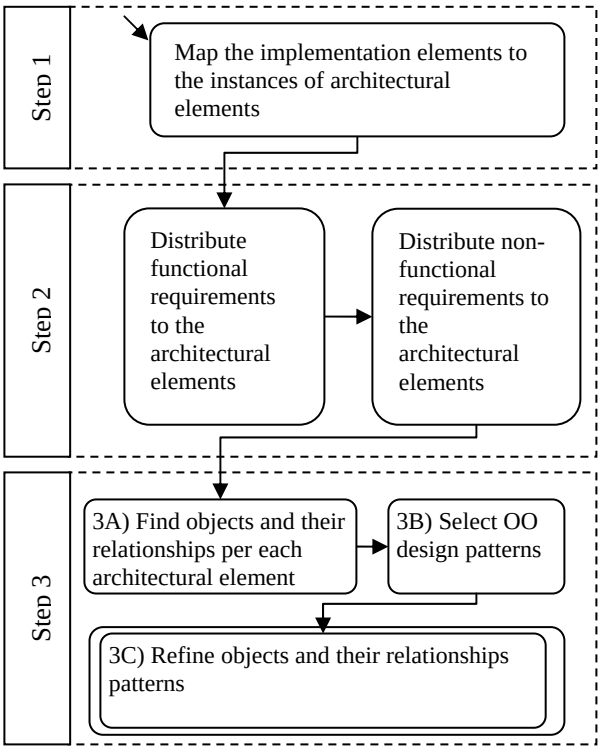

Figure 2. The IMI Process

In Step 1, the major architectural elements are component, connector, ports and constraints[4] in CCA and they are assumed to be composed of objects and their relationships. Mapping the CCA to implementation elements is a key activity in this step. In this activity, how to realize these elements is considered based on the architectural elements identified.

[4] Constraints of the software architecture are about type constraints or object binding rules, which don't affect the overall structure or design. Thus, in this approach, they are not considered to be a modeling target.

In Step 2, once the implementation elements for each architectural element are decided, the functional and non-function requirements are distributed among architectural elements. To distribute non-functional requirements to architectural elements is not as simple as functional requirements because quality attributes that represent non-functional requirements can be supported both implicitly and explicitly by the architectural elements. When an architectural element is used in order to support a specific quality attribute, the attribute can be assigned to the elements. .

In Step 3, there are three sub-steps. For each functional requirement, objects that are needed to satisfy the requirement has to be identified in 3A.

And the second sub-step 3B, OO design patterns can be considered. This step is required because it can be very hard to make OO design without applying design patterns. Considering design pattern already proven to be useful to the specific problem before you jump into making a solution to the problem helps developers make better design than later considering. Considering the quality attributes and requirements, the patterns which can be applied to the component can be listed up (Table 2) in the exemplified project. A design pattern gives us a generic solution that solves a family of recurring problems in a certain context [14].

Lastly, in Step 3C, the objects identified and the relationships between them can be rearranged based on the selected design patterns.

Table 2. Design patterns and their purposes

Design pattern	Purpose
Command	By wrapping a method in an object, it can be passed to other methods or objects as a parameter.
Template	It is defined in the base class and cannot be changed.
Façade	It is used to expose simple interface which handles confusing collection of classes and interactions.
Observer	It is often used for the specific case of changes based on other object's change of state, but is also the basis of event management.
Adapter	It takes a type and produces an interface to some other types.

5. Applying the IMI approach to the Metadirectory Project

In this section, the IMI approach explained in the previous section is applied to the Metadirectory system

introduced in Section 2. In Section 5.1, the three steps of the IMI approach is applied one by one to one of the architecture element. In Section 5.2, the final detail design is exhibited. Finally in Section 5.3, it is verified that the architectural structure and the quality attributes supported by it are preserved in the final OO design.

5.1 Applying the IMI approach

Among many subsystems of the Metadirectory system, the Adapter Manager (AM) component and its connectors and ports (Figure 3) are selected to demonstrate IMI approach because it utilizes two protocols for ports and has complex functionality which may easily cause inconsistency in realization.

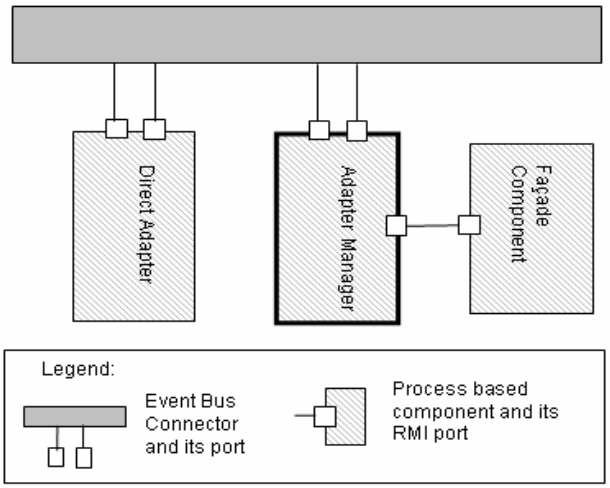

Figure 3. Adapter Manager Component and its connectors and ports

Step 1) Three major architectural elements are considered (Table 3), which affect the detail design in terms of identifying key objects. Two kinds of middleware were used for connectors. Java Message Services (JMS) was adopted for the event bus and RMI (Remote Method Invocation) embedded in Java Development Kit (JDK) was used for the basic client-server connector.

Step 2) AM is responsible for handling Direct Adapters[5](DAs) and relaying user commands to a DA. According to the Software Requirement Specification of the project, DAs have to be added to the system and removed simply by issuing handling command without doing additional actions like starting adapter process manually or stop the whole system. By satisfying this requirement, one of important quality attribute, extensibility, will be supported.

Table 3. Architectural elements and their implementation elements

Architectural Elements	Instances of Architectural Elements	Implementation Elements
Component	AM Component	Java Process
Connector	Event Bus Connector	JMS
	RMI Connector	RMI
Port	Event Bus Receiver Port	JMS Listener
	Event Bus Sender Port	JMS Sender
	RMI Receiver Port	RMI Stub
	RMI Sender Port	RMI Proxy

The user can control DAs through AM. Once “stop” or “start” commands are issued by the user, they are first sent to the AM with a destination tag and resend the command to the proper DA after verifying the command. Besides relaying commands, the component has to monitor DAs to assure availability of the system by resizing the queue size of the event bus. The detailed requirements for this component can be summarized as in Table 4.

Table 4. Functional and non-functional requirements for AM

Functional requirements	Non-functional requirements
1. To remove DAs 2. To add DAs 3. To verify commands from the façade component 4. To relay commands to the target DAs 5. To check loads of all DAs 6. To control the queue size of event bus according to the load	1. Extensibility: A. The system can allow a user to remove or add DA easily (without restarting the system or changing configurations) B. A new functionality can be added to this component easily 2. Usability: A command issued can be canceled by undo command by one level

Step 3A) After mapping the implementation elements to the architectural elements and distributing functional and non-functional requirement to each element, major objects are identified and then several candidate design

[5] Direct Adapter is a component that allows an external system to join the Metadirectory system.

patterns are considered and the most proper one is chosen by considering both brief objects list and non-functional specifications. Lastly, based on the selected design pattern and its implied structure, supplementary objects can be added and all objects are arranged and are to have relationship each other.

Table 5 shows required objects of each functional requirement. Extracting objects from requirement is the first step OOD, where the identified objects are very rough and later can be split into smaller objects or need additional objects for structural completeness.

Table 5. Objects needed for functional requirements

Requirements	Objects needed
To remove DAs To add DAs	DA_handler
To verify commands from the façade component	Command_verifier
To relay commands to the target DAs	Command_sender
To check loads of all DAs	Load_checker
To control the queue size of event bus according to the load	Queuesize_ controller

Each object extracted has interactions with other objects invoking methods of other object or sharing variables with others. Because these interactions are embodied as class relationships in detail design, it is important to characterize relationships between these objects (Figure 4).

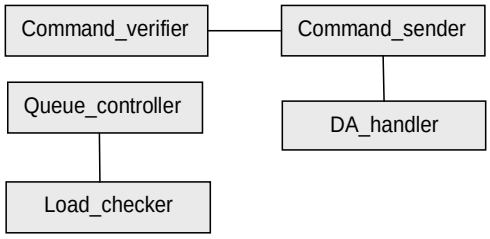

Figure 4. Overall relationships between major objects

Step 3B) Once the overall structure with objects and their relationships is determined, the design patterns can be selected to fulfill the non-functional requirements and to refine the overall structure of the system. By using the factory pattern, the adapter proxy can be easily created and destroyed in AM. Although the major part supporting the "Extensibility" is to use publish-and-subscribe architectural style, using this pattern strongly support the attributes also. Template pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. And finally through command pattern, user commands can be realized as an object with which the command can be canceled or reissued. The list of selected patterns is in Table 6.

Table 6. Design patterns based on the non-functional requirements (* and ** indicate the kinds of 'extensibility' attributes of Table 4)

Requirements	Design pattern
Extensibility A*	Factory
Extensibility B**	Factory, Adapter, Template, Singleton
Usability	Command

Step 3C) After applying design patterns to the model, several new objects (dark boxes) required for the patterns are added as in Figure 5. Factory and adapter objects are such ones. Relationships are refined according to the patterns. At this stage, however, the apparent feature of the pattern does not appear because IMI approach is confined to the relationships not using inheritances or compositions. But it is very meaningful to find out key object required for patterns or functional and non-functional requirements in terms of making the later designs consistent under the selected design pattern. Figure 5 and the patterns of Table 6 are the output artifacts of the IMI approach.

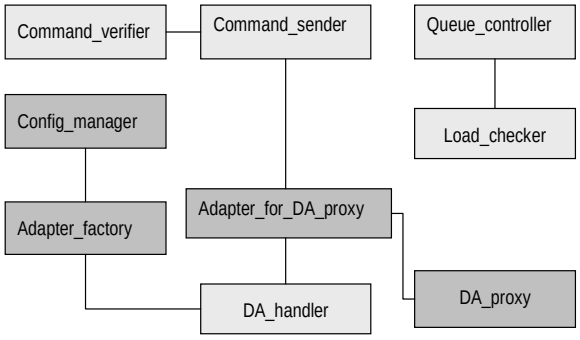

Figure 5. IM for the AM Component

5.2 The Final OO design

In the previous section, only the IM for the AM component was demonstrated. IMs for other architectural elements like connector and port can be also built in a similar way. Based on theses IMs the class diagram of the AM component containing ports is described in Figure 6.

To show how the component is embracing the port for connector, the part for commands processing and load

controlling is omitted and only adapter proxy creation which is connected to the port is emphasized. In this diagram, unlike CCA, it is hard to distinguish which part is for the component and which part is for the ports unless the two kinds of classes are differentiated with oblique line filled box.

Then what is the difference between the class diagram directly from the CCA and the one from the IM? The most important difference is that the classes obtained by the IMI approach are managed in a uniform way. At the same time, the structure and quality attributed can be maintained throughout the development process.

5.3 Verification

It is, of course, possible to develop a system without an IM. But from the aspects of consistency, reverse engineering, and maintenance, development may be very difficult because there is no concrete process for converting CCA to OOD. In this section, the effectiveness of the intermediate model introduction will be demonstrated from two viewpoints. One is by answering the question "Is the architectural structure maintained from CCA to OOD?" and the other by answering the question "Are the quality attributes preserved in the process of applying IMI approach?"

Architectural structures

The IMI approach is performed per each architecture element. The first step is to map implementation elements to the CCA elements. The remaining part of the process is done for each implementation element without considering the other elements. It can be said that the overall structure defined by the architecture is maintained to the OOD. Furthermore, with the class diagram the designer can easily distinguish which part is a port and which part is a component from the IM.

Quality attributes

Many researchers say that architecture is important because it can contain quality attributes required for a system. But often quality attributes are realized by making subsystems or objects support the quality attributes. Table 7 describes how the quality attributes are preserved through the modeling. The second step of the modeling process assigns non-functional requirements to each implementation element, during which each quality attribute is explicitly assigned to a certain implementation element. All "mapped implement elements" of Table 3 has its responsibility for each quality attribute that is expected to be maintained from the SRS to the code.

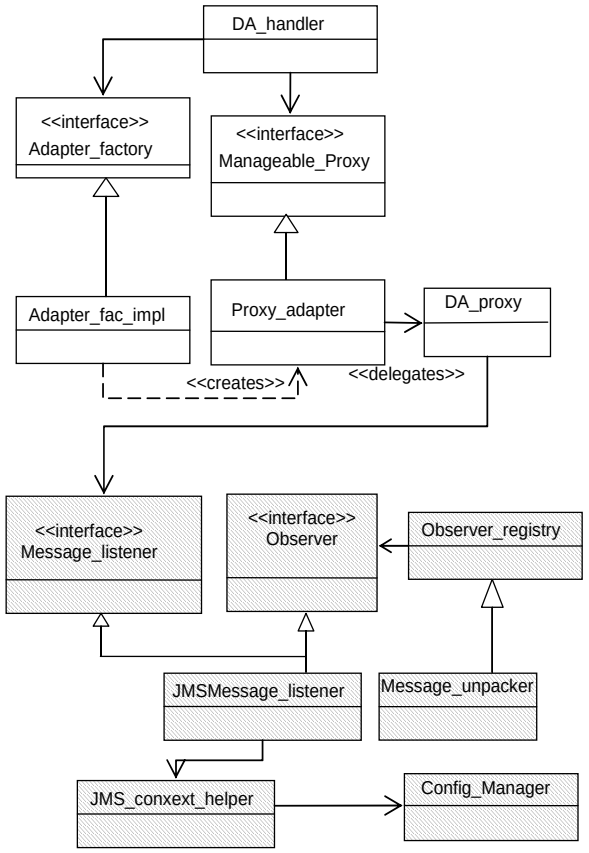

Figure 6. A simplified final Class diagram for the AM component with ports

Besides the preservation of the attributes, the modeling makes it possible to develop design artifacts in a managed way focusing on quality attributes. For instance, the extensibility is strongly supported by adopting the adapter pattern and the command pattern which allow the developer to add additional code without changing the existing code.

Table 7. Quality attributes supported in IM

Architectural Elements	Implementation Elements	Supported Quality Attributes
Component	Adapter Manager	Extensibility, Usability
Port	JMS sender JMS receiver	Integrity
Connector	JMS RMI	Usability

In this way, the architectural structure and quality attributes are maintained from the CCA to OOD in the IMI approach. Now let's see if the problems described in

Section 2 that we encountered during the Metadirectory project can be solved.

The first problem P1 is solved because IM takes CCA as an input and produces objects and their relationships which can be used as an input to the class diagram as needed for the UP. For P2, since objects list and design patterns define how many objects are required and what their relationships are to be like in order to realize the architecture, the implementation level has been defined. Without this implementation baseline, the detail designs can diverge. For example, where one programmer introduces one class, another programmer can introduce a set of classes per a component. The third problem P3 is solved by the fact that the overall architectural structure is maintained to the detail design through intermediate design because the architectural structure contains architectural style. For P4, it is also solved by ensuring backward traceability of non-functional attributes with the IM.

6. Conclusion

Mapping implementation entities such as a middleware or a process to the architectural elements is the baseline of the IMI approach. This step ensures that the architectural structure is maintained when the architecture is realized with the OO way. The IMI approach adds two major activities to this base step. The first one is to extract the major objects required and determine the relationships between them, both of which are the key inputs for the OOD. And the second one is about design patterns. Based on the extracted objects and the non-functional requirements, design patterns are selected and applied to the model. This process ensures that the quality attributes are still preserved in the OOD. Besides, by making a list of objects and determining their relationships, the OOD can be developed in a consistent way because the level of detail or the way of realization is confined.

Selecting design patterns and their invariants is not a simple task. This paper used a heuristic approach for selecting design patterns that can help developers adopt and refine. Using a design pattern may cause its invariants to be adopted in certain situations and adopting several similar invariants can require another higher level design pattern to be used. Formalizing the process of selecting design patterns remains a future work.

7. References

[1] D. Garlan and M. Shaw, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[2] B. Schmerl and D. Garlan, “AcmeStudio: Supporting Style-Centered Architecture Development," In *Proc*. of the 26th International Conference on Software Engineering, Edinburgh, Scotland, May, 2004, pp. 23-28.

[3] D. Garlan, R. Allen, and J. Ockerbloom, “Exploiting Style in Architectural Design Environments,” In *Proc.* of SIGSOFT '94: Foundations of Software Eng., Dec. 1994, pp. 175-188.

[4] N. L. Kerth and W. Cunningham, “Using patterns to improve our architectural vision,” Software, IEEE, vol. 14, issue 1, Jan.-Feb. 1997, pp. 53-59.

[5] N. Medvidovic and R. N. Taylor, “A classification and comparison framework for software architecture description languages,” *IEEE Transactions on Software Engineering*, vol. 26, issue 1, Jan. 2000, pp. 70-93.

[6] M. Bjorkander and Cris Kobryn, “Architecting Systems with UML 2.0,” IEEE Software, 2003.

[7] S. Cheng and D. Garlan, “Mapping Architectural Concepts to UML-RT,” Int’l Conf on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Monte Carlo Resort, Las Vegas, Nevada, USA, Jun. 2001.

[8] Object Management Group, CORBA 3.0 New Components Chapters, ptc/99-10-04, Oct. 1999.

[9] Microsoft, The global XML Web services architecture GXA; available at http://msdn.microsoft.com/webservices/understanding/gxa/default.aspx

[10] N. Medvidovic, “On the Role of Middleware in Architecture-Based Software Development”, ACM 1-58113-556-4/02/0700, 2002.

[11] J. Aldrich, C. Chambers, and D. Notkin, “ArchJava: Connecting Software Architecture to Implementation,” In *Proc.* of ICSE 2002, May 2002.

[12] A. H., Eden and R. Kazman, “Architecture, design, implementation,” In *Proc.* of the 25th Int’l Conference on Software Engineering, May 2003, pp. 149-159.

[13] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin, “Language Support for Connector Abstractions,” In *Proc.* of the European Conference on Object-Oriented Programming (*ECOOP '03*), Jul. 2003.

[14] E. Gamma, R. Helm, R.E. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1994.