Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Today, our daily life depends largely on computer software with high quality and full functionality. To satisfy functional requirements and achieve quality attributes, software development is becoming quite complicated no matter whether the software runs on a desktop computer or is distributed in a network [Gorton, 2011] [Bass *et al.*, 2003] [Zhu, 2005]. Thus, software engineers frequently face the challenges of designing, implementing, and maintaining software. In order to meet the challenges, a number of techniques have recently emerged to address the problems of software development process. For example, software architecture has received a lot of attention in software engineering. The main objective of software architecture is to provide frameworks and methods for analyzing and resolving the problems of software. Bass *et al.* [2003] gave definitions of the notion: architecture is a combination of components and connectors. Components are a set of independent elements that have well-defined interfaces that are replaceable within environment, but they do not have observable states from outside [Szyperski *et al.*, 2002] [OMG, 2010]. Connectors are concerned with the interaction among components. They are typically regarded as software elements for delivering data and control between components in software system [Bass *et al.*, 2003]. The techniques about the interaction are highly developed, but there are many problems. For example, in the case of the remote method invocations (RMI) mechanism, its tight coupling is frequently considered as a major drawback [Tanenbaum *et al.*, 2007].

Thus, interaction of components is a significant and necessary part of software architecture, particularly for software system in network. In order to design and implement interactions among components, current research introduces connectors to this area of

knowledge. However, today's definitions of connectors vary according to the level of understanding for connectors.

From the view of ***relationship*** *among components*, Shown *et al.* [1996] gave the definition of connectors: connectors were the implementation of the relationship of components. Each connector has a protocol specification that is used to define its features.

From the ***transport*** *point of view*, other researchers [McGovern *et al.*, 2003] described that connectors were software modules used to provide a channel to link components together. Connectors may merely utilize network protocols or adopt complex transport system for exchanging data each other.

From the view of system ***integration***, connectors are used to integrate existing systems or data sources for sharing processes and data [Sharma *et al.*, 2001]. Despite the fact that there are various definitions and different level of understanding for connectors, most of them are of the same opinion—connectors provide software system with one or more of four services which are identified as communication, coordination, conversion, and facilitation [Mehta *et al.*, 2000].

1.1 Motivation

In software development, there are some traditional connectors, such as procedure call (method call), global variable, association class, shared memory, etc. These connectors are frequently used in local application (or called desktop application) where the connectors

and components run on a same computer sharing the CPU and memory resource. In practice, some of them may combine to build a complex connector [Mehta *et al.*, 2000]. The diagram (Figure 1.1) below describes a complex connector which was shown in Mehta *et al.*'s connector [2000]. The shared Memory Connector consists of Procedure Call and Data Access connectors.

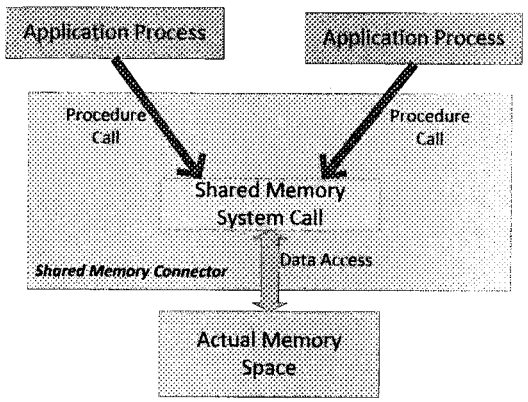


Figure 1.1: Shared memory connector [Mehta et al., 2000]

These typical connectors are widely used in desktop applications. Some of them are recognized as fundamental design methods (for example, association class of OOP); some of them have been absorbed by basic programming approaches (for example, procedure call) provided by most of development tools. The typical connectors and their combinations are good at linking components together in local environment. As a result of the strong support of development tools, developers may neglect them in design and implementation phases.

However, in the recent years, significant changes in computer and network are occurring, in particular the large increases in performance of hardware, such as computer chip, mobile computing device, network equipment and so forth. And then these changes introduced new research areas of computer science, for example distributed system.

Distributed system is defined as a set of computers that communicate across network, hosting processes which use a set of distributed protocols to help the coherent execution of distributed activities [Veríssimo *et al.*, 2001]. In this way, components in distributed systems are deployed in separate computers. In other words, they run on different types of CPU, different memory spaces and operating systems. The main benefit and purpose of distributed systems are *resource sharing* which varies from hardware components such as disk and printers to software entities such as database and software objects [Coulouris *et al.*, 2005]. However, there are many challenges of distributed system, mainly because most knowledge and experiences from traditional software are not suitable any more for the development of distributed systems. Tanenbaum *et al.* [2007] discussed some ***False Assumptions*** that are often made by developers when designing connectors in distributed systems:

1) The network is reliable;
2) The network is secure;
3) The network is homogeneous;
4) The topology does not change;
5) Latency is zero;
6) Bandwidth is infinite;
7) Transport cost is zero, etc.

Therefore, to resolve the problems which are caused by one or more assumptions, connectors of distributed system should be the main focus of architecture design and developers should put more effort on development of connectors. Furthermore, these false assumptions are main motivating factors for constructing connectors with high qualities.

For example, availability and security (quality attributes) should be taken into account during design time for leading to the impossibility of making the first two false assumptions.

Another motivation is identified by the limitations of existing approaches and traditional mechanisms for designing connectors. The traditional mechanisms for developing connectors cannot be directly used for describing interaction of distributed components [Qiu, 2005] [Tanenbaum *et al.*, 2007] [Tari *et al.*, 2001]. For example, procedure call is well-known by developers in structured programming, but it cannot be used in different address spaces [Tanenbaum *et al.*, 2007] [Tari *et al.*, 2001]. The existing approaches (for example, middleware solutions and Web service) have their drawback to design of connectors, as is discussed in the following content (section 2.2 and 2.3).

1.2 Example Scenario: A Push Mail Solution for Mobile Device

Electronic mail (frequently called email) is a common approach to delivering digital messages on the internet. The email system employs a set of protocols for storing and forwarding messages, such as SMTP, MIME, POP3, IMAP [RFC3501, 2002] [RFC5321, 2008] [RFC1939, 1996] [RFC1341, 1992] and so forth. A typical procedure of sending and receiving emails involves four steps. Firstly, mail user agent (MUA) sends a composed message to local mail submission agent (MSA) using SMTP; secondly, MSA resolves the domain name of destination address and delivers the message to message transfer agent

(MTA) using SMTP; thirdly, the message is sent to message delivery agent (MDA) and MDA puts the message to the mailbox of the destination user; finally, the destination user (another MUA) needs to pick up the message using POP3 or IMAP protocols.

Nowadays, the demand for mobile mail is growing. With the wireless data communication, people can get messages, anywhere, anytime. *The challenge for realizing mobile mail is the design of connector between two different systems (e-mail system and mobile wireless network).* A technology market research [THE RADICATI GROUP, 2009] predicts that the wireless email market revenue will increase significantly in three years 2011 – 2013, as described below (Figure 1.2).

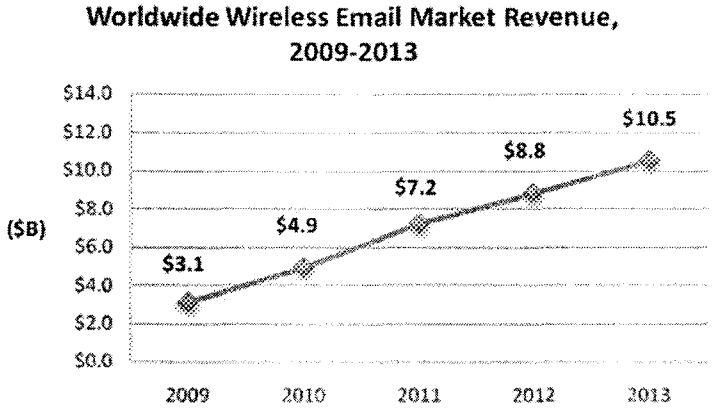


Figure 1.2: Wireless Email Market Revenue 2009-2013 [THE RADICATI GROUP, 2009]

In contrast to traditional internet email, the polling mode of the MUA residing in mobile device can greatly reduce the battery life and increase the traffic of wireless network, because of the need to constantly poll the mail server. Therefore, wireless email systems introduce push mail solutions to overcome the problem. Compared to polling mode, push

mail enables MUA of mobile device to automatically get new instant emails without polling the server to check for changes.

More recently, some technologies related to push mail have been applied to commercial system, for example EMN push [OMA, 2007] which is adopted by BlackBerry email system. In BlackBerry email system, blackberry enterprise server (BES) is a key part that can be regarded as a connector to link the IP network and wireless communication network (GPRS or CDMA).

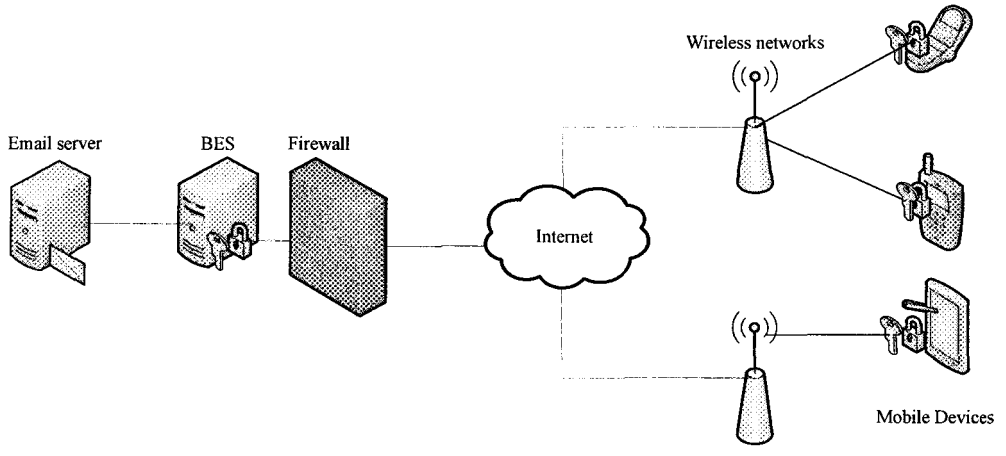


Figure 1.3: BlackBerry mobile mail system

Figure 1.3 briefly shows the architecture of BlackBerry mobile mail system. Before accessing the system, users must firstly activate their mobile devices for building the encryption keys (3DES or AES) and registering an account from BES. Then, BES commences monitoring the email server for new messages. When a new message arrives at email server, BES will notify the wireless network operators to ready mobile devices for receiving the new message. And then, BES pushes the message to mobile device. The

whole message delivering process provides a true push email service for users and the service is instant and secure.

However, for offering the push mail service, system must involve extra work of wireless network operators. In other words, the current solutions have to seek the supports from operators. And as a result, this type of support can probably increase the cost of push mail service. Thus, for overcoming this problem, the scenario in this chapter addresses such questions as:

- Can a real push mail system be designed without the additional support of operators?
- Can the implementation of the push mail system avoid modifying the existing deployment of wireless network?

In chapter 5, we will propose our solution to these questions and develop this scenario in more detail, to illustrate how the demands of this system are met through a dedicated connector between email server and mobile devices without involving extra support of wireless operators.

1.3 Statement of Problems

In general, software engineers face more challenges of building distributed system which are strongly related to problems of connectors; these problems are identified through different aspects of connectors in many research work: heterogeneity, openness, security, scalability, failure handling, concurrency, and transparency [Coulouris *et al.*, 2005]

[Tanenbaum *et al.*, 2007] [Mehta *et al.*, 2000]. Naturally, these general problems raise a question: why does the development of connectors become more difficult? The answers described below to this question results in framing the problem area of our research.

First, according to current research, there are various definitions and descriptions for connectors. For example, Shown *et al.* [1996] regarded connector as an element to represent the relationship among components; however, McGovern *et al.* [2003] defined connector as a software conduit between components. The definitions and descriptions address only certain aspects of connectors. And then it leads researchers to merely focus their attention on some parts of connectors and ignore others. For example, *certain descriptions of connector focus on delivery mechanism, but ignore relationship of components and presentation of information. For instance, middleware is regarded as connector in some research, but it cannot be used to present information (type, format...)* [Tanenbaum *et al.*, 2007] [Hohpe *et al.*, 2004].

Second, today's classifications of connectors are confusing and they easily cause researchers and developers to misunderstand connectors. Mehta *et al.*'s taxonomy of software connectors [2000] is frequently studied and several related research areas are based on that taxonomy. However, Balek *et al.* [2000] doubted that Metha *et al.* made a proper classification of connectors, because certain types of connector are situated in the different levels of software (across many layers).

Third, there are very few of design models for building connectors in distributed systems. Like common software development process, existing approaches to design of

connectors are only adopted to satisfy functionalities of connectors without taking the quality attributes into account [Kontio *et al.*, 2002]. Furthermore, in practice, multiple technologies and development tools must be combined to design and implement connectors. But the selection of development tools is rarely talked about.

Finally, the area for maintenance of connectors is seldom identified as a problematic issue. In contrast to design of connectors in new system, this area (maintenance) focuses on how to add connectors in legacy software systems and how to avoid more changes to existing systems, according to the concept of software maintenance [ISO/IEC 14764, 2006] [Grubb *et al.*, 2003]. Thus this problem is very different to building connectors in a new system from scratch. This thesis focuses on building new connectors in both legacy and new systems.

The problems of our research can introduce the specific questions. Firstly, how do developers describe connectors for covering all aspects of connectors in distributed systems? Secondly, in order to fully satisfy functionalities and to achieve qualities of connectors, how do software developers analyze and design connectors in distributed architectures by weaving relevant technologies together? Thirdly, can our description of connectors help to provide a new classification of connectors? Fourthly, is there any model to help developers to design connectors? What is the benefit of the design model? Finally, when updating or adding connectors in legacy software system, how do developers avoid major modifications in the existing components? In addition, in existing source code, where do developers

automatically find the key points (or statements), which other components are designed to obtain through desired connectors?

1.4 Contributions of the Thesis

This thesis is concerned with the design of connectors in distributed systems with aim of satisfying both quality requirements and functional requirements. For the benefit of industry, the thesis provides technology and expertise dedicated to raising the quality and reducing the cost of software development and maintenance. The main contributions of our research work are the following:

Firstly, this thesis presents a ***new definition of software connector***. The definition takes all aspects of connectors into consideration, especially for the connectors in distributed systems. Additionally, various knowledge areas can be introduced for designing connectors, including the system quality attributes.

Secondly, based on attribute-driven design method (ADD), we create a new approach which is called as ***Extended Attribute-Driven Design method (EADD)*** for both architecture design and selection of developments tools. *For designing connectors, EADD is equipped to drive architecture design by providing the instantiated modules and suitable development tools for developers.*

Thirdly, according to EADD design method, we propose ***a design model for generic purpose*** that involves various technologies and approaches for design of connectors in distributed systems. The model (consisting of LCM and LDM models) is capable of

producing architecture design of connectors and selecting a group of development tools. *The goal of the model is to enhance high level design of development by satisfying functional requirements and achieving quality attributes before diving into detailed implementation.*

Finally, based on the new definition and design model of connectors, our research performs an analysis and attempts to make ***a classification of connectors*** by combining the functionalities with qualities of connectors in distributed systems.

1.5 Organization of the Thesis

The rest of this thesis is structured as follows. In chapter 2 we examine current research about connectors and describe in detail the technologies and approaches involved in development of connectors. In particular, we show the attribute-driven design method for laying the foundations for following chapters. Chapter 3 introduces EADD design method for architecture design and selection of development tools and highlights the benefit which design of connectors can gain from the design method. In chapter 4 we propose our design model and related approaches to the use of the model in practice. And next, we perform an analysis of our model according to software development process. In addition, we show the advantages of our design model and attempt to give a classification of connectors based on the new definition and the design model. Chapter 5 presents a case study based on the example scenario of section 1.2: architecture design of connector for a push mail system in

wireless network, showing how the architecture of connector is designed and how the related development tools are selected based on our design model. Finally, Chapter 6 summarizes our research of this thesis and examines how the work can be taken further.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Introduction

First chapter introduced the research problems and questions as well as the objective of this thesis. This chapter lays the groundwork for the rest of this thesis by presenting the foundation of our research that builds on previous work in several areas. It is founded on middleware in distributed systems, particularly message-oriented middleware; Web services; relationship between components; message construction; aspect-oriented programming; quality attributes of connector; classification of connector; program slicing; and attribute-driven design method.

The research work in current literature presented in this chapter is closely related to design of connectors. Middleware solutions are often used to build connectors for carrying information; aspect-oriented programming is combined with pattern approaches for building the relationship of components (a part of connectors); attribute-driven design method and related analysis of quality attributes for connectors are introduced to facilitate development of connector to achieve high qualities. In this chapter, the related research to each of these areas is discussed in greater detail.

2.2 Middleware

In distributed systems middleware is dedicated software which is designed to deliver information between components without involving knowledge of network protocol and hardware platform [Tanenbaum *et al.*, 2007]. In other words, with aim of providing the

means for components to communicate with each other, they are introduced for hiding all differences and complexities of protocols, operating systems, and hardware platforms [Tanenbaum *et al.*, 2007]. Generally, middleware is considered as a software layer in systems. However, for designing connectors, middleware solutions have some disadvantages; for example, they cannot be adopted to present information (by type, format, etc.) [Tanenbaum *et al.*, 2007] [Hohpe *et al.*, 2004]. These issues are discussed in the following subsections.

2.2.1 TCP/IP and Sockets

TCP/IP protocol stack is the foundation of the Internet [Tanenbaum, 2003]. The protocol stack is involved in most modern operating systems by providing standard APIs-sockets for applications [Stevens, 1997]. Socket APIs are organized as a software layer situated between application layer and TCP/IP stack (including UDP) for facilitating network programing. Using the APIs enables applications to make use of TCP/IP protocols stack without needing to understand more about the mechanism of network and communication protocols. From this point of view, sockets can be regarded as a lightweight middleware which has been widely discussed [Mueller, 2002].

2.2.2 RPC and Object Oriented Middleware

In light of interaction of components, connectors of distributed system are more complicated than traditional connectors (or called classical connector running in desktop computer). This is because components are distributed in separate computers across network and the traditional approaches are not directly available for interacting with the

distributed components. For example, conventional procedure call is known by programmers in structured programming; but it cannot be used easily in different address spaces [Tanenbaum *et al.*, 2007] [Tari *et al.*, 2001]. Thus, in early 1980s developers extended the concept of procedure call and named it Remote Procedure Call (RPC) [Birrell *et al.*, 1984] for applying it to interaction of distributed components. However, RPC has the limitation: it is difficult to build applications that are dynamic [Pryce, 2000].

Last decade, object oriented middleware (OO middleware) is applied to component interaction in distributed systems. Compared to the RPC mechanism, OO middleware provides an easy programming model for the interaction of distributed programs using current programming language features [Pryce, 2000], such as CORBA (Common Object Request Broker Architecture) [OMG, 2004] and Java RMI (Remote Method Invocation). OO middleware implements the interaction by modeling procedure call. Thus, in some research OO middleware is regarded as an upgrade of RPC [Emmerich, 2000]. Even though OO middleware and RPC can help developers to construct the interaction between distributed components, there are some disadvantages: *due to synchronous mechanism of RPC, the flow of applications is always blocked for waiting for a call returning from remote site* [Cook *et al.*, 2005]; *it makes the technology a poor choice for object-oriented systems* [Cook *et al.*, 2005]; *and different OO middleware provides different support ranges of platforms and often specifies certain OO languages* [Cook *et al.*, 2005].

2.2.3 Message Oriented Middleware

Message oriented middleware (MOM) is also called as messaging system. It is a separate software system that provides messaging capabilities for distributed components [Hohpe *et al.*, 2004]. In contrast to OO middleware, MOM implements the interactions of components by formatting them as messages or events instead of modeling the interaction as procedure calls. For transmitting messages among components, MOM provides components with APIs that can give users the capability of providing stable communications in an unstable network environment.

Hohpe *et al.* [2004] showed two basic characteristics of MOM: (1) send and forget; (2) store and forward. In addition, MOM can offer non-blocking mechanism which is used to support asynchronous working mode. To transfer messages and support non-blocking, MOM maintains message queues and normally makes use of TCP/IP protocol. Both socket APIs and MOM can be used to delivery messages among components in distributed systems. But they are quite different, as described in table 1.1.

	Queue	Efficiency	Coupling	Mode
Socket	No	High	Tight	synchronous
MOM	Yes	Low	Loose	asynchronous

Table 2.1: TCP/IP and MOM ([Hohpe et al. 2004; Tanenbaum. 2003])

Compared with other middleware (Socket API, RPC and OO-middleware), MOM takes major advantage of the asynchronous working mode in network. Furthermore, MOM is easily deployed in distributed systems and provides loosely coupled linkage for distributed components. *However, like other middleware, MOM is solely responsible for moving data across network. They cannot be used to describe the relationship of*

components and they do not have the capability to present content (or format) of messages [Hohpe *et al.*, 2004] [Tanenbaum *et al.*, 2007]. In addition, the asynchronous communications lead to overload of messages in network and servers may delay to process all messages; and the MOM must run on every platform in systems, if one of them cannot support a MOM solution, the whole system will not work [Cook *et al.*, 2005].

2.2.4 Message Oriented Middleware and JMS

Java Message Service (JMS) [Gabhart, 2003] defines the rules for message delivery in Java enterprise systems, and also declares interfaces to facilitate message exchange between application components and MOM (messaging system). In Sun Developer Network (SDN), the goals of JMS are presented as followed [SDN]: the design goal is to provide a consistent set of interfaces that MOM clients can use independent of the fundamental message system provider. If a MOM (for example, WebSphere) in a system is replaced by another MOM (for example, MSMQ), all components based on the MOM do not need to be modified, because JMS keeps and uses the same interfaces with the components. In other words, MOMs provide functionality to client for exchanging messages. But the different MOMs providers always have different API for processing messages. Thus, Java gives its solution (JMS) to support most of commercial and open source MOMs. Figure 2.1 shows the relationship among TCP/IP protocol stack, MOM and JMS, when the component1 and 2 need to communicate with each other.

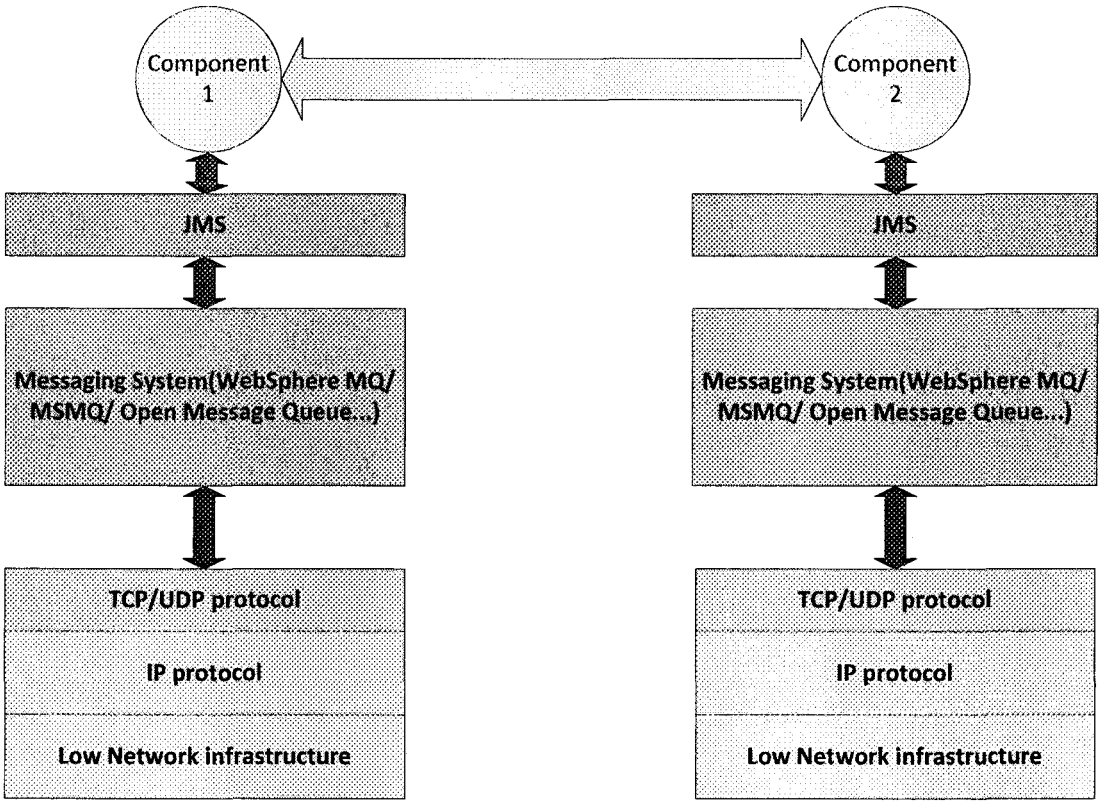


Figure 2.1: TCP/IP, MOM and JMS ([SDN; Gabhart. 2003])

2.3 Web Services

Web services are designed to facilitate development of Internet-scale distributed computing, by providing general services for remote applications [Vogels, 2003]. Nowadays, the term is widely used, but the exact meaning of Web services varies greatly, depending on the underlying concepts and technologies [Alonso, 2004]. For example, Web services are traditional services (with functionalities) which are made available across the Internet [Tanenbaum *et al.*, 2007]; on the other hand Web services are defined by a set of protocols (SOAP, WSDL, and UDDI) or called network accessible interface to which a

number of vendors conform for defining, describing, and discovering services [Karastoyanova *et al.*, 2003] [Snell *et al.*, 2002].

In terms of interaction between programs in network, W3C gives a definition of Web services [W3C, 2004]: "*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.*" Thus, from this point of view, Web services can be regarded as an evolution of middleware. According to the point, this section will discuss the basic principles of Web services and criticize it.

The principles hidden in Web services are quite simple. And there is nothing new beyond the area of distributed computing. The principle is that client application can make a request for Web services provided in remote servers. According to the requests, the servers execute program of desired services and give responses back. All services are described, defined and discovered based on standardization [Tanenbaum *et al.*, 2007]. Below we analyze the main protocols (standardization) used in Web services.

SOAP (Simple Object Access Protocol) was initially designed to remote-procedure call services for computers across network. Now it has been become a simple and lightweight protocol for exchanging XML messages [Singh *et al.*, 2005]. SOAP has two communication approaches: SOAP RPC and SOAP Messaging. However, the SOAP protocol does not specify any certain type of underlying protocols. In other words, SOAP can bind with various transport protocols between SOAP nodes, such as POP3, JMS, MOM, TCP/IP sockets and so forth [Nagappan *et al.*, 2003]. Thus, the XML messages can be delivered to the destination by using transport protocols. Similarly, the receiver can receive

the XML message. In this way, service consumer is capable of invoking the local or remote service and gets the result returned by service providers.

WSDL (Web Service Description Language) is an XML language for describing network services as a set of interfaces based on messages that contain the document-oriented or procedure-oriented information [Christensen *et al.*, 2001]. In other words, WSDL is used to define the name of service, the type of input/output parameters and the related data structures. In addition, it provides the service consumers with network addresses, protocol bindings and message format of input/output.

UDDI (Universal Description Discovery and Integration) provides directory service that is based on XML and SOAP [Singh *et al.*, 2005]. It develops a specification for building framework to register and find services across network [UDDI, 2000]. After service provider submits a service to UDDI (in broker), the service location, type and description of service API are registered for invocation of application builders (service consumers).

Web services technology is very limited. *They are about interoperable document centric computing, not distributed objects [Vogels, 2003].* Web services technology is developed to deliver XML documents using standard Internet protocols, instead of modeling interfaces with methods. In addition, Web services cannot support the ***stateful*** distributed computing that most distributed object systems have offered as basic functionality [Vogels, 2003].

Web services evolve from middleware and are classified as a heavy weight middleware [Schmidt *et al.*, 2002]. Hence, like other middleware we mentioned previously, Web services do not have the capability to present content (or format) of messages. In terms of SOAP messages, all data and application-specific data types are formatted in XML, however there is no generic mechanism to serialize application-specific data types to XML [Nagappan *et al.*, 2003]. In addition, if we exchange data between SOAP nodes and we do not want others to get the information, the security of the communication should be provided through encryption. But the SOAP specification does not define encryption for XML Web Services [Howard, 2001].

Web services cannot replace current middleware solutions, because i) Web service are particularly used to prevent incompatibilities between middleware of different vendors and application integration in distributed systems; ii) not all networked applications need the heavyweight middleware [Schmidt *et al.*, 2002]; iii) and *current Web services are hard to meet the functionalities and quality attributes of other middleware solutions, such as reliability, scalability, and performance* [Karastoyanova *et al.*, 2003]. Furthermore, a web service is implemented by organizing components; and in some situation, these components are so complex that they are completely distributed across network [Tanenbaum *et al.*, 2007]. Hence, the current middleware solutions can be used to construct Web service.

For designing connectors, like other middleware solution, Web service can be adopted to build connectors. And for developing a service, connectors may be used to connect components of Web service in network.

2.4 Relationship between Components

Relationship and dependency analysis among software components are always a challenging issue for developers and researchers [Li, 2003] [Gupta *et al.*, 2010]. Based on component-based systems (CBSs), much research in this area has been done. In terms of CBS, Ratneshwer *et al.* [2011] identified some types of dependence analysis, such as data dependence, control dependence, interface dependence, and real time dependence. Their study may be helpful in understanding CBS software. In Edwards's research [1997], the problems of relationships are defined as follows: (1) relationship of component should concisely describe dependencies of component compositions; (2) it should provide precise meanings for clients and developers; (3) it should reflect a clear conceptual view of component-based software engineering. In order to address these issues related to design of connectors, this section discusses relationship and dependency from pattern point of view.

2.4.1 Object Oriented Programing (OOP)

The relationships among classes (the object or class is commonly regarded as a kind of lightweight component) are analyzed in Booch *et al.*'s research [2007]. There are three basic kinds of class relationships: generalization/specialization, whole/part, and association relationships [Arranga *et al.*, 1997] [Zamir, 1999]. In these relationships, associations are most useful approaches to connect two components. In class diagrams, association relationships in a C/C++ application represent the following things [IBM Corporation]: (1) A semantic relationship between two or more classes that specifies connections among

their instances; (2) a structural relationship in which objects of one class are connected to objects of a second class.

2.4.2 Design Pattern (DP)

Design patterns can be used to describe relationships and interactions between components (classes or objects). The design patterns in Gamma *et al.*'s book [1995] are descriptions of communication of objects and classes that are customized to deal with general design problems. The communication of object and class should be loosely coupled without becoming entangled in each other's data models and methods [Cooper, 2000]. Gamma *et al.*'s design patterns particularly deal with problems at the level of software design, especially object-oriented software design. They can be classified by criterion scope which is used to specify whether the pattern is mainly used to classes or objects. Thus, the design patterns have two kinds: class design pattern and object design pattern [Gamma *et al.*, 1995]. In particular, object patterns can be applied to object relationships, which can be modified at run-time and are more dynamic, such as Proxy Pattern, Observer Pattern etc.

2.4.3 Architectural Pattern

An architectural pattern provides a fundamental structural organization schema and a set of predefined subsystems for software systems. It includes rules and guidelines for organizing the relationships between them [Buschmann *et al.*, 2007]. The term architectural style has also been used to describe the same concept [Bass *et al.*, 2003]. The architectural patterns concentrate on large-scale elements of architecture (heavy components) especially in the distributed system, compared to design patterns. Actually, some patterns are often

researched in different areas at same time. For example, publish-subscribe is sometimes regarded as design pattern while it can also be considered as architectural pattern.

There are many architectural patterns which have been well known for a long time in the software engineering, such as Model-View-Controller (MVC), Pipes/filters and Layers etc. The developers can apply them to implementation of the interaction of components in distributed systems.

2.5 Message Construction

In section 2.2, the limitation of middleware solutions is mentioned: lack of capability to present information between components. This section introduces a research area to address this problem.

Software message is a packet of data that can be delivered between functions, classes, processes, computers and so forth. In order to transmit messages, message senders (message producers) need to split data into one or more packets. And next, each packet is encapsulated in passing information to form a message. Likewise, message receivers (message consumers) need to remove passing information and find out desired content. From the view of network protocol stack, upper protocol message is always wrapped by lower protocol [Tanenbaum, 2003].

A message may comprise two basic parts: message header and message body. Normally, message header is composed of passing information, such as message type,

address of receiver; message body consists mainly of data or part of data. In practice, messages that may be transmitted without message body are employed in notification and the messages are also called as event according to certain type of use. Therefore, in contrast to message body, message header is more important.

In Hohpe *et al.*'s research [2004], they presented some message types in terms of different application styles of use: command message, document message, event message and so forth. When beginning to design connectors, thinking about the type of message which will be carried by connectors is a necessary step, because the separate type of message, in practice, requires different transport mechanism of connector. For example, a researcher [Thekkath *et al.*, 1994] studied a remote procedure call (RPC) model in distributed systems, in which separating data transfer and control transfer can much improve the performance of the system. In addition to the types of message, there are other aspects described below that are typically taken into account when designing connectors.

Moreover, the amount of information exchanged in a distributed system can deeply influence the design of connector [Mattmann, 2007]. For example, P2P protocols are always chosen as communication mechanism of some connectors to deal with large scale data.

Message format is another important aspect. One loosely coupling solution [Hohpe *et al.*, 2004] adopts a standard format that is self-describing and platform-independent instead of sending information directly to a specific machine. Normally, this format of message is

not human-readable. For example, XML formatted message are applied to SOAP for exchanging data between Web server and client [Erenkrantz, 2004].

Message security not only depends on the communication mechanism but also the process of message. Several technologies emerge to address this issue: digital signatures, message encryption, public key cryptography, digital certificates and so forth [TechNet, 2005].

2.6 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is combined with object-oriented programming (OOP) for achieving a basic objective [Gradecki *et al.*, 2003]: describe and divide the concerns by crosscutting components. AOP has no intention of replacing OOP or other object-oriented mechanisms. Moreover, it is employed to process crosscutting concerns that are described as system-wide concerns covering a set of unrelated classes [Laddad, 2003]. In order to process crosscutting concerns of classes, AOP provides a mechanism to describe the crosscutting concerns by bringing in a new programming element- aspect. In AOP, the goal of aspect is to crosscut normal classes of OOP without modifying the source code of them. The core mechanism of AOP uses an aspect weaver that is a compiler-like tool. This tool compiles the whole program by combining the crosscutting concern (aspect) with other classes and the compiling process is defined as weaving in AOP [Laddad, 2003]. Therefore,

the working mode of AOP technology is static. The whole program cannot be dynamically changed after weaving them.

According to the properties of AOP, it has the capability to build relationships (following a design pattern) between separate components without modifying much of the existing source code of components. As a result of this point, using AOP enable the developers to easily design, implement, extend and maintain software system.

Despite this, AOP has some disadvantages. For example, crosscutting classes can lead to breaking program flow and encapsulation of OOP. Moreover, in terms of distributed system, AOP has a major limitation. It is merely applied to process classes situated in one local program, because it can only provide the mechanisms for developer to weave aspects and base code of classes together into a coherent program [Elrad *et al.*, 2001]. As a result of this limitation, AOP cannot be directly adopted in distributed systems. For overcoming the limitation, developers are meant to combine object oriented middleware (for example, remote procedure call) or message oriented middleware with AOP technology [Bishop *et al.*, 2006]. However, the combination is still a challenge for developers.

2.7 Program Slicing

Program analysis is concerned with automatically extracting information from programs [Aiken, 1999]. It involves many knowledge areas, disciplines and approaches. In Nielson *et al.*'s research [1999] they addressed some approaches to program analysis: data

flow analysis, constraint based analysis, abstract interpretation, and type and effect systems. To perform these program analysis, program slicing is a well-known mechanism and transformation method that is particularly good at performing data flow analysis and control flow analysis (constraint based analysis).

Program slicing uses program statement dependence information to identify parts of a program that influence or are influenced by an initial set of program points of interest (called the slice criteria) [Ranganath *et al.*, 2006]. Program slicing helps to reduce a large program to a small subset that does not change the behavior of original program [Weiser, 1982]. Therefore, the subset of program (called a slice) makes the large program more easily understood. Hence, program slicing can greatly facilitate program maintenance that consists of program debugging [Weiser, 1979], program comprehension [Harman *et al.*, 2003], program testing and so forth.

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes [ISO/IEC 14764, 2006]. Designing a new connector for an existing distributed system belongs to software maintenance. Developers often face the difficult task of maintaining connectors. Particularly, when they are not familiar with the systems and they do not have enough documents about the program, the developers must spend much time analyzing and understanding the existing source code. Sometimes, software developers may put themselves in bad situation: they cannot find the right points (or statements) in large and complex source code files to provide the information for other components through

connectors. Or after adding a new connector, the source code becomes more and more difficult to understand and maintain.

The program slicing technique will be talked in chapter 4 as a powerful approach to design of connectors.

2.8 Quality Attributes of Connector

Component interaction is generally identified as the basic functional requirement of connector. Like other software systems, connectors not only have functionalities but also possess corresponding quality attributes. In addition, quality attributes are more important than functionalities of connectors [Carmichael, 1998]. Particularly, connector quality attributes in distributed systems play more critical role, compared to the attributes of traditional connectors residing in local applications.

Bass *et al.*'s research [2003] addressed the system quality attributes of software, such as availability, modifiability, performance, security, testability, usability, and so forth. In order to achieve quality attributes of connector in distributed systems, these qualities should be firstly considered in design, implementation and maintenance stages.

In terms of connectors in distributed systems, we talk about these quality attributes based on Bass *et al.*'s research [2003]:

- Availability of connectors is concerned with system failure, related fault detection and recovery. It involves network failure, crashed components, system exceptions etc.
- Modifiability of connectors is about the cost and time for changing connectors and maintaining connectors in existing system. Localize modification [Bass *et al.*, 2003] is a main solution to control the cost and time.
- Performance of connectors is relating to giving a response to an action in an expected time. Reducing the resource consumption of connectors is basic idea to achieve the attribute.
- Security is concerned with resisting attacks and detecting attacks. In terms of distributed system, attacks are identified as remote access to computers, unauthorized users etc.
- Testability is about how connectors can be easily validated and tested in static and runtime.
- Usability of connectors refers to how connectors can be easily deployed in a distributed system. Well-defined interfaces and suitable architectures are two goals for achieve the quality of connectors.

2.9 Attribute-Driven Design Method

Software engineering methods provide structures for software engineering activities to make the activities systematic [Abran *et al.*, 2001]. In SWEBOK, the authors classified

engineering methods as heuristic methods, formal methods and prototyping methods [Abran *et al.*, 2001]. For example the attribute-driven design method and traditional object-oriented methods both belong to heuristic methods.

Attribute-driven design (ADD) method is designed by the Carnegie Mellon Software Engineering Institute (SEI). *ADD is an approach to designing software architectures in which the requirement analysis and the high level design process rely on the software's quality attribute* [Wojcik *et al.*, 2006]. *ADD method is structured by a set of recursive steps which is iterated until all architecturally important requirements are satisfied* [Bass *et al.*, 2003]. These steps are [Wojcik *et al.*, 2006][Wood, 2007]: 1) confirm the requirement information; 2) choose an element to decompose; 3) confirm the architectural drivers; 4) choose design concept to satisfy the architectural drivers; 5) instantiate architectural elements; 6) define interfaces for elements; 7) verify the elements and refine requirements; 8) repeat the process as necessary.

The input of ADD involves functional requirement, design constraint and quality attribute requirement; the output of ADD is part of the high levels design of some views of architecture. The views of architecture include module decomposition, concurrency, and deployment. Figure 2.2 shows the input/output of ADD method.

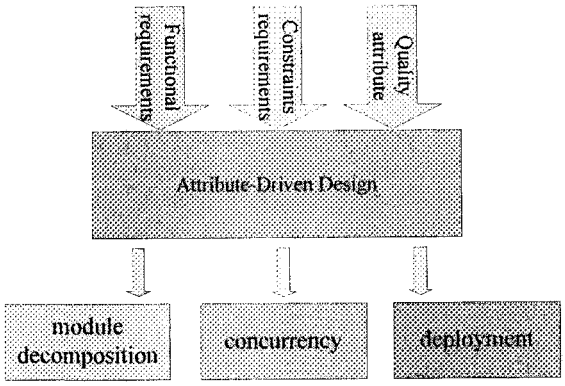


Figure 2.2: Attribute-Driven Design Process

Quality attribute requirements that belong to non-functional requirements show the different properties which a system must exhibit; functional requirements indicate what functions a system must be provided for stakeholder's needs when the software works under specific conditions; design constraints which can be also regarded as a kinds of non-functional requirements are decisions about a system's design that must be involved into final design [Wojcik *et al.*, 2006]. Sometimes, developers do not need to classify them. Particularly, functional requirements and design constraints sometimes refer to similar requirements. However the more important thing is to find all the related requirements as the input of ADD.

Architectural drivers are the combination of functional requirement and quality requirements which are used to form the architectures (modules) [Bass *et al.*, 2003]. Architectural patterns and tactics are applied to satisfy the quality attributes which are used to define types of elements and interactions by following the relationship (in step 4 of ADD). While functionality requirements are chosen to instantiate the module types (in step 5 of ADD).

In Bass *et al.*'s research [2003], the authors presented several types of quality. Here we only concentrate on qualities of the system, such as availability, modifiability, performance, security, testability, and usability. System quality attributes are the backbone of a software system. Therefore, they are critical to the success of whole system.

However, ADD method has its limitation, which is, that ADD does not have the capability to select development tools for satisfying quality requirements in high level design. In the following chapter (chapter 3), ADD design method will be extended to overcome this limitation.

2.10 Classification of Connectors

Classification of software connector helps to choose a suitable connector or a composition of connectors in design process. Many useful research work related to this area emerge to talk about classification of software connector [Allen *et al.*, 1997] [Oreizy *et al.*, 1998]. In Mehta *et al.*'s research [2000], they defined four general categories of software connector in terms of the services that connectors provide for components: *communication service* is concerned with transmission of data among components; *coordination service* is relating to transfer of control among components; *conversion service* is applied to conversion of the interaction required by one component to that provided by another; *facilitation service* is defined to mediate and streamline component interaction. According to these services, they divided connector into 8 major types: 1) procedure call, 2) event, (3)

data access, 4) linkage, 5) stream, 6) arbitrator, 7) adaptor and 8) distributor [Mehta et al., 2000]. Each type of connector provides one or more interaction services and can only be chosen under specific circumstances that are considered by requirements of the interactions. We draw a conclusion (Table 2.1) from Mehta *et al.*'s research to illustrate the relationship between the connector types and the services. From this table, we can find that different connector type can and only can support specific services. For example, one connector that belongs to type procedure call is suitable for a situation in which two components need coordination service. On the contrary, procedure call is not a proper connector to provide a conversion service.

	Communication	Coordination	Conversion	Facilitation
Procedure Call	*	*		
Event	*	*		
Data Access	*		*	
Linkage				*
Stream	*			
Arbitrator		*		*
Adaptor			*	
Distributor				*

Table 2.2: The relationship between the connector type and the service ([Mehta *et al.*, 2000])

Mehta et al.'s service categories are the foundation of their classification framework. But some problems of the categories are identified. Communication and coordination services are defined in terms of the type of information (data or control) that the connectors transfer. However, in practice, one connector of a system is often designed to transmit all information no matter what it is. It is true that it is necessary to enable connectors to understand the content of information that belongs to data or control. But most connectors

are typically devised to provide the both services. In addition, sometimes it is often difficult to distinguish between data and control, because data can also affect the execution of components. Conversion and facilitation services are both fundamental functionalities of connectors. Connectors are frequently designed using facilitation of protocols and mechanism to achieve conversion. Therefore, it is unnecessary to particularly talk about the two services. For example, the type of connectors- event mentioned in their classification can often provide concurrency control service (facilitation) to systems; procedure call is frequently used to compose the conversion service. But the authors did not consider them.

Furthermore, in Balek *et al.*' research work [2000], they discussed that Mehta *et al.*'s connector types are situated in different software levels. For example procedure calls are the assembly language of software interconnection [Shaw, 1994]. It is not suitable to be put together with event or data access. In addition, some connector types have overlapping functionalities, such as event and stream. Another problem is that they lack the content which addresses approaches to design of connector type.

2.11 Conclusion

This chapter laid the foundations for this thesis. All research area presented in this chapter is immediately related to design of connectors in distributed systems. Middleware solutions are chosen to develop connectors for delivering information; the combination of aspect-oriented programming and pattern approaches have the capability to build the

relationship of components in terms of connectors; message construction is concerned with the presentation of information exchanged among components; the discussion of classification of connectors help developers to understand the different functionalities offered by connectors; program slicing, which is a static analysis technique, can help to maintain connectors for legacy systems; attribute-driven design method (ADD) and related analysis of quality attributes for connectors greatly facilitate development of connectors for achieving high qualities. But ADD is not capable of making a selection of development tools to enhance the qualities of connectors. Thus, next chapter will extend the ADD method, with the aim of adding new feature of choosing development tools.

CHAPTER 3

EXTENDED ADD METHOD FOR DESIGN OF CONNECTORS

3.1 Introduction

In previous chapter, we have presented the background and related work about connector. Particularly, we highlighted ADD design method and quality attributes of connector. In order to prepare the way for the design model of connectors proposed in chapter 4, this chapter extends the ADD method by supporting the selection of development tools.

Software connector with high quality and full functionality are always required by distributed system. However, in the development process engineers frequently face many problems which come from selection of development tools and frameworks, multiple requirements of quality attributes, use of design processes and design methods. These issues are identified with software engineering. In Puntambekar's research [2009] software engineering is defined as a layered technology. It comprises 4 parts: quality layer, process layers, methods layer and tools layer. The process layer combines methods layer with tools layer for developing high quality software. In the method layer, functionality-based design is regarded as a common method, such as object-oriented design method (OOD) [Bass *et al.*, 2003]. OOD mainly takes into account the functional requirements. The output of the method is a collection of objects rather than functions [Abran *et al.*, 2001]. In contrast to OOD, as mentioned early, attribute-driven design (ADD) method takes quality attributes, design constraint and functional requirements as input [Bass *et al.*, 2003]. The output of ADD is high level architecture design made up of a group of decomposed modules. These

modules are presented by different views of architecture comprising module decomposition, concurrency, and deployment.

However, ADD has its limitations. The design method does not have the capability to select suitable development tools for satisfying quality of software. To develop connectors in distributed systems is required to adopt multiple development tools and technologies. This is because i) connectors should provide a set of functionalities and possess many quality attributes (next chapter will discuss this point in detail) which the use of one or two development tools cannot achieve; ii) components linked by connectors are distributed in network and they are developed by different tools running in separate operating systems and under different network setting etc.; iii) the development is complex so that multiple development tools and frameworks combine to build connectors in design, implementation, and maintenance stages. These limitations are main motivating factors for extending ADD design method.

In this chapter, we attempt to address the problems of development tools and frameworks in software engineering area. In particular, we mainly focus on the problems of developing software connectors.

- In order to satisfy quality attributes of software connector, how do engineers choose development tools and frameworks in high level design stage?
- Is there any design method that is able to help engineers select suitable tools and frameworks that are capable of being organized together to construct software connectors?

In this chapter, we propose an extended attribute-driven design (EADD) approach to selecting development tools and frameworks. According to our approach, the chosen tools and frameworks have capability to achieve quality attributes. In addition, every decomposed module (output of ADD) of architecture design corresponds to a set of development tools and frameworks.

3.2 Software Engineering and Design Methods

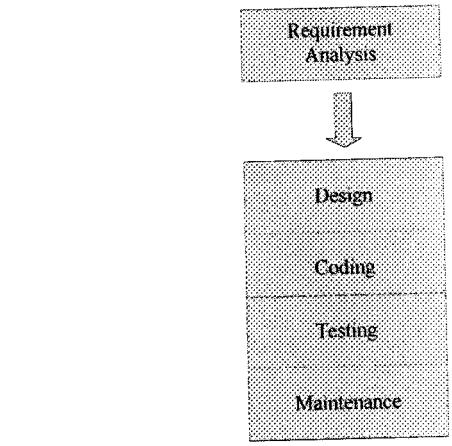
Before presenting our design method of software architecture, we lay the software engineering foundations for our design method by: 1) reviewing software development process; 2) discussing software requirement analysis and architectural design; 3) analyzing software development tools.

3.2.1 Software Development Process

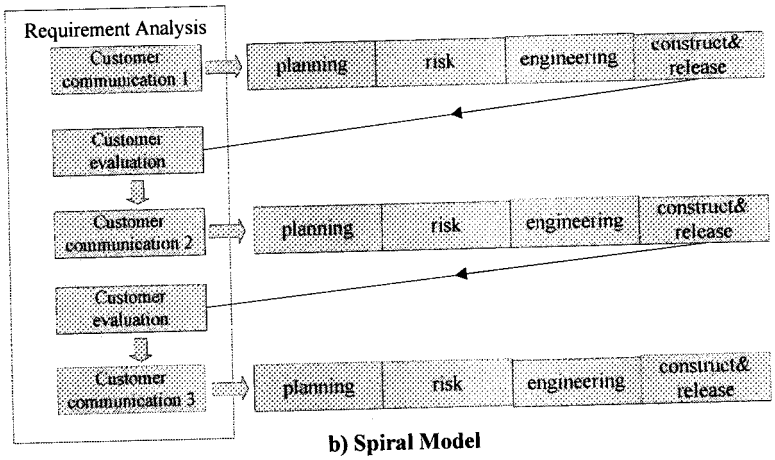
Software development process is an important part of software engineering. Puntambekar's definition of the process is a group of activities which are organized to help developers construct software systems [Puntambekar, 2009]. In the software development process, software life cycle models are defined and used to address the problem of software development for developers.

Software life cycle models divide the whole development process into separate phases for controlling and managing development. They consist of waterfall model, throwaway model, incremental development, evolutionary prototype, reusable software, spiral model

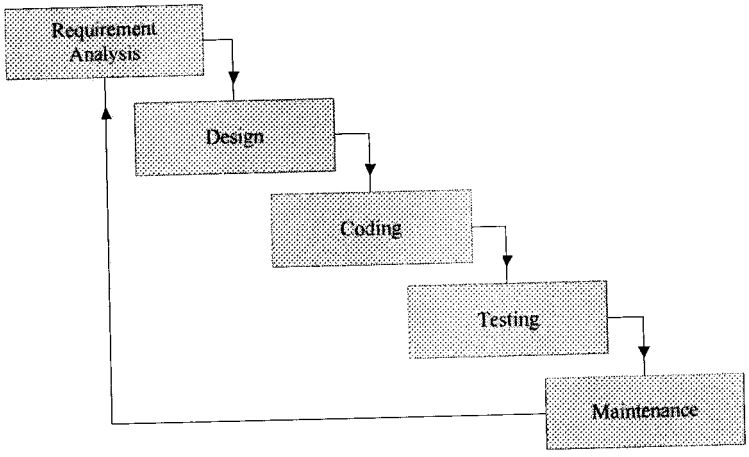
etc. [Comer, 1997] [IEEE/EIA, 1996]. These models vary greatly in practice depending on the skills of staff, schedules, tools, and funding of projects. However, they all have a common phase- requirement analysis (or called users' needs, communication with customer) which drive the software life to be cycled. For example, in the waterfall model (in Figure 3.1.a) the phase "requirement analysis" is situated on the top of other phases, and the results of the phase will lead the next phase (design) to start; in the spiral model (in Figure 3.1.b), each iteration of the process begins with the region "customer communication" that actually is the input of the requirement from customers. Compared to waterfall model, the phase of requirement analysis is also required in the spiral model but the inputs of requirements are split among the iterations; like waterfall and spiral model, incremental development (in Figure 3.1.c) also has a requirement analysis step in its model. Therefore, the requirement analysis is considered as a significant phase in software development process.



a) Waterfall Model



b) Spiral Model



c) Incremental Model

Figure 3.1: Requirement analysis in waterfall, spiral, and incremental models

3.2.2 Software Requirement Analysis

Requirements analysis is highly significant for the software development process. The procedure of the requirement analysis is described in Abran *et al.*'s research [2001]: i) collecting the requirements; ii) defining the bounds of the software project and its interfaces between software and users; ii) giving more details about system requirements which are derived from software requirements.

System requirements are concerned with the detailed functionalities of system and constraints [Sommerville, 2004]. They are typically classified as functional requirements and non-functional requirements [Puntambekar, 2009]. Functional requirements are the functionalities and services offered by software systems. In contrast to user requirements [Sommerville, 2004], functional requirements should be presented in great detail by using low-level statements according to the services and functionalities of the system. Non-functional requirements are the properties and constraints of system [Puntambekar, 2009]. They do not directly relate to the specific functions provided by the system. They mainly concentrate on the qualities delivered by the system, such as reliability, performance, availability, modifiability and so forth. In particular, non-functional requirements are more important than functional requirements, because one non-functional requirement that is not entirely satisfied may lead to a total failure of a software system.

3.2.3 Architectural Design

Software architectural design is normally regarded as the high-level design. It mainly focuses on three design issues [Bass *et al.*, 2003]: i) how to structure the system (as a group

of modules); ii) how to design the runtime properties of the modules and the interactions between them; iii) how to deploy the modules.

In practice, the processes of requirement analysis and architectural design are not cleanly separable from one another [Sommerville, 2004]. Architectural design process often overlaps with the requirements process, because the separate modules are used to satisfy the different requirements, while the system must be divided into different modules according to the requirements. In addition, during the analysis process the interactions between modules often lead to new system requirements.

3.2.4 Software Development Tools

Software development tools enable repetitive activities to be automatically completed in the software life cycle processes. Tools are often intended to support particular software engineering methods [Abran *et al.*, 2001]. According to SWEBOK [Abran *et al.*, 2001], software development tools consist mainly of 5 types which correspond to five different software processes. They are software requirements tools, design tools, construction tools, testing tools and maintenance tools.

New software technologies and requirements frequently introduce new software development tools for building software. For example aspect-oriented programming brings in AspectJ [Gradecki *et al.*, 2003] for constructing aspects in java programs; program comprehension promotes the Indus slicing tools [Ranganath *et al.*, 2006] for software maintenance. Recently, network technologies and distributed system are rapidly growing. In distributed systems components are situated in network. These components, which may

run on top of different operating systems, may be developed by separate teams who use different tools. Hence how to choose proper development tools under different development processes is frequently a challenge for engineers.

3.3 EADD for Selection of Tools and Frameworks

3.3.1 Quality Attributes Supported by Tools

As mentioned early, five types of development tool are identified in software engineering. This section concentrates on 3 of them: construction tools, testing tools, and maintenance tools, because these tools are directly related to quality attributes and they have capability to drive architecture design. Although the selection from these tools depends on functional requirements, non-functional requirements (system quality attributes), cost of development and developers' skills, in this section we merely focus on the effect of the system quality attributes: availability, modifiability, performance, security, testability, and usability. Below, Table 3.1 demonstrates how the development tools are organized to satisfy the quality attributes of software.

Quality	Features of the Development Tool	Example
Usability	capability to develop user interface, particularly, the graphical user interface	wysiwyg tools: Adobe Dreamweaver
Modifiability	capability to modify the program without changing much more	AOP: aspectj; program slicer: Indus
Performance	provide efficient solution to system.	compiled languages: C/C++
Portability	enable the application to be easily ported to other system	virtual machine codes: Java
Availability	support fault recovery mechanisms	exception handling mechanisms: java
Security	provide the protection mechanism	Sandbox: Java applets and Silverlight
Testability	have powerful IDE with debugging tools to help the test of application	IDE: Eclipse; GNU tool chain

Table 3.1: Quality Attributes Supported by Tools

3.3.2 Quality Attribute Supported by Frameworks

In computer software, a framework is a reusable set of software system that is made up of development tools and typical approaches for providing functionalities for developers. Normally, software framework cannot work independently. It needs to be combined with specific software in run-time or compile phases for achieving the objective of software systems.

Typical approaches and related development tools are key parts of software framework. For example, the Microsoft Foundation Class (MFC) is a famous desktop framework that wraps most parts of the Windows API (development tools), event-driven mechanism, and provides MVC approach (document-view) for developing windows-based applications, especially programs with GUI. In addition to functionalities, MFC framework is capable of helping to achieve usability of applications, because of the support of GUI.

Thus, like development tools described previously, the use of frameworks also places significant importance on the quality attributes of software. In fact, this feature of framework is distinguished by corresponding development tools that are involves in frameworks. In other words, from quality attributes point of view, the selection of frameworks can be analyzed and regarded as certain complex type of development tools. Hence, for creating our design method in the following section, our research considers that the effect of frameworks is same as development tools. For example, when talking about the Spring framework [Johnson *et al.*, 2005], we mainly concentrate on Inversion of Control (IOC) and AOP tools.

3.3.3 EADD

Choosing development tools becomes key determinant of software development process. Before implementing software (coding phase), developers should normally make the selection of development tools in high level design. Hence the phase of choosing tools must be situated between requirement analysis and low-level design. Generally, it relies heavily on the quality attribute requirements, functional requirements, design constraints and so forth. Based on the mechanism of ADD method, we create a new output for ADD method- a view related to software development (described in Figure 3.2). The contents of the view are the chosen tools which are arranged to deal with specific decomposed modules. The goal of the adjustment is to find more suitable tools to develop the software for satisfying the quality attributes of software. In other words, the selection of tools can also be driven by quality attributes. In our research we call this adjusted ADD method as Extended ADD (EADD) design method.

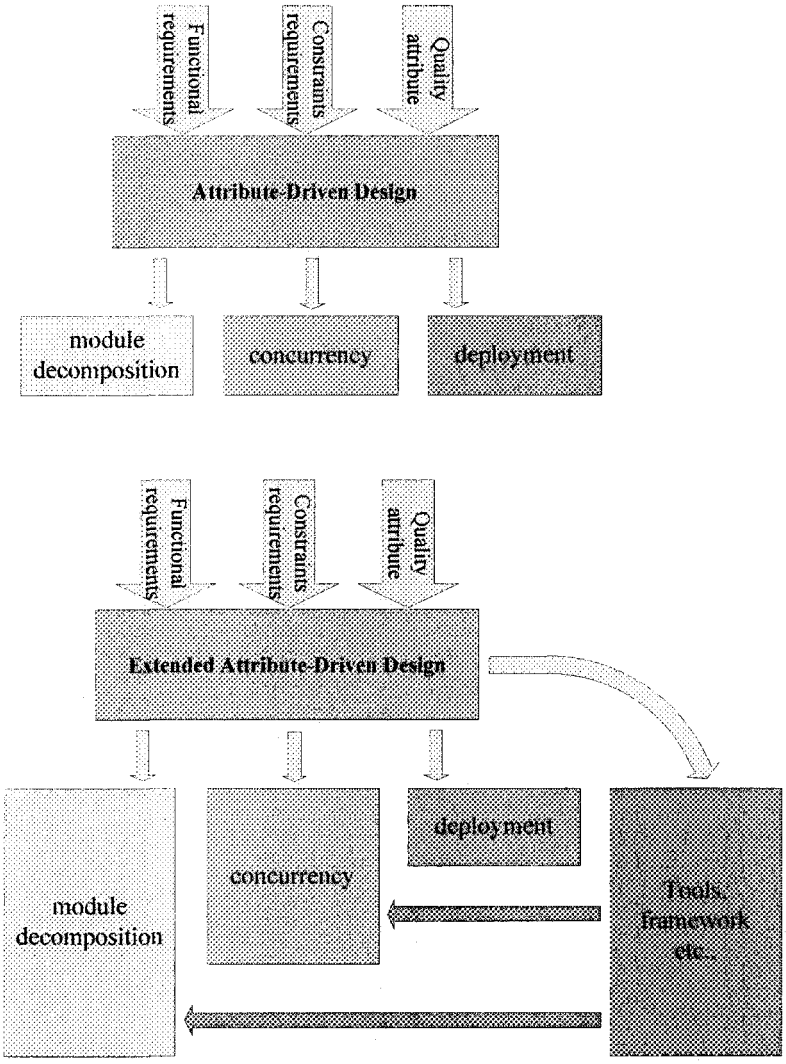


Figure 3.2: Adjustment to Attribute-Driven Design, Software Development tools (new output)

According to the steps of ADD method specified by SEI [Wojcik *et al.*, 2006], EADD makes an adjustment to the step 5 of SEI'ADD method (Instantiate elements and allocate responsibilities). In the adjusted step 5, after instantiating the types of software elements and allocating responsibilities, EADD is capable of helping developers choose proper development tools for every instantiated module according to the qualities requirements

that they must satisfy. After the EADD process is completed, the decomposed modules and corresponding development tools are both obtained in high level design phase.

Software connectors are normally made up of different modules that may lead to complicated architectures for providing functionalities and ensuring qualities. Moreover, the implementation and maintenance of connectors may involve multiple development tools because connectors are often used to link different components which are developed using various tools and may run on separate platforms, particularly connectors in distributed systems. As a result, the EADD design method is very suitable for architecture design of connectors. For designing connectors in high level design, EADD has capability not only to drive the architecture design but also to select development tools for satisfying the quality attributes.

3.4 Conclusion

This chapter discusses the selection of development tools in software engineering. We mainly concentrate on the problems in software development process: how to choose tools (including frameworks) which are applied in development process to ensure the qualities of software connectors? Before addressing the problems, we analyze the quality attributes of software which the development tools support, according to development process in software engineering. Then we propose an approach (EADD) to selection of tools based on ADD design method. Our design method is capable of not only producing architecture

design but also offering selection of development tools according to functional requirements and quality attributes. At last we highlight the effect of our design method on design of connectors.

The following chapter will create a design model for designing connectors using the EADD method. Based on EADD, the design process of the model can be greatly facilitated.

CHAPTER 4

DESIGN MODEL FOR CONNECTORS BASED ON EADD

4.1 Introduction

In chapter 3, we developed the EADD design method to help developers create architecture design and select development tools according to functionalities and quality attributes. This chapter uses EADD to develop a model for design of connectors.

In the recent years, distributed system and relevant technologies have grown steadily. As a result, the design of connectors in distributed architecture makes architecture design become more complicated than the design in desktop applications. This is because the components are distributed in separate computers over the network and the traditional approaches to designing connector (for example, the method-based mechanism) are *not directly available* for interacting with the distributed components [Qiu, 2005] [Tanenbaum *et al.*, 2007] [Tari *et al.*, 2001]. In order to design connectors in distributed systems, multiple technologies are applied in practice. However, when facing these technologies, developers often meet some problems about the design and implementation of connectors. For example, messaging system or socket APIs (TCP/IP) is used to construct connectors in distributed systems for delivering messages in network. But they do not have the ability to build the relationship of components. On the other hand design pattern is most effective for building relationship when combined with aspect-oriented programming (AOP). However, AOP is merely applied to classes situated in local applications. As mentioned previously in section 2.6, AOP cannot be directly used in distributed systems. Furthermore, functionalities and qualities of connectors may be seldom discussed in the design process of connectors, particularly when these technologies are put together for solving problems.

In this chapter, we focus on these design issues: in order to fully satisfy the functionalities and quality attributes of connectors, how to analyze and design connectors in distributed systems by weaving the relevant technologies together, such as design pattern, aspect-oriented programming, messaging system, network protocol and so forth. In the following content we present a new description about connectors. Next, we propose our design model to design connectors using EADD method in distributed systems. Finally, we perform an analysis of our design model.

4.2 New Definition of Connector

Connectors residing in distributed systems are frequently required to satisfy full functionalities: the transport of information, description of relationship, presentation of information, etc.

Moreover, according to requirements of systems, connectors should normally be designed to achieve some system quality attributes, such as availability, modifiability, performance, security, testability, usability, and so forth. Today, these attributes have become key determinants of design of connectors.

Thus, according to the functionalities and quality attributes of connectors, we provide a definition of software connectors:

Connectors with high quality attributes are capable of linking components together, which implements information delivery by carrying wrapped data in certain formats; the delivery is based on some types of dependence (relationship) between components.

This definition covers all aspects of connectors including the quality attributes. In particular, it is well suited to the connectors in distributed systems. In the following content (sections 4.3 and 4.4), our design model is created according to the definition.

4.3 Life Cycle Model of Connectors

In distributed systems the working process of connectors can be separated into some working phases. Different phase has separate functionalities and corresponding quality attributes. In order to fulfill these requirements (functionalities and quality attributes) of connectors and based on finite state machine (FSM) theory [Wagner *et al.*, 2006], we introduce a **Life Cycle Model** (called **LCM** model) of connectors which consists of 6 basic states: idle, connecting, disconnecting, active, modifying and error. These states can be used to represent all possible situations of connectors during working process. Below, we describe each state and relevant functionalities and quality attributes in detail.

In idle state, connector is in an initial situation. It has nothing to do but to wait for the start commands.

In connecting state, connector initiates the connection between components. In particular, it needs to build the relationship of components and should prepare all factors

(for example the password for accessing a system) for data transmission in active state. In order to satisfy the requirements, the working process of connecting depends largely on the reliable and stable mechanisms of delivery. Thus the availability and security that both belong to system quality attribute are the critical architecture drivers in connecting state.

Disconnecting state is responsible for closing the connection among components. In this state, the allocated software and hardware resources should be released.

Connector in active state keeps the connection built in connection state and provides data transmission between components. According to different requirements, availability, performance and security may be taken into account in design process.

In modifying state, the parameters of connector can be modified, such as the size of packet, the format of data, the response time etc. Modifiability should be supported in the state.

Error state is in charge of fault process, such as the fault recovery. Thus, the availability is the main system quality attribute which should be achieved in this state.

In order to describe the behavior of the Life Cycle Model, we draw a state diagram (Figure 4.1) that presents the 6 basic states of connector and the transition between them in distributed systems. In the state transition diagram, after idle state starts to initiate a connection, the state of connector switches from idle to connecting; then according to the result of connecting process, the state of connector can go to active or error state; in the active state the parameters of the connector can be changed by shifting the state from active

to modifying; or the state goes to error state when meeting an exception; finally, connector can shift the state from active to disconnecting and then it can come back to idle state.

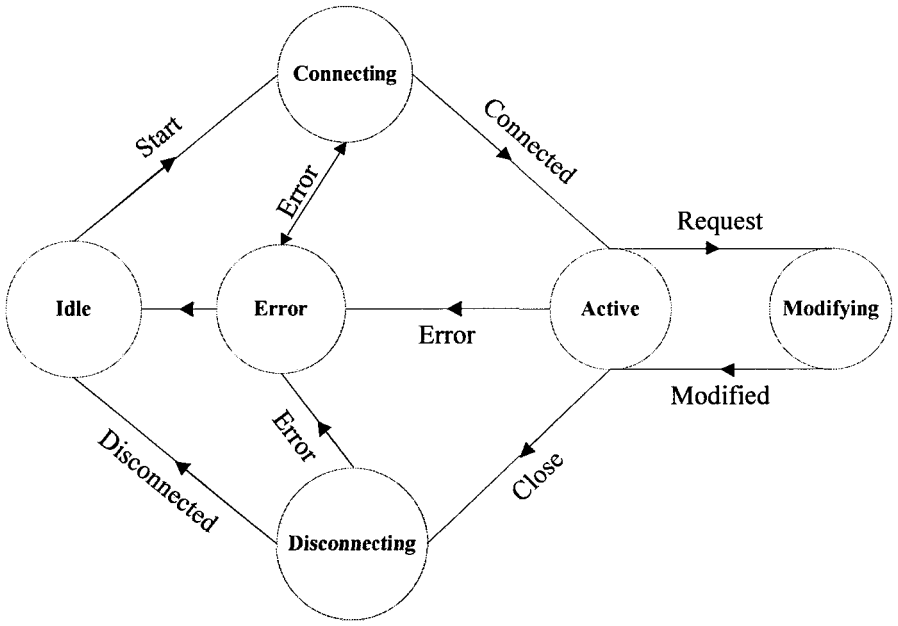


Figure 4.1: State Transition Diagram of LCM of Connector

In LCM model, connector's state: connecting, active and disconnecting are directly related to the relationship of components, mechanism of delivery, and the format of data. Hence, the three states of connectors are more complicated than the other three states (idle, error and modifying). In addition, there are many system quality attributes: availability, performance and security that are considered as significant quality attributes when designing and implementing them. As a result of this consideration, we continue to develop a more detailed design model in following section based on LCM model of connectors by using EADD design method.

4.4 Layered Design Model Based on EADD and LCM

In this section we focus on the design issues of connecting, disconnecting, error, modifying and active states of the Life Cycle Model (LCM model) of connectors. According to the requirements and characters of these critical states, we develop a design model of connectors in detail for each state of LCM model by using EADD method in distributed systems. The design model applies layered structure to each state of LCM. Thus, it is named as **Layered Design Model** (LDM).

4.4.1 Active State of Connector

Below, we present the design process for designing the architecture of active state by using EADD design method.

Step 1: collect the architectural drivers: specific quality scenarios, functional requirements and design constrain (Table 4.1). As mentioned early, architectural drivers are defined as the combination of functional requirement and quality requirements.

Quality Attribute	Scenarios
Availability	Connectors can also deliver the messages when the network is in bad situation or even the component is shutdown.
Modifiability & Scalability	When designing and implementing a new connector, the existing design and implementation of components should not be modified too much.
Performance	Connectors can provide efficient service for the delivery of messages.
Security	The connector can protect the messages and data transferred against the attack by an unauthorized attempt.
Testability	Most parts of design and implementation of the connector can be independently tested and verified. The internal state and inputs of the connector can be easily controlled and then the outputs of connector should be also observed.
Functionality	
Connectors must link components and transfer data and control by using some formatted information according to the dependence (relationship) between components	
Constraints	
Components which are linked by the connector can be situated in network or one application. The platform in which the components are situated must be Windows and Linux compatible.	

Table 4.1: Architecture drivers of active state

Step 2: choose the module to decompose. We consider the active state of connector as the primary element.

Step 3: identify chosen architectural drivers. From the step 1, multiple quality attributes (for examples availability, performance, modifiability etc.) should be taken into account in the active state. Even though the listed quality attributes are all important for our

design model, they may not remain the same priority according to the requirements in concrete projects.

Step 4: choose the patterns and tactics to satisfy the architectural drivers. According to the requirements of multiple quality attributes and the purposes of our design model, we apply a layered architecture pattern [Saleh, 2009] to the design process.

Based on Saleh's research [2009], layered software architecture is typically adopted to distribute software into different layers for reducing the complexity of software design. Because software is distributed in separate layer, the corresponding functionalities and quality attributes are also separated into layers. Thus, according to the architectural drivers, we propose a layered model called as "Connector Stack" which comprises three modules (or called layers): *Transport Layer, Dependence Layer,* and *Presentation layer*. Figure 4.2 shows the layered structure of connector stack, related development approaches, and quality attributes.

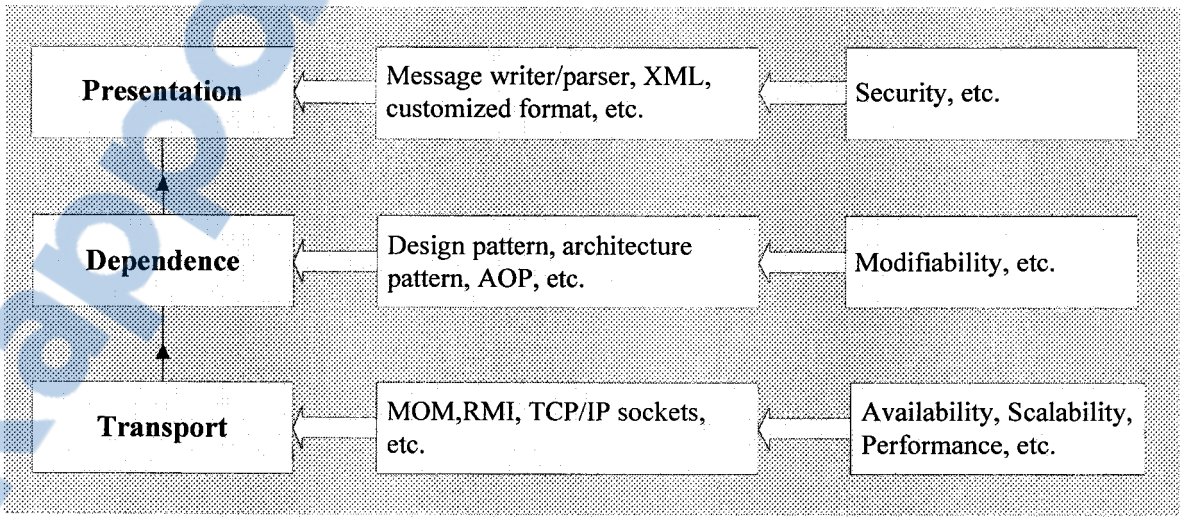


Figure 4.2: Connector stack, related development approaches, and quality attributes

Transport layer (module) situated in the base of the connector is responsible for data transmission through the network. It is used to support the dependence or relationship between components by linking them together. Typically, it is implemented by network protocols (such as TCP/IP, HTTP protocol stacks) or some kind of messaging systems (WebSphere MQ/ Open Message Queue etc.). Messaging systems can satisfy the availability and scalability, because that messaging systems have the ability to provide the reliable communication and support high scalability for a large number of clients. However, there is a side effect for messaging system: poor performance. Thus, if the performance is highly critical to the system, the socket API (TCP or UDP) should be chosen for delivery.

Dependence layer (module) describes relationship between components. In order to build relationship, we can use object design patterns or architecture patterns to design it, for example the Publish-Subscribe pattern. Without modifying much of existing components, the construction of the relationship among components can derive great benefit from the combination of Aspect-oriented programming and design pattern. In particular, when designing connectors in legacy systems, we can adopt program slicing technology to automatically locate the interesting values in source code statements of components which should be provided for other components through desired connector. In this way, the modifiability can be achieved by composing these approaches. In the following section 4.5, we will show the use of those approaches in detail.

Presentation layer (module) is concerned with messages exchanged between components, based on the type, format, amount, and meaning of data. We can use Markup

language such as XML or a customized format defined by developers according to the type and amount of the data. In addition, secure encryption of data and messages are always required in this layer. To achieve them, composition of the development tools and algorithm can help to satisfy the security.

"Connector Stack" provides an approach to designing and analyzing the active state of connector by dividing the state into separate modules (layers). Each module is an important part of connectors. They affect each other and are also independent of each other. Additionally, in this way these modules can be tested and verified, respectively. In other words, the testability is supported.

Step 5: instantiate architectural modules and define responsibilities. In this step, we instantiate the modules achieved in step 4.

i) In presentation layer, we adopt a virtual machine approach to achieving the modifiability quality in light of the main tactic of the virtual machine (localize changes). In Figure 4.3, the parsing virtual machine is responsible for understanding all formats of packet data. If the system needs to add new type of packet data, the virtual machine is only one module that should be modified.

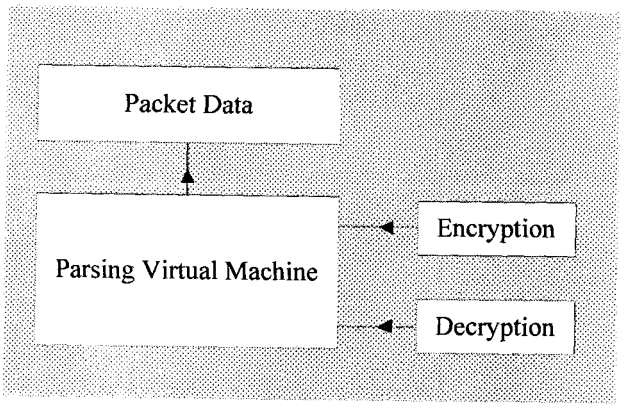


Figure 4.3: Presentation layer of active state

The format of packet data is the meaning and logic of messages which are also called message protocol. For example, if one component wants to access a database, it may use SQL language to describe the logic of message content. In Figure 4.3, the packet data module is used to input or output the packet data. It is normally considered as non-performance critical computation.

With the aim of improving the security, two related modules are instantiated. They are charge of encryption and decryption. These modules may be a performance-critical module or not, depending on the complexity of algorithm.

Based on the EADD design method, after instantiating the modules, we can select the development tools for each corresponding instantiated module in order to achieve better quality attributes. In this way, we obtain the output of EADD about development tools selection (Shown in table 4.2). In this table we assume that the modules Encryption and Decryption are performance critical modules (high complexity of algorithm).

Module	Quality	Development tools
Packet Data	Non-Performance	Applications modules, do not affect the quality
Parsing VM	Modifiability	Considering the "localize changes" tactics (more modifies may happen), should choose tools which support slice or crosscut for analysis and modification in future, such as Indus Slicer.
Encryption Decryption	Performance Critical	Choose efficient tools, such as compiled language: C++

Table 4.2: Selection of tools for presentation layer

ii) In dependence layer, we apply two types of modules: proxy and software pattern. Below, we show a diagram about the design of the dependence layer (Figure 4.4).

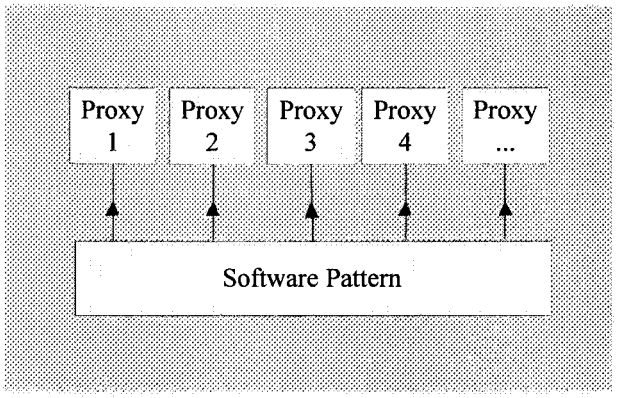


Figure 4.4: Dependence layer of active state

Firstly, we talk about the software pattern module. In our research, software pattern consists mainly of design pattern and architecture pattern that can be both employed to build the relationship of components. For example, we apply a design pattern to build the relationship of components in distributed MVC architecture [Qiu, 2005] [Lee, 1994]. When the data maintained in Model is changed, the Model must notify the Views of the changes. Thus, Publish-Subscribe pattern [McGovern *et al.*, 2003] is suitable to build the relationship between Model and View. For the detailed design, we use Observer design

pattern [Hannemann *et al.*, 2002] to implement the Publish-Subscribe and apply AOP to implement the Observer design pattern. The reason why we prefer utilizing AOP is analyzed below: the component Model should concentrate on the data processing of application and all the functionalities of Model should be separated from all things related to display and user interfaces that should be done by component View. In other words, the developers of Model do not need to know when to notify Views and what data should be delivered to Views. Fortunately, the AOP enables developers to establish the relationship between two components (Model and View) by crosscutting the two components without modifying much of the logic of both components. Particularly, when planning to update one MVC architecture by adding a new connector between components (for example, a new View wants to get some information from an existing Model.), the developers can get maximum benefit from the usage of AOP. From this point of view, AOP technology helps to achieve the modifiability.

Secondly, we discuss the proxy modules. In the dependence layer, proxies represent remote components. Figure 4.4 shows the arrangement of those proxy modules for illustrating the design idea. But in practice one proxy may correspond to multiple remote components, according to different requirements. The relationship between proxy and component (local) reflects the relationship of corresponding components (remote). For example, Figure 4.5 shows that the relationship between component A and proxy B represents the relationship of component A and B. The proxy B and component A are situated in one program that runs in a same computer. It means that they share the same memory space and CPU. One purpose of proxy is to relay the relationship between "A and

B" to "A and proxy B" in dependence layer. The other purpose of Proxy will be described in transport layer.

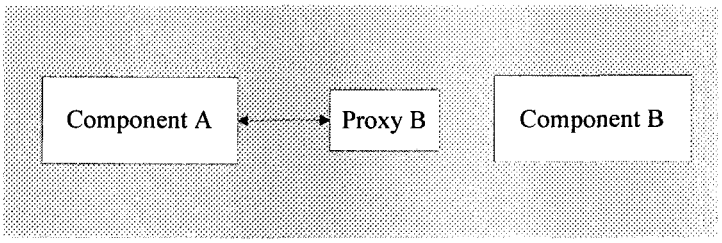


Figure 4.5: Proxy and components

Like the development tools selection in presentation layer, we apply the selection method of EADD to the dependence layer. The result is shown below in Table 4.3. Particularly, the combination of AOP and program slicer is logically capable of helping to achieve modifiability for designing new connectors of legacy system in software maintenance. We will talk about the typical tools and approaches later.

Module	Quality	Development tools
Proxy	Non-Performance	It is used to help to work with AOP. Thus the tools should be same as them in software pattern module.
Software Pattern	Modifiability	AOP, such as AspectJ; Program slicer, such as Kaveri.

Table 4.3: Selection of tools for dependence layer

iii) In transport layer, according to the quality of network and requirements of whole system, developers choose one or more type of middleware to implement the task of transport in network.

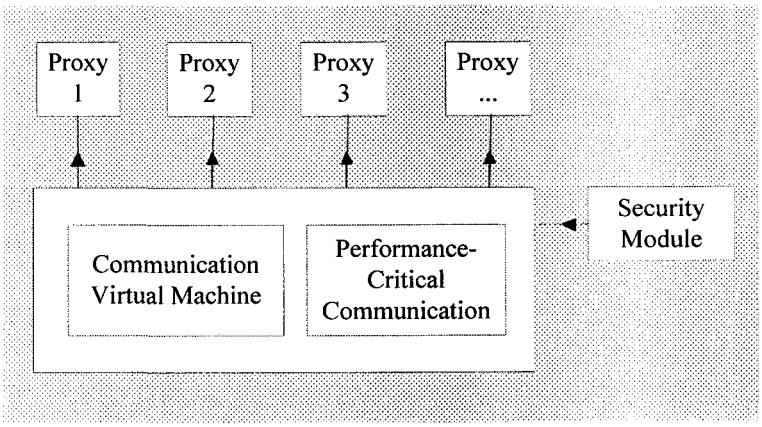


Figure 4.6: Transport layer of active state

In Figure 4.6, the module proxies are described in the design of dependence layer (shown in Figure 4.4). Here, we talk about another purpose of the Proxy. It is used to send package data to its represented component (remote) by using an instantiated module (communication VM or a performance-critical communication). During the delivery process, the security attribute should be taken into account depending on requirements.

Communication virtual machine is made up of MOM or other messaging systems. It is responsible for stable message delivery in an unstable network environment. In order to achieve the modifiability (easy to configure and change the transport mechanism), we use this VM to limit all changes to the module.

However, the communication VM has its own limitation: low performance. Thus, we provide an alternative approach to communication. It is implemented in performance-critical communication module in which applications can directly call the interface of network protocols (for example, using socket API) for enhancing the performance.

Security may also be a key quality attribute in transport layer. Even though it is done in communication VM, developers are often meant to design an independent module to achieve high security.

The module proxy plays an important role in our design model. It works as a "relay": the relationship of components is relayed to network by the proxy; next, it links the dependence layer and transport layer together.

Like the development tools selection in the other two layers, we apply the selection method of EADD to this layer. The result is shown below in Table 4.4.

Module	Quality	Development tools
Proxy	Non-Performance	Proramming tools should support AOP and have efficient API of communication.
Communication VM	Modifiability & Availability	MOM, JMS etc.
Performance-Critical Communication	Performance	Socket API (TCP/UDP)
Security module	Security & Performance Critical	Choose efficient tools, such as compiled language: C++

Table 4.4: Selection of tools for transport layer

After Step 5, the design model for active state of connector is complete. According to EADD methods, the Step 6 (define interfaces) and Step 7 (refine requirements) should be accomplished in an actual project by following different requirements.

In the end we provide a whole architecture of active state for designing connector by joining the three layers together (shown in Figure 4.7). The Figure 4.7 shows that the

architecture comprises 9 modules: picket data, parsing VM, encryption, decryption, software pattern, proxies, security, communication module and performance-critical communication module. Each of them is designed to satisfy functionalities and achieve quality attributes: the parsing VM and communication VM module are employed to support modifiability; two kinds of security modules situated in presentation and transport layers are responsible for security attribute; the software pattern modules is charge of the construction of relationship; availability is achieved by adopting the communication VM which consists mainly of MOM; proxy modules are applied to representing components in order to build relationship in dependence layer and take part in delivery in transport layer.

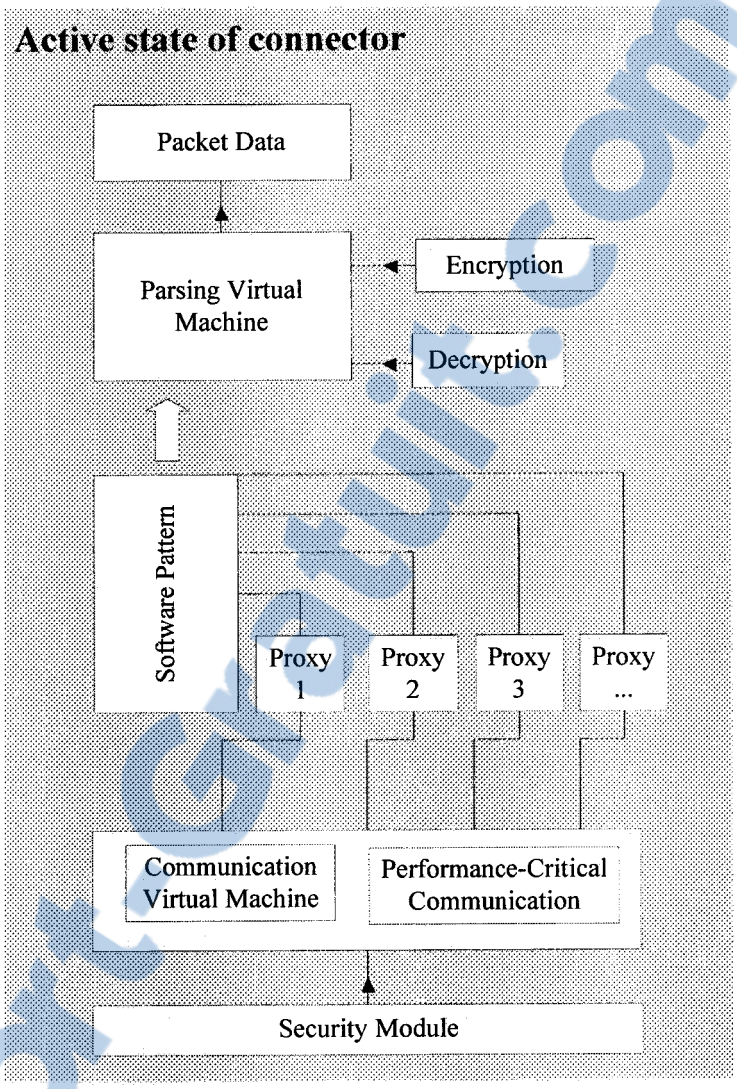


Figure 4.7: Architecture of active state for designing connector

Based on the EADD design method, we get not only the decomposed modules but also corresponding development tools (such as AspectJ, JMS C++, Indus, MOM, Socket API and so forth). As a result, these selected development tools greatly improve the quality attributes in the implementation stage.

4.4.2 Connecting States of Connector

In the connecting state of connector, availability and security are both the critical architecture drivers. In this section we show the architecture design of the connecting state by following EADD method.

Step 1: collect the architectural drivers: specific quality scenarios and functional requirements (Table 4.5).

Quality Attribute	**Scenarios**
Availability	When connector fails to initiate the connection, in the connecting state, connector should detect the error and make a recovery from the operation.
Security	During the process of initating connection, connector should start without being attacked.
Functionality	
Connector needs to build the relationship of components and should prepare all parameters for data transmission in active state.	

Table 4.5: Architecture drivers of connecting state

Step 2: choose the module to decompose. We choose the connecting state as the primary element.

Step 3: identify chosen architectural drivers. According to the architecture driver of connecting state, two quality attributes (availability and security) are taken into account.

Step 4: choose the patterns and tactics to satisfy the architectural drivers. In order to achieve the availability quality attribute, we apply Ping/Echo pattern to making sure that the communication path between components is available. Next, the security can be achieved by using limit access pattern for resisting attacks. Figure 4.8 shows an architecture

pattern which is the combination of the tactics listed above. In the figure, like the transport and presentation layers in the active state, we also employ a transport and a presentation layers in the architecture pattern of connecting state.

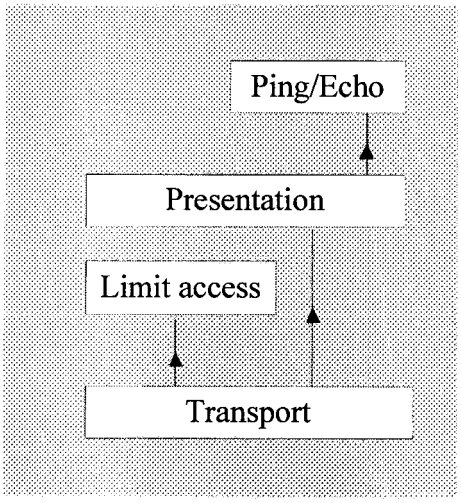


Figure 4.8: Architecture pattern of connecting state for designing connector

Step 5: instantiate architectural modules and define responsibilities. In this step, we instantiate the modules analyzed in step 4.

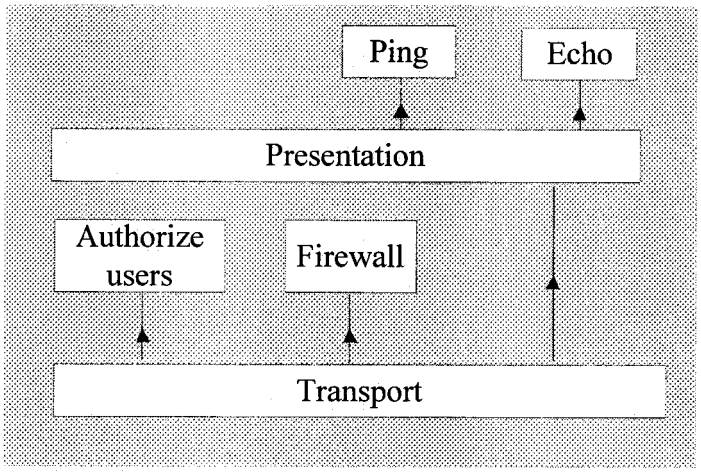


Figure 4.9: Architecture of connecting state for designing connector

The figure 4.9 shows that the design model of connecting state is also a layered structure like the architecture of active state. Compared to the active state, the architecture of connecting state does not have the dependence layer, because there is no relationship at the beginning of initiation. Next, we design Ping, Echo, Firewall etc. modules to achieve the availability and security quality attribute. Based on the EADD method, we should select the development tools for improving the quality attribute of design in the end of this step. Because that the presentation and transport layers are also discussed in active state, we only present the tools selection of other modules of connecting state (as shown in Table 4.6).

Module	Quality	Development tools
Ping	Availability	For measuring the expected performance bound, excellent timer mechanisms are required.
Echo		
Authorize User	Security	The tools should provide access control patterns for security [Delessy et al., 2007].
Firewall		

Table 4.6: Selection of tools for connecting state

According to EADD methods, the Step 6 (define interfaces) and Step 7 (refine requirements) should be done in an actual project by following different requirements. Thus, the model for designing connector in connecting state is completed.

4.4.3 Disconnecting States of Connector

In disconnecting state, the connection is closed. Moreover, the allocated software and hardware resources should be released. When closing a connection across network,

connector may need a period of time to complete the closing process. Typically, this period of time is called as timeout. In order to correctly shut connector down, timeout is an approach to achieving the availability in disconnecting state. In the step 5 of EADD, we obtain the result of instantiated architectural modules (shown in Figure 4.10). Figure 4.10 shows that both modules (release of software and hardware resources) act based on timeout mechanism.

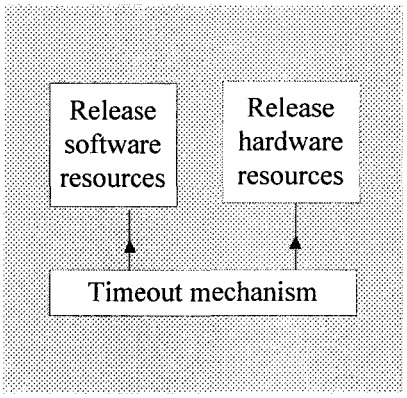


Figure 4.10: Architecture of disconnecting state for designing connector

Based on the EADD, we can obtain the development tools for disconnecting state (as shown in Table 4.7).

Module	Quality	Development tools
Timeout	Availability	For measuring the expected time, a good timer mechanisms are needed.
Release software resources	None	do not affect the quality
Release hardware resources		

Table 4.7: Selection of tools for disconnecting state

In contrast with active and connecting states, disconnecting state is not very complicated. Particularly, some connectors do not need to shut down in their working process. Hence, we just briefly describe the design model for disconnecting state.

4.4.4 Error and Modifying States of Connector

Error state is responsible for system recoveries and repairs of systems, when an error occurs and is detected in other states of connector. In light of the LCM model, once an error is met in active, connecting and disconnecting states, these states will normally throw exception and then jump to the error state. At this moment, error state can output error information. And then the state switches from error to connecting or to disconnecting state. The switch decision (to restart or close connection) depends on the requirements of system. The availability is the main architectural driver in error state. But it must be combined with other states to achieve the quality attribute.

Like error state, the modifying state is designed to support the modifiability by working together with active state. It is only responsible for configuring the parameters of active state during runtime or compile-time.

Because the error and modifying states do not directly relate to quality attributes and the logic behind them is easily understood, we are not meant to perform the further analysis and do not create the detailed design about them.

4.5 Typical Approaches to Use of LDM Model in Practice

In addition to the architecture design, our model introduces several development tools and involves multiple approaches, patterns etc. in high level for future implementation in practice. This section highlights the key tools and approaches that will be used in implementation and maintenance phases. We especially examine the combination of them when designing new connectors in legacy systems.

4.5.1 Typical development Tools and Approaches

AspectJ is created to fulfill the purpose of Aspect-Oriented programming (AOP) for the Java language. Generally, it is regarded as an extension to the OOP of java. AspectJ that is made up of a compiler and a set of JAR files has capability to compile Java code of OOP and aspects code of AOP into standard Java byte-code, which can run on top of any JVM [Gradecki *et al.*, 2003].

The Gang-of-Four (GoF) design patterns [Gamma *et al.*, 1995] provide developers with flexible and practical solutions to overcoming software development problems. GOF defines 23 well-known design patterns. Each pattern comprises purpose, intent, applicability, solution structure, and sample implementations [Hannemann *et al.*, 2002]. Software developers can derive maximum benefit from them or combination of them to design the dependence layer of active state of connector.

Indus Java program slicer belongs to project of Indus [Indus project]. The Indus slicer is regarded as the first and only publicly available Java slicing framework which supports almost all features of Java [Ranganath *et al.*, 2006]. Kaveri is an eclipse plug-in [Holzner, 2004] that adopts the Indus slicer to perform slices of java programs. In addition, it can

show the sliced program in the IDE of Eclipse. Kaveri is an effective tool with GUI that helps to analyze source code and maintain program [Ranganath *et al.*, 2006]. Moreover, Kaveri plug-in has capability to automatically slice the java source code and present the data and control dependence. Based on the data and control dependence of program, developers can easily find the crosscutting point for designing new connectors in legacy system.

4.5.2 Implementation of Architecture Design

The multiple development tools and approaches should combine to create program of connectors in implementation and maintenance phases based on our design model. Particularly, in order to reduce the cost of maintenance, implementing a new connector in existing system involves more development tools. For illustrating the use of these tools and key steps of implementation, we show an example about how to combine the development tools to implement a connector in maintenance phase.

In the example, we assume that the architecture design is completed by following our design model and the example only highlights the use of those key tools in implementation. We choose Java and AspectJ as main developing language; Observer design pattern is adopted to build the dependence layer of active state of connector; next, we employ UDP/IP socket API in transport layer of connecting and active states; and we utilize Kaveri plug-in of Eclipse to slice source code of legacy program to get "pointcut" for implementing connector. Below, we select a section of source code from one existing component:

```
public class Employee {
    private String name;
    private int age;
    private String numberID;
    private double salary;
    private int officeID;
    private String officeName;

    public String getName(){      return name;   }
    public void setName(String newName){ this.name = newName; }
    public int getOfficeID(){ return officeID; }
    public String getOfficeName(){ return officeName; }
    public void setOfficeName(String newName){ this.officeName = newName; }
    public void setOfficeID(int newID){ this.officeID = newID; }
    private void notifyHR(){ … }
    public void updateOfficeInfo(int newID, String newName){
            notifyHR();
            setOfficeID( newID);
            this.officeName = newName;
            if (newID > 800)
                    increaseSalary();
    }
    public void ChangeOffice(int newID, String newName, String newNumberID){
                    updateOfficeInfo(newID, newName);
                    this.numberID = newNumberID;      }
```

```
    private void handleRetire_1(){        }
    private void increaseSalary(){ salary = salary + 1000; }
    private void handleCommand(String cmd){  ...      }
    private void saveCommand( String cmd){        }
    public void giveChangeCommand(int newID, String newName, String
newNumberID, String cmdName){
        handleCommand(cmdName);
        ChangeOffice(newID, newName, newNumberID);
        saveCommand(cmdName);
    }
    private void process_1(){     ...     }
    private void process_2(){     ...     }
    public void retire(){
        if ( age > 60 ){
            this.officeName = "empty";
            handleRetire_1();
            if ( age > 65){
                process_2();
            }
            process_1();
            increaseSalary();
        }
    }
}
```

In the simple component, there is one class (Employee) that organizes all information of an employee, such as name, age, office ID and salary. But now, the system has a new

requirement: a component (Manager) situated in network requires to be notified when an employee's office ID is modified. Thus, according to the new requirement, we must design a connector between them in the existing system.

After producing the architecture design using our design model, we also obtain the development tools and related approaches for future implementation. Among these approaches, we choose Observer design pattern to build the relationship between Employee and Manager. In order to use Observer design pattern and AspectJ, we should firstly find the point where the office ID is changed in the program. The point is called as pointcut according to AOP.

We assume that the code of Employee is very large and complex. As a result, if we manually read and analyze the whole source code and there are no enough documents related to the program, we maybe take more time to understand the program or finally find some wrong statements. Thus, we perform a program slicing for the system using Kaveri. We pick statement "*currentOfficeID = person.getOfficeID();*" as a criteria which is in main method. Next we choose "value of the expression" and set "backward program slicing". Then start the slicing process. After a short while, the slice result is described as below:

```
public class Employee {
    private String name;
    private int age;
    private String numberID;
    private double salary;
    private int officeID;
```

```
    private String officeName;

    public String getName(){  return name; }
    public void setName(String newName){      this.name = newName;        }
    public int getOfficeID(){       return officeID;  }
    public String getOfficeName(){       return officeName;    }
    public void setOfficeName(String newName){ this.officeName = newName; }
    public void setOfficeID(int newID){          this.officeID = newID;        }
    private void notifyHR(){       ...       }
    public void updateOfficeInfo(int newID, String newName){
            notifyHR();
            setOfficeID( newID);
            this.officeName = newName;
            if (newID > 800)
                    increaseSalary();
    }
    public void ChangeOffice(int newID, String newName, String newNumberID){
                    updateOfficeInfo(newID, newName);
                    this.numberID = newNumberID;      }
    private void handleRetire_1(){        }
    private void increaseSalary(){ salary = salary + 1000;  }
    private void handleCommand(String cmd){  ...       }
    private void saveCommand( String cmd){             }
    public void giveChangeCommand(int newID, String newName, String
newNumberID, String cmdName){
            handleCommand(cmdName);
```

```
            ChangeOffice(newID, newName, newNumberID);
            saveCommand(cmdName);
    }
    private void process_1(){       ...       }
    private void process_2(){       ...       }
    public void retire(){
            if ( age > 60 ){
                    this.officeName = "empty";
                    handleRetire_1();
                    if ( age > 65){
                            process_2();
                    }
                    process_1();
                    increaseSalary();
            }
    }
}
```

From the slicing result, it is easy to find the statements marked in highlight green that are all points which are related to the changed value of office ID. Kaveri helps to filter out unwanted code using program slicing technology. Hence, the sliced program confirms that *giveChangeCommand()* calls *ChangeOffice()*; *ChangeOffice()* calls *updateOfficeInfo()*; and then *updateOfficeInfo()* calls *setOfficeID()*. Because the office ID is merely modified in *setOfficeID()*, we only need to put the pointcut around statement "*this.officeID = newID;*" avoid taking other methods into account. In this way, we reduce the time of analysis and

are able to locate the pointcut quickly. At last, the pointcut is set in aspect file using AspectJ, described as below:

```
/* Set the pointcut. Advise the method setOfficeID (). */
Protected pointcut subjectChange(Subject subject):
  call( void Employee.setOfficeID () ) && target(subject);
```

After creating all code for the new connector, the existing component Employee is not changed. It is kept very clean in the implementation. We only added some new files about aspect of AOP to the system and the implementation can be easily removed or changed without affecting the logic of original components.

4.6 Analysis of Design Model

The purpose of our model is to design architectures of connectors and to select the development tools in distributed systems. Functionalities and quality attributes are two significant requirements that drive the architecture design of connectors and help to select related development tools by following our design model. According to different design phases, our design model is divided into two parts: Life Cycle Model (LCM) and Layered Design Model (LDM). In the following content, we analyze the design model in 3 aspects.

4.6.1 Design Model in Software Development Process

Software development process is defined as a group of activities which are organized and managed to build the software system [Puntambekar, 2009]. In the software development process, software life cycle and related processes (assessment and

measurement) are adopted to divide the whole development process into separate phases for controlling and monitoring development. They comprise waterfall model, throwaway model, incremental development, evolutionary prototype, reusable software, spiral model etc. [Comer, 1997] [IEEE/EIA, 1996].

In the subsection we show where our design model is situated in the software life cycle and when we can apply our design model based on software development process by taking waterfall model for example. First, we present a waterfall model, described in Figure 4.11.

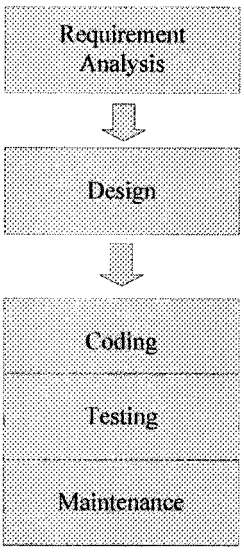


Figure 4.11: Waterfall life cycle model

In terms of the waterfall life cycle, the design model is situated between requirement analysis and design phase. In addition, it normally overlaps the design phase and the overlapped area is called as architecture design. Figure 4.12 shows the waterfall cycle after exerting our design model. The design process following the LCM model situated in architecture design (or called high level design) can be started after performing functional requirement analysis. Similarly, the design process following the LDM model should be

produced after performing the non-functional requirement analysis and completing general design (followed LCM).

In summary, our design model (comprising LCM and LDM models) belongs to high level design that mainly focuses on the design issue about architectures. The design work following the design model should be done before low level design and after requirement analysis.

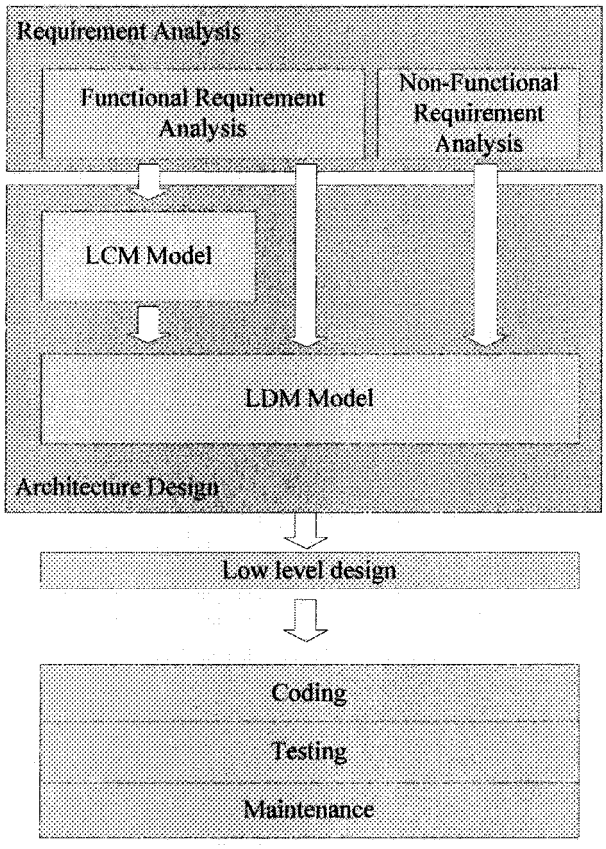


Figure 4.12: Waterfall life cycle model when combined with our design model

4.6.2 Comparison with Existing Approaches

Layered design model (LDM) employs a clearly layered structure in active, connecting, and disconnecting states of Life cycle model (LCM). In this way, the functionalities and

quality attributes are separated into layers and the design fulfills the architectural drivers by using EADD methods. Hence, each layer is responsible for satisfying its functionalities and corresponding quality attributes. Developers can be guided through the process of high level design of connectors by following our design model.

A major difference between our model and other approaches is that the model separates the architecture design into different parts (states and layers) that are independent and collaborative. These parts help developers design, implement, and understand all aspects of connectors. Particularly, connectors designed by following our model are capable of achieving high system quality attributes. In addition, the development tools can be decided in the high level design for supporting implementation (coding, testing, and maintaining). In contrast, other research works regard network protocols (TCP/IP, P2P etc...) or messaging systems (middleware solutions) as connectors without considering that presentation of data and independence of components belong to the knowledge area of designing connectors. As a result, this thinking can lead developers to concentrate merely on the functionalities in delivery and result in neglecting to study and design the relationship of components, the presentation of data etc.

Active state is a main state of LCM model. And relationship layer is the core of the active state in terms of LDM. When adopting this design model, developers must put a lot of effort into the research of relationship among components. To simplify the analysis of relationship, pattern approaches combined with AOP are applied to the design of connectors in distributed systems. Middleware (especially messaging systems) and other

transport mechanisms are designed to ensure that the messages can be successfully delivered. In our opinion, the middleware should be organized in transport layer, even though some of them have capability to describe the independency among components.

4.6.3 Analysis of Classification of Connectors

As mentioned early in chapter 2, classification of connectors helps to choose proper connectors between components. However, current research on the classification often varies, depending on the level of understanding of connectors. This section attempts to analyze the classification of connectors based on our design model and definition of connectors.

According to our description (shown in section 4.2), connector provides four fundamental functionalities: integration, division, extension, and distribution. According to the functionalities, we identify four types of connectors. From quality point of view, these types of connectors possesses certain types of system quality attributes, such as availability, modifiability, performance, security, testability, usability, and so forth. These quality attributes may be achieved in different states of the LCM model as discussed above. Below we give the general description and discuss some quality attributes and functionalities for each type of connectors.

Integration connector is employed in the process of combining two or more components so that they work together effectively. The combination of components largely depends on the connector that is designed using our designing model. Generally, availability is a critical attribute for integration connector, because the connector requires

every component is available and the transport of active and connecting states should work under any kind of situation for providing integrated services for system.

Division connector is used to separate a system into two or more different components. The division is created based on requirements of system. For example, a web server may crash under a heavy load. Hence, one solution is to split it into multiple components residing in network. As a result, division connector has capability not only to link the components together but also to schedule these divided components.

Extension connector is adopted to add new components to an existing system. Extension is typically a process of software maintenance. For the act, the modifiability is an important quality attribute of extension connector. According to the development tools and approaches selected by using our design model, the design and implementation of the type of connector should be limited in new connectors, instead of existing components.

Distribution connector is applied to the act of sharing the functionalities of components across network in a designed way. The connector helps to link authorized components for accessing a shared functionalities of components. Thus, security is the main quality attribute of distribution connectors for resisting attacks. Hence, the developer should put more effort on the design of connecting state of the design model.

4.7 Conclusion

In this chapter, we introduce a new description of connector and design model for designing connector based on EADD method. The model addresses the main design issue: for fully satisfying the functionalities and qualities of connectors, how developers build the architecture design of connectors in high level by weaving the multiple technologies together.

Firstly, our design model is produced to have two parts: LCM model and LDM model. The LCM model is constructed merely based on functionalities. According to the separate working stages and corresponding functional requirements, LCM divides connector into 6 states: idle, connecting, active, disconnecting, error and modifying. LCM model does not directly produce an architecture design. Instead, it provides an approach to decomposing connector and helps to obtain a rough structure for further design by applying detailed design model.

Based on the LCM, functional requirements and quality attributes drive to build LDM model using EADD. By following LDM model, developers have capability to build detailed architecture design of connectors and choose related development tools for each module of the design.

Secondly, we highlight the key tools and approaches in implementation phase. Particularly, we present an example about how the developers gain the benefit from the combination of the tools and approaches when designing new connectors of legacy system in software maintenance.

Finally, we discuss our design model from the software development process point of view for showing where our design model is situated in the software life cycle and when we can apply it. And then we compare the design model with existing approaches. The comparison highlights the features of our design model from the following aspects: the layered structure, analysis of relationship and implementation. At last, we analyze the effect of the design model on classification of connectors and attempt to give a classification of connectors based on our description and the design model.

CHAPTER 5

CASE STUDY: A PUSH MAIL SOLUTION FOR WIRELESS NETWORK USING THE DESIGN MODEL

5.1 Introduction

In previous chapter, we presented our definition of connectors and proposed a design model for designing connectors in distributed systems. This chapter will show the benefit of using our model to design a connector in distributed systems. The connector is a solution to the push mail system in wireless network. As mentioned earlier in section 1.2, the desired connector is capable of linking email system (based on internet) with components running across wireless network. Figure 5.1 describes the position and the basic functionality of connector.

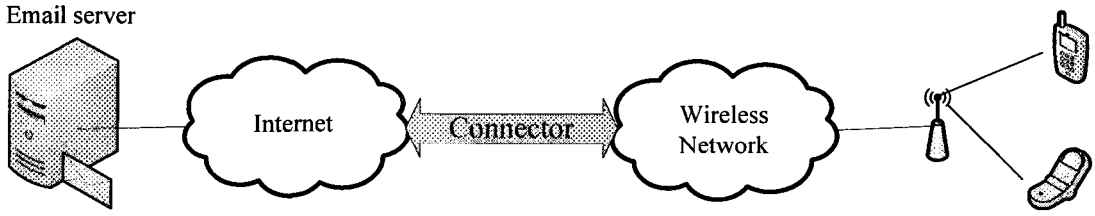


Figure 5.1: Connector between Internet and wireless network

The goal of the connector is to help email server push mails to mobile devices without more special support of wireless network operator. The specification of push mail system is presented as below:

- Email server should be independent of the push mail system;
- Wireless network is not required to adjust its deployment (including software and hardware);
- When there is no new mail, mobile devices are not required to access Internet;

- Changes can be easily made to the implementation of legacy systems.
- The system should provide users with secure and available services.

The rest of this chapter will demonstrate how the architecture of the connector in push mail system is designed by following our design model for meeting the specification.

5.2 Requirement Analysis

According to the specification of the connector, this section performs a requirement analysis for architecture design. The requirements of connector can be divided into 2 types: functional requirements and qualities attributes. Thus, we analyze the requirements of connectors based on this classification.

We produce a detailed requirement analysis (shown in Table 5.1) with respect to the specification of connector. Table 5.1 shows the functional requirements and quality attributes that correspond to the specification.

Specification	Functionalities	Quality Attributes
Email server should be independent of the push mail system;	Connector should provide standard interfaces for accessing email server without modifying the implement of email server.	Modifiability/Usability
Wireless network is not required to adjust its deployment	Connector has capability to link mobile device and mail server together using bypass working mode (without involving much more work of operators).	N/A
	Connector should run on top of normal wireless service.	N/A
When there is no new mail, mobile devices are not required to access Internet;	Connector can be allocated or released depending on the emergence of new mail.	N/A
	Connector should be installed in real time when new mail is coming.	Performance
Changes can be easily made to the implementation of legacy systems.	N/A	Modifiability
The system should provide users with secure and available services.	N/A	Security and availability

Table 5.1: Requirement analysis based on specification of connector

5.3 General Architecture Design of Connector

This sub-section presents a general architecture for designing connector of push mail system by following the Life Cycle Model (LCM). Based on the model, we can similarly design the 6 states of connector for the design of push mail system: idle, connecting, active, disconnecting, modifying, and error states.

In idle state, the connector is suspended and it is ready for new emails.

In connecting state, connector initiates the connection and builds the relationship between mail server and mobile devices. During the initiation, login information should be required for accessing the system.

Disconnecting state is responsible for closing the connection between mail server and mobile devices. In this state, the allocated software and hardware resources should be released. For example, this state should turn off online mode of devices and should clear all notification of new mail.

Active state keeps the connection and relationship built in connection state. It is employed to deliver all content or part of mail to mobile devices according to the relationship between mail server and mobile devices.

In modifying state, the parameters of connector can be modified, such as the size of packet, the format of data, the response time etc.

Error state of the connector is in charge of fault process. When exception occurs, error state should inform the system of the exception and examines it to decide what to do for error process. Typically, the connector is capable of restarting the delivery process. If it fails to restart the delivery, it tries to close the connector.

As described previously in Figure 4.1, the transitions among states are driven by different events. For example, when new email is coming, the state will switch from idle to connecting. According to the requirements of push mail and the LCM model, we put the descriptions of event to every transition of state for presenting the event driven process of the connector (shown in Figure 5.2).

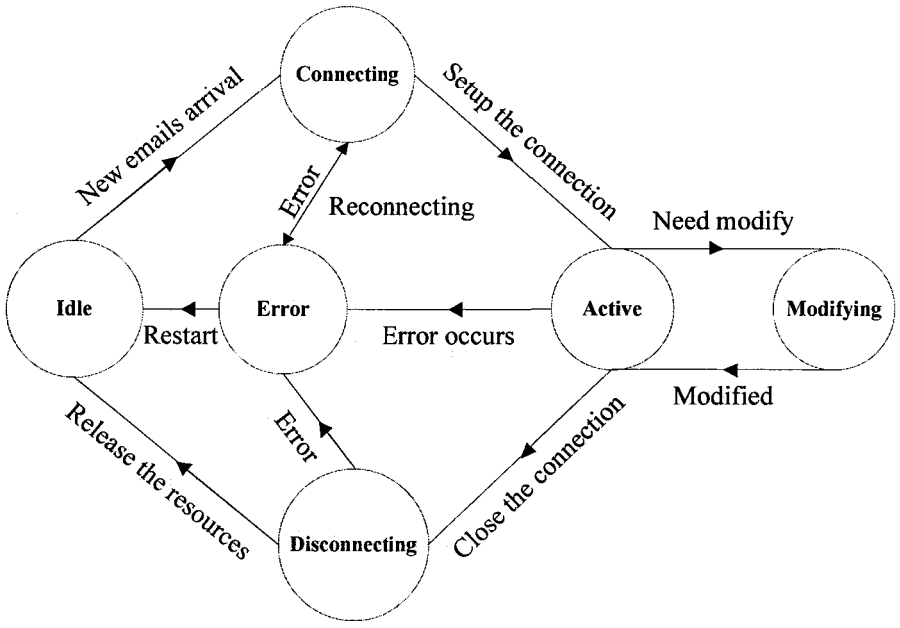


Figure 5.2: State Transition Diagram of Connector in push mail system

Figure 5.2 shows a general view for the architecture of connector. The each state is designed in the following content.

5.4 Detailed Architecture Design of Connector

In this section we produce a detailed architecture design of connector in push mail system using the Layered Design Model (LDM). The detailed design evolves from the general architecture so that it comprises 6 parts corresponding to the different states. Among the 6 states, connecting and active states are most complicated and important to the design of connector. Thus, in the following section we mainly focus on the two states for illustrating the detailed architecture design of connector.

5.4.1 Active State of Connector

According to our design mode, active state is designed as a layered structure that comprises transport layer, dependence layer and presentation layer. As mentioned earlier in chapter 4, each layer has its functionalities and possesses quality attributes that drive the architecture design. This sub-section creates the detailed design for each layer of active state, respectively.

i) According to the design model (shown in the Figure 4.3), presentation payer is made up of three modules: packet data, parsing VM and encryption modules. Packet data module is used to input/output email data. Moreover, it is adopted to select which parts of mail (such as email title, content, attachment etc.) should be chosen as the source packet of connector. Parsing VM module is designed as a virtual machine which has capability to compose or decompose the packets using XML format. We can apply DES to the encryption module for achieve security in this layer. Figure 5.3 shows the architecture of this layer by instantiating the corresponding model.

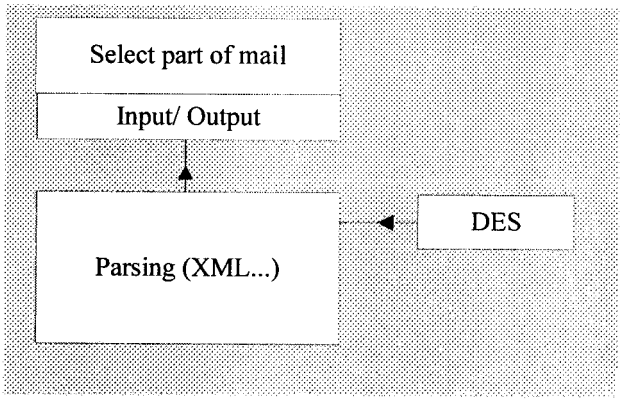


Figure 5.3: Presentation layer of active state for connector in push mail system

ii) Dependence layer comprises two kinds of modules: design pattern and proxies. Observer design pattern is employed to describe the dependence between mail server and mobile devices, because multiple devices may be interested in same messages. The proxies represent the mobile devices to build the relationship with mail server based on the observer pattern. Figure 5.4 shows the architecture of dependence layer by instantiating the corresponding layer in our model. And it also presents the position between mobile devices and mail server in practice.

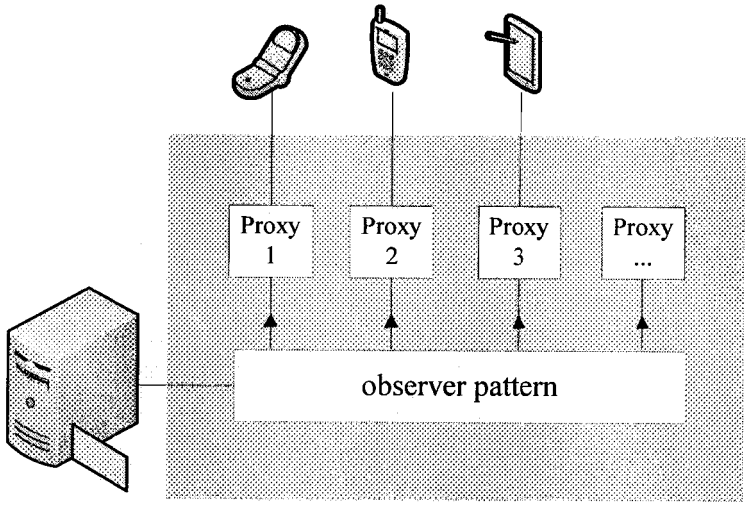


Figure 5.4: Dependence layer of active state for connector in push mail system

iii) In transport layer, there are two communication mechanisms: communication virtual machine (supporting message system) or network protocol API. Due to the limited resource of mobile devices, we adopt the network protocol API (Socket API) for obtaining the performance. Because the security of connector is already achieved in presentation layer and connecting state, the security module can be omitted in this design according to the requirements. The instantiated architecture of this layer is shown in Figure 5.5.

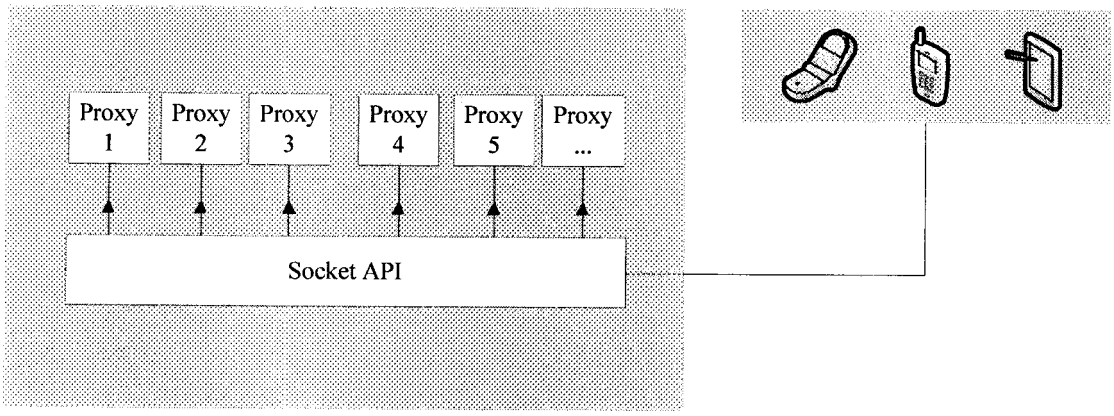


Figure 5.5: Transport layer of active state for connector in push mail system

At last we give an entire architecture for active state of connector in push mail system in Figure 5.6. Figure 5.6 not only shows the internal architecture of connector but also illustrates the external structure about how to setup it between mail server and mobile devices.

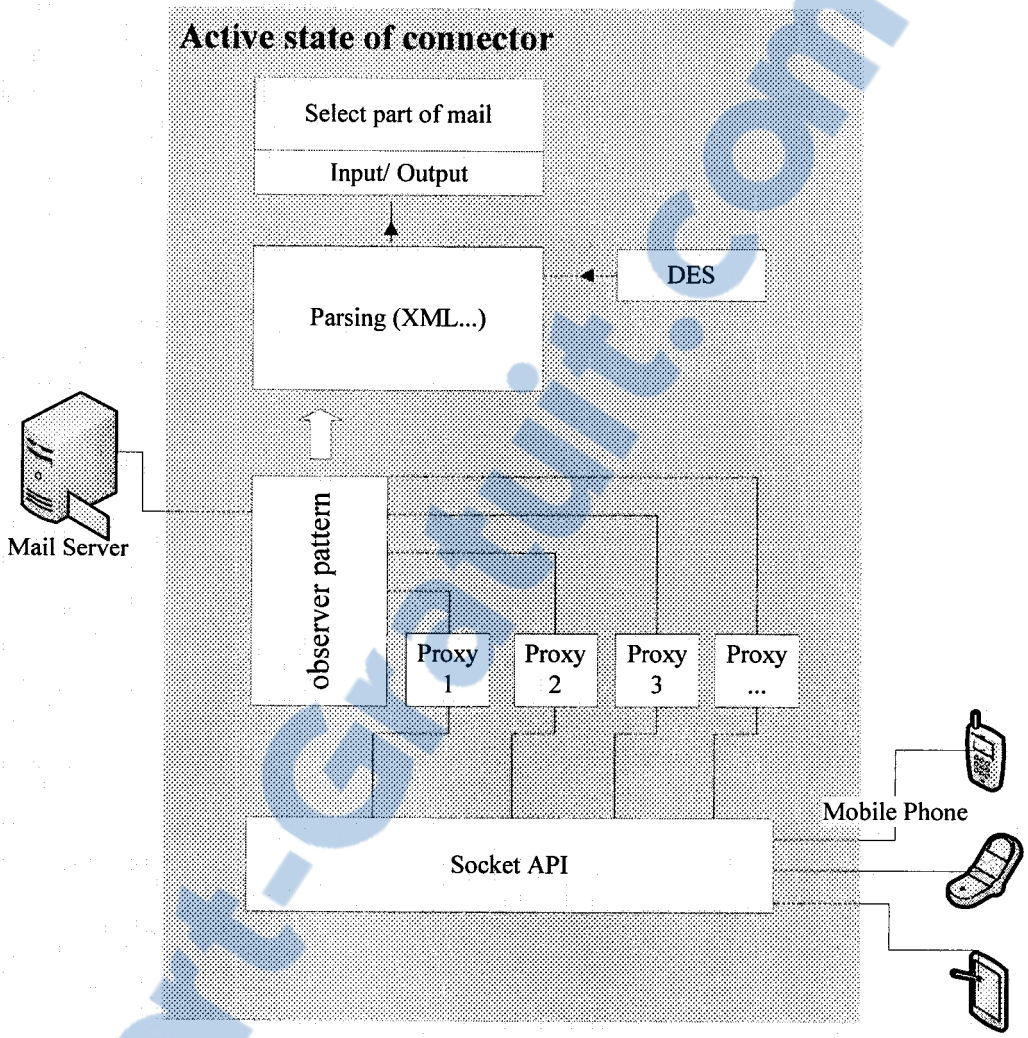


Figure 5.6: Entire architecture of active state for connector in push mail system

According to our design model, after producing the architecture design, the model helps developers select the development tools for the instantiated modules of connector in order to possess high quality attributes. Thus, we can get development tools for the each module shown in Figure 5.6 (shown in Table 5.2).

Module	Quality	Analysis of tools selection	Tools selected
Select part of mail Input/Output	Non-Performance	Applications modules, do not affect the quality	Java
Parsing VM	Modifiability	Considering the "localize changes" tactics (more modifies may happen), should choose tools which support slice or crosscut for analysis and modification in future.	Indus Slicer and Java
DES	Performance Critical	Choose efficient tools, such as compiled language.	C/C++
Proxy	Non-Performance	It is used to help to work with AOP. Thus the tools should be same as them in software pattern module.	Java and AspectJ
Observer Pattern	Modifiability	Aspect-oriented programming	AspectJ
Socket API	Performance	Tools have capability to directly invoke socket API.	Java

Table 5.2: Selection of tools for active state of connector in push mail system

5.4.2 Connecting State of Connector

Connecting state of the connector is used to setup the connection and help to build the relationship between mail server and mobile devices. Particularly, in the push mail system, when new mail arrivals, mobile devices should be notified of the event. However the devices may not be in packet switched mode (GPRS, 3G etc.). In other words, at this time the mobile devices are in circuit switched mode. Hence, it is difficult to allow the typical internet protocols to run on top of circuit switch mode for pushing messages. Our solution to the problem is to adopt SMS system [Clements, 2003] to notify mobile devices of the event of new mail. The detailed approach and architecture design are described below in this sub-section.

Firstly, we address our approach to the notification problem in push mail system. When the mail arrivals at email server, a mail user agent (MUA) detects it and sends a SMS to mobile devices using circuit switched mode without requiring data mode. And then the

mobile devices prepare to switch to data mode for receiving the new mail from mail server. After that, mobile devices may need to send back SMS with network parameters. The workflow of the connecting state is illustrated below (Figure 5.7):

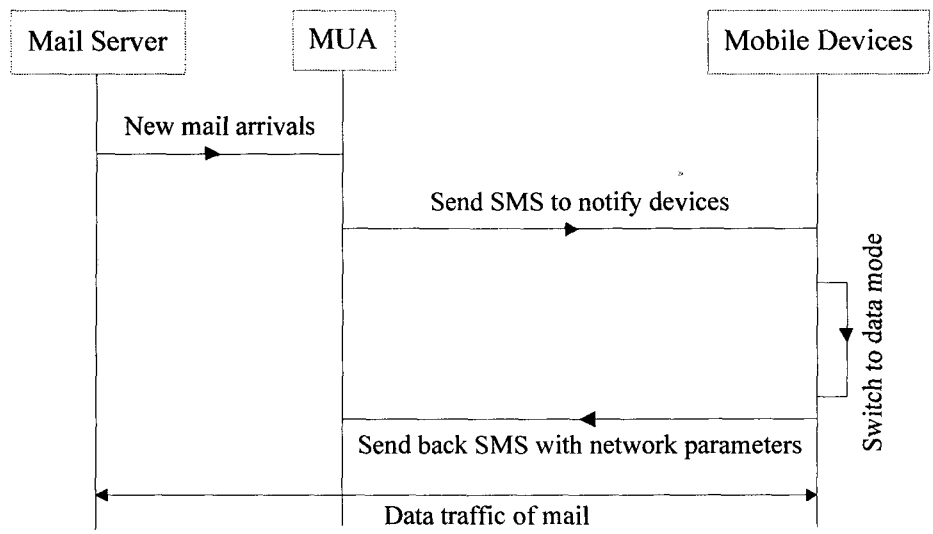


Figure 5.7: The workflow of the connecting state

According to the design model of connecting state (Figure 4.9), there are six instantiated modules in the architecture design. Presentation module is designed by using a customized text format that is simple and short for delivery. Ping module is used to verify if the mobile devices are available with notifying them of new mail event. Similarly, the echo is adopted to respond by sending back network parameters. In the push mail system, firewall and authority modules are designed to avoid certain types of network attacks through applying phone ID that is utilized to determine if allow a mobile device to access the system. SMS is adopted to support the transport for carrying the data of presentation. Figure 5.8 gives a whole architecture design of connecting state for connector in push mail system.

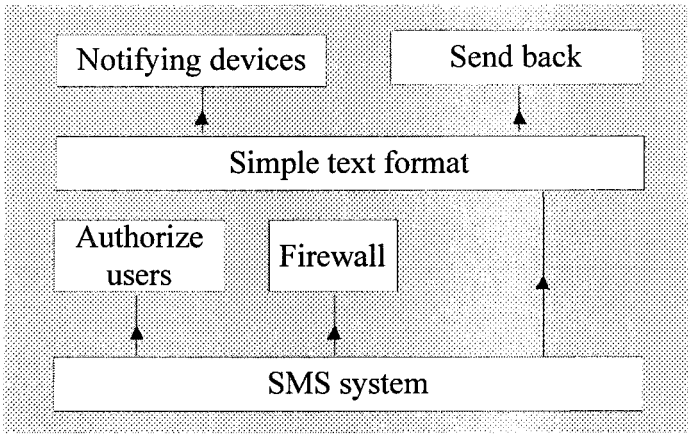


Figure 5.8: Entire architecture of connecting state for connector in push mail system

Finally, after analyzing the architecture design, we examine the selection of development tools for all modules of the architecture following our design model. Although the tools are required to support timer mechanism and access control approach (Table 4.6), for the modern languages it is easily to satisfy them based on normal operating systems. For example we choose Java as the development tools for this state.

5.5 Implementation

According to the design of connector the chapter produced, we develop a prototype by running a simulation of the push mail system in software environment.

The push mail server is developed using Java and the connector is implemented by Java and AspectJ. For both the server and the connector, we choose Eclipse as IDE to develop them. In the client side (mobile phones), we adopt Windows Phone emulator and

use C# (in Visual Studio 2010 Express for Windows Phone) for developing push mail client under Windows Phone OS (version: "Mango").

In the prototype, we implement main parts of connector. For simulating the SMS mechanism, UDP socket is applied in the software environment. The simulation of prototype shows the result that the connector can push emails to the Windows Phone, when emails arrive in the mail server.

5.6 Conclusion

In this chapter, we perform a case study for designing connector in distributed systems using our design model. The case study is concerned with the architecture design of connector for a push mail system in wireless network. First, we give specifications about the desired connector. Second, according to the specifications we carry on the requirement analysis for providing architecture drivers. Third, we create a general architecture design which consists of 6 states of connector. In the end, we concentrate on the active and connecting states for producing a detailed architecture design following LDM model. In addition, we present the related development tools for each module.

During the design process, the case study illustrates how the design model operates in practice for attaining the objective of functionality and achieving the quality attributes of connector in high level design. Following our design model (LCM and LDM), the functionalities and quality attributes of the connector are separated into layers so that each

layer is especially responsible for satisfying its functionalities and corresponding quality attributes. Therefore, the design fulfills the architectural drivers by using EADD methods and developers can be guided through the process of high level design of connectors by following our design model.

Moreover, the design model provides suitable selection of development tools for implementation of the designed architecture for supporting implementation (coding, testing, and maintaining). For example, the pattern module of active state chose Aspectj and Observe design pattern for achieving the Modifiability attribute.

Finally, we build a prototype for simulating the push mail system in software environment in order to check our objective of the case study: design of connector in a push mail system without the additional support of operators and avoiding modifying the existing deployment of wireless network.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Summary

This thesis has given a definition of connectors and illustrated a model for designing connectors in distributed systems. Furthermore, we developed a design method (EADD) to facilitate the architecture design process and the selection of development tools in high level design.

For achieving these objectives of the thesis, we firstly provided the motive and the example scenario and then we identified a set of problems of our research in chapter 1. These problems of development of connectors comprised: 1) various definitions, 2) confusing classification, 3) lack of design model, 4) and few approaches to maintenance.

Secondly, chapter 2 reviewed the current state-of-art of connectors and presented the relevant technologies that are mainly made up of middleware, relationship of components, AOP, and program slicing. Furthermore, we analyzed quality attributes of connectors and the ADD method to prepare basis for the following chapters.

Thirdly, in chapter 3 we created EADD design method based on ADD mentioned in section 2.9. EADD typically takes functional requirements and quality attributes as input and returns architecture design and selection of development tools as output. And then we underlined the use of the method for designing connectors.

Fourthly, based on EADD, chapter 4 proposed our design model and related approaches to the model in practice. Then, from the development process point of view, we analyzed the design model in order to indicate the stage of its use in the software life cycle

and examined the timing to adopt it in development. Additionally, we highlighted the advantages of our design model and gave a classification of connectors according to our new definition and model.

Finally, in chapter 5 we demonstrated a case study according to the example scenario of section 1.2: design of connector for a push mail system in wireless network. The design process followed the procedure: specifications analysis, requirement analysis, general architecture design, and detailed architecture design. In this way, the case study shows the design process of the architecture of connector and the selection of the related development tools based on our design model.

6.2 Contributions

By answering the questions given in the introduction of chapter 1, we summarize the contributions of this thesis to the area of computer science as below.

This thesis gives a new definition of software connectors. The definition can be distinguished by three major features: information delivery and wrapped data, as well as independence of components (see section 4.2). The definition especially considers quality attribute as an important property of connectors. Hence, in this way, software connectors are clearly and fully described in computer science. In addition, it also introduces various knowledge areas for designing connectors, including the system quality attributes.

We create a new approach (see section 3.3) which is called as extended attribute-driven design method (EADD) for both architecture design and selection of developments tools, based on attribute-driven design method (ADD). For designing connectors, EADD is equipped to drive architecture design by providing the instantiated modules and suitable development tools for developers. Our new method, in contrast to ADD, can enable engineers to select proper development tools and frameworks that can combine to construct software connectors in high level design stage. From the development point of view, EADD can be also applied to design of all software.

Based on EADD method and the new definition of connectors, we develop a model for designing connectors (see sections 4.3 and 4.4) in order to fulfill functional requirements and to achieve quality attributes by creating architecture design (a set of modules) and organizing a group of development tools. The model can be applied to design of generic connectors in distributed systems, for example connectors in component-based distributed systems, connectors in service-oriented architecture (SOA), etc. From the software life cycle point of view, our model may enhance high level design phases and to choose suitable tools for following development phases. In addition, because requirements are separated into some parts according to states and layers of the model, different developers can focus on different requirements. In this way, the design model can be regarded as a determinant to arrange for proper developers to deal with different requirements.

Compared with existing approaches to designing connectors, the layered structures of major states provided by our design model enable developers to understand all aspects of

connectors for design, implementation and maintenance. Moreover, the key approaches to use of design model are provided to implement connectors by weaving the relevant technologies and tools together, particularly for connector maintenance (adding connectors in existing systems, automatically locating the key points of source code and preventing major modifications of legacy components). In particular, connectors designed by following our model are capable of achieving high system quality attributes.

According to the new definition and design model, our research analyzes the effect of the design model on classification of connector and attempts to make a classification of connector by combining the functionalities with qualities of connectors in distributed systems. Four types of connector are identified: integration, division, extension, and distribution (see section 4.6.3).

6.3 Future Work

The results presented in this thesis can be extended and improved in several ways.

6.3.1 Validation Model for Developing Connectors

According to the software development life cycle, a test process is typically involved in development. Thus, we can create a validation model for checking intended functionalities and quality attributes of software connectors. Based on the definition of connectors in this thesis, the validation model should be designed for following purposes:

- The model can deal with all test cases that should cover functionalities and qualities;
- The model can verify every state of connector, such as active, connecting, disconnecting, and so forth;
- The model can help developers locate bugs according to layers of state;
- The model should be deployed without major modification of connectors.

6.3.2 Framework for Statistics

From the thesis, the design of connectors largely depends on some working parameters of connectors, for example amount of data, overhead of packet etc. Particularly, when updating connectors, developers must firstly obtain this information. In light of this, these parameters are dynamically captured during execution of connectors. Thus, the statistics framework should be embedded in measured connectors and does not affect any features of connectors.

6.3.3 Taxonomy of Software Connectors

As mentioned early in chapter 4, we have discussed the classification of connectors. According to this, we can develop a classification framework to understand connectors from taxonomy point of view. The objectives of the framework are described as below:

- The framework can help to select suitable connectors for design;
- The framework can provide typical approaches and patterns for each type of connector;

- In terms of functionalities and qualities, combination of connectors can gain benefit from the framework.

BIBLIOGRAPHIE

[Bass et al., 2003] Len Bass, Paul Clements, Rick Kazman (2003). Software Architecture in Practice, Second Edition. Addison-Wesley.

[Szyperski et al., 2002] Clemens Szyperski, Dominik Gruntz, Stephan Murer (2002). Component software: beyond object-oriented programming. Addison-Wesley.

[Shown et al., 1996] Shown and Garlan (1996). Software Architecture: perspectives on an emerging discipline. Prentice-Hall.

[McGovern et al., 2003] James McGovern, Scott W. Ambler (2003). A Practical Guide to Enterprise Architecture. Prentice Hall.

[Mehta et al., 2000] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke (2000). Towards a Taxonomy of Software Connectors. In Proceedings of the 22th International Conference on Software Engineering (ICSE 2000), Limerick, Ireland.

[Balek et al., 2000] D. Balek and F. Plasil (2000). Software connectors: A hierarchical model. Technical Report 2000/2, Charles University.

[Veríssimo et al., 2001] Paulo Veríssimo and Luís Rodrigues (2001). Distributed systems for system architects. Springer.

[Coulouris et al., 2005] George F. Coulouris, Jean Dollimore, Tim Kindberg (2005). Distributed systems: concepts and design. Addison-Wesley.

[Thekkath et al., 1994] Chandramohan A. Thekkath, Henry M. Levy, Edward D. Lazowska (1994). Separating data and control transfer in distributed operating systems. ACM, USA.

[Mattmann, 2007] C. Mattmann (2007). Software Connectors for Highly Voluminous and Distributed Data-Intensive Systems, Ph.D. Dissertation, USC.

[Edwards, 1997] Stephen H. Edwards, David S. Gibson, Bruce W. Weide, Sergey Zhupanov (1997). Software Component Relationships. IN PROC. 8TH ANNUAL WORKSHOP ON SOFTWARE REUSE.

[Booch, 2007] Grady Booch, Robert A. Maksimchuk, Michael W. Engle (2007). Object-oriented analysis and design with applications. Addison-Wesley.

[IBM Corporation] IBM Corporation. Developing application code, Association relationships.
http://publib.boulder.ibm.com/infocenter/rsdvhelp/v6r0m1/index.jsp?topic=%2Fcom.ibm.xtools.viz.cpp.doc%2Ftopics%2Fcassociation.html.

[Gamma et al., 1995] Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[Cooper, 2000] James W. Cooper (2000), Java design patterns: a tutorial. Addison-Wesley.

[Buschmann et al., 2007] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt (2007). Pattern-oriented software architecture: On patterns and pattern languages. John Wiley and Sons.

[Hohpe et al., 2004] Gregor Hohpe, Bobby Woolf (2004). Enterprise integration patterns: designing, building, and deploying messaging solutions. Addison-Wesley.

[Gabhart, 2003] Kyle Gabhart (2003). J2EE pathfinder: Enterprise messaging with JMS. IBM developerWorkers. http://www.ibm.com/developerworks/java/library/j-pj2ee5/

[SDN] SDN. MOMs and the Java Message Service (JMS). http://java.sun.com/developer/technicalArticles/Networking/messaging/

[Tanenbaum, 2003] Andrew S. Tanenbaum (2003). Computer Networks. Prentice Hall.

[Shaw, 1994] Mary Shaw (1994). Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Carnegie Mellon University Pittsburgh, PA, USA.

[Ranganath et al., 2006] Venkatesh Prasad Ranganath, John Hatcliff (2006). Slicing Concurrent Java Programs using Indus and Kaveri. International Journal on Software Tools for Technology Transfer (STTT), Vol.9, Numbers 5-6, pp. 489-504.

[ISO/IEC 14764, 2006] ISO/IEC 14764 (2006). Software Engineering- Software Life Cycle Processes- Maintenance.

[Wood, 2007] William G. Wood (2007). A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0. TECHNICAL REPORT CMU/SEI-2007-TR-005 ESC-TR-2007-005.

[Sharma et al., 2001] Rahul Sharma, Beth Stearns, Tony Ng (2001). J2EE Connector architecture and enterprise application integration. Addison-Wesley.

[Birrell et al., 1984] Andrew D. Birrell and Bruce Jay Nelson (1984). Implementing remote procedure calls. ACM Transactions on Computer Systems, 2(1): 39-59.

[Pryce, 2000] Nathamid G. Pryce (2000) Component Interaction in Distributed Systems. PhD Thesis, Imperial College of Science, Technology and Medicine.

[OMG, 2004] OMG (Object Management Group, 2004). The Common Object Request Broker: Architecture and Specification, Version 3.0.3.

[Emmerich, 2000] Wolfgang Emmerich (2000). Engineering Distributed Objects. John Wiley and Sons.

[Gradecki et al., 2003] Joseph D. Gradecki and Nicholas Lesiecki (2003). Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley.

[Bishop et al., 2006] Sasa Subotic and Judith Bishop (2006). Aspect-Oriented Programming for a Distributed Framework. SACJ, Submission.

[Elrad et al., 2001] Tzilla Elrad, Robert E. Filman and Atef Bader (2001). Aspect-oriented programming: Introduction. Magazine Communications of the ACM Volume 44 Issue 10, Oct. 2001 ACM New York, NY, USA.

[Stevens, 1997] W. Richard Stevens (1997). Unix Network Programming: Networking APIs: Sockets and XTI (Volume 1). Addison-Wesley.

[Allen et al., 1997] Robert Allen and David Garlan (1997). A Formal Basis for Architectural Connection. ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY.

[Oreizy et al., 1998] Peyman Oreizy, David S. Rosenblum, Richard N. Taylor (1998). On the Role of Connectors in Modeling and Implementing Software Architectures. Department of Information and Computer Science, University of California.

[Laddad, 2003] Ramnivas Laddad (2003). AspectJ in action: practical aspect-oriented programming. Manning.

[Erenkrantz, 2004] Justin R. Erenkrantz (2004). Web services: Soap, uddi, and semantic web. Technical Report UCI-ISR-04-3, Institute for Software Research.

[TechNet, 2005] Microsoft TechNet Library (2005). Understanding Message Security. http://technet.microsoft.com/en-us/library/bb124482%28EXCHG.65%29.aspx

[Aiken, 1999] Alexander Aiken (1999). Introduction to set constraint-based program analysis. Journal Science of Computer Programming Volume 35 Issue 2-3.

[Nielson et al., 1999] Flemming Nielson, Hanne R. Nielson, Chris Hankin (1999). Principles of Program Analysis. Springer.

[Weiser, 1982] Mark Weiser (1982). Program Slicing. ICSE '81 Proceedings of the 5th international conference on Software engineering.

[Weiser, 1979] M. Weiser (1979). Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, MI.

[Harman et al., 2003] Mark Harman, David Binkley and Sebastian Danicic (2003). Amorphous program slicing. Journal of Systems and Software, 68 (1), pp. 45-64.

[Wojcik et al., 2006] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord and B. Wood (2006). Attribute-Driven Design (ADD), Version 2.0. TECHNICAL REPORT, CMU/SEI-2006-TR-023, ESC-TR-2006-023.

[Carmichael, 1998] Andy Carmichael (1998). Developing Business Objects. CUP Archive.

[RFC3501, 2002] RFC 3501 (2002). INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev.

[RFC5321, 2008] RFC 5321 (2008). Simple Mail Transfer Protocol.

[RFC1939, 1996] RFC 1939 (1996). Post Office Protocol - Version 3.

[RFC1341, 1992] RFC 1341 (1992). MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies.

[THE RADICATI GROUP, 2009] THE RADICATI GROUP (2009). Wireless Email Market, 2009-2013. http://www.radicati.com

[OMA, 2007] OMA (2007). E-Mail Notification. http://www.openmobilealliance.org/Technical/release_program/emn_v10.aspx

[Puntambekar, 2009] A.A. Puntambekar (2009). Software Engineering. Technical Publications.

[Abran et al., 2001] A. Abran, J.W. Moore, P. Bourque and R. Dupuis (2001). Guide to the Software Engineering Body of Knowledge. IEEE Computer Society.

[Comer, 1997] E. Comer (1997). "Alternative Software Life Cycle Models", Software Engineering. IEEE CS Press.

[IEEE/EIA] IEEE/EIA 12207.0-1996/ISO/IEC12207:1995, Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes, vol. IEEE, 1996.

[Sommerville, 2004] Ian Sommerville (2004). Software engineering. Pearson/Addison-Wesley.

[Qiu, 2005] Xiaohong Qiu (2005). Message-based MVC Architecture for Distributed and Desktop Applications. Syracuse University PhD.

[Wagner et al., 2006] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, Peter Wolstenholme (2006). Modeling software with finite state machines: a practical approach. Auerbach.

[Saleh, 2009] Kassem A. Saleh (2009). Software Engineering. J. Ross Publishing.

[Lee, 1994] G. Lee (1994). Object Oriented GUI Application Development. Prentice Hall.

[Hannemann et al., 2002] Jan Hannemann and Gregor Kiczales (2002). Design Pattern Implementation in Java and AspectJ. ACM, USA.

[Puntambekar, 2009] A.A. Puntambekar (2009). Software Engineering. Technical Publications.

[Clements, 2003] Tom Clements (2003). SMS— Short but Sweet. Sun developer network, http://developers.sun.com/mobility/midp/articles/sms/

[Indus project] Indus project, available at http://indus.projects.cis.ksu.edu/

[Holzner, 2004] Steven Holzner (2004). Eclipse. O'Reilly Media, Inc.

[Johnson et al., 2005] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg and Colin Sampaleanu (2005). Professional Java development with the Spring Framework. John Wiley & Sons.

[Tanenbaum et al., 2007] Andrew S. Tanenbaum and Maarten van Steen (2007). Distributed systems: principles and paradigms. Pearson Prentice Hall, Second Edition.

[Alonso, 2004] Gustavo Alonso (2004). Web services: concepts, architectures and applications. Springer.

[Karastoyanova et al., 2003] Dimka Karastoyanova, Alejandro Buchmann (2003). "Components, Middleware and Web Services", In IADIS International Conference WWW/Internet 2003, Volume II, IADIS Press, 2003, pp. 967-970.

[Snell et al., 2002] James Snell, Doug Tidwell, Pavel Kulchenko (2002). Programming Web services with SOAP. O'Reilly Media, Inc.

[W3C, 2004] W3C Working Group Note (2004). Web Services Architecture. http://www.w3.org/TR/ws-arch/#whatis

[Singh et al., 2005] Munindar Paul Singh, Michael N. Huhns (2005). Service-oriented computing: semantics, processes, agents. John Wiley and Sons.

[Nagappan et al., 2003] Ramesh Nagappan, Robert Skoczylas, Rima Patel Sriganesh (2003). Developing Java Web Services: Architecting and Developing Secure Web Services Using Java. John Wiley & Sons.

[Christensen et al., 2001] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana (2001). Web services description language (WSDL) 1.1. www.w3.org/TR/wsdl

[UDDI, 2000] UDDI. UDDI technical white paper. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[Howard, 2001] Rob Howard (2001). Encrypting SOAP Messages. Microsoft Corporation, MSDN. http://msdn.microsoft.com/en-us/library/ms972410.aspx

[Cook et al., 2005] Diane J. Cook, Sajal K. Das (2005). Smart environments: technologies, protocols, and applications. John Wiley and Sons.

[Li, 2003] Bixin Li (2003). "Managing dependencies in component-based systems based on matrix model", Proceedings Of Net.Object.Days 2003, pp. 22-25.

[Gupta et al., 2010] R. Gupta and A.K. Tripathi (2010). "Dependence analysis of software component", presented at ACM SIGSOFT Software Engineering Notes, 2010, pp.1-9.

[Zamir, 1999] Saba Zamir (1999). Handbook of object technology. CRC Press.

[Arranga et al., 1997] Edmund C. Arranga, Frank P. Coyle (1997). Object-oriented COBOL. Cambridge University Press.

[Gorton, 2011] Ian Gorton (2011). Essential Software Architecture. Springer.

[Zhu, 2005] Hong Zhu (2005). Software Design Methodology. Butterworth-Heinemann.

[OMG, 2010] OMG (2010). "OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3", OMG Document Number: formal/2010-05-05. Object Management Group.

[Kontio et al., 2002] Jyrki Kontio, Reidar Conradi (2002). "Improving Software Quality in Product Families through Systematic Reengineering", Software quality-ECSQ 2002: quality connection--7th European Conference. Springer.

[Grubb et al., 2003] Penny Grubb, Armstrong A. Takang (2003). Software maintenance: concepts and practice. World Scientific.

[Schmidt et al., 2002] Douglas C. Schmidt, Stephen D. Huston (2002). C++ Network Programming: Systematic reuse with ACE and frameworks. Addison-Wesley Professional.

[Mueller, 2002] Stephen M. Mueller (2002). APIs and protocols for convergent network services. McGraw-Hill Professional. Page 94.

[Tari et al., 2001] Zahir Tari, Omran Bukhres (2001). Fundamentals of distributed object systems: the CORBA perspective. John Wiley and Sons. Page 17.

[Vogels, 2003] Werner Vogels (2003). "Web services are not distributed objects", IEEE Internet. Computing, vol. 7, no. 6, pp. 59–66.

[Ratneshwer et al., 2011] Ratneshwer, A K Tripathi. "Dependence Analysis of Component Based Software through Assumptions", International Journal of Computer Science Issues. Volume 8, Issue 4, pages 149-159.