A SURVEY AND A DETAILED CASE STUDY USING OMG IDL: THE ROLE OF IDL IN COMPONENT COMPOSITION

BY

EMRAN AL-SHAHROURI

Bachelor of Science

Mu'tah University

Al-Karak - Jordan

1993

Submitted to Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2003

A SURVEY AND A DETAILED CASE STUDY USING OMG IDL:

THE ROLE OF IDL IN COMPONENT COMPOSITION

Thesis Approved:

[illegible]

Thesis Advisor

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

PREFACE

The dream of building large software systems out of well-defined independent components is gradually coming true. Modern software systems are rarely developed entirely from scratch; rather they are constructed using tested and reliable pieces called components.

Component Based Software Development (CBSD) still faces some major obstacles. One of these problems is composing the different components that make up a system. Interface Definition Language (IDL) plays a vital role in composing components. IDL is used to describe the contracts (interfaces) between the components of a system. Object Management Group (OMG) is a leader in defining standards for software components. One of the IDL standards is OMG IDL. This thesis reports a study of OMG IDL and the role it plays in component composition. One case study (a library system) was investigated using OMG IDL.

The case study is introduced informally, then it is analyzed and designed as a component-based system, subsequently an OMG IDL is presented for the case study. The functional and extra functional properties of the system are then discussed. The following lessons and conclusions were learned from the case study. OMG IDL was originally designed to specify the functionality of the components of a system, but its function has been extended to compose the components together as well. Using the OMA standard

services, CORBAservices helps control the extra-functional properties. The software designer should be knowledgeable about the standard components and services in the component model in order to use them when they are needed and not write them again. In the library system case study, it was not necessary to develop new Naming and Trading services, Transaction services, or Security services, and the OMA standard services were used instead. A good design is essential for a component system to succeed. It is hard to cover all parts of OMG IDL in one case study. Also, there is lack of standardized components in the general library system domain.

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I would like to extend my sincere appreciation to my thesis advisor, Dr. Mansur H. Samadzadeh, for assisting me with his guidance, wisdom, encouragement, and patience throughout my graduate studies at Oklahoma State University. My special gratitude is also extended to Dr. George E. Hedrick and Dr. Blayne E. Mayfield for their valuable help and participation while serving as members of my graduate committee.

My appreciation goes to the soul of my late father who was my inspiration to fulfill my dreams. Especial admiration goes to my mother for her great love and support. My thanks also extend to my brothers and sisters for their encouragement and trust. I would like to extend my special gratitude to my fiancé, Noor, whose patient love has enabled me to complete this thesis.

I am also obliged to my friends for their support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Software development is a time consuming and expensive process. A major concern for researchers and software engineering specialists is how to minimize the time and cost needed to develop reliable software systems. One of the effective ways to reach this goal is through software reuse, and one of the preferred practices for software reuse is Component-Based Software Development (CBSD). CBSD emerged in the late 1990s [Ivers et al. 02] but still lots of work needs to be done such as predicting component and system properties.

The idea of reusing software began gaining widespread acceptance since the inception of object-oriented programming and software libraries. Instead of spending time and effort doing the same thing repeatedly, and building software systems from scratch, one can reuse the software already made, and develop software systems from components as the menu functioning process is done in other engineering disciplines.

The components that make up a system could reside on the same computer or be distributed over a network. Such components need to communicate with each other and with the environment in which they are deployed. These communications are typically done through components interfaces. Component interfaces work as contracts among the different components in the composed system. There is a special type of language used to describe these software contracts (interfaces). These languages are called Interface Definition Languages (IDL).

Middleware is software that manages the communication and data exchange among the different components in component-based systems [Sommerville 01]. The most widely used component middleware technology is CORBA (Common Object Request Broker Architecture) that is a product of OMG (Object Management Group). CORBA 3 refers to the CORBA Component Model that includes a suite of ten specifications [OMG 02], one of these ten specifications is the OMG IDL.

The purpose of this research was to investigate the important role that IDL plays in composing the different components of a component-based system. A case study (a library system) was studied using the OMG IDL. The library system was specified and analyzed as a component-based system, and the different components of the system were identified. OMG IDL routines were written for the different interfaces of the components. Subsequently, the properties of the system were analyzed based on the types of properties of component systems: functional properties and extra-functional properties. Extra-functional properties include performance, security, latency, and accuracy. These properties are usually referred to as quality properties or quality of service properties when they are attached to service [Bachmann et al. 00].

The organization of this thesis is as follows. Chapter II provides an overview of Component Based Software Development (CBSD) including software components, problems with CBSD, the CBSD process, designing Component-Based systems, component composition, and component interfaces. Chapter III introduces some of the popular middleware component technologies. Chapter IV provides a general introduction to the Object Management Group (OMG) Interface Definition Language (IDL). Chapter V presents the Object Management Architecture. Chapter VI provides the informal specifications, the object oriented analysis and design, and the component-based specification for the case study "library system", then the case study is captured in OMG IDL. Chapter VII gives a discussion of the functional and extra-functional properties of the "library system". Finally, Chapter VIII discusses the summary and future work.

CHAPTER II

COMPONENT-BASED SOFTWARE DEVELOPMENT (CBSD)

The sections in this chapter present an overview of the main issues of Component-Based Software Development (CBSD): software components, discussing some problems with CBSD, introducing the CBSD process, designing component-based systems, component composition, and component interfaces.

2.1 Software Components

Components are the core of CBSD and thus we need a clear definition for them in order to understand the fundamentals of CBSD. In the absence of universal standards and guidelines in this area, there is no definition for the term "component" on which everyone agrees. Basically, a component has the following main features: 1) a software component is an independent and replaceable entity of a system that performs a clearly defined function, 2) a software component plays a role in a well-defined architecture, 3) a software component interacts and communicates with the other component through its interface and it also provides services through its well-defined interface [Cai et al. 00] [Clements et al. 99].

According to Alan Brown, one of the co-authors of the book *Constructing*

Superior Software [Clements et al. 99], "a component is a software package which offers services through interfaces". UML 1.0 and 1.1 define a component as: "a reusable part that provides the physical packaging of model elements" [Clements et al. 99]. Microsoft Component Object Model (COM) defines a component as "a piece of compiled software which is offering a service" [Crnkovic and Larsson 02]. Another definition for a software component is given below.

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party [Crnkovic and Larsson 02].

A component is defined by its interfaces. In general, components have two different and related interfaces: 1) interfaces provided, which are the services a component provides, and 2) interfaces required, which are the services that have to be available from the system in order to use the component [Sommerville 01] (see Figure 1).

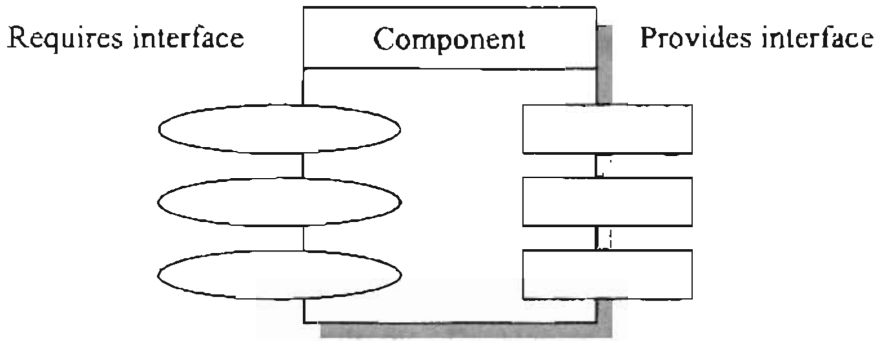


Figure 1. Component interfaces [Sommerville 01]

One of the great features of a component is the separation of its interfaces from its implementation [Crnkovic and Larsson 02]. A component can also be conceptualized as an encapsulated implementation of functionality that can be used by a third party, and that complies with a component model [Bachmann et al. 00].

Components vary in size from a component that performs a simple mathematical function to a component that is an entire application by itself. The latter level of component reuse (e.g., MS Excel as a component) is called commercial off the shelf (COTS) component reuse [Sommerville 01].

Component-based software development (CBSD) or, as some people like to call it, component-based software engineering (CBSE) is closely related to the development of distributed systems, which consist of components that are distributed among computers on a network.

2.2 Problems with CBSD

Component-based software development (CBSD) suffers from some problems and extra costs, which are basically associated with software reuse in general. These costs and problems are [Crnkovic and Larsson 02] [Sommerville 01]: high maintenance cost, lack of supporting development tools, frequent updating of the component library, and the effort involved in finding the right components and adopting them.

Another problem with CBSE is the issue of component trust, which is referred to as the "not-invented-here" syndrome. Developers usually find it hard to trust other developers' work especially from outside their organization. This becomes more pronounced when it comes to COTS products. Mittermeir suggested some techniques to improve software comprehension [Mittermeir et al. 01] in order to help trust components especially obtained from outside an organization.

Component composition is another major problem in CBSE. Predicting the overall system properties is one of the important research trends in CBSE [Crnkovic et al. 02]. When a system is composed from different components, these components sometimes act in an unexpected way because of the environment and the effect of the other components in the system (see Section 2.5). This can only be discovered at the time of composing the system [Moreno et al. 02].

2.3 The CBSD Process

The traditional practice of software engineering process does not give software developers the full advantage of component-based software development (CBSD). In order to gain the benefit of CBSD, software developers should think differently in the way they design and construct software systems [Clements et al. 99]. Developers should shift their goal from developing an individual application to developing well-defined reusable software components that can be used to build families of application. Some of the main issues that a typical CBSD process should address are [Clements et al. 99]: defining acceptable sizes for software components, describing the dependencies among the components in a system, documenting a component and making it handy for others to use, and examining the impact of CBSD on testing, maintaining, and evaluating component-based systems. CBSD should focus on interfaces and interface-based design, and it should support selection, evaluation, and assembly of components to create new applications.

Sommerville presented two different approaches for component-based development [Sommerville 01]. The first approach consists of four steps (Figure 2), it starts with designing the system architecture, then it is time to specify the components in the system, next comes the search for components that comply with the specifications and the design, and the last step is incorporating the discovered components. This approach may indeed lead to a good level of component reuse, but at the same time it contrasts with other engineering disciplines where component reusability drives the development and menu functioning process. Typically, engineers first search for the necessary components and then they design the system based on these components. In the second approach (Figure 3), the specifications get modified according to the available components.

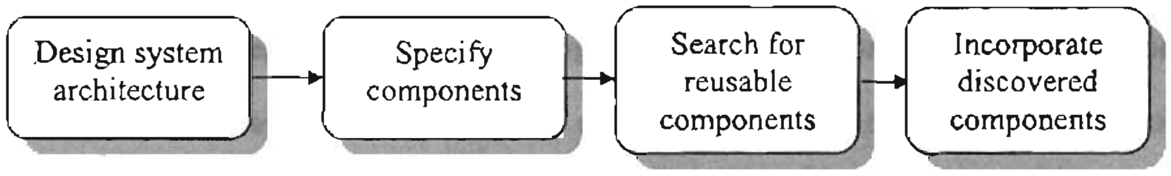

Design system architecture
Specify components
Search for reusable components
Incorporate discovered components

Figure 2. An opportunistic reuse process [Sommerville 01]

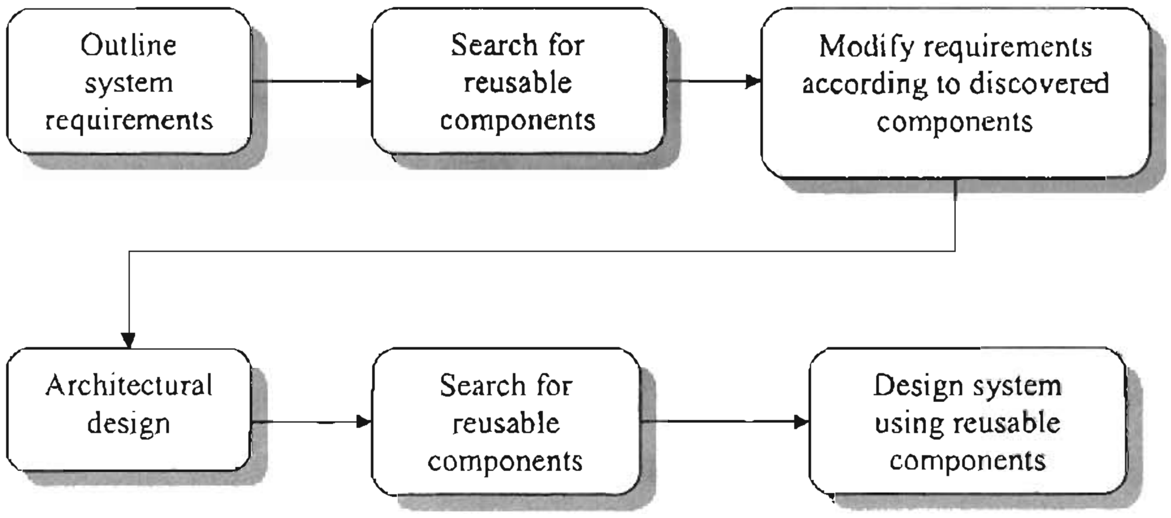

Outline system requirements
Search for reusable components
Modify requirements according to discovered components
Architectural design
Search for reusable components
Design system using reusable components

Figure 3. Development with reuse [Sommerville 01]

2.4 Designing Component-Based Systems

As stated earlier (see Section 2.3), the development of component-based systems should focus on interfaces and interface-based design. As a result, we need tools to support this kind of development. Some advances in tools and modeling techniques have taken place in recent years. Some of these advances are mere standardization on a common notation for behavior-based design systems and the emergence of component design targeting this notation.

One of the popular tools used to model software systems is the unified modeling language (UML) [Clements et al. 99] [Kurchten 98] [Siegel 00]. UML is another product of OMG. It provides a notation for capturing many features and issues of components and component-based systems. But still UML cannot address all of the different aspects of component and component-based systems. Kruchten [Kruchten 98] proposed a number of techniques to represent component-based systems in UML.

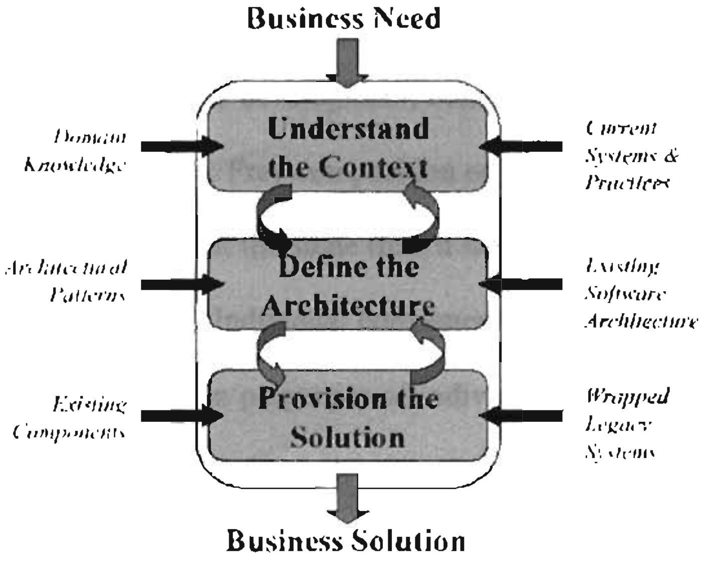


Figure 4. Elements of a component-based development approach [Clements et al. 99]

Alan Brown from Sterling Software presented three basic steps for component-based systems modeling [Clements et al. 99]: understanding the context, defining the architecture, and provisioning the solution (Figure 4). These steps may occur in any order.

2.5 Component Composition

In CBSD, the term composition is used instead of integration to refer to how a system is assembled [Bachmann et al. 00]. Different blocks (components) are composed to form a component assembly or a system. A number of components may be composed to generate larger components. Components have different levels of communication [Bachmann et al. 00]. These levels are: component-to-component, component-to-framework, and framework-to-framework. A framework manages the different resources shared by components and provides basic mechanisms that facilitate interaction among components [Bachmann et al. 00].

In the traditional software development, component integration and composition is a critical phase of the process. Free composition of reused components can reduce the development cost and time, but at the same time it has a number of potential risks and it may incur a high price later on. Individual components occasionally do not act the same when they are composed. Certain properties of individual components may not hold for the assembly. Individual components sometimes make some assumptions about other components, these assumptions may not hold when they are integrated with one another. This can cause a phenomenon that is called architectural mismatch [Dong 02]. In other words, the actual behavior of component assemblies is only discovered after their integration [Moreno et al. 02]. Some research has been done on predicting the system properties or the component assembly properties based on the properties of the constituent components [Crnkovic et al. 02], but still there are no clear results in this area of research.

2.6 Component Interfaces

Software systems that have problems and bugs may result in big losses in money, effort, and even in human lives [Dong 02]. In CBSD, interface specification has a critical role in constructing software systems based on the building blocks (components). As stated by Dong [Dong 02]:

> Imprecise, ambiguous, and incomplete specification of components may lead to wrong choices, and therefore mismatches in the compositions. These mismatches may require high cost and expert skills to find and correct, thus compromise the benefits of component-based software development.

So designing the interfaces of the components should be precise, clear, unambiguous, and complete.

Component interfaces govern the way a component communicates with the outside world. These interfaces represent the boundaries between components. These boundaries could be thread boundaries, process boundaries, programming language boundaries, or machine boundaries [Gudgin 01]. Component interfaces are the way we integrate components into groups called assemblies [Bachmann et al. 00].

CHAPTER III

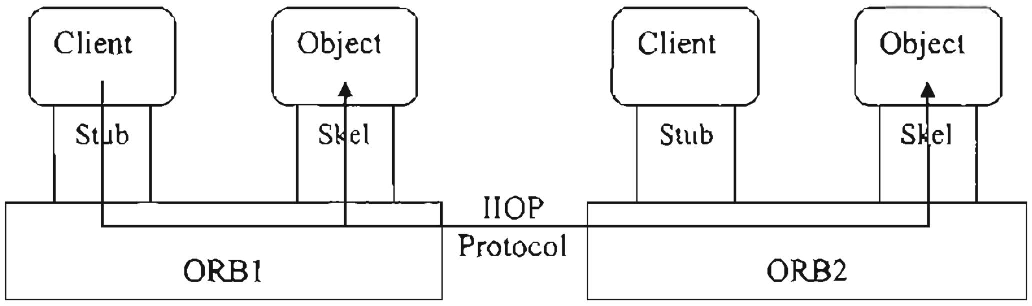
POPULAR MIDDLEWARE COMPONENT TECHNOLOGIES

Components of a system may be implemented in different languages, these components may be distributed over a network, and they may run on different platforms [Sommerville 01]. Also, the components need to communicate and coordinate through the component infrastructure (sometimes called a component model [Cai et al. 00]). Component infrastructure acts as the "pluming" or "middleware" that allows different components to communicate with each other [Cai et al. 00]. There are some standardization efforts done on these component middleware infrastructures like OMG CORBA, Microsoft's COM and DCOM, and Sun's JavaBeans and Enterprise JavaBeans [Cai et al. 00]. In each one of these component infrastructure implementations, there is a vision of how to build an enterprise-scale component-based application supported by a set of tools and standards [Clements et al. 99].

3.1 Common Object Request Broker Architecture (CORBA)

This is an open standard for component interoperability [Cai et al. 00]. This standard is defined by the Object Management Group (OMG), which is made up of over 800 companies [Clements et al. 99] [Siegel 00] to promote object-oriented software development. The role of this group is to provide standardization for object-oriented development but not to provide a specific implementation, and it is available free of charge. OMG does not just define standards for CORBA, it also defines other standards like UML and OMG IDL [Sommerville 01]. OMG is attempting to achieve consensus on an appropriate component-based model for building component-based distributed applications [Clements et al. 99].

OMG defined its vision of component-based systems in its Object Management Architecture (OMA) model (see Chapter V). CORBA has three major features [Clements et al. 99]: 1) Interface Definition Language (IDL) that describes how business functionality is packaged to be accessed from external interfaces, 2) CORBA component model that describes how components make requests for other components' services, and 3) the Internet InterOperability Protocol (IIOP) that allows the different CORBA implementations to communicate and interoperate. Figure 5 shows how this communication happens.



Stub: client side part of the compiled IDL file
Skel (Skeleton): object side part of the compiled IDL file
ORB: Object Request Broker

Figure 5. Interoperability uses ORB-to-ORB communication [OMG 02]

Figure 5 is a simplification of what actually transpires in terms of communications among the components. The stub and skeleton act as proxies for the client and object implementations, respectively. The client passes its IDL-based invocation containing an object reference (each object has a unique object reference) to its local Object Request Broker (ORB). If the object reference is to a local object implementation, the ORB routes it to its target object implementation. If not, the ORB will route it to a remote ORB, through the IIOP protocol that all ORBs implement [Siegel 00], and then the invocation will be routed to a remote object implementation.

The communication and interaction among components in CORBA are done through middleware called the Object Request Brokers (ORB). Using the ORB, a client may invoke the methods of other objects, and the location of these objects will be transparent to the client. The client does not need to know where the objects are located, in what languages they were developed, or under what platforms they are running [Cai et al. 00].

A set of standardized capabilities has been defined in the CORBA services standards. The following services are most often found in the currently available implementations [Clements et al. 99] [Sommerville 01]: 1) life cycle services that are responsible for creating and terminating component instances, 2) naming services that allow different components to identify and find the different services over the network or on the same computer, and they also allow the components to know different information about the other components and the services that the other components may have, 3) security services that provide a secure private connection between a client and the provider of services, 4) transaction services that give the user control to start and complete distributed transactions between components, and in addition they facilitate a rollback mechanism in case of failure, and 5) notification services that let the objects notify other objects of the occurrence of some events. Figure 6 shows a request passing from a client to an object implementation in the CORBA model.

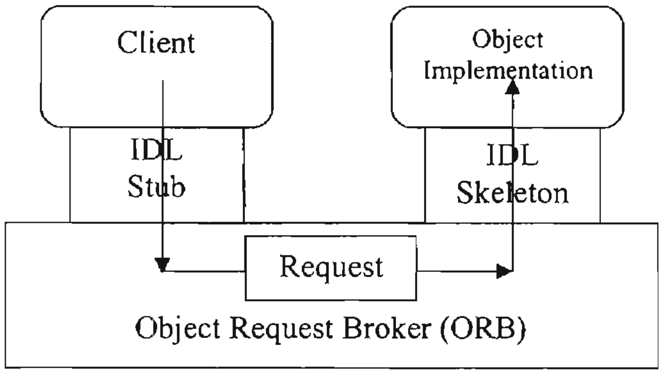


Figure 6. A request passing in CORBA [OMG 02]

There are a number of implementations for the CORBA standards, from different venders, on different platforms for distributed systems running across heterogeneous platforms. This indicates that implementing a component-based application using the OMG standard is feasible and practical. There are a number of successful examples of component-based applications using the OMG approach in different application domains such as banking, retail, and telecommunications [Clements et al. 99]. According to Cai [Cai et al. 00] "CORBA is widely used in object-oriented distributed systems".

There are different versions of the OMG CORBA model [OMG 02]: CORBA 2 and CORBA 3. CORBA 2 is sometimes referred to as the CORBA interoperability and the IIOP protocol, and CORBA 3 is sometimes referred to as the CORBA Component Model.

3.2 Component Object Model (COM) and Distributed COM (DCOM)

Microsoft introduced the Component Object Model (COM) technology in 1993 as a general architecture for component software. It is language independent and based on Windows and Windows NT platforms. COM defines how components communicate with their clients. The main purpose of COM was to enable the sharing of functionality among different desktop applications. After Microsoft realized the advantages of the generic approach [Clements et al. 99] for the desktop applications, it made an extension of COM called Distributed COM (DCOM), which is a protocol that allows components to communicate over a network directly in a reliable, secure, and efficient manner [Cai et al. 00].

There are three major features in DCOM [Clements et al. 99]: 1) the MIDL (Microsoft Interface Definition Language) that describes how the functionality of a component can be accessed externally through its interface, 2) the COM model describes how components can communicate and request services from one another, and 3) the DCOM addition to COM adds support for locating different components across a network and makes the process location transparent to the other components.

Microsoft provides other component infrastructure services through two other products. These two products are: Microsoft Transaction Service (MTS) and Microsoft Message Queue (MSMQ) [Clements et al. 99]. The main disadvantage of the COM/DCOM component infrastructure is that it is platform dependent and works only with Microsoft platforms.

3.3 Sun Microsystems JavaBeans and Enterprise JavaBeans

In the last few years, Java has gained rapid acceptance and has been adopted as a language for developing client-side applications for the Web. Java is in an advanced position to be the backbone for the development of component-based distributed systems. According to Brown [Clements et al. 99], this is a result of a number of features that Java has as a programming language. These features are: 1) Java was originally designed to build network-based applications and it contains support for distributed multi-threaded applications, 2) Java's runtime environment permits modifying a Java application while it is running, 3) memory management simplification in Java has made Java easier to utilize for component-based applications, and 4) Java includes constructs that support the key principles of component-based software engineering such as separating implementations from specifications.

There are two different products that Java provides as infrastructures for component-based development [Cai et al. 00]: the client-side component development, which is JavaBeans, and the server-side component development, which is the Enterprise JavaBeans.

Enterprise JavaBeans provides a definition for the minimum set of services that must be on any server to comply with the specifications of developing enterprise-scale distributed applications. These services are: process and thread dispatching and scheduling, resource management, naming and directory services, network transport services, and transaction management services.

JavaBeans supports applications in a multi-platform environment with reusable client–side and server-side components [Cai et al. 00]. JavaBeans and Enterprise

JavaBeans are platform independent but they are language dependent.

CHAPTER IV

OBJECT MANAGEMENT GROUP (OMG) INTERFACE DEFINITION LANGUAGE (IDL)

Interface Definition Languages such as OMG IDL and COM IDL describe interface abstractions that control the dependencies that exist among different parts of a program or a system [Bachmann et al. 00]. An IDL definition of an interface forms a contract among a client, an object, and the runtime component model [Gudgin 01].

An interface definition written in OMG IDL is programming language independent, but it maps to popular programming languages through the OMG standards (these languages are C, C++, Java, Cobol, Smalltalk, Ada, Lisp, Python, and IDLscript) [OMG 02].

For an IDL to work well for a distributed system, it needs to specify the operation that is going to take place as well as the input and the output parameters with their respective types, and it should also have an error handling mechanism. The OMG IDL has all these three requirements [Siegel 96]. What IDL really does is that it constitutes a contract with the clients of the components. These clients use the same interfaces (to call, build, and dispatch the invocations of the different methods) that the implementations use (to receive and to respond).

The OMG CORBA architecture separates the interfaces definitions from the interface implementations. The interface (the contract) is written using the OMG IDL and the implementation (the fulfillment) is written using a programming language like C++, C, or Smalltalk. An interface represents a promise to a client, but at the same time it represents an obligation for the object that supports and implements that interface [Siegel 96] (see Figure 1 on page 5).

OMG CORBA also enforces object encapsulation, the object of a component can only be accessed through its announced IDL interfaces [Siegel 96]. The IDL compiler maps an IDL script to the desired programming language. Every ORB comes with at least one IDL compiler. When an IDL script runs through the OMG IDL compiler, first the IDL compiler checks for errors. If the IDL script is error free, then the IDL compiler produces at least two files, one for the client stub and the other for the object skeleton [OMG 02]. The client and object implementations are isolated by at least three different components: an IDL stub on the client side, a related IDL skeleton on the object implementation side, and one or more ORBs [Siegel 96] (see Figure 5 on page 13 and Figure 6 on page 15).

It has been reported that interface definitions written in OMG IDL are generally simple, easy to understand, and easy to construct [OMG 02]. OMG IDL has the appearance of ANSI C++ in many ways [Siegel 96]. An OMG IDL script example is given below.

```
// defining the interface for the object Copy

interface Copy{

boolean CheckOut(in string BorrowerName, in
CopyNumberType CopyNumber) raises
(NotValidCopyNumber, UserNotFound)

boolean Return(in CopyNumberType CopyNumber)
raises (NotValidCopyNumber)

}
```

This is the interface to a Copy object that checks out and returns a copy of a book in a library system. The object's type is Copy and it can perform two operations: CheckOut and Return. The CheckOut Operation takes two input parameters. The first parameter, BorrowerName, is a string and the second parameter, CopyNumber, is of type CopyNumberType, which is a user-defined type. The return value, which does not need a name, is a boolean. The CheckOut Operation raises two exceptions: NotValidCopyNumber and UserNotFound. The second operation, Return, takes one input parameter, CopyNumber, which is of typc CopyNumberType. The return value is a Boolean, and it raises one exception: NotValidCopyNumber.

One of the motivations for developing CORBA and IDL is getting all computers in an enterprise to work together regardless of what hardware or software or platform these computers are consist of [Siegel 00], ironically, Interface Definition Languages (IDLs) in general give only a weak guarantee [Borgida and Devanbu 99] [Dong 02] that a software service will work in a particular context as expected. Borgida [Borgida and Devanbu 99] proposed an approach based on description logics to describe component interfaces. Interface Definition Languages (IDLs) describe only the syntax of the component interfaces but not the semantics [Dong 02], the lack of information may cause serious problems when it composed with others. Another problem with IDLs is that IDLs only describe the services offered by an object but not the services required [Dong 02].

CHAPTER V

OBJECT MANAGEMENT ARCHITECTURE (OMA)

Object management architecture (OMA) is OMG's vision for the component model. OMG breaks up the component architecture into four different types (categories) of components [OMG 03]. These four categories are CORBAservices, CORBAfacilites (Horizontal CORBAfacilities), CORBAdomain (Vertical CORBAfacilities), and application objects (see Figure 7). In Figure 7 each service is composed of a number of CORBA objects, each service is accessed by a standard IDL interface, and clients access all services through the Object Request Broker (ORB) [Siegel 00].

IDL (see Chapter IV for more detail) serves as an alphabet [Siegel 00] that different applications could use to create their own interfaces for particular functions. But these applications, even though they use the same alphabet (IDL), may not be able to interoperate because they need a common interface for particular functions in order to interoperate with each other. If the IDL was the common alphabet, OMA is the common language [Siegel 00] among the different application components. OMA is a foundation for the standard services that every component might need for low level of system communication (CORBAservices) such as Naming and Trader services, Transaction services, Security services, and other basic services. OMA is also a foundation for the common functions that different applications from different domains use (CORBAfacilities) such as printing services, or for the common functions that applications from the same domain use (CORBAdomain). Examples of such domains are: healthcare, telecommunications, transportation, electronic commerce, and utilities.

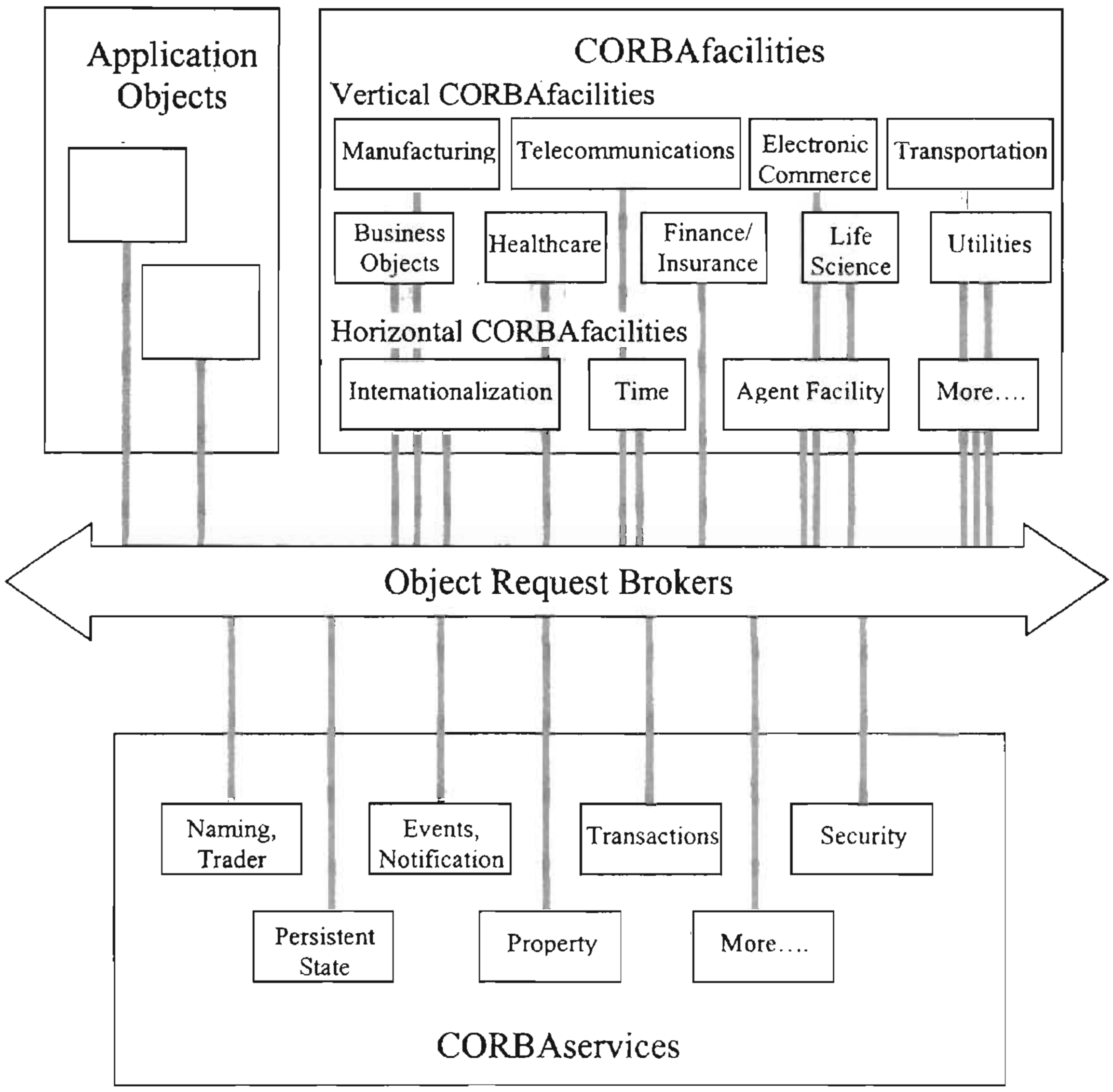


Figure 7. Object Management Architecture [Siegel 00]

Application Objects that constitute the higher part of the hierarchy do not need to be standardized. They are customized for the application according to the application's specification and needs. In another word, these objects are the objects that are not affected by OMG standardization [OMG 02].

The power of CORBA is with the standardized common services and functions [Siegel 00]. The component interfaces of the OMA standardized components and services are written in IDL. As mentioned earlier (Section 3.1), OMG just issues specifications without implementation, so one might find more than one implementation for a certain standardized service or function (component) from different venders. And sometimes the implementation may have extended functionality compared to the specification issued from OMG. But, on the other hand, some services might not have any implementations in spite the fact that an implementation for that standardized service would help the software architects, designers, and developers significantly.

Since the common services and functions in OMA categories (CORBAservices, CORBAfacilities, and CORBAdomain) have a standardized interface written in a common alphabet (i.e. the IDL) one does not have to buy these services and ORBs from the same vender. As stated earlier (Section 3.1), all ORBs implement the common protocol IIOP and all the standardized services use the same interface. As a result, one can replace these common services with others from another vender. One can even change the ORB itself. IDL and interface standardization generally affords great flexibility.

CHAPTER VI

"LIBRARY SYSTEM" CASE STUDY

Library System is a common problem that has been used frequently in the software engineering research efforts as an illustrative example. It has been used because of its clarity, familiarity, and ease of understanding.

In this chapter an informal specification for the library problem is introduced (Section 6.1). The specification of the system is given in Section 6.2 as a component-based system. An IDL was written for the interfaces of different components of the system (Section 6.3). A discussion of the functional and non-functional prosperities of the system is given in Chapter 7.

6.1 Informal Description of the "Library System"

What follows is a description of the library problem as it was informally described by Wing [Wing 88].

Consider a small library database with the following transactions:

1. Check out a copy of a book. Return a copy of the book.
2. Add a copy of a book to the library. Remove a copy of a book from the library.
3. Get the list of books by a particular author or in a particular subject area.
4. Determine the list of books currently checked out by a particular borrower.

5. Find out what borrower last checked out a particular copy of a book.
There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can also perform transaction 4 to determine the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for check-out or be checked out.
- No copy of a book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

6.2 Component-Based Specification of the "Library System"

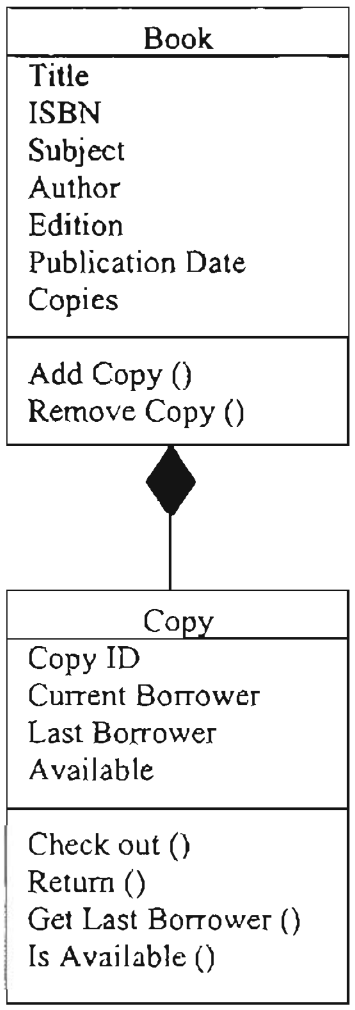
The early systems were generally developed in an ad hoc software development approach [Yourdon and Argila 96]: every system was unique, developers did not take the reuse concepts into considerations, no formal methods were used, and these systems were difficult to maintain and evolve. As the time went and developers became more concerned about developing maintainable and scalable systems, there was a need for a standardized process as well as methods and techniques to develop software systems. A large number of tools and methods have been developed for this purpose. Object Oriented Analysis (OOA) is one of these methods.

OOA has a number of supporting tools to model and represent the real world object of a system and the relationships and the roles that the objects play in a system [Brown 02]. OOA is mainly a design approach that can be performed using different supporting tools and programming languages. Usually, the object oriented analysis model serves two purposes [Yourdon and Argila 96]. First, it serves the formalization of the view of the real world in which the system will be built. Secondly, the object oriented analysis model establishes how the different objects of the systems work together to perform the tasks of the system being modeled. The main advantage of using OOA is to take advantage of the object oriented way of thinking that makes systems generally easy to maintain and debug by using a clearly defined structure. The object oriented way of thinking is claimed to be a natural way of thinking of systems as objects with attributes and methods representing real world objects [Yourdon and Argila 96].

The first step of object oriented analysis is specifying the different objects of the system. These objects represent the basic building blocks of the system. This step is fundamental because all of the other steps are built on this step. The basic objects of our case study (library system) are library catalog, book, copy, author, user, borrower, and staff user.

Figure 8 shows the class diagrams for the book and copy classes, the copy class is part of the book class. Figure 9 shows the class hierarchy diagram for the Library User class; Staff user and Borrower both inherit The Library User class. Figure 10 shows the Library Catalog class.

Figure 11 shows the relationships between objects in the library system and it also shows some constrains. The library system has one library catalog, one or more staff users, and one or more borrowers. The library catalog class has one or more books, and each book has one or more copies. Each book has one or more authors. Every borrower cannot have more than a predefined number of books checked out at the same time. Every copy of a book can only be checked out by one borrower at a time.



◆ : UML notation represents composition relationship

Figure 8. Book and Copy class diagram

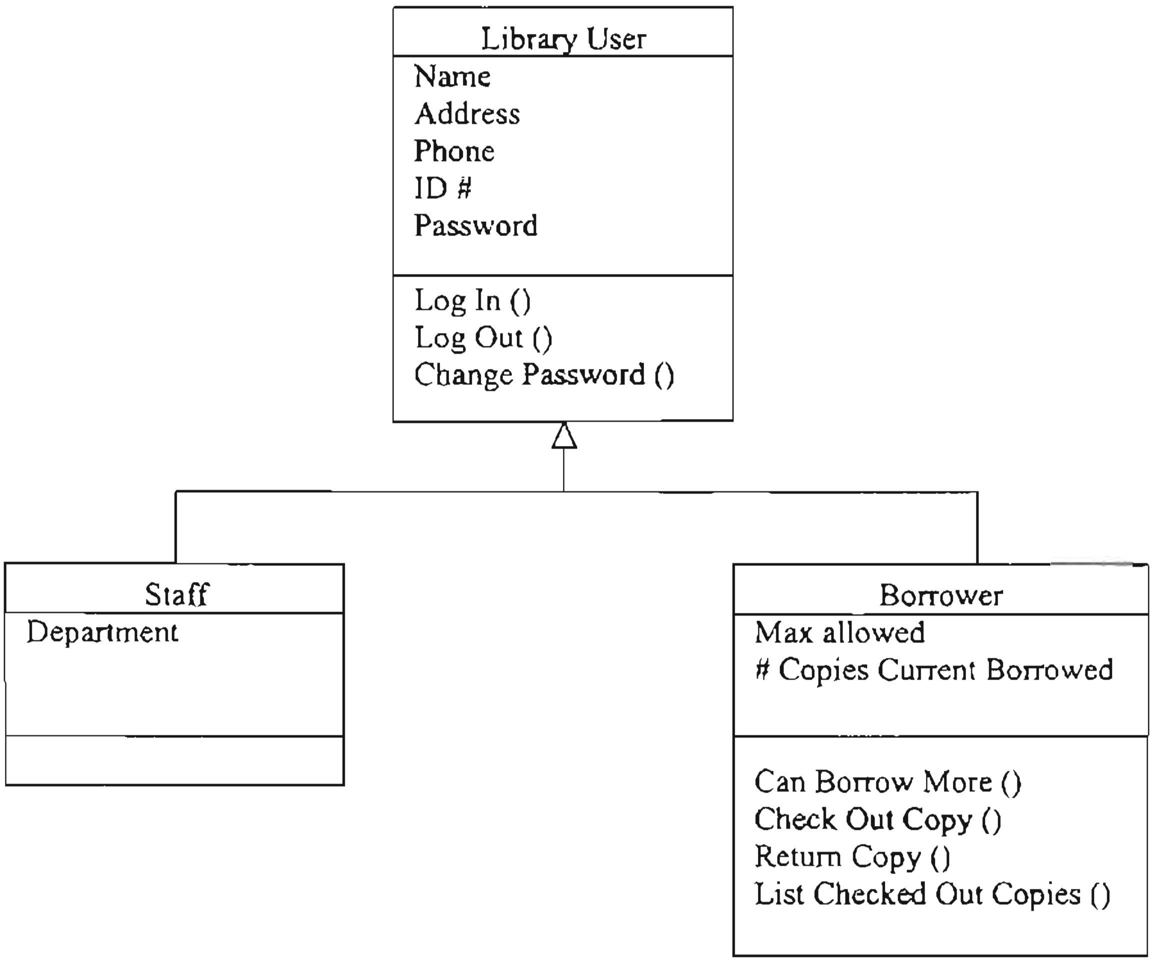


Figure 9. Library User class hierarchy diagram

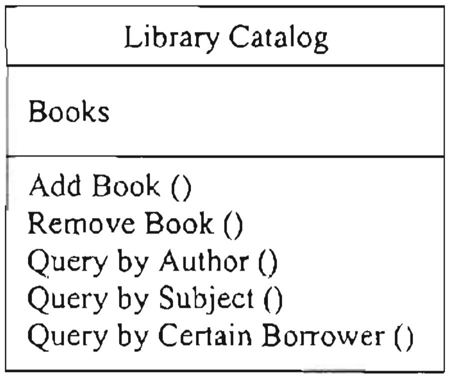
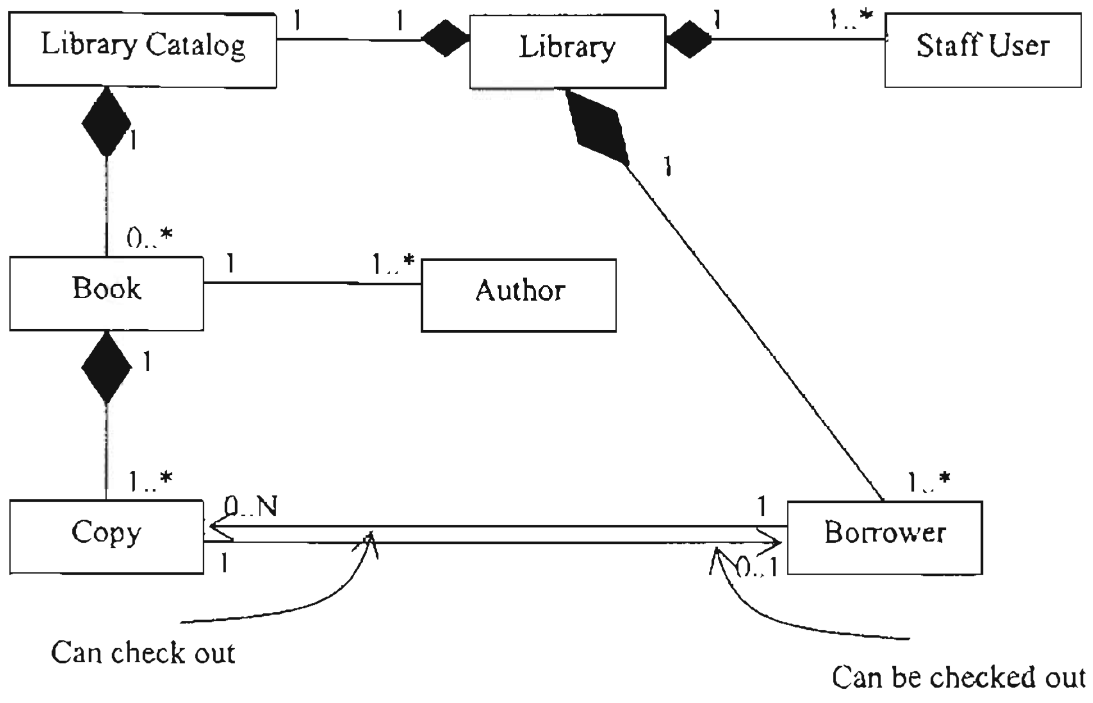


Figure 10. Library Catalog class

A use-case diagram is a UML diagram used to "document what functions the system should offer to the users" [Brown 02]. Use-case diagrams show how different actors can use the system (according to the specifications). They show what the system does but not how, i.e., they show the black-box behavior of the system rather than its mechanisms. There are two actors in the library system: ordinary borrower and staff user. The first actor, i.e., the ordinary borrower, can ask the system to perform three different tasks according to the specifications (see Figure 12). These tasks are: show the list of books by a particular author, show the list of books in a particular subject area, and get the list of the currently borrowed books by that borrower.

The second use-case diagram (see Figure 13) is for the staff user, who can use the system in eight different cases. Two of the use-cases are performed on behalf of the borrower. They are: checking out a copy of a book and returning a copy of a book. Another three are analogous to the borrower use-cases. They are: show a list of books by a particular author, show a list of books in a particular subject area, and get a list of the currently borrowed books by a particular borrower (any borrower). Another task is to get what borrower last checked out a particular copy of a book. The remaining tasks are related to keeping the library catalog updated by adding copies to the library and removing copies from the library. We might also add some actors or some use-cases that are not in the problem specifications, for example there should be an administrator for the system that adds new staff users, Also, staff users should be able to do more tasks like adding and deleting borrowers. However, in the rest of this chapter the original specifications [Wing 88] will be adhered to closely.



◆: UML notation represents composition relationship

N: Max number of books can be checked out at the same time by a particular borrower
1..*: means can have one or more objects
0..1: means can have zero or one object but not more than one
0..*: means can have zero or more objects
0..N: means can have zero to N objects

Figure 11. Object relationships and constrains

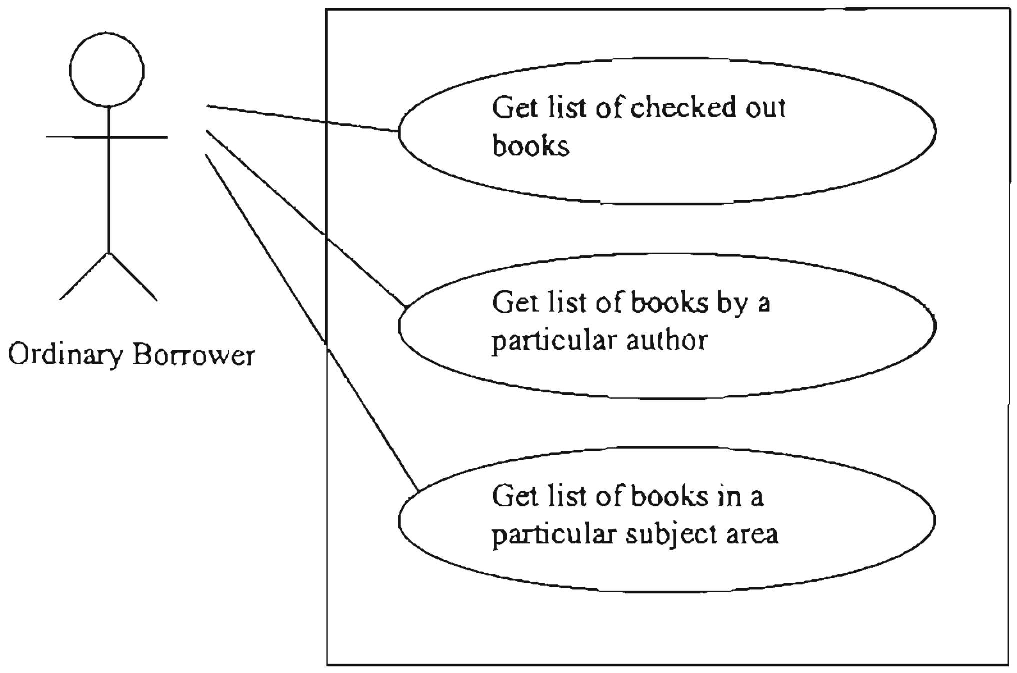

Get list of checked out books
Get list of books by a particular author
Get list of books in a particular subject area
Ordinary Borrower

Figure 12. Ordinary Borrower use-case diagram

Now we will group the similar services and objects into components. In this case study, the services were grouped into three components: catalog based services, library access services, and library station services. Of course, this is in addition to the standardized OMA components that the system will use. Figure 14 shows the library system components with their interfaces and suggested deployment for these components.

In the following paragraphs, the different components will be introduced along with their interfaces and some implementation details.

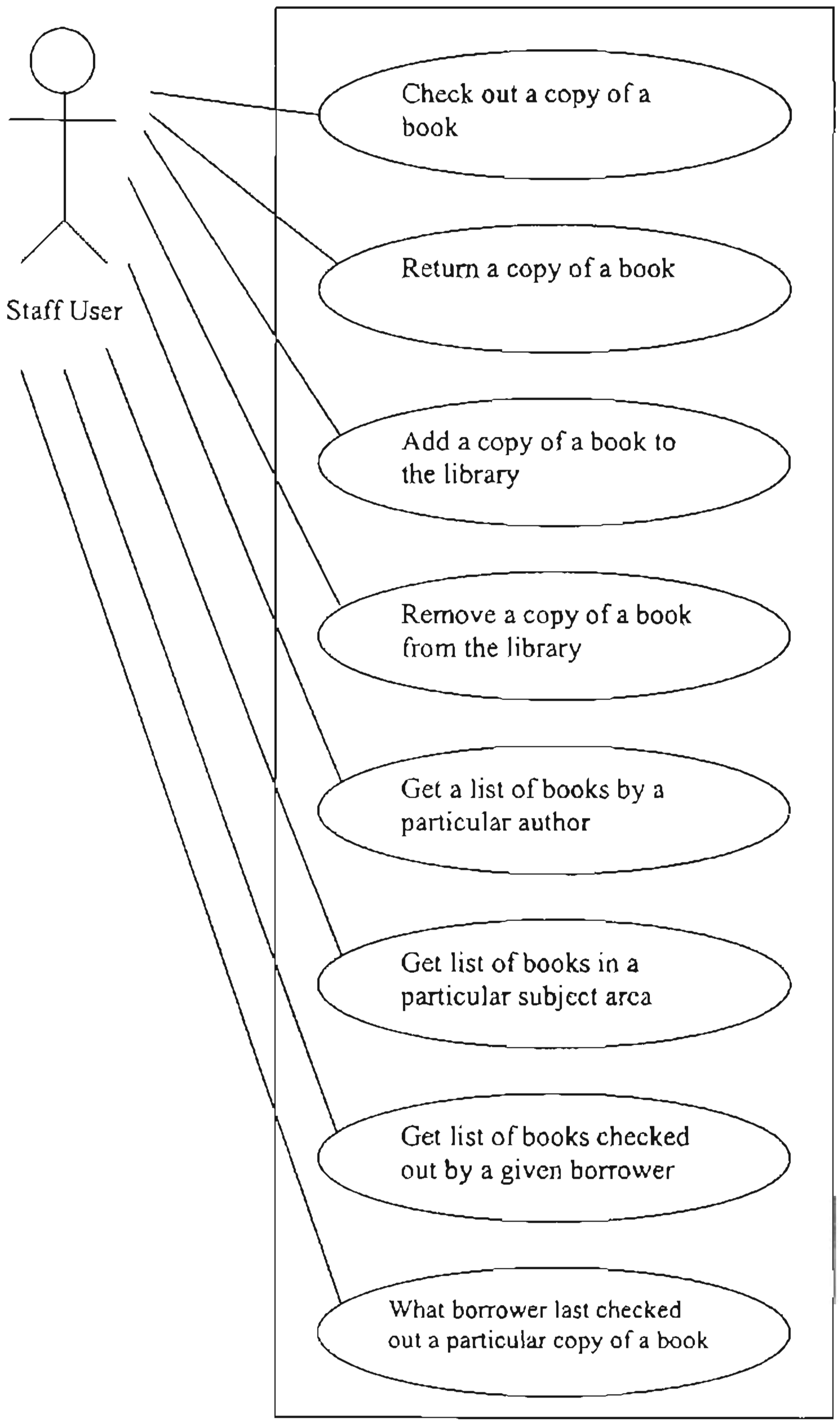
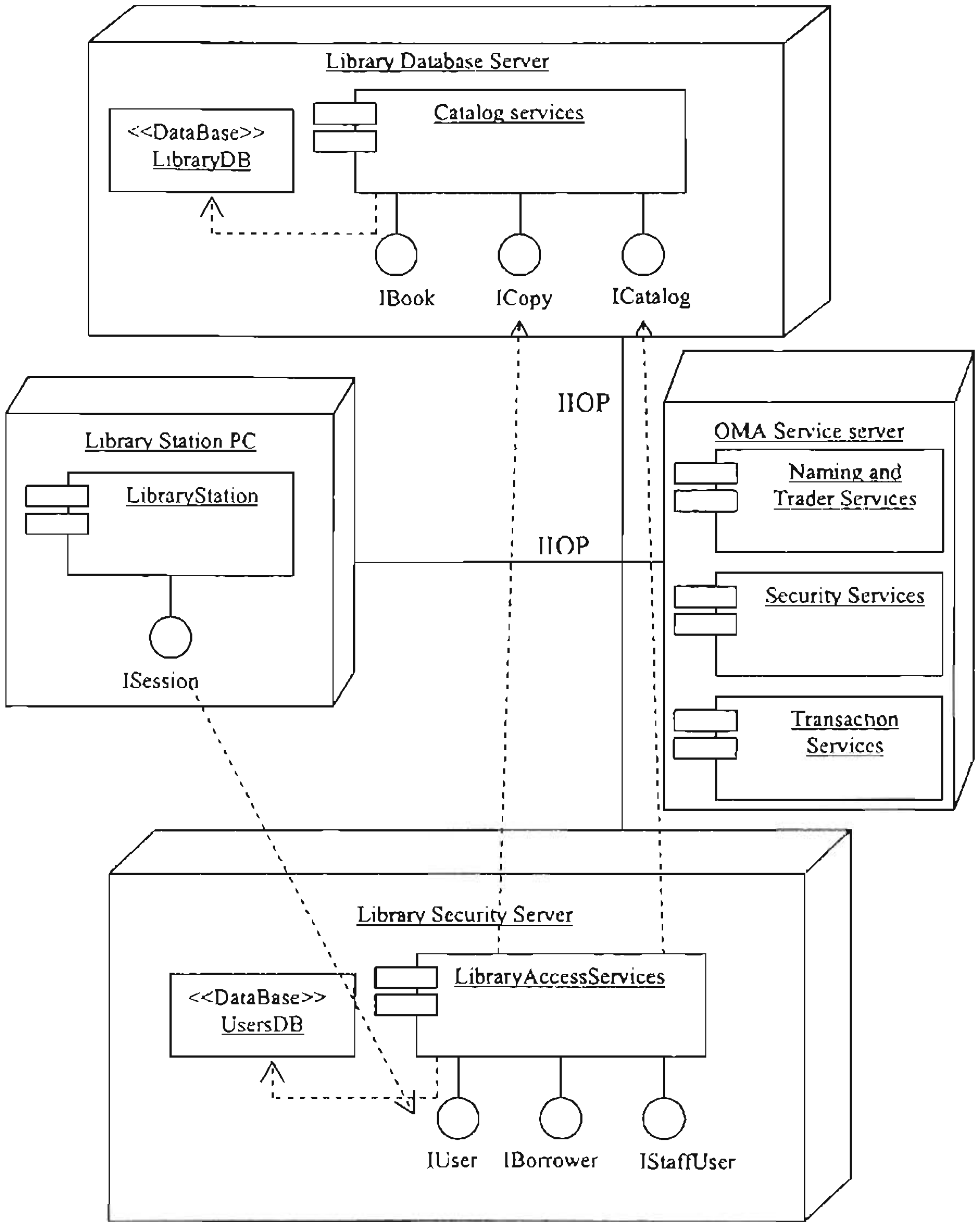

Staff User
Check out a copy of a book
Return a copy of a book
Add a copy of a book to the library
Remove a copy of a book from the library
Get a list of books by a particular author
Get list of books in a particular subject area
Get list of books checked out by a given borrower
What borrower last checked out a particular copy of a book

Figure 13. Staff User use-case diagram



IIOP: the Internet Interoperability protocol.

The "I" at the beginning of an identifier stands for Interface; for example, IUser stands for User Interface.

- - - - - - dependency

——— connection

Figure 14. Library system components and deployment diagram

A point about notation is in order at this point. In the rest of this section and the next section, the OMG's style guide for identifier formation (how to make up variable names) [Siegel 00] was used. Interfaces, datatypes, and exceptions start with capital letters, and if they consist of more than one word, the first letter of each word is capitalized with no spaces between words. Names of operations, parameters, and structure elements will be all lower case letters with underscores to separate the words. Constants and enumeration values will be all capital letters with underscores to separate the words.

The system has one global user-defined data type (`UserID`) and consists of three application components. The system will get the benefit of the following OMA services: Naming and Trader services, Security services, and Transaction services (see Figure 14).

6.2.1 `CatalogServices` Component.

• Description: This component deals with the services concerning the library catalog, book, and copy objects. The operations are: adding and removing books and copies of books, checking out copies of books, and querying the library catalog database.

• User-Defined Types and Structures: `BookTitle, BookISBN, BookSubject, BookAuthor, BookPublicationDate, CopyID, BookCopies, Books.`

• Interfaces:

•• `Catalog` interface

••• Glossary: This interface is for adding and removing books from the library catalog.

••• State Variables: None.

••• Operations:

add_book

Description: adding a book to the library. This operation is called when adding the first copy of a book.

Parameters: input details of type BookDetails; no return.

Implementation: first check if the book is in the database or not; if it is not in the database, add the book to the catalog and use the input parameter details for the new book's details.

Exceptions: raises one exception BookAlreadyInCatalog if a book with the same ISBN number already exists in the library catalog database.

remove_book

Description: removing a book from the library catalog. A book is removed from the library catalog after its last copy has been removed.

Parameters: input book_isbn of type BookISBN; no return.

Implementation: find the book in the library database and then remove it, make sure that the Book doesn't have any copies attached to it.

Exceptions: raises one exception BookNotInCatalog when trying to remove a book that is not in the library catalog.

get_copy_ref

Description: get a reference to a Copy object in the library catalog.

Parameters: copy_id of the CopyId; return a reference to a Copy object.

Implementation: find the `Copy` object with the `copy_id;` and if it was not active, activate it.

Exceptions: raises `CopyIsNotFound` if the `copy_id` is not found in the library catalog database.

`query_by_author`

Description: find the list of books written by a certain author.

Parameters: input `author_name` of type `Author`; output `result_books` of type `Books`; no return.

Implementation: find the list of books one of whose authors matches the first input parameter author, return the list of books in the output parameter `result_books`.

Exceptions: none.

`Query_by_subject`

Description: find the list of books on a certain subject.

Parameters: input `book_subject` of type `BookSubject`; output `result_books` of type `Books`; no return.

Implementation: find the list of books in the subject area passed in the first input parameter `book_subject`, return the list of books in the output parameter `result_books`.

Exceptions: none.

`Query_by_certain_borrower`

Description: find the list of books borrowed by a certain borrower.

Parameters: input `borrower` of type `UserID`; output `result_books` of type `Books`; no return.

Implementation: find the list of book currently borrowed by the borrower with the UserId passed as the first input parameter, return the list of books in the output parameter result_books.

Exceptions: none.

•• Book interface

••• Glossary: This interface represents the object Book in the library catalog with the operations concerning the Book object.

••• State Variables: details of type BookDetails, copies of type BookCopies, and copies_count of type short.

••• Operations

add_copy

Description: add a copy of a book to the library catalog.

Parameters: output number_of_copies of type short; return of type CopyId.

Implementation: assign a CopyId, add one copy to the book, add the copy to the attribute copies, add the copy to the database, add one to the copies_count, update the number_of_copies with copie_count (we can benefit from Transaction services, which is part of OMA, since this operation contains more than one step, and in case of failure of any kind, we can call rollback or submit changes).

Exceptions: none.

remove_copy

Description: remove a copy of a book from the library catalog.

Parameters: input id of type CopyId; output number_of_copies of type short; no return.

Implementation: search the copies related to the object, remove it, then remove the copy from the database, subtract one from the copies_count, update number_of_copies with copies_count (we should benefit from the OMA Transaction services here also).

Exceptions: raises CopyIdNotFound if there is no match in the list of copies related to the current object.

•• Copy interface

••• Glossary: This interface represents the Copy object. Each copy is connected to a book and has a unique CopyId. Copy represents the physical object copy, while Book has no physical existence in the library system.

••• State Variables: copy_id of the type CopyId, available of the type boolean, last_borrower of the type UserId, and current_borrower of the type UserId.

••• Operations

check_out

Description: check out a copy by a certain borrower. This operation is not called directly, it is called from the Borrower object operation check_out_copy because every borrower can borrow up to a certain pre-defined number of books at the same time; so, before calling this operation, the system should make sure that the borrower does not exceed that number.

Parameters: input borrower of the type UserId; no return.

Implementation: first check to see if the book is available to be checked out; if true,set the available attribute to false and then set the `current_borrower` to the input parameter `borrower` (we should benefit from the OMA Transaction services here also).

Exceptions: raise `CopyCheckOut` if the copy was checked out by another borrower (by checking the attribute `available` we can find out if the copy has been checked out or not).

`return`

Description: return a copy of a book to the library after having borrowed it for some time. This operation is not called directly either. It is called from the `borrower` object to adjust the number of books checked out by a borrower.

Parameters: none; no return.

Implementation: first check to see if the copy was checked out or not; if it was checked out, then the operation sets the `available` attribute to `true` and updates the `last_borrower` with the `current_borrower` (we should benefit from the OMA Transaction services here also).

Exceptions: if the copy was not checked out (attribute `available` is `true`), then the operation raises `CopyNotCheckedOut`.

`get_last_borrower`

Description: return the last borrower of a copy of a book.

Parameters: output `borrower` of type `UserId`; no return.

Implementation: set the output parameter `borrower` with the `last_borrower` attribute.

Exceptions: if the copy has not been checked out before it raises the exception CopyNotBorrowedBefore. We can find out whether or not the copy has been checked out by checking the attributes last_borrower and current_borrower, if both are null, then the copy has not been checked out.

6.2.2 LibraryAccessServices Component

• Description: This component deals with the services concerning accessing the library system through the User, Borrower, and StaffUser objects operations.

• User-Defined Types and Structures: UserPassword, UserType, UserAddress, UserDetails.

• Interfaces:

•• User interface

••• Glossary: This interface has common attributes and operations for the Borrower and StaffUser interfaces. The Borrower and StaffUser interfaces inherit this interface.

••• State Variables: details of type UserDetails, type of type UserType, and current_state of type boolean.

••• Operations

login

Description: login to the library system with a valid password.

Parameters: input pw of type UserPassword; output user_type of type UserType; no return.

Implementation: cross check the passed parameter pw with the user password in the details attribute; if there is a matched, return and set the attribute current_state to true.

Exceptions: raises WrongPassword if the password that is passed does not match the password in the details attribute of that object.

logout

Description: log out of the library system.

Parameters: none; no return.

Implementation: set the current_state to false.

Exceptions: no exceptions.

change_password

Description: change the user password.

Parameters: input old_password of type UserPassword; input new_password of type UserPassword; return boolean.

Implementation: first determine whether or not the new password is empty. If not empty, then cross check the old_password with the password in the attribute details; if there is a match, set the password in the user_details attribute to the new_password.

Exceptions: raises OldPasswordDoesnotMatch if the old_password does not match the password in the attribute struct details; and if the new_password parameter is an empty string, it raises NewPasswordEmpty.

•• Borrower interface

••• Glossary: This interface deals with the operations that the borrower object can perform.

••• State Variables: max_books_allowed of type short and currently_borrowed_books of type short.

••• Operations

check_out_copy

Description: check out a copy of a book by this borrower object from the library and the service can be requested by a StaffUser.

Parameters: input service_requester of type UserId; input copy_id of type CopyId (which is part of the CatalogServices component); no return.

Implementation: make sure that the service_requester (first input parameter) is authorized to request this service by verifying it to be of type StaffUser. Call the can_borrow_more operation; if it returns true, add one to the attribute num_currently_borrowed_books. Then, get a reference to the Copy object with the copy_id (second parameter) by calling get_copy_ref from the Catalog interface. Last, call operation check_out from the Copy object with the UserId of this object as a parameter. We should benefit from the OMA Transaction services here also.

Exceptions: when calling the can_borrow_more operation, if it returns false, raise the CanNotBorrowMore exception. If the service_requester is not authorized to perform the operation, it raises UnAuthorisedRequester. If the copy_id is not in the system, it will raise the same exception that the Catalog interface raises, i.e., CopyIdNotFound.

return_copy

Description: return a copy of a book, the service can be requested by a StaffUser.

Parameters: input service_requester of type UserId; input copy_id of type CopyId (which is part of the CatalogServices component); no return.

Implementation: make sure that the service_requester (first input parameter) is authorized to request this service by verifying that it is of type to be StaffUser. Subtract one from num_currently_borrowed_books. Then, get a reference to the Copy object with the copy_id (second parameter) by calling get_copy_ref from the Catalog interface. Last, call operation return from the Copy object. We should benefit from the OMA Transaction services here as well.

Exceptions: if the return operation of the Copy interface raises the exception CopyNotCheckedOut, then raise the same exception. If the service_requester is not authorized to perform the operation, it raises UnAuthorisedRequester. If the copy_id is not in the system, it will raise the same exception that the Catalog interface raises, i.e., CopyIdNotFound.

borrowed_books

Description: list the currently borrowed books by this borrower.

Parameters: input service_requester of type UserId; output result_books of type Books; no return.

Implementation: if service_requester is not the same as the borrower, make sure that the service_requester (first input parameter) is authorized to request this service by verifying that it is of type StaffUser. Then, call the query_by_certain_borrower operation from the Catalog interface with the

UserId of the current object as a parameter. Books, the second output parameter, will hold the returned list of books.

Exceptions: if the service_requester is not authorized to perform the operation on this object, it raises UnAuthorisedRequester.

can_borrow_more

Description: check to see if this Borrower object can borrow more books.

Parameters: none; return boolean.

Implementation: check to see if num_currently_borrowed_books fewer than the max_books_allowed, then return true otherwise return false.

Exceptions: none.

•• UserStaff interface

••• Glossary: This interface represents the staff user object

••• State Variables: department of type string.

••• Operations: inherits the User object operations.

6.2.3 LibraryStation Component

• Description: This component deals with the services concerning the opening and closing of a session with the library system from a library station.

• User-Defined Types and Structures: None.

• Interfaces:

•• Session interface

••• Glossary: The Session interface enables the library station to access the library system.

••• State Variables: `is_open` of type `boolean`; `current_user_type` of type `UserType`.

••• Operations

`get_user_ref`

Description: get a reference to the `User` object with the `UserId`.

Parameters: `user_id` of type `UserId`; return a reference to the `User` object.

Implementation: find the `User` with the `user_id`, and if it was not active, activate it.

Exceptions: if the `user_id` is not in the users database, raise `UnknownUser`.

`open`

Description: open a session with the library system from a library station.

Parameters: input `user` of type `UserId`; input `password` of type `UserPasswrod`; output `user_type` of type `UserType`; no return.

Implementation: call operation `get_user_ref` with `user_id` as a parameter. When getting the reference for the `User` object, the object should call operation `login` of the `User` object with `password` and `user_type` as parameters. The `login` operation (if successful) will return the user type `STAFF_USER` or `BORROWER` in the second parameter `user_type`. Then the `is_open` attribute is changed to `true` and the `current_user_type` attribute is set to the `user_type` returned from the `login` operation.

Exceptions: if operation `get_user_ref` raised `UnknownUser`, this operation will raise the same exception as well. Also, if the operation login of the `User` object raised exception `WrongPassword`, this object will raise the same exception.

Close

Description: close the open Session.

Parameters: none; no return.

Implementation: changing the attribute is_open to false and deleting the Session object.

Exceptions: none.

6.3 "Library System" in OMG IDL

The following code is the OMG IDL for the library case study according to the specifications given in Section 6.1:

```
typedef long UserId; // Global definition for UserId type

// Catalog Services Module declarations
module CatalogServices
{
     // user defined type declarations
     typedef string BookTitle;
     typedef string BookISBN;
     typedef string BookSubject;
     typedef string BookAuthor;
     typedef long BookPublicationDate;
     typedef long CopyId;

     // BookDetails structure holding the detailed
     // information of Book
     struct BookDetails
     {
          BookTitle title;
          BookISBN isbn;
          BookSubject subject;
          sequence<BookAuthor> author;
          short edition;
          BookPublicationDate publication_date;
     };

     // Catalog Services exceptions declarations
```

```
exception CopyCheckedOut;
exception CopyNotCheckedOut;
exception BookNotInCatalog;
exception CopyIdNotFound;
exception CopyNotBorrowedBefore;
exception BookAlreadyInCatalog;

interface Copy; //forward reference

typedef sequence<Copy> BookCopies;
typedef sequence<BookDetails> Books;

// Catalog interface
interface Catalog
{
     boolean add_book(in BookDetails details)
          raises (BookAlreadyInCatalog);

     void remove_book(in BookISBN book_isbn)
          raises (BookNotInCatalog);

     Copy get_copy_ref(in CopyId copy_id)
          raises (CopyIdNotFound);

     void query_by_author(
               in Author author_name,
               out Books result_books);

     void query_by_subject(
               in BookSubject book_subject,
               out Books result_books);

     void query_by_certain_borrower(
               in UserId borrower,
               out Books result_books);

};



// Book interface definition
interface Book
{
     attribute BookDetails details;
     attribute BookCopies copies;
     attribute short copies_count
     CopyId add_copy(out short number_of_copies);
```

```
            void remove_copy(
                    in CopyId id
                    out short number_of_copies)
                raises CopyIdNotFound;
        };

        // Copy interface definition
        interface Copy
        {
            attribute CopyId copy_id;
            attribute boolean available;
            attribute UserId last_borrower;
            attribute UserId current_borrower;
            void check_out(in UserId borrower)
                raises (CopyCheckedOut);

            void return()
                raises (CopyNotCheckedOut);

            void get_last_borrower(out UserId borrower)
                raises (CopyNotBorrowedBefore);
        };

    };

    // Library Access Services Module declarations
    module LAServices
    {
        // user defined types and structures
        typedef string UserPassword;
        enum UserType {BORROWER, STAFF_USER};
        struct UserAddress
        {
            string line1;
            string line2;
            String city;
            string zip_code;
            String state;
        };

        struct UserDetails
        {
            string name;
            UserAddress address;
            String phone;
            UserId user_id;
            UserPassword password;
```

```
};

// Library Access Services exceptions declarations
exception UnAuthorizedRequester;
exception CanNotBorrowMore;
exception UnKnownUser;
exception WrongPassword;
exception OldPasswordDoesnotMatch;
exception NewPasswordEmpty;

// User interface declaration
interface User
{
     attribute UserDetails details;
     attribute UserType type;
     attribute boolean current_state;

     void login(
               in UserPassword pw;
               out UserType user_type)
          raises WrongPassword;

     void logout();

     boolean change_password(
               in UserPassword old_password,
               in UserPassword new_password)
          raises (OldPasswordDoesnotMatch,
                  NewPasswordEmpty);
};

// Borrower interface declaration
interface Borrower:User
{
     attribute short max_book_allowed;
     attribute short num_currently_borrowed_books;

     boolean check_out_copy(
               in UserId service_requester,
               in CatalogServices::CopyId copy_id)
          raises (CanNotBorrowMore,
                  UnAuthorisedRequester,
                  CatalogServices::CopyIdNotFound);

     boolean return_copy(
               in UserId service_requester,
               in CatalogServices::CopyId copy_id)
```

```
                raises (CatalogServices::CopyNotCheckedOut,
                        UnAuthorisedRequester,
                        CatalogServices::CopyIdNotFound);

           void borrowed_books(
                     in UserId service_requester,
                     out CatalogServices::Books
                     result_books)
                raises (UnAuthorisedRequester);

           boolean can_borrow_more();
      };

      // Staff User interface declaration
      interface StaffUser:user
      {
           attribute string department;
      };
};

// Library Station Module declaration
module LibraryStation
{
      //Session interface declaration
      Interface Session
      {
           attribute boolean is_open;
           attribute LAServices::UserType current_user_type;

           LAServices::User get_User_ref (in UserId user_id)
                raises (LAServices::UnknownUser);
           void open(
                     in UserId user_id,
                     in LAServices::UserPassword password,
                     out LAServices::UserType user_type)
                raises (LAServices::UnknownUser,
                        LAServices::WrongPassword);

           void close();
      };
};
```

CHAPTER VII

DISCUSSION

The two sections in this chapter discuss the different properties of the library system case study that was described in detail in Chapter VI. The first section discusses the functional properties of the library system and the second section discusses its extra-functional properties.

7.1 Functional Properties

The goal of this section is to make sure that the component interface design introduced for the library system in Sections 6.2 and 6.3 meets the requirements that were mentioned informally in Section 6.1. What follows describes how these specifications were handled in the design of the component interfaces. For transaction number 1 in the description (Section 6.1), 'Check out a copy of a book. Return a copy of a book', two interfaces are responsible for performing this transaction. The first object is the `Borrower` object with its methods `check_out_copy` and `return_copy`. These two operations, after ensuring that the call was initiated by an authorized user, call the `Copy` object operations `check_out` and `return`, respectively.

For transaction 2 (add a copy of a book to the library and remove a copy of a book from the library), two objects are responsible for performing this transaction: Catalog and Book. If the copy was the first copy of a book to be added to the library, then the add_book method of the Catalog object will be called first. Subsequently, the add_copy method of the Book object will be called. If the book is already in the library and we just want to add another copy to the library, the add_copy method of the Book object will be called directly. Correspondingly, when removing a copy of a book from the library, after calling the Book object's remove_copy method, we check to see if it was the last copy in order to remove the whole book from the library catalog database by calling the remove_book method of the Catalog object.

Transactions 3 and 4 are basically queries on the library catalog database with different criteria for each one. Database queries for books written by a particular author are handled by the Catalog interface through the query_by_author operation. Database queries for books on a particular subject are handled by the Catalog interface through the query_by_subject operation. Querying the database for the list of books checked out by a particular borrower involves the Catalog and Borrower interfaces. After ensuring that the request was initiated by an authorized requestor (here the authorized requestor could be the borrower herself/himself or any staff user) in the borrowed_books operation of the Borrower interface, the borrowed_books operation calls the query_by_certain_borrower operation of the Catalog interface.

Transaction 5 (find out what borrower last checked out a particular copy of a book) is conducted by the interface Copy through the get_last_borrower operation.

In order that the system impose the restriction of what kind of tasks a staff user or a borrower can perform, the system should have a way to distinguish among the current users, and there should be a logging system to control what kind of functions the current users can perform on the different objects of the system. The `User` interface with its two operations `login` and `logout`, and the `Session` interface with its two operations `open` and `close` control this by keeping track of the type of the current user in the attribute `current_user_type` of the `Session` interface. The `User` interface components can use there operations to initiate the right tasks according to the user types.

The restrictions (all copies in the library must be available for check-out or be checked out and no copy can be both available and checked out at the same time) are both imposed by the state attribute `available` (`boolen`) of the `Copy` interface. Since `available` is of type `boolean`, its value can be either `true` or `false` (available or checked out), and not both at the same time.

The last restriction (a borrower may not have more than a predefined number of books checked out at one time) is imposed through the two state variables `max_book_allowed` and `num_currently_borrowed_books` of the `Borrower` interface.

IDL was originally designed [Clements et al. 99] [Siegel 00] to describe the functionality of black box components through their interfaces. Thus it is not surprising that by looking at and studying an IDL, one can tell what functionality the components have, but one cannot tell how those functionalities are affected. This leaves a software developer with some flexibility in how to implement a specified component according to the specification and the design documents.

7.2 Extra-Functional Properties

Extra-functional properties, or as they also called Quality of Services (QoS) prosperities, cannot be predicted from the component properties or totally controlled by the application components. As it was mentioned earlier (see Section 2.5) one cannot predict the overall system properties from the components properties, because the over all system properties do not just depend on the system components and also because of what is referred to the as architectural mismatch phenomenon. This phenomenon depends on the component model used as well as the infrastructure and the surrounding computing environment.

The quality of service properties may include performance, security, latency, and accuracy. Most of these properties cannot be really tested unless the system is completely developed and deployed in the targeted environment. What follows contains some of the implementation details of the library system case study that help in controlling these properties. It was mentioned in Section 6.2 that the system could benefit from using the following OMA services: Naming and Trader services, Transaction services, and Security services. All of these services interfaces were written using OMG IDL, and the only way to access them is through their interfaces. Using these services help control the quality of services properties. In this case study Transaction services were used when a new copy was added to the library, when a copy was removed from the library, when a copy was checked out from the library, and when a copy was returned to the library. All of the previous operations consist of more than one step (operation) and some of them may include accessing more than one database resource. These databases may exist on one machine or on different machines across a network. One of these operations might fail, we need a mechanism, in case on operation would fail, to roll back the other operation. This is done using the Transaction service which helps in the reliability and accuracy properties.

Installing and configuring the Security services of the OMA along with the application components supplies the system with the security it needs to support its operations. Security service prevents unauthorized access to the system components from other components that might exist in the environment. The developers and users might not be aware of all of the objects that they are interacting with. As Siegel stated [Siegel 00] "An OMA security architecture should allow for environments where mistrust between objects is ubiquitous".

Naming and Trader Service locates the system components and helps the components know about one another in runtime environment. This service will indeed increase the reliability and performance of the overall system. In case a component is moved from its place, one still can locate the new place of the component exactly. This is analogous to address forwarding in real life through ORB and the Naming and Trader service. Or, if there is a great demand on a certain component, one could have more than one copy of that component running on different places, and the only way to make all of this transparent is by the Naming and Trader service.

CHAPTER VIII

SUMMARY AND FUTURE WORK

We live in fast changing and growing world. Reliance on software systems is increasing every day. As a result, developing reliable, easily maintained, scalable, and efficient software systems in short time and with affordable cost is essential for this modern life. CBSD (Component-Based Software Development) is one of the preferred ways [Dong 02] [Cai et al. 00] to develop software systems that meet the above criteria. CBSD faces some problems [Crnkovic and Larsson 02] [Sommerville 01] such as: high maintenance cost, lack of supporting development tools, the "not-invented-here" syndrome, frequent updating of the components library, finding the right components and adopting them, and component composition. System components are relatively easy to be developed but hard to combine or compose together. In particular, one should make sure that components fit in a new environment when they are reused.

OMG IDL was originally designed to specify the functionality of the components of a system, but its function has been extended to compose the components together. Using the OMA standard services helps in controlling the extra-functional properties. A software designer should have a good knowledge of the standard components and services in the component model in order to use them when (s)he needs them rather than writing them again. A good design is essential for a component system to succeed.

Software architecture is strongly related to study of component-based development. It is hard to cover all parts of OMG IDL in one case study. Object Management Architecture (OMA) and standardized components shorten the time and the cost required to develop component-based systems [Cai et al. 00].

Chapter II introduced a general overview of the Component Based Software Development. Chapter III presented some of the popular middleware technologies including CORBA, COM and DCOM, and JavaBeans and Enterprise JavaBeans. Chapter IV discussed Object Management Group's Interface Defining Language. Chapter V presented the Object Management Architecture (OMA), which is the Object Management Group's view for a component technology. Chapter VI discussed a simple library system as a case study. The library system was first presented informally, then a component-based analysis and design of the library system was conducted, and finally an OMG IDL for the library system components was written. Chapter VII discussed the functional and extra functional properties of the library system.

A future work in this area might be studying the OMA and suggesting new services and components, such as standardized general library components, in one of the three different standardized categories which are CORBAservices, CORBAfacilites, and CORBAdomain. An extension to the OMG IDL might be developed to give OMG IDL more power in component composition. Other future work include developing CASE tools to help in component assembly and visualization tools to show the existing dependencies among the various components of a system.

REFERENCES

[Bachmann et al. 00] Flex Bachmann, Len Bass, Charles Buhman, Sniago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering", Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 2000.

[Borgida and Devanbu 99] Alex Borgida and Prem Devanbu, "Adding More "DL" to "IDL": Towards More Knowledgeable Component Inter-Operability", *Proceedings of the 21st International Conference on Software Engineering*, pp. 378-387, Los Angeles, California, May 1999.

[Brown 02] David William Brown, *An Introduction to Object-Oriented Analysis Objects and UML in Plain English,* Second Edition, John Wiley & Sons, Inc., New York, NY, 2002.

[Cai et al. 00] Xia Cai, Michael R. Lyu, Kam-Fai Wong, and Roy Ko, "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes", *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC 2000)*, pp. 372-379, Singapore, December 2000.

[Clements et al. 99] Paul C. Clements, Len Bass, L. Belady, Alan Brown, Peter Freeman, Scott Isensee, Rick Kazman, Herb Krasner, John Musa, Shari Lawrence Pfleeger, Karel Vredenburg, and Tony Wasserman, *Constructing Superior Software*. Macmillan Technical Publishing, Indianapolis, IN, 1999.

[Crnkovic and Larsson 02] Ivica Crnkovic and Magnus Larsson, *Building Reliable Component-Based Software Systems*, Artech House, Inc., Norwood, MA, 2002.

[Crnkovic et al. 02] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, "Anatomy of a Research Project in Predictable Assembly", *Fifth ICSE Workshop on Component-Based Software Engineering white paper. URL: http://www.sei.cmu.edu/pacc/CBSE5/CBSE5_whitepaper.pdf*, Orlando, Florida, May 2002.

[Dong 02] Jing Dong, "Design Component Contracts: Modeling and Analysis of Pattern-Based Composition", Ph.D. Thesis, School of Computer Science, University of

Waterloo, Waterloo, Ontario, Canada, 2002.

[Gudgin 01] Martin Gudgin, *Essential IDL Interface Design for COM*, Addison-Wesley, Pearson Education, Upper Saddle River, NJ, 2001.

[Ivers et al. 02] James Ivers, Nishant Sinha, and Kurt Wallnau, "A Basis for Composition Language CL", Technical Note CMU/SEI-2002-TN-026, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 2002.

[Kruchten 98] Philippe Kruchten, "Modeling Component Systems with the Unified Modeling Language", a position paper presented at *The 1998 International Workshop on Component-Based Software Engineering, URL: http://www.sei.cmu.edu/cbs/icse98/papers/p1.html*, Kyoto, Japan, April 1998.

[Mittermeir et al. 01] Roland T. Mittermeir, Andreas Bollin, Heinz Posewaunig, and Dominik Rauner-Reithmayer, "Goal-Driven Combination of Software Comprehension Approaches for Component Based Development", *Proceedings of the ACM Symposium on Software Reusability (SSR' 01)*, pp. 95-102, Toronto, Canada, May 2001.

[Moreno et al. 02] Gabriel A. Moreno, Scott A. Hissam, and Kurt C. Wallnau, "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling", *Online Proceedings of the fifth International Conference on Software Engineering (ICSE), Workshop on Component-Based Software Engineering, URL: http://www.sei.cmu.edu/pacc/CBSE5/Moreno-cbse5-final.pdf*, Orlando, Florida, May 2002.

[OMG 02] Object Management Group's official Internet web site URL: http://www.omg.com, Last Updated: May 21, 2002, Date Accessed: January-March 2003.

[Siegel 00] Jon Siegel, *CORBA 3 Fundamentals and Programming*, Second Edition, John Wiley & Sons, Inc., New York, NY, 2000.

[Siegel 96] Jon Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons, Inc., New York, NY, 1996.

[Sommerville 01] Ian Sommerville, *Software Engineering*, 6th Edition, Pearson Education Limited, Essex, England, 2001.

[Wing 88] Jeannette M. Wing, "A Study of 12 Specifications of the Library Problem", *IEEE Software*, Vol. 5, No. 4, pp. 66-76, July 1988.

[Yourdon and Argila 96] Edward Yourdon and Carl Argila, *Case Studies in Object Oriented Analysis & Design*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

APPENDICES

APPENDIX A

GLOSSARY

Assembly	A set of components and their interconnections [Crnkovic et al. 02].
CASE	Computer Aided Software Engineering, programs used to support software engineering process activities such as requirements analysis, system modeling, and testing [Sommerville 01].
CBSD	Component-Based Software Development, the process of developing software systems from small pieces (black boxes), called components, by composing them together to form the final system.
CBSE	Component-Based Software Engineering, the engineering discipline that is concerned with developing software systems from small pieces (black boxes), called components, by composing them together to form the final system.
COM	Component Object Model, a middleware component technology (a Microsoft product).
Component	A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party [Crnkovic and Larsson 02].
CORBA	Common Object Request Broker Architecture, a middleware component technology (an OMG product).
COTS	Commercial Off The Shelf, ready-made software components that are available in the component market place.
DCOM	Distributed Component Object Model, the distributed version of Microsoft COM that allows the different components of a system to reside on different machines.

Framework	A Component framework is a piece of software that manages resources shard by a number of components, and provides the underlying mechanisms that enables communication (interaction) among components [Bachmann et al. 00].
IDL	Interface Definition Language, a definition language used to define component interfaces in component-based system.
IIOP	Internet Interoperability protocol, a communication protocol in CORBA that allows the different components to communicate over a network remotely.
MIDL	Microsoft Interface Definition Language, a definition language used to define component interfaces in the COM/DCOM component model.
MSMQ	Microsoft Message Queue, a piece of software that provides support for asynchronous communication between components via a message queue [Clements et al. 99].
MTS	Microsoft Transaction Service, a Microsoft product that provides security and transaction management services [Clements et al. 99].
OMA	Object Management Architecture, a component-based architecture standard that represents the OMG's vision for the component software environment which categorizes objects into four categories: the CORBAservices, CORBAfacilites, CORBAdomain objects, and Application Objects [Siegel 00].
OMG	Object Management Group, an international not-for-profit software consortium that sets standards in the area of distributed object computing. OMG was founded in April 1989 by eleven companies to create a component-based software market place, now it contains more than 500 companies. Some of the standards the OMG has developed include CORBA, UML, OMG IDL, and IIOP [OMG 02].
OOA	Object Oriented Analysis, a design approach used to analyze software systems as objects with attributes and methods.
ORB	Object Request Broker, communication software that allows the different components to communicate with each other in CORBA, and makes the location of the components transparent.

UML	Unified Modeling Language, a standard modeling language that is mainly used for object-oriented modeling. UML is an OMG standard [Sommerville 01].

VITA

Emran Al-Shahrouri

Candidate for the Degree of

Master of Science

Thesis: A SURVEY AND A DETAILED CASE STUDY USING OMG IDL: THE ROLE OF IDL IN COMPONENT COMPOSITION

Major Field: Computer Science

Biographical:

Personal Data: Born in Amman, Jordan, On October 26, 1971, son of Khalil Al-Shahrouri and Asia Al-Shareef.

Education: Received the Bachelor of Science degree in Computer Science from Mu'tah University in June 1993; completed the requirements for the degree of Master of Science in Computer Science at the Computer Science Department at Oklahoma State University in December 2003.

Experience: Working with the Jordan Armed Forces - General Head Quarters as a Computer Programmer and Software Analyst since 1993. Employed by the Information Technology Division - Client Services in Oklahoma State University as Lab Assistant from August 2002 to May 2003.

Professional Memberships: Jordan Computer Society.