INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

CORBA-based Test Architecture For E-commerce Application

By

Wujun Li

A thesis submitted to
The School of Graduate Studies and Research
In partial fulfillment of the degree of

Master in Computer Science

School of Information Technology and Engineering
University of Ottawa
(Ottawa – Carleton Institute for Computer Science)

ACKNOWLEDGEMENTS

I would like to express my heartiest appreciation and gratitude to my thesis supervisor Professor Robert L. Probert for his support, encouragement, and financial help through out my thesis research. I would also like to sincerely thank Louise Defrochers, Rodd Lamarch and Alan Williams for their unconditional help and useful suggestions.

My deepest love and respect go to my family, whose support and understanding have helped me pursue my education.

Finally, I gratefully acknowledge IBM Toronto Lab for their financial support.

To my family

Table of Contents

List of Figures

List of Tables

ABSTRACT

E-Commerce systems are specialized instances of distributed processing systems that involve critical financial transactions. The Common Object Request Broker Architecture (CORBA) is an industrial standard framework to provide sophisticated infrastructure to develop and deploy distributed objects in an open environment. Using CORBA to facilitate the development and testing of e-commerce system can greatly improve the test effectiveness, and shorten the time-to-market cycle. This thesis proposes a cost-effective generic testing architecture for function and performance testing of e-commerce applications. We relate this architecture to middleware standards such as CORBA. We show how the CORBA framework support testing of e-commerce system for robustness and reliability with the testing architecture which is open, highly distributed, and scalable. We also give a high-yield strategy to testing e-commerce system and demonstrate how our testing architecture supports this strategy. We also provide an assessment of the approach by a small, but realistic case study (Online Currency Exchange) involving design, implementation and function testing in detail. During testing a significant design error was detected. Finally we conclude this thesis by summarizing our results and giving some recommended future work.

Chapter 1

Introduction

1.1 Research Motivation

1.1.1 Statement of the problem

Electronic Commerce has become the new frontier for businesses around the world. It is poised to revolutionize the way we do business in the 21st century [Kamthan, 1998].

Electronic commerce is the exchange of secure messages among various distributed processing entities for the purpose of completing an atomic financial transaction. E-commerce applications are financially critical systems performing various distributed business functions or transactions [Zwass, 1996].

In the open distributed environment, such as the Internet, e-commerce is architecturally composed of multiple, decentralized and autonomous processing components or objects exchanging messages across the network. To frequently deploy distributed software components in a timely way with high quality, reliability, responsiveness and availability is the critical challenge in developing e-commerce systems. Testing these highly distributed software components in a heterogeneous environment becomes even more challenging. An advanced distributed test architecture for e-commerce must be developed to meet those challenges.

Obviously, traditional black-box and white-box software test techniques [Beizer, 1990] are very useful to ensure that each software component conforms to its specification. However, integrating these components over a distributed environment requires more sophisticated test environments and procedures.

1.1.2 Motivation

E-commerce application testing is a new research area, and there are relatively few research publications on this problem [Zwass, 1996]. Our research is focused on distributed test execution architectures for the following reasons:

- There have been many different validation methods corresponding to what we call the Test Generation Phase, which starts with the system specification and ends with the semi-automated generation of test suites. However very little research has been conducted based on realistic test execution environments.

- E-commerce is a special instance of advanced distributed systems – new and complex services will be deployed in the context of various distributed object environments. Thus, cost-effective methods to validate and test large and heterogeneous e-commerce systems is an important research topic in its own right, with immense implications for practice.

1.2 Thesis Objective and Contribution

Our main objective is to develop a new, practical, cost-effective design and validation approach (high-yield test strategy) and standards compatible test execution architecture for testing e-commerce products. In other words, the subject of this thesis is to capture requirements for an effective test architecture for e-commerce applications to prepare a open distributed test architecture for e-commerce, and to illustrate and evaluate this architecture for high-yield (defect-directed) test strategies in a realistic case study.

The thesis makes the following contributions:

- Customizes a high-yield (bug finding) test strategy to e-commerce products and demonstrates that this strategy is i) measurable, ii) UML(Unified Modeling Language) – compatible and iii) risk-directed.

- Reviews the established testing environments and test methods for distributed system.

- Proposes a new cost-effective generic test architecture for function and performance

testing of e-commerce application.

- Maps the generic test architecture onto a middleware standards, namely CORBA (Common Object Request Broker Architecture) and proposes the CORBA-based test architecture for e-commerce. This testing architecture is open, highly distributed and scaleable.

- Provides an assessment of this approach (high-yield test strategy and cost-effective test architecture) by a small case study (Online Currency Exchange) involving design, implementation and function testing. This case study will also serve as a live system for teaching and research purpose.

- Provides recommendations for future study based on our findings.

1.3 Organization of the Thesis

In chapter 2 we will give more details of e-commerce, the definition, the history, the models, and the critical quality issues. In chapter 3 we will describe distributed object-oriented technology and CORBA, the underlying technology used in our test architecture. In chapter 4 we will provide details of some existing techniques for testing distributed system. Chapter 5 discusses the practical test issues of e-commerce applications, and outlines the requirements of a good test architecture. Chapter 6 gives our approach to e-commerce testing, including i)a high-yield test strategy for functional test design, and ii)CORBA-based e-commerce test architecture. In Chapter 7, we will describe the case study to demonstrate feasibility and assess our test architecture with respect to the criteria in chapter 5. Chapter 8 will summarize our results and contributions, then describe some recommended future work. A glossary is added to help the reader with some terms and acronyms.

Chapter 2

Basic Concepts of E-commerce

In this chapter, we give a detailed introduction to key aspects of e-commerce and the underlying technologies that enable the development of e-commerce systems. This will set the stage for understanding the scope of e-commerce test requirements.

2.1 Definition of Electronic Commerce

When we talk about Electronic Commerce, the first phrase that jumps to mind is "doing business on the Web".

Most people think that e-commerce means online shopping. But Web shopping is only a small part of the e-commerce picture. The term also refers to online stock and bond transactions, and buying and downloading software without ever going near a store. In addition, e-commerce includes business-to-business connections that make purchasing easier for big corporations.

Here are some definitions from industry and academia. [Zwass, 1996] defines e-commerce as "... the sharing of business information, maintaining business relationships, and conducting business transactions by means of telecommunications networks." It points out that e-commerce includes not only buying and selling goods, but also the various processes within individual organizations that support those functions. Applegate etal. also view e-commerce as more than simply buying and selling goods electronically [Applegate, 1996]. They point out that e-commerce involves using network communications technology to engage in a wide range of activities up and down the value-added chain both within and outside the organization. In addition, Kalakota and Whinston state that four different types of information technology are converging to create the discipline of e-commerce [Kalakota, 1996]. These include electronic

messaging such as fax and email, sharing a corporate digital library to promote collaborative work, electronic document interchange utilizing EDI and electronic funds transfer, and electronic publishing to promote marketing, advertising, sales, and customer support.

Here we would like to adopt [ECFAQ, 1999] as the formal definition of e-commerce:

Electronic Commerce is the paperless exchange of routine business information and transactions using Electronic Data Interchange (EDI) and other technologies, including Electronic Mail (E-Mail), electronic bulletin boards (EBBs), facsimile machines (faxes) and Electronic Funds Transfer (EFT).

Narrowly defined, e-commerce means doing business online, or selling and buying products and services through web storefronts. Products being traded may be physical products such as used cars or services, e.g., arranging trips, remote education. Increasingly, they include digital products such as news, audio and video, databases, software and all types of knowledge-based products.

E-commerce, broadly defined, is concerned with the electronic marketplace. It is a virtual image of physical markets, and resembles physical markets in many aspects, including:

- *players*: market agents such as firms, suppliers, brokers, shops, and consumers,
- *products*: goods and services,
- *processes*: supply, production, marketing, competition, distribution, consumption, etc.

The difference is that, in the electronic marketplace, at least some of these components are electronic, digital, virtual or online.

E-commerce is transforming the way individual companies operate and compete, and the economics and boundaries of entire industries. Through process engineering, new organizational models, and client/server technologies, enterprise functions are being more closely integrated[G. Winfield Treese 1998]. As more enterprises conduct more business electronically, large number of providers, suppliers, and customers are becoming more closely coupled. Networks of

complimentary enterprises may consequently emerge, boasting global reach, reduce cost structures, rapid response times, and market profitable knowledge assets. It is obvious that the process of value creation and value extraction is being greatly facilitated by e-commerce.

2.2 History of E-commerce

The following are some significant milestones of revolutionary technological innovation in human history, leading up to the development and widespread acceptance of e-commerce:

- 1876 Alexander Graham Bell invented the telephone, people were able to communication regardless of the physical distance.
- 1946 the world's first computer was built, starting the digital age.
- 1966 ARPANET was created and dominated by researchers and scientists.
- 1974 Transmission Control Protocol (TCP) was established by Vint Cerf and Bob Kahn to meet the needs of an open-architecture network with the ability to maintain effective communication even in the face of disturbances or intermittent blackout.
- 1978 Internet Protocol (IP) was proposed as a way to separate TCP's routing functions.
- 1982 the University of Wisconsin developed the Domain Name Service(DNS).
- 1989 Tim Berners-Lee developed HTTP, and the concept of the World Wide Web. The Web allows consumers to search for information on the Internet with a straightforward graphical interface. The Web remains a critical element in emerging electronic markets.
- 1989 CompuServe and MCI became the first commercial e-mil providers.
- 1990 Andrew Parkinson started Peapod, the first online grocery shopping service.
- 1995 Jeff Bezos opened Amazon.com.
- 1996 Dell began selling computers via Internet. Now Dell's sales via Internet reach $30 million per day. Dell and Amazon are two of the biggest success stories of e-commerce.

Sine 1990 the growth of the Internet has been exponential. The growth of hosts on seven continents from the Internet Domain Survey [IDS, 1997] is shown in Table 2.1.

Table 2.1 Number of Hosts

Region	Hosts, 1994	Hosts, 1995	Hosts, 1996	Hosts, 1997
Northe America	1,685,715	3,372,551	7,088,754	11,216,035
Europe, West	550,933	1,039,192	2,699,559	4,352,151
Europe, East	19,867	46,125	168,142	784,225
Middle East	6946	13,776	44,484	58,930
Africa	10,951	27,130	84,715	104,838
Asia	81,355	106,664	151,773	672,495
Pacific	113,482	192,390	475,505	647,948

In 1997 Forrester Research and Yankee Group reported and projected the following statistics for online retail sales and business-to-business sale.

- Online Retail

  1996 $600 Million

  1997 > $2 Billion

  2001 >$46 Billion

- Online Business-to-Business

  1997 $8 Billion

  2002 >$327 Billion.

E-commerce on Internet has great potential. It is evolving from its present form of human information collection, decision making and buying to forms of automated information collection, decision making and buying.

2.3 E-Commerce Models

Here we present and distinguish between two kinds of models, the *business model* and the *architectural model.* An e-commerce business model describes the business's operational and functional structure. As such, it focuses on high-level details such as goals, strategies, and the interactions of entities (eg., customer and business). The architectural model of e-commerce system defines the basic components and important constraints of the system, as well as the relationships among them. In this section, we look at four business models, and four architectures and discuss how they are constructed. This will give us better understanding of e-commerce from both business and technology perspective in order to consider the test strategy and architecture later in chapter 6.

2.3.1 Business Models

There are two primary business categories of electronic commerce business model

- Business-to-Consumer.
- Business-to-Business.

These two categories specify the basic content of any business operation. Some business fall strictly within one category while some fall into both but with different focus. Based on these criteria's, [Saleh, 1999] defines six business models of e-commerce while [Julta, 1999] defines three business models. Based on the fact that some models in [Saleh, 1999] overlap each other and new technologies and applications often create new business models, here we combine both into four models. The object interaction diagram of each model is also provided. The four models are:

2.3.1.1 Store-front Business Model

As shown in Figure 2.1, the business simply buys the finished product from suppliers and resells it. This model has many advantages: It reduces the inventory management overhead associated with staffing and office space; and frees capital that would otherwise be tied up in inventory control. More importantly, it provides specialization (marketing, production, delivery) across the supply chain from manufacturers to customers. An example is Amazon.com.

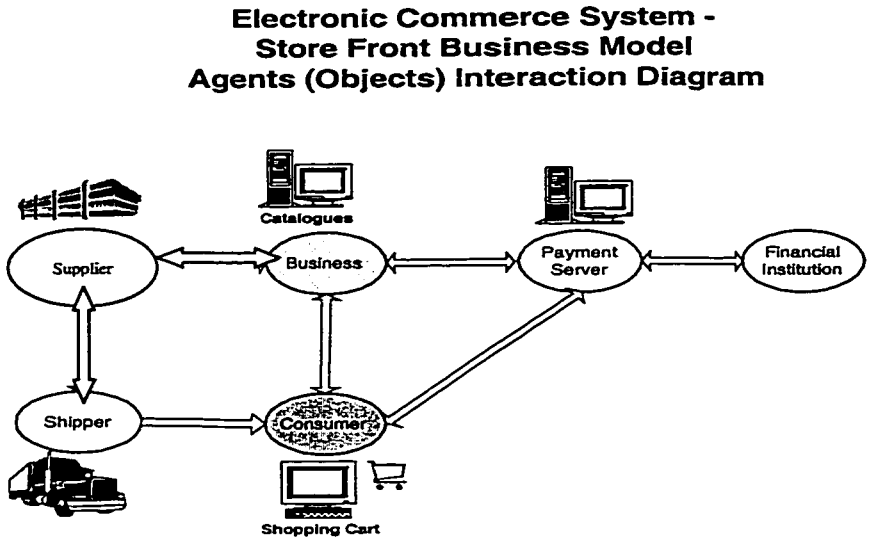


Figure 2.1 Store Front Business Model

2.3.1.2 Manufacturing Model

As shown in Figure 2.2, the business adds value through its internal manufacturing processes – by either developing a product from scratch, integrating components or enhancing an existing product. This model works best for businesses with configurable products, mature marketing staff, and sophisticated customer service processes. It usually introduces e-commerce extranets with other business(suppliers). This model is a type of Business-to-Business category. A well-known example is Dell computer.

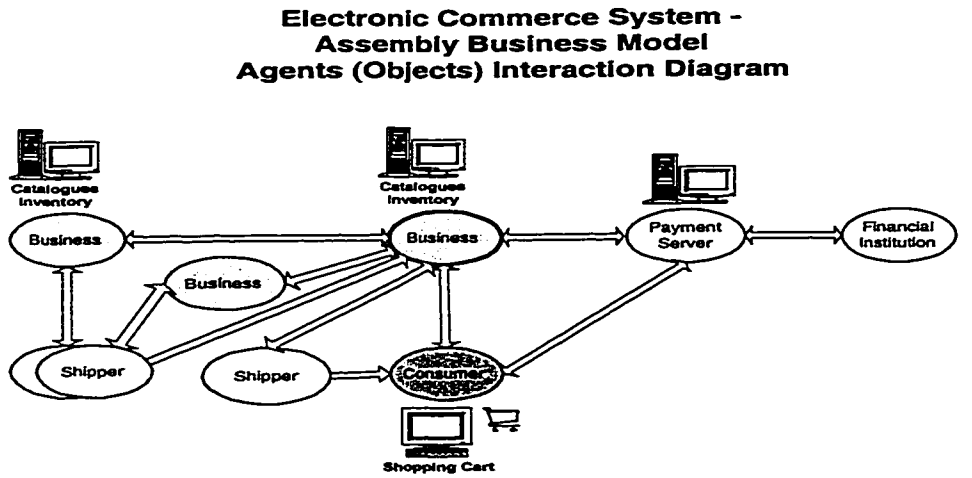


Figure 2.2 Manufacturing Model

2.3.1.3 Auction Model

As shown in Figure 2.3, potential buyers submit a bid, and the product is sold when the supplier accepts a bid. Suppliers are considered customers. The auction site's job is to connect the customers with someone interested in the services or products. This model is popular with organizations that want very little involvement with stocking physical inventory from suppliers or processing customer sales. A good example of an auction site is Ebay.

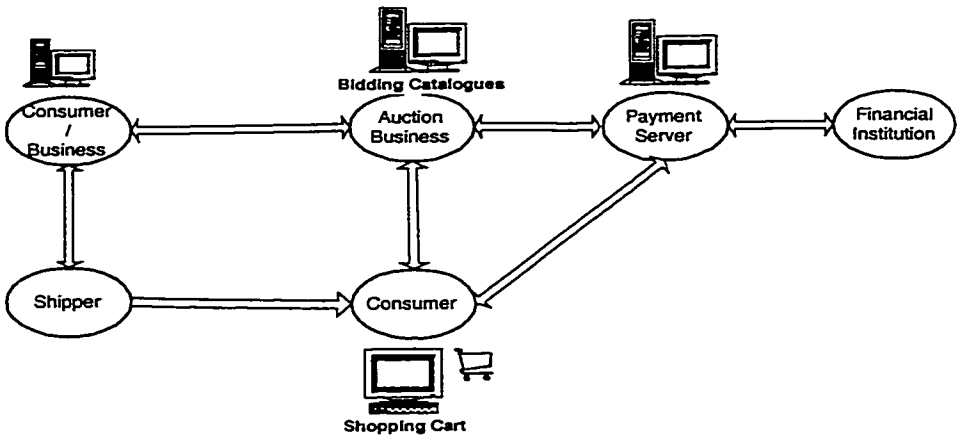


Figure 2.3 Auction Model

2.3.1.4 Broker Model

As shown in Figure 2.4, the business associates with many suppliers specialized in a specific service. Acting on behalf of the consumers, the business selects the supplier with the least cost, shortest delivery time, or most suitable stock. Its business model supports shipping the product from a supplier to the consumer directly through another third-party delivery channel, such as Federal Express or UPS. An example of this is E-Trade. Also, our case study product, On-Line Currency Exchange follows the broker model.

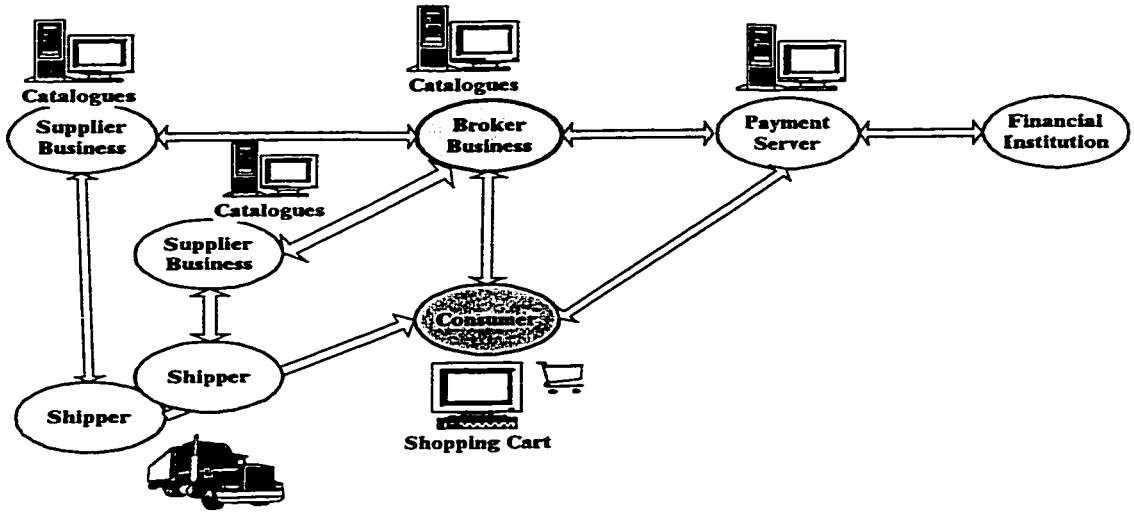


Figure 2.4 Broker Model

2.3.1.5 Test Issues Related to Business Models

Analysis of e-commerce business models can help us better understand e-commerce from a business perspective - how different parties are interacting with each other. Thus we can better tailor our test strategy to meet the different needs of different business model.

Each party in each model has their unique business objectives and needs unique underlying system to support these objectives. Test strategy must address this uniqueness. For example, models focus more on *Business-to-Consumer* has potentially much more end-customer (millions per day) to deal with simultaneously than models focus on *Business-to-Business*. So load and stress testing should be emphasized. Section 5.1 gives a list of test issues that are common to all models and can be combined to address the uniqueness of different model.

2.3.2 Architectural Model

For comparing architectures, we have found it convenient to consider four primary components of e-commerce systems:

- *Client*

  The client is a computer system, typically a PC, connected to the Internet directly via an Internet Service Provider (ISP), or indirectly via a corporate network. The customer uses the client computer to browse and purchase.

- *Business Server*

  The business server is the computer system or systems that contain business's electronic catalog.

- *Transaction server*

  The transaction server is the computer system or systems that process a particular order and are responsible for payment, record keeping, and other business aspects of the transaction.

- *Payment gateway*

  The payment gateway is the computer system that routes payment instructions into existing financial networks such as for credit card authorization and settlement.

The following four architecture models are commonly used in real world applications [G. Winfield Treese 1998].

2.3.2.1 Web Server with Order Form Architecture

A web server with catalog pages and an order form is one of the simplest and most common ways of constructing an e-commerce system. A diagram of a representative system is shown in Figure 2.5.

In this example, a single merchant Web server provides both the catalog content and the order form. In other words, the business server and transaction server is combined into one system, and there is no explicit payment gateway. The primary virtue of this basic architecture is its simplicity. On the other hand, it may be more difficult to expand it as the online business grows

or to incorporate new technologies and components as they become available.

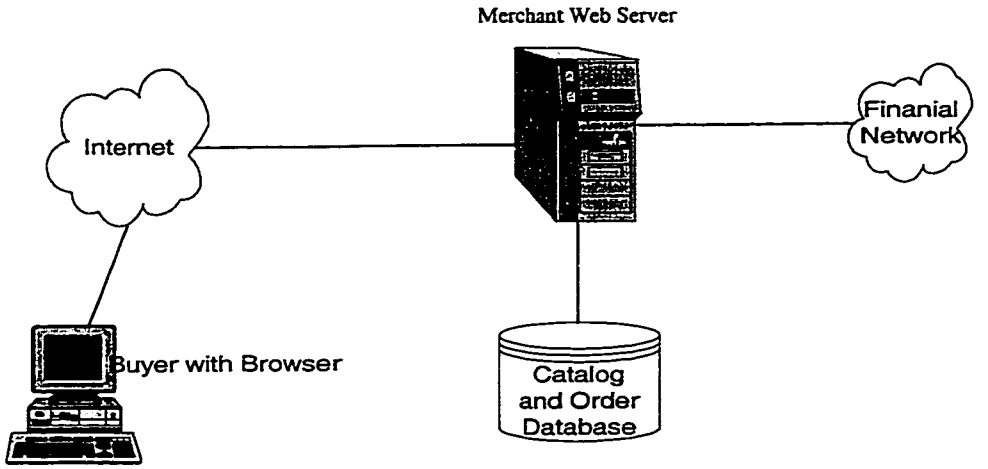


Figure 2.5 Web Server with Order Form

2.3.2.2 Secure Electronic Transactions (SET) Architecture

SET, for Secure Electronic Transactions, is a standard for the way credit card payment transactions should be handled on the Internet. Details of SET and its specification can be found in [SET, 1999]. In a SET system, a payment gateway is added which is distinct from the transaction server. A diagram of a SET-enabled Internet commerce system is shown in Figure 2.6.

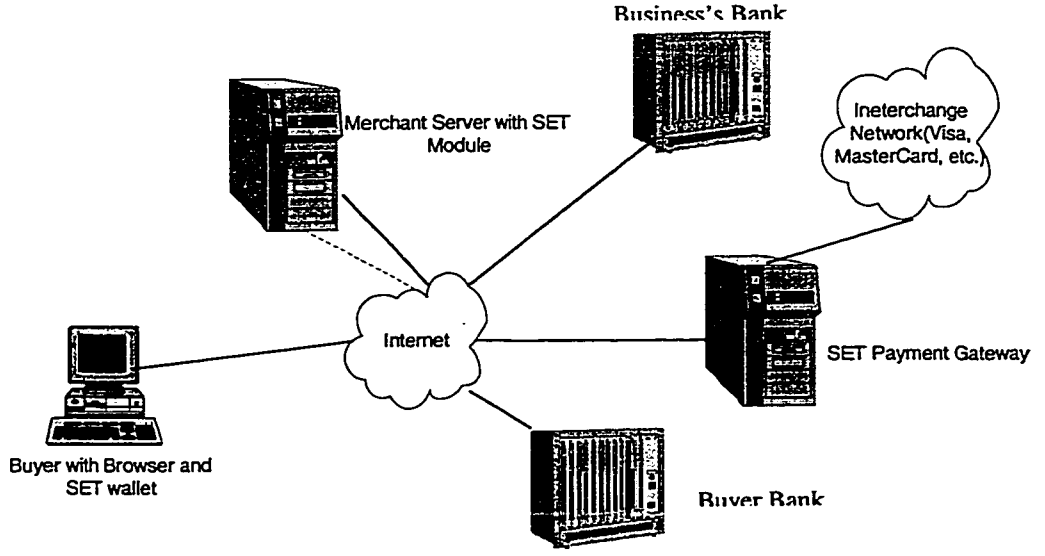


Figure 2.6 SET System Architecture

SET wallet, a specialized client application, implements SET payment methods that require additional processing, such as cryptographic operations, on the client computer. SET module is the application with similar functionality on Merchant Server.

In its simplest form, the SET architecture takes over from the business server order form at the point where payment by credit card is initiated. The business server, rather than connecting directly to a credit card authorization network, instead invokes a SET module. When the SET module is called upon to handle payment, the following steps occur.

- The business's SET module sends a message to the SET wallet located on the buyer's computer, containing a description of the order and the total price.
- The buyer uses the SET wallet to select a payment card and to approve the purchase.
- The SET wallet communicates via the business server with the SET payment gateway at the business's bank.
- The payment gateway connects to a traditional financial network to authorize the transaction.
- The business server stores the acknowledgment and sends a receipt to the buyer.

2.3.2.3 Open Market Commerce Architecture

The core idea of this architecture is to separate the business content from the payment transactions through a technology called *SecureLink*. This idea permits multiple catalog servers to share the capacity of a single transaction engine and allows the business content-oriented parts of the system to scale independently from the transaction-oriented parts of the system. The approach also permits service organizations to become commerce service provider, who provide transaction management services on an outsource basis to other companies.

Figure 2.7 shows the physical architecture of this approach. The transaction server is separated from the business server and there may or may not be separate payment gateway depending on which payment methods are supported.

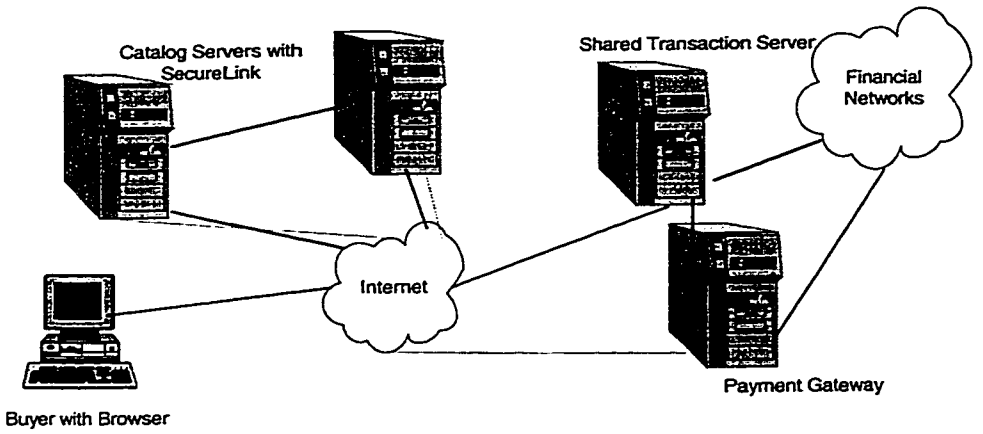


Figure 2.7 Open Market Commerce Architecture

2.3.2.4 Open Buying on the Internet (OBI) Archiecture

Open Buying on the Internet, or OBI as shown in Figure 2.8, is a standards proposal released by the OBI consortium. The consortium is a group of buy-side, sell-side, payment processing organizations, and technology companies that is addressing the problem of business-to-business commerce on the Internet. The core idea of OBI is to split the functionality of the commerce system between buy-side activities and sell-side activities so that each organization manages those functions logically connected to it.

The real benefits of the OBI can be seen when there are multiple buy-side companies trading with multiple sell-side companies. When this happens, the buy-side is able to centrally manage its requisitioner database and approval system and to use those systems seamlessly with multiple trading partners. Similarly, the selling organization can use a master catalog and order management system against purchasers. In this ideal situation, information is not duplicated on either side. More details of OBI can be found in [OBI, 1999].

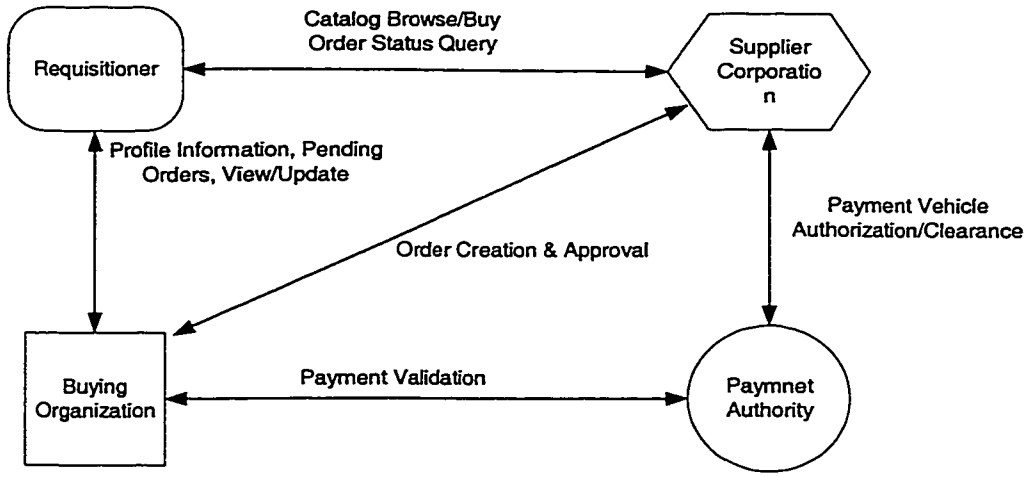


Figure 2.8 Open Buying on the Internet

2.3.3 Test Issues Related to Architectural Models

Analysis of e-commerce architecture model can help us better understand the distributed nature of the e-commerce system – how various functions are distributed among different sub-systems. Thus we can pinpoint where the bottle neck or hot spot are, and design our test strategy to address them.

For example, *Secure Electronic Transaction Architecture* is based on the credit card payment standard SET, conformance testing must be performed to make sure the system provide standardized secure transaction service. Section 5.1 gives a list of test issues that are common to all architectures and can be combined to address the unique requirements of each architecture.

2.4 Technology Related Issues in E-commerce Development and Test

Technology is, of course, what makes e-commerce possible. The invention and subsequent spread of the World Wide Web, in particular, provided the technical foundation for many different applications. Since its introduction the Web has changed dramatically, with rapid growth in usage and evolution in protocols, systems, and applications. For e-commerce systems,

there are two key technology issues: what technology to use and how to deal with the fast pace of technological change.

E-commerce applications bring together many technology components: the Web, databases, high-speed networking, cryptographic algorithms, multimedia, and others. Putting them together to form a secure, reliable, high-performance, integrated system can be challenging, but the principles and ideas presented here should provide some useful guidance. The earliest Internet commerce systems were custom software. More recently, it has become possible to assemble a commerce system by using toolkits to integrate software modules and applications from different suppliers. We are now seeing the emergence of packaged application software for Internet commerce, called software components, which are integrated into a complete or nearly complete system. This promises to make development faster, but will require extensive acceptance testing for components.

The second technology issue, the pace of change, is a fact of life in e-commerce today, and there is no end in sight. To be successful, any e-commerce system must be open to accommodate and incorporate new technologies as they become available. The key to such adaptability is a coherent system architecture, which lays out what is to be accomplished and why. By focusing on the fundamental principles, we can adopt new technologies that help us achieve these goals, while avoiding new technologies that may seem exciting, but in reality do not fit in with our goals or the system. Many toolkits and frameworks help a great deal in coping with technology change. An example is IBM's net.commerce framework [Shurety, 1998].

2.4.1 Design Principles of the Internet

The Internet has been successful because of some fundamental decisions about its design made early in its history. These decisions are often invisible to the end user, and even to application developers, but understanding them gives insight into why the network is the way it is. The main design ideas in the Internet are as follows [Dahl, 1996]:

- *Interoperability* – In the context of e-commerce interoperability means that buyers and sellers do not have to buy and upgrade software simultaneously from the same vendor to

conduct commerce; i.e., the systems can be assembled using client/server system with hardware and software from different vendors. In practice, this means that the software from different vendors is based on common open standards.

- *Layering* – Internet protocols are designed to work in layers, with each higher layer building on the facilities and services provided by lower layers. For example, TCP builds on IP to create reliable byte streams, and application protocols such as those for the Web or e-mail are built on the capabilities of TCP. This layered architecture is shown in Figure 2.9.

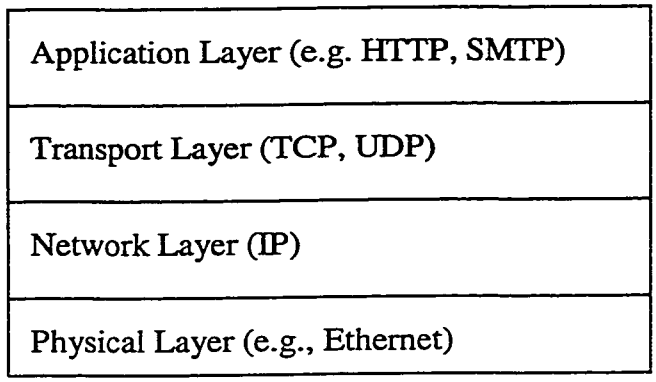


Figure 2.9 Layering of Internet Protocol

- *Simplicity* – One way to look at the layering of the Internet is that it grows both up and down from IP. IP itself is very simple, providing only addressing and formatting of packets. Below the level of IP is the complexity of many different kinds of network hardware, topologies, routers, Ethernet, dial-up connections, and so on. IP hides that complexity from applications. Above IP, higher-level protocols such as TCP offer service abstractions that are easy for application programmers to understand and use. As a consequence, application developers and users are insulated form the complexities of different network devices as well as from the complexities of implementing low-level network protocols.

- *Uniform naming and addressing* – The IP layer offers a uniform addressing structure

that assigns a 32-bit address to each computer connected to the network. Addresses are hard for people to remember and work with, so the Domain Name System (DNS) offers a uniform way to translate human-readable names for computers, such as "www.openmarket.com" to the numeric address for that computer. These two addressing systems, together with the interoperability of implementations, let the Internet function.

- *End-to-End* design – The Internet is designed around end-to-end protocols. That is, the interpretation of data happens on the sending and receiving systems. The underlying network won't look at the data content but only the destination address for delivering the packet. End-to-end protocols have several advantages. They hide the internal structure of the network, including the wide range of physical hardware used on the Internet, from users and applications. In addition, they can provide simple abstractions to programmers, shielding them from the messy details of recovering from low-level errors.

From these simple ideas has evolved a powerful, robust, and reliable network that has scaled up to a global level. It is important to be aware of these principles when testing e-commerce systems, because they provide some guidelines for testing e-commerce. For example, *interoperability* is one of the test issues of e-commerce system; layering and end-to-end are important concepts of protocol testing [CTMF, 1993]. The test effort should conform to these principles.

2.4.2 Core Network Protocols

The Web depends on a number of lower-level protocols, particularly TCP/IP, and DNS. To properly test e-commerce system requires that the underlying protocols have been properly implemented, configured, and integrated. This requires conformance testing, described briefly in section 4.2.

Core Network Protocols: TCP/IP, DNS, HTTP, Secure Socket Layer (SSL), etc. Figure 2.10 shows the network protocol stack.

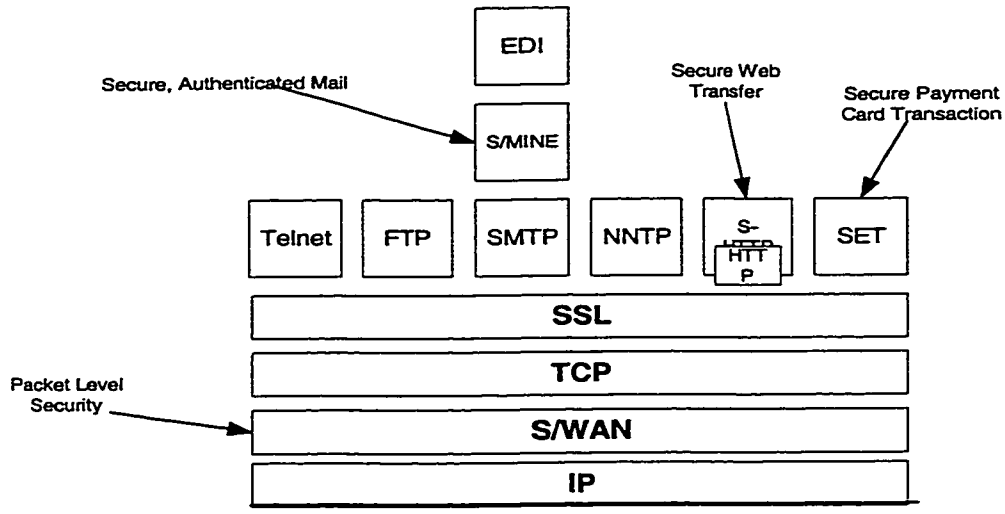


Figure 2.10 Core Network Protocols for Internet Applications

2.4.3 Distributed Object Technology: DOC, CORBA, COM, Java Beans/RMI.

In order to take advantage of these network protocols to deliver end-user services, we need to rely on software design technology, especially distributed object technology to implement and deploy these services.

The essential idea of a software object is a package containing both data attributes and methods that operate on the data. Object technology not only provides better and more efficient ways(through software re-use) to construct large software applications, but also provides the possibility that complex systems can be built using software pieces designed and implemented by different groups or different companies. This idea is called component based software development.

At present, the three main underlying distributed object (middleware) technologies for e-commerce and other distributed Web applications are CORBA, DCOM, and JAVA, and their APIs.

CORBA specifies interfaces and protocols for a distributed infrastructure. It is an extremely powerful and extensible framework that companies such as IBM and SUN have adopted in deploying their e-commerce solutions. Netscape's ECXpert/BuyerXpert/SellerXpert solution is built on Visigenic's ORB.

Alternatively, Microsoft's DCOM is very easy to use, and its components can be written in many languages, including Visual Basic, C++, Java, and Cobol. Plans are under way to interface DCOM and CORBA components, thus preserving existing investments in each technology [OMG, 1995]. DCOM's strength is in managing presentation and user interaction, while CORBA's strengths are in providing a distributed infrastructure and integration medium. How COM, CORBA, and Java-based technologies will perform by themselves or integrated with each other in practice remains an open question.

The third approach uses Java. Java began as a platform-independent language for creating client-side applets (small processes) that run inside the Web browser. In addition, JavaSoft also has a component model, JavaBeans. Enterprise JavaBeans extends the lightweight JavaBeans model with multi-user security and resource management capabilities. Java provides a mechanism for components to discover and use each other's interface at runtime and can run on different platforms because of the Java Virtual machine (JVM). Java and CORBA are commonly used together to combine strengths of the Java client and CORBA distribution medium [OMG, 1995]. Many businesses use Java on their client-side applications and languages like C++ for the server side. C++ is faster than Java; Java, on the other hand, offers portability and better presentation services [SUN, 2000].

Details of CORBA, DCOM, Java along with their comparisons will be discussed in chapter 3.

2.4.4 Summary of Technology Issues for Testing E-commerce

- The needs of fast pace of technology change requires not only the e-commerce system but also the test system built on an architecture with great openness, interoperability and scalability.

- Software component technology makes e-commerce system development easier and faster, but requires extensive product testing, system testing and acceptance testing.

- The frequent e-commerce product deployment also requires effective regression test, thus the automated test become increasingly important.

Section 5.1 will detail the test issues of e-commerce.

2.5 Critical Issues

2.5.1 Customer's Concerns

The reasons for customers not purchasing online were identified by GVU's 10th WWW User Survery (1998) [GVU, 1998], shown in Figure 2.11.

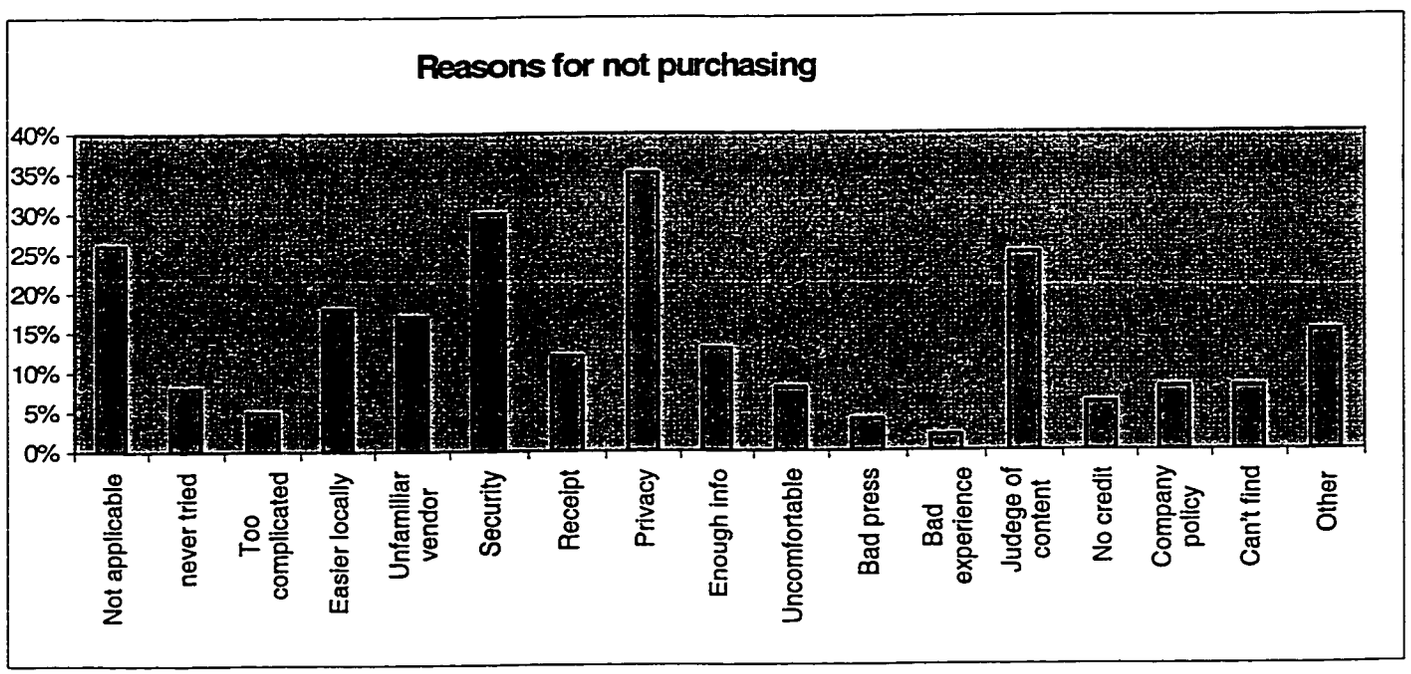


Figure 2.11 Customer Concerns

2.5.2 Business's Concerns

2.5.2.1 No second chance

Web sites sometimes form the customer's crucial first impression of the business. They are often the first points of contact with customers. *No second chances* means the site must be up available all the time, key features must be accessible, and performance must be adequate. If business does not at least provide this level of functionality, robustness and reliability, it will lose customers or viewers even before it has had a chance to sell to them. Thus, testing functionality, robustness, reliability and performance must be supported by our test framework.

2.5.2.2 Minimal control over customers' environments

This is not a client/server architecture, in which there is only one supported configuration for the client and one for the server. In e-commerce for example, end users use PCs, Macs, Unix boxes, WebTVs, Internet-enabled phones, and 1024 x 760 or 800 x 600 monitor resolutions. They can use Netscape Navigator, Microsoft Explorer versions 1, 2, 3 or 4.0, Opera, Konqueror, some plug-ins (but perhaps not the ones you assume), different security settings (such as turning off cookies), 1,200 or 56,000 baud modems, T1 lines, etc.

A most difficult part of ensuring the quality of an installed e-commerce application is ensuring the quality of the client environment.

2.5.2.3 Do not know the customers

Most software products have been designed for a specific type of customer. And most testers have tested products for a very specific type (operating profile) of user. But with e-commerce applications this is not the case. We have no idea who will come into our webstore. Since the customers are anywhere, we might be shipping to Mongolia. Does the one-day shipping guarantee still hold? Can their addresses even fit in the forms? What currency do they use? What measurement system?

Will the customers know how to navigate the site? There are few opportunities for training and

documentation. Individual training is not possible. Printed manuals are out. The 800 number will not be used. The Web site must be usable "out of the box".

Another challenge in not knowing the customers is the inability to always pre-qualify them before doing business. They might be selling alcohol to minors, for example, without realizing it.

2.5.2.4 WebTime

WebTime is the biggest risk. With an electronic commerce site, the content will be changing frequently, and the software development process will begin to more closely resemble a publishing process. Typically there are many small changes to the user interface throughout the development cycle, Release cycles are measured in days, not months. And you'll be doing more maintenance mode testing with an Internet application than with a normal client/server application.

2.5.2.5 Security

Security is a crucial concern of e-commerce system. The following requirements for secure communication must be met:

Privacy – The message is secret, only the sender and intended recipient will know the contents of the message.

Authentication – The recipient knows that the message is not a forgery, but was in fact sent by a particular sender. The sender knows that the message is going to the proper recipient.

Integrity – The recipient knows that the message was not modified (intentionally or accidentally) while in transit.

Nonrepudiation – The author of the message can't later deny having sent the message.

Those requirements apply to not only the transaction data on the fly, but also the warehoused data. A common example for e-commerce application is protecting customer's payment credentials, such as a credit card number, when the customer sends it to payment server. In such

cases, the encryption mechanism is typically used to keep the messages private. After the credit card number has been communicated securely, if it is processed immediately and discarded, no additional protection is needed; otherwise, we may choose to encrypt that data when it is stored on a disk on the business end system, also we may use a firewall to prevent the untrusted party from accessing and interpreting the data. Cryptography and firewall are the most commonly deployed security technology today.

For more details on e-commerce security requirements, see [Bhimani, 1996]

2.5.3 Characteristics of Robust and Reliable E-commerce Transactions

What are the properties of a reliable e-commerce system? Functions performed by Internet commerce applications are very complex. Each is composed of a number of other activities. For example, a sale event might include:

- Debit buyer's account
- Credit seller's account
- Record event for business records
- Transmit order to fulfillment center
- Issue receipt to buyer with order number (for tracking delivery)

Good quality transaction systems make sure that all of these steps happen if any of them do, that the activities of multiple buyers do not interfere with each other, and that records are not lost. From this perspective, transactions must have four essential characteristics, known by the acronym ACID. ACID requires that transactions be atomic, consistent, isolated, and durable. Distributed ACID transaction must be robust and prevail in the face of network outages, replay attacks, failures of local hardware, and errors of human users [Gray, 1993].

- *Atomicity* – An *atomic* transaction must either fail completely or succeed completely. Either all of the steps must be taken, or it must be seen to all observers that the

transaction never happened at all. For example, consider what happens when a customer transfers funds from a savings account to a checking account. Both the checking account is credited and the saving account is debited, or neither account balance changes. There is no case where money either disappears from any one account or is credited to only one account.

- *Consistency* – If a transaction is *consistent*, all relevant parties agree on critical facts of the exchange. If a customer makes a one-dollar purchase, then the merchant, the customer, and the bank (if it is involved) all agree that customer has one less dollar and the merchant one more. Accounts must balance both before and after the transaction.

- *Isolation* – Transactions that do not interfere with each other are said to be *isolated.* Because an Internet commerce business transaction takes some period of time, and because there may be multiple transactions underway, it is important the activities of one buyer do not interfere with those of another buyer, and even the multiple transactions of one customer do not interfere with each other. The result of a set of overlapping transactions must be equivalent to some sequence of those transactions executed in nonconcurrent serial order.

- *Durability* – When any transaction can recover to its last consistent state, it is *durable.* Once a transaction is complete or committed, it should be impossible for its effects to come undone due to component failure. In practice, this means that the results of transactions are reliably recorded on a stable storage medium that is resilient to failure, such as duplicate or mirrored disk drives. For example, if the customer physically drops a dollar when making a purchase, that dollar does not disappear. When the customer retrieves the dollar, the last consistent state is restored. Similarly, money that was available to a station before it crashed should not disappear when the station's machine reboots.

2.6 Case Study System: Online Currency Exchange

For the purpose of relating some of the e-commerce concepts used throughout this thesis to a

concrete example, we will describe a simple e-commerce system – Online Currency Exchange. This example will be referred to throughout the thesis. In chapter 8 we will describe the system in much more detail.

To demonstrate the e-commerce test architecture, a demo e-commerce system – *Online Currency Exchange (EX)* was developed. The following is a simple English description of the customer's initial functional requirements for such a system.

2.6.1 Online Currency Exchan ge (EX) Initial Requirements

1. EX provides online currency exchange from CND to USD in Canada.
2. EX promises to provide the cheapest price among linked banks.
3. EX provides delivery in 24 hours.
4. EX accepts payment by

   Debt card or

   CND Dollar Account Number
5. EX returns customer's money in the form of:

   Money order or

   Traveler's check

   Direct deposit in customer's USD account in anyone of the associated banks.
6. EX authenticates customer's order by customer's password & customer ID.
7. Every customer has a unique customer ID and confidential password associated with EX.
8. For a first time customer, EX provides an off-line registration from which the customer obtains his/her password & customer ID, and EX obtains the customer's information.
9. EX links up with Canada Post to deliver the money order or traveler's check to customers.

2.6.2 Example of Sequence of Transactions

Here is a basic scenario of the system (shown in Figure 2.12).

1.1 Client browses on Internet, visit EX Online Currency Exchange service page.

1.2 Client inputs his/her user ID and password to login to account page in EX.

2.1 Client asks the exchange rate for certain amount of CND.

2.2 EX negotiates with associated five banks and return best exchange rate.

3.1 Client fills orders form and place the order.

3.2 EX forwards the order to the client's bank which holds client's CND account

3.3 Client's bank asks account information

3.4 Client gives account information

4.1 Client's bank contacts with selling bank which offers the best exchange rate

4.2 Selling bank confirms transfer

4.3 Selling debits client's CND account and transfers equivalent amount of USD to client's USD account (if customer asks for direct deposit)

5. Client's bank transfers commission into EX's CND account

6.1 Client's bank notifies EX that transaction is finished

6.2 EX notices client that order is completely processed.

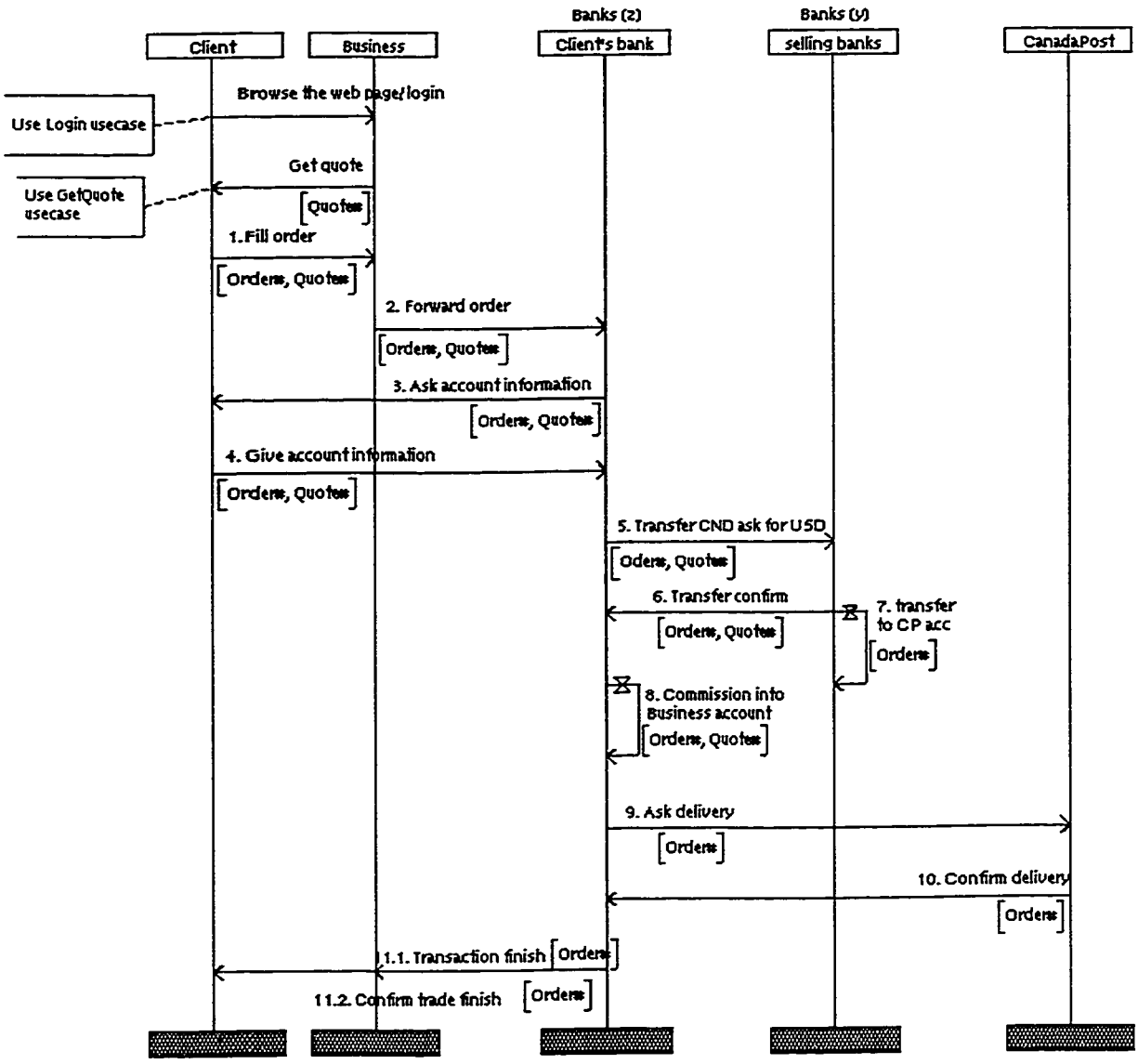


Figure 2.12 Basic Transaction Scenario

2.6.3 Assumptions of Online Currency Exchange

Base on the customer's initial requirement, we find the requirement is quite general, ambiguous, and incomplete. Some basic assumptions are made as following:

1. Before setting up EX business, all the parties agree about the process and responsibility of each party. The process will follow the primary scenario.

2. EX opens exactly one CND account and one USD account in any one of its linked banks.

3. Canada Post has USD account in each of EX's linked banks.

4. Bank servers and Canada Post server are robust and reliable.

5. Network protocol TCP/IP has robust fault recovery functionality.

Details of EX system and its implementation will be given in chapter 7.

2.7 Summary

In this chapter, we gave a detailed introduction to the aspects of e-commerce which are relevant to test considerations. In the next chapter, we will describe the basic design principles of e-commerce applications and the CORBA distributed object technology that provides the framework for e-commerce software development. In the next chapter and in the case study we will show how a CORBA-design for e-commerce assists testability.

Chapter 3

CORBA Based E-commerce Architecture

E-Commerce systems are highly distributed software applications. They have unique characteristics and challenges that were described in chapter 2. This chapter addresses some these challenges from a software testing perspective. We also describe the common distributed object technologies and the popular CORBA framework. CORBA is a very important enabling technology that facilitates the development of distributed e-commerce systems, and will be used to build an effective test framework (described in Section 6.3).

3.1 Object-Oriented Technology

As requirements for more imaginative e-commerce systems arise, it is necessary to find a cost-effective, timely means to capture requirements, validate these requirements, and develop these products according to an accelerated development schedule (less time to market). Object oriented design is seen as an enabling technology for these goals.

The adoption of object-oriented technology is a major trend in the software industry. The basic element of this technology is the *object*, a software entity that encapsulates both data and code, and interacts with the outside world through well-defined interfaces called methods. This technology emphasizes the importance of reusability, maintainability, flexibility and modularity in the software development process. Key features are:

- Encapsulation – Objects hide the details of their implementation.

  Objects are a step beyond subroutine libraries, or APIs, as they are now called. The programmer who develops software using APIs must understand how data is represented and stored. With object technology, the object is responsible for manipulating the data;

the programmer only makes use of the object to accomplish some higher goal. Encapsulation not only makes it unnecessary for the programmer to learn about the internal details of the object, it actually makes it impossible. This sounds heavy-handed, but it greatly reduces coupling and imposes cohesion [Jacobson, 1992]

- Polymorphism – There can be multiple implementations of an object.

  Once a particular concept, such as a bank account, has been represented as an object, variations of the concept, such as a checking account or a saving account, can be represented as different implementations of an object, each with the same interface. When this is done, applications that operate on the account, such as the application which computes interest due each month, do not even need to know that there are different kinds of accounts.

- Language binding – Object can be implemented in different languages.

  The internal communications between one part of a program and another part are usually orchestrated by the language and its compiler. However, because the object technology carefully defines the ways in which one object can call another, there is no need for all the objects in a system to be coded in the same language. Instead, compiler provides a translation to the standard communications machinery defined by the object system; for example, Java translates to byte code which run on JVM.

3.2 Distributed Object Computing

3.2.1 DOC Overview

From the marriage of object-oriented computing and distributed computing comes the Distributed Object Computing (DOC) paradigm [IEEE Com., 1997]. This paradigm exploits the benefits of object-oriented technology and uses the Internet and its communications infrastructure as a vehicle for the delivery of a wide range of sophisticated value-added distributed services. Applications of the paradigm are characterized as open, client/server, multi-tiered and using collaborating distributed objects.

The telecommunication industry is one of the main industries investing in client/server and distributed applications. The DOC paradigm has a great potential for integrating different telecommunication services including video, audio and data retrieval, as well as management of heterogeneous computer networks. Another emerging industry that involves the deployment of collaborating distributed objects is e-commerce.

Typical distributed objects consist of cooperating objects. Some of them act as clients requesting services, others act as servers providing these services, with the provision that the same object acting as a server could act as a client to another server and vice-versa. These objects may reside in the same process in the same machine, or in different processes in the same machine or in completely different machines.

3.2.2 Generic Architecture of Distributed Application

Typically, distributed applications are multi-tiered (3 tiers or more). For example, the first tier is normally the *client tier* handling the interface with the user. The second tier is *server tier,* offers functionality and connectivity (a bridge between the first and third tier) achieving the business function and logic of the application. The third tier is the *data tier*. Hence, we are dealing with *clients* with minimum responsibilities, connected to data storage through a tier that contains the business logic. These applications benefit from the object-oriented design paradigm. The building objects of these applications have a clear separation between their interface and their implementation. Client objects will only need to know the interface of their servers to access their services without any concern about the implementation of these services.

Moreover, distributed objects interoperate transparently. A client object is not concerned about the location of the server object. A client object invokes the operations offered by a server object as if it is residing in the same machine. Ideally, finding these objects in a distributed computing environment and the involved communication details should not be the responsibility of the client. Similarly, the server object receives a request, processes it and sends its reply to the client without caring about the client's location and the communication details.

The Internet is a perfect vehicle for delivering distributed object applications and services.

Millions of heterogeneous computers are connected to the network of networks (the web). The web provides a typical client/server architecture, in which a web browser is the client and a web server is the server.

3.3 CORBA and Testability

Distributed systems are heterogeneous in terms of interconnected networks, operation systems and programming languages used to develop individual components. The goal of open distributed systems is to enable access to components of a distributed system from anywhere in the distributed environment without concern for its heterogeneity. The rapid growth of distributed processing systems has led to a need for coordination and standardized frameworks. It's also the need for testing distributed system since standardized frameworks can help to build a open, generic test architecture for distributed system. At present, the three main underlying technologies for e-commerce and other distributed Web applications are CORBA, DCOM, and JVM with its APIs.

3.3.1 CORBA and OMA Overview

The Common Object Request Broker Architecture (CORBA) is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them.

CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) which defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defined true interoperability by specifying how ORBs from different vendors can interoperate.

OMG has in fact developed two architectures: OMA (Object Management Architecture) and CORBA. OMA defines its object terminology and various facilities required for object-oriented distributed processing. CORBA provides the mechanisms necessary to identify, locate, access

and manipulate the OMA objects through ORB.

ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In doing so, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In typical client/server applications, developers use their own design or a recognized standard to define the protocol to be used between the devices. Protocol definition depends on the implementation language, network transport and many other factors. ORBs simplify this process. With an ORB, the protocol is defined through the application interfaces via a single implementation language-independent specification, the IDL. Also ORBs provide flexibility. They let programmers choose the most appropriate operating system, execution environment and even programming language to use for each component of a system under construction. More importantly, they enable integration with existing components. In an ORB-based solution, developers simply model the legacy component using the same IDL they use for creating new objects, then write "wrapper" code that translates between the standardized bus and the legacy interfaces.

Figure 3.1 illustrates the primary components in the OMG Reference Model architecture. Descriptions of these components are given below. Portions of these descriptions are based on material from [Vinoski, 1997].

The OMA architecture consists of:

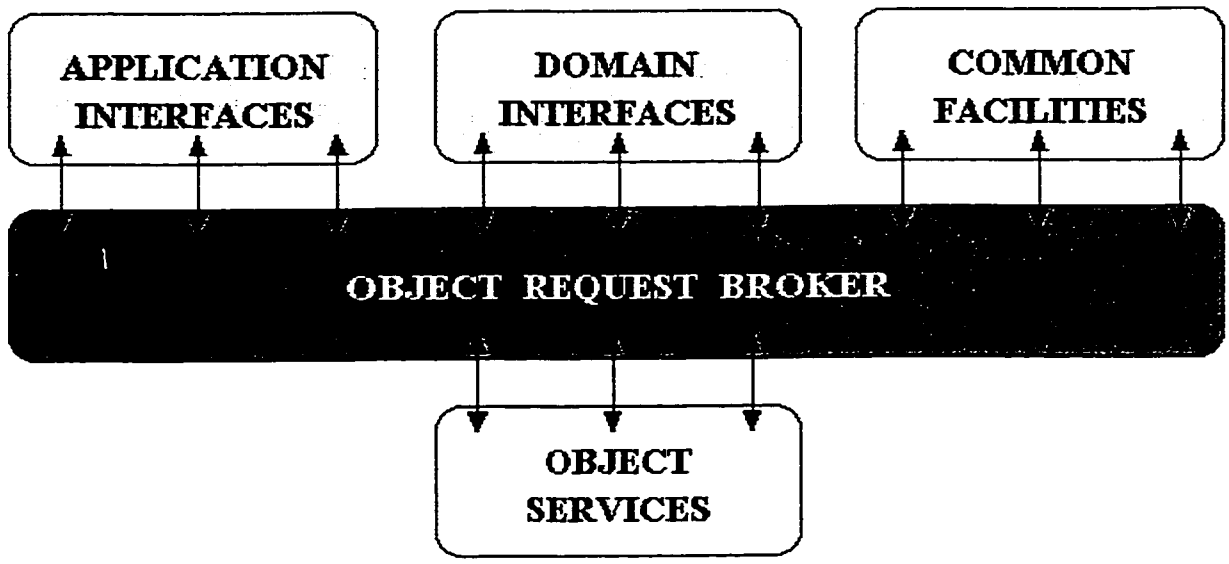


Figure 3.1 OMG Reference Model Architect

- **Object Services** -- These are domain-independent interfaces that are used by many distributed objects. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of Object Services that fulfill this role are:
    - The *Naming Service* -- which allows clients to find objects based on names;
    - The *Trading Service* -- which allows clients to find objects based on their properties.
- **Common Facilities** -- Like Object Service interfaces, these interfaces are also horizontally-oriented, but unlike Object Services they are oriented towards end-user applications. An example of such a facility is the *Distributed Document Component Facility* (DDCF), a compound document Common Facility. DDCF allows for the presentation and interchange of objects based on a document model. For example, DDCF facilitates the linking of a spreadsheet object into a report document.
- **Domain Interfaces** -- These interfaces fill roles similar to Object Services and Common Facilities but are oriented towards specific application domains. For example, one of the

first OMG Request For Proposals (RFP) issued for Domain Interfaces is for Product Data Management (PDM) Enablers for the manufacturing domain. Other OMG RFPs is being issued in the telecommunications, medical, and financial domains.

- **Application Interfaces** - These are interfaces developed specifically for a given application. Because they are application-specific, and because the OMG does not develop applications (only specifications), these interfaces are not standardized. However, if over time it appears that certain broadly useful services emerge out of a particular application domain, they might become candidates for future OMG standardization.

3.3.2 CORBA ORB Architecture

Figure 3.2 illustrates the primary components in the CORBA architecture. Descriptions of these components are given below the figure.

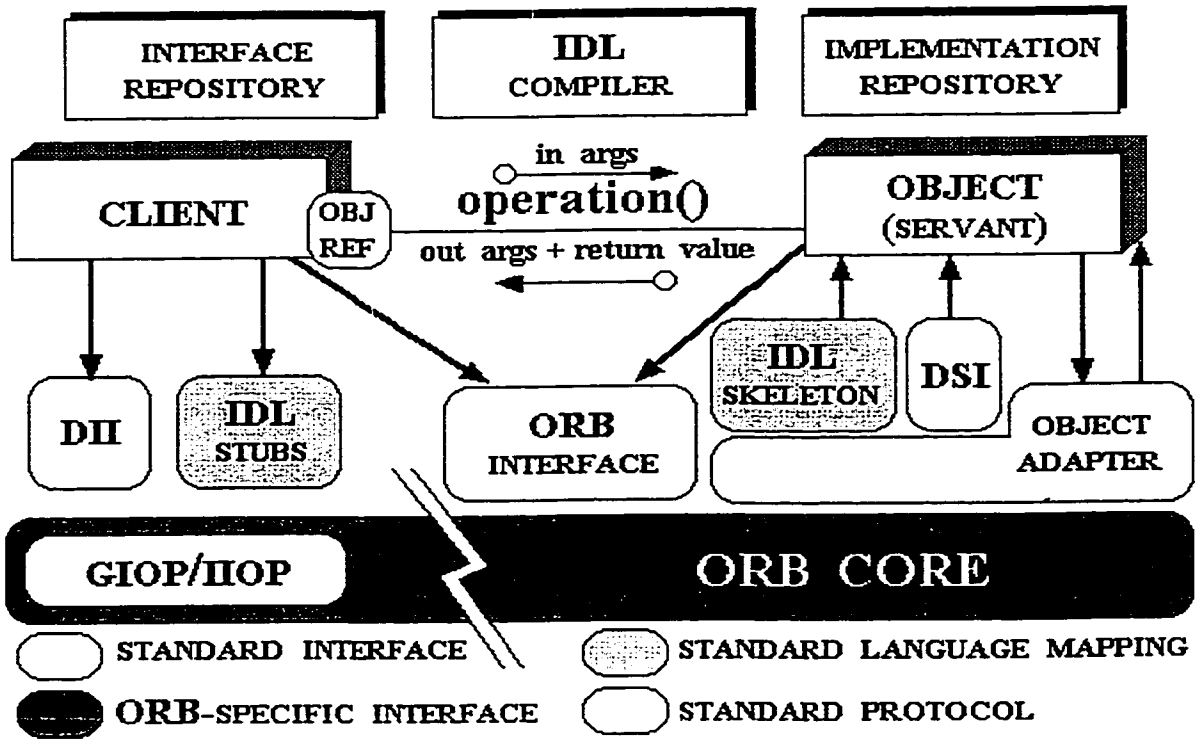


Figure 3.2 CORBA ORB Architecture

- **Object Implementation (Servant)** -- This defines operations that implement a CORBA IDL interface. Object implementations can be written in a variety of languages including C, C++, Java, Smalltalk, and ADA. An object implementation provides the semantics of the object, usually by defining data for the object attributes and code for the object's methods. Often the implementation will use other objects or additional software to implement the behavior of the object. In some cases, the primary function of the object is to have side effects on other things that are not objects.

  A variety of object implementations can be supported, including separate servers, libraries, a program per method, an encapsulated application, an object-oriented database, etc. Through the use of additional Object adapters, it is possible to support virtually any style of object implementation. Generally, object implementations do not depend on the ORB or how the client invokes the object. Object implementations may select interfaces to ORB-dependent services by the choice of Object Adapter.

- **Client** -- This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Clients generally see objects and ORB interfaces through the perspective of a language mapping, bringing the ORB right up to the programmer's level. Clients are maximally portable and should be able to work without source changes on any ORB that supports the desired language mapping to any object instance that implements the desired interface. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it.

- **Object Request Broker (ORB)** -- The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller. In the architecture, the ORB is not required to be implemented

as a single component, but rather it is defined by its interfaces.

Any ORB implementation that provides the appropriate interface is acceptable. The interface is organized into three categories:

- Operations that are the same for all ORB implementations
- Operations that are specific to particular types of objects
- Operations that are specific to particular styles of object implementations

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different properties and qualities. There may be multiple ORB implementations (also described as multiple ORBs) which have different representations for object references and different means of performing invocations. It may be possible for a client to simultaneously have access to two object references managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their object references. It is not the responsibility of the client to do so. General Inter-ORB Protocol (GIOP) is the protocol to transfer message between ORBs. Internet Inter-ORB Protocol (IIOP) is the GIOP message format sent over TCP/IP.

- **ORB Interface** -- An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface (DII) described below.

- **CORBA IDL stubs and skeletons** -- CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. A CORBA IDL compiler automates the transformation between CORBA IDL definitions and the

target programming language. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for compiler optimizations.

- **Dynamic Invocation Interface (DII)** -- This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow Remote Procedure Call: RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* calls (separate send and receive operations) and *oneway* (send-only) calls.

- **Dynamic Skeleton Interface (DSI)** -- This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

- **Object Adapter** -- This assists the ORB in delivering requests to the object and in activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

- **Object References** -- An Object Reference is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and thus are insulated from the actual representation of objects. Two ORB implementations may differ in their choice of Object Reference representations for the same object.

  The representation of an object reference handed to a client is only valid for the lifetime of that client. All ORBs must provide the same language mapping to an object reference (usually referred to as an Object) for a particular programming language. This permits a

program written in a particular language to access object references independent of the particular ORB.

- **Interface Definition Language** -- The OMG Interface Definition Language (OMG IDL) defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. Note that although IDL provides the conceptual framework for describing the objects manipulated by the ORB, it is not necessary for there to be IDL source code available for the ORB to work. As long as the equivalent information is available in the form of stub routines or a runtime interface repository, a particular ORB may be able to function correctly.

- **Mapping of OMG IDL to Programming Languages** -- Different object-oriented or non-object-oriented programming languages may prefer to access CORBA objects in different ways. For object-oriented languages, it may be desirable to see CORBA objects as programming language objects. Even for non-object-oriented languages, it is a good idea to hide the exact ORB representation of the object reference, method names, etc. A particular mapping of OMG IDL to a programming language should be the same for all ORB implementations. Language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB. It includes the structure of the client stub interface (not required for object-oriented languages), the dynamic invocation interface, the implementation skeleton, the object adapters, and the direct ORB interface.

- **Implementation Repository** -- The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

CORBA architecture is open and standardized, communication between distributed objects are

totally transparent, this can greatly improve the testability of applications built on it.

3.3.3 CORBA Development Procedure

A CORBA application can be developed using the following procedural steps:

- The interface of the server object is written in IDL. This interface is actually a skeleton containing the methods, which provide both the services of the server object and access to its instance variables. In the code, we find the signatures of methods but not their implementations.
- The IDL code is compiled (mapped) to the language of implementation using special compilers (such compilers exist for Java and C++). The compiler will produce two types of classes: stub classes are linked to the client-object code, and skeleton classes are linked to the server-object code.
- The methods of the skeleton is implemented in the language to which the IDL code was mapped.
- The code of the client and server applications are compiled using the native language compilers.
- After performing the above procedure, client and server objects are ready to invoke each other's methods. These objects will be bound to different client and server applications.

3.3.4 Current Status and Prospects for CORBA

3.3.4.1 Recognition in Industry Practice

Currently, CORBA is a standard specification and common framework for distributed application development. It is being used as the infrastructure for technology companies like Netscape, Oracle, Sun, IBM and hundreds of others. It is also used worldwide to develop and deploy distributed applications for vertical markets, including Manufacturing, Finance, Telecom, Electronic Commerce, Realtime systems and Health Care. There also are hundreds of vendors of

CORBA products. One leading CORBA vendor – IONA -- has already shipped over 40,000 CORBA products (Orbix developer kits) to more than 3,500 companies worldwide. Below are some success stories of CORBA practice from various e-commerce related industries.

- American Airlines (wwwr3.aa.com)

  The American Airlines website is implemented with Orbix. It handles 12.5% of all travel reservations on the Internet. The American Airlines site has seen on-line bookings increase by 400%, and has generated over $500 million in revenue in 1999. According to Media Matrix, AA.com is the number 1 airline site on the web.

  "The integration capabilities of IONA's Orbix enabled us build a dynamic and interactive web service that relies upon communication between the browser at the front-end and a variety of back-end servers and databases," John Samuel, managing director of the interactive marketing group at American Airlines.

- Chicago Stock Exchange (www.chicagostockex.com)

  The Chicago Stock Exchange is the fastest growing exchange center in the U.S. In order to be the most technologically advanced stock exchange in U.S., it replaced its core trading system with a new, more powerful, trading system developed on Orbix.

  "Orbix products are an integral component to all of our existing and future distributed trading applications - the value of our technology allows our traders to offer lower costs, more flexibility, reliability and faster service than other exchanges." stated John Kerin, vice president, Chicago Stock Exchange application development. "The inherent scalability that CORBA delivers enables us to easily add to our trading system at any point in the future without the headaches of ripping out and replacing everything."

- CNN Interactive (www.CNN.com)

  CNN Interactive, a division of Cable News Network, has built and is deploying a system using CORBA to gather, store and disseminate news material. The project has enabled the organization to tie together its various clients and servers that include Windows and

Mac desktops, Windows NT, Sun Solaris and Netscape server platforms so that all internal users can access the same material and all servers can talk to one another.

Al Issa, Senior Manager of Software Development and Application Services at CNN Interactive, says that "CORBA is the only way to go for distributing to CNN's complex systems." News and information are received in various media formats including text, audio and video as well as other electronic formats. These are wrapped into CORBA objects which can be stored and easily retrieved by users using a variety of clients including Macs, Windows PCs.

3.3.4.2 Difficulities in Adopting CORBA

Although CORBA is industry-wide, vendor-neutral, and adheres to open standards, it is regarded as a relatively new technology. This raises concerns which may or may not be specific to CORBA, but they become the main reasons that CORBA is not used in some cases. As technology evolves, those concerns may be solved. In this section, we summarize the concerns raised in our investigation into industrial practice.

- TCP/IP related problems

  CORBA IIOP is running on top of TCP/IP. TCP/IP messages are transferred between the client computer and server computer by IP address and Port Number. In the CORBA model, the Port Number via which the client communicates with the server object is assigned dynamically when that server object instance is invoked. This makes it difficult to deploy for applications that have to be accessible via the Internet (not Intranet). For security, most organizations tend to impose restrictions on what kind of communications traffic they allow into or out of their internal network. The most common way for organizations to control traffic flows is to setup *firewall* between the internal network and the Internet. A firewall is a system consisted of hardware, software, or both that isolates a private system or network from a public network.

  To set up a firewall filter, the specific Port Number that allows IP traffic to go through

needs to be specified. That's why the CORBA model does not fit well into this environment. CORBA provides the functions to fix a static port number for CORBA objects, but it will not be easy to convince corporate security officers to open up non-HTTP ports (normally 80) through their corporate firewall.

Some products attempt to get around this problem by transporting all CORBA communications (IIOP messages) on top of HTTP. This method, however, incurs huge performance overhead because:

- CORBA objects are remote objects. Every time you invoke a method, however simple it may be, e.g. get a property value of the object, you incur a communication round trip between the client and the server;
- HTTP is a 'connectionless' protocol, that is, every time the (Web browser) client sends a request to the server, it has to set up a TCP/IP connection to the server.

Some products get around this problem by sticking with only HTTP communications between the browser client and the server. The messages transmitted over HTTP are in XML format. (How to use XML and CORBA together is described later)

- Performance

CORBA doesn't promise performance. It is very difficult to determine ahead of time what the performance of a system will be like, if people are putting together a small application on a machine with lots of CPU power and high bandwidth then this is not a concern. If they want to manage several hundred old workstations over slow, bandwidth-limited connections, this is a concern. So performance depends on what infrastructure an organization has for its e-commerce system. So far, there are no concrete statistics of how CORBA could slow down the system.

- Integration with Third-party Software

As explained previous section, the CORBA IDL compiler generates C++ code from IDL

file. In most CORBA products, e.g. Orbix, the code is compiled using HP ANSI C++ compliant compiler. But some of C++ code in the organization's old system in fact is not ANSI C++ compliant. Trying to get all of this code to correctly compile and link is a potential problem. This contradicts a main selling point of CORBA, namely its ability to wrap up legacy applications with IDL interfaces.

- Learning curve

  This is a debatable point since there are very different point of views. For people who are familiar with Object-Oriented concepts and distributed computing, learning CORBA is easy. For people just coming into this field, the learning curve for CORBA will be long. Another complaint about CORBA is the memory management model used by CORBA. Memory used to send and receive responses must be managed by the programmer. The rules are complex as they differ depending on the data being sent (integer, string, structure, etc) and the direction of transmission (server to client or client to server). But memory management is a common issue of distributed computing.

- Business Decision

  Whether or not to use CORBA has to be factored into many business considerations. The truth is: business issues normally dominate technical issues in industry.

  In other words, there is no way to answer the CORBA question without knowing the business considerations of the organization that is deciding to use these distributed object technologies. There is no universal notion of better and worse and no easy answer in relation to the business decision. The vast majority of these decisions are dominated by business considerations, not by technical considerations.

3.3.5 CORBA Products

The following is a list of CORBA implementations that are available today:

- Orbix from IONA

- VisiBroker from Visigenic
- CORBAplus from Expersoft
- ORBacus from OO Concepts
- OmniORB AT&T Research Labs
- TAO from Washington University
- Chorus COOL ORB from Sun
- OAK from Paragon Software
- Component Broker from IBM
- The MICO ORB
- RCP-ORB from Nortel

3.3.6 Benefits and Limitations of CORBA to Testing Distributed System

- The key feature of CORBA is transparency. ORB hides the following characteristics of objects: object location, object implementation, object execution state, and object communication mechanisms. It simplifies the observation and monitoring of the object and system behavior, provides high testability.

- CORBA makes application (including the system under test and test system) construction and integration much easier, since CORBA provides the standard ways to allow objects to be accessed from anywhere in a distributed system. It greatly increases the development efficiency.

- IDL defines the software interface. CORBA provides a IDL compiler to compile the IDL code and generate the language specific interface for both server side and client side. There are two advantages of this feature. Firstly, the software interface is verified before it is

implemented, thus it can limit errors in early stage of the software development, reduce costly rework in the future; secondly, the interfaces of both server side and client side are generated from the same IDL file, it prevents both sides from the possible inconsistency in their implementation.

- CORBA provides great interoperability for software implemented on different operating systems and in different programming languages. Developers of the test architecture don't need to worry about the operating system and the language of the system under test, they can choose to built the test system in the most suitable environment, and still get the communication across the heterogeneous network.

- Of course, as mentioned in section 3.3.4.2, CORBA is a relatively new technology. There are still some concerns around it. The answer will rely on evolution of technology.

3.4 Other DOC Frameworks

3.4.1 Microsoft Technology

Microsoft Component Object Model (COM) is a competing architecture OMG's CORBA for application development.

COM includes the Component Object Model and its implementations on Microsoft and other operating systems. It is an integration infrastructure, used to integrate components that interact within a single address space (called *in-process*) or between processes on a single host (called *local* or *out-of-process*). It underlies OLE, ActiveX, and an increasing number of other services provided by Microsoft's operating systems, such as the DirectX multimedia services [COM,1995].

DCOM (Distributed COM) is the extension of COM that allows interaction between objects executing on separate hosts in a network. DCOM arrived with Windows NT 4.0 and as a separate add-on for Windows 95 in late 1996.

COM+ is the recently announced plan for a new generation technology. It will provide a variety

of new features, including runtime object management and language-based development tools.

Figure 3.3 shows the present COM architecture. (Note that this diagram is not an authoritative Microsoft depiction of the architecture. Furthermore, it will likely evolve when COM+ is released.) Microsoft Transaction Server (MTS), ActiveX, and OLE are client services that depend on the COM Infrastructure. In addition, Message Queue (MSMQ) is currently available as a separate service that supports distributed processing. Microsoft has announced that this service will be integrated into the COM Infrastructure.

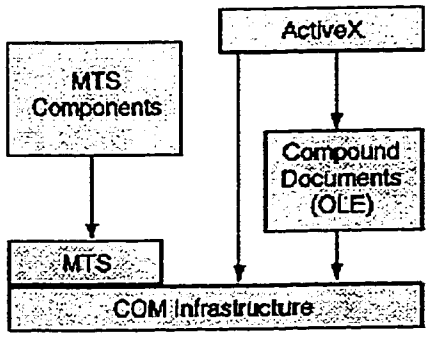


Figure 3.3 Component Object Model Architecture.

3.4.2 Java Beans/RMI

Java is the newest member of the object community. Java is an object-oriented programming language, and Remote Method Invocation, or RMI, is the standard way for Java objects to communicate with one another across a network. As an object-oriented language, Java has a built-in object model. The other key feature of Java is platform independence. Java applications are built to run under the control of the Java virtual machine, a kind of abstract computer platform which supports Java by implementing the Java virtual machine. Thus any Java program can run almost anywhere.

Java Beans are the Java version of component software. Beans implement a set of standard interfaces and behaviors similar to ActiveX, which permit bean behavior to be customized at design time. Java Beans can also use a set of bridges, which permit beans to interoperate with RMI-based objects, ActiveX controls, or CORBA objects.

Java and CORBA are commonly used together to combine strengths of the Java client and CORBA distribution medium. Many businesses use Java on their client-side application and languages like C++ for the server-side. C++ is faster than Java; Java, on the other had offers portability and better presentation services. Because Java RMI is only for a Java object to communicate with another across a network, in practice, business mostly prefers the DOC framework which supports multiple platforms and programming languages. CORBA and DCOM are the two prevailing technologies in this regard.

3.4.3 CORBA vs. DCOM

DCOM and CORBA apply object technology to the design and use of software components. Both provide application interoperability and limited portability in distributed computing environments. Both are intended for enterprise-scale development. However they also have many important differences. This section provides some comparison between the two architectures and relates these differences with respects to e-commerce development and testing. For more details see [OMG]

3.4.3.1 Specification

DCOM and CORBA are both described by specifications, but the documents differ substantially in authority, style, precision, and purpose.

Microsoft controls the COM specification [COM, 1995] and its implementation. Furthermore, the company has always maintained that the implementation is the authoritative specification. The specification is thus always incomplete, making it difficult in some cases to determine the formally correct behavior.

The CORBA specification results from a defined adoption process. The specification is formal and expressed in IDL. (OMG IDL is also an ISO standard, number 14750.) Despite the predictable shortcomings of any standards process, CORBA implementations have achieved substantial technical and market acceptance, particularly among large enterprises. This has fostered competition and innovation. Consistent and standardized specification is also a key

aspect of testing process.

3.4.3.2 Object Model

DCOM and CORBA both make the distinction between *interfaces*, which are abstract collections of related methods, and *implementations*, which are concrete embodiments of the methods. This distinction is the fundamental concept that permits isolating clients from implementation details. An interface definition allows only method invocations, so that, like a traditional API, it hides the details of the implementation from the client of the interface. Clients invoke methods on objects *by reference*, never directly. The actions are ultimately carried out in the address space occupied by the object, which might be remote from the client. Some object distribution architectures, e.g., ODBMS, take a different approach, actually caching the object temporarily in the client address space, having reserved it from a central store.

3.4.3.3 Errors and Exceptions

CORBA specifies an extensible exception capability that maps naturally into languages that have native exceptions, like C++ and Java, and maps into exception data in languages that do not. It is based on user-defined exception types in CORBA IDL. In practice, this mechanism works well.

COM has a standard way of handling error data through return of a 32-bit value called an HRESULT. An exception mechanism was recently added that transports an exception object from object to client. This shows evidence of an ad-hoc origin. It does not intrinsically support user-defined exception types, but provides an exception object that will convey a fixed set of user-supplied data. Since the exception object is a COM object, users can provide an alternate implementation, but it will not automatically marshal the exception data by value when returned to the client, as the default exception implementation does. (To achieve the same effect with a user-defined exception object, that object must employ custom marshaling.)

From testing point of view, CORBA's exception handling is based on user-defined error and exception type, so it is more flexible and straightforward and easy for error locating and analysis.

3.4.3.4 Identity and Persistence

COM and CORBA have distinctly different notions of object identity. CORBA assumes an instance can have some meaningful existence that persists over time, so it provides *object references* that identify instances uniquely within a network. Objects can be activated, saved to persistent storage and deactivated, and reactivated again, without losing identity. Object references can be used in a directory or name service to point to significant, long-lived objects.

COM has two concepts of identity, neither of which is as strong as CORBA's. First, two active objects are identical if their `IUnknown` pointers are identical. However, this comparison has nothing to do with the persistent state of an object, and no longer has meaning when the object is deactivated. Second, the identity of a persistent object can be encoded as a string name, and subsequently resolved into a usable interface pointer by using a moniker. Using monikers involves more steps than an object reference. A system can have multiple moniker implementations. A client must have access to the correct moniker implementation for the kind of persistent name string being used.

Since COM's concepts of object identity is not as strong as CORBA's, it is not easy to continuously trace one object's behavior, thus inhibits testability of the object.

3.4.3.5 Platform and Tool Support

COM has been limited to Windows NT and Windows95 prior to 1998. However, Microsoft now supports the implementation of COM on other platforms.

CORBA is designed to support many widely used programming languages without requiring their modification. CORBA's language mappings specify how IDL is translated into the target language in a relatively natural way. This makes it possible to use a CORBA implementation with one's choice of language, compiler, development tools, and operating system. No specific tool support is required to make the system work. This freedom of choice also means a lack of standard tools, which can lead to inconsistency in the tool sets developers use on different platforms. The OMG has produced standard CORBA language mappings for Java, C, C^{++}, Smalltalk, Cobol, and Ada95. Some vendors have implemented mappings for other languages.

For the reasons given above, CORBA has become an increasingly popular standard for distributed objects in the industry.

3.4.4 CORBA vs. XML

XML stands for Extensible Markup Language. XML and its numerous brethren are created and controlled by the World Wide Web Consortium (W3C). Like CORBA, XML is one of industry-wide, vendor-neutral, open standards.

XML is a meta-language for defining tag-based markup languages. It uses a portable format that is both machine and human readable, and is used to produce documents that convey content with semantic structure. An XML document need not simply be a file; it can be generated dynamically.

After XML came out, there was quiet a lot of noise that XML will replace CORBA. Both XML and CORBA are important in their own right; they are the best tools for the job they are designed for. So do XML and CORBA conflict or cooperation? The answer seems to be to make XML and CORBA interoperable.

It is still important to distinguish CORBA from XML.

- CORBA is an enabling technology for creating sophisticated, distributed object systems on heterogeneous platforms. XML is a technology for conveying structured data in a portable way.
- CORBA allows users to connect disparate systems and form object architectures. XML allows users to transmit structured information within, between and out of those systems, and to represent information in a universal way in and across architectures.
- CORBA sets up a persistent connection, sends requests back and forth and maintains the transaction states. XML doesn't have support for transactions, security, session management or long-term association of client with a state at the server end.
- CORBA defines the smaller packets of transient, machine readable data that are exchanged

between the components of a distributed application. When the requirement is to exchange data between cooperating computer applications for the data exchanges that tie together the components of a distributed system, CORBA is the better candidate. XML is intended for marking up human-readable, textual data. When the requirement is to store data for the long term and extract human-readable summaries and reports, then XML would be the more appropriate approach.

Both technologies are platform, vendor, and language-independent. The conceptual fit is perfect. To see where and how this fit is best realized, we will examine how to actually combine CORBA and XML from a series of widening perspectives.

- The information perspective

  XML is a natural way to represent configuration information components, services, servers and the like. In the past, configuration has often been achieved with custom, non-portable data formats and mechanisms. XML-based configuration is portable and usable in a generic fashion. This is straightforward and already being done today.

  Systems must maintain application data such as session states. This must often be communicated between subsystems or systems, and is thus a good candidate for standardization via XML. A ubiquitous example of this is logging information that all services pass to a central archive service.

  Using XML to convey data in a CORBA-based system helps make the system more flexible. Instead of narrowly tailoring IDL interface operations for a given set of data, the operations accept/emit XML. This leaches some of the semantics out of the interface. The advantage is that the data can be changed without changing the interface (a task that usually snowballs in an expensive way).

- The layer perspective

  Most CORBA architectures comprise multiple tiers or layers. XML can be used at the interfaces between layers to provide greater decoupling and flexibility, serving as the

glue in the architecture.

An application layer can produce XML output not just for client consumption, but for data interchange with other systems. This is particularly useful for business-to-business interactions.

- The communications perspective

  XML provides a way of conveying messages across many traditional technology boundaries separating systems. Some current efforts are exploring the use of XML via HTTP as an alternative to IDL/IIOP for accessing CORBA systems. This can support non-CORBA-enabled simple clients, or clients using a non-IIOP-compatible protocol. In addition, using HTTP to transport XML eliminates many firewall issues. A request/response message XML format can be interpreted by an XML-CORBA bridge which translates XML into CORBA requests and vice versa. Essentially, such a system is an XML-to-CORBA DSI gateway.

In summary, XML and CORBA are complimentary technologies. XML is intended for the storage and manipulation of text making up humane-readable documents like Web pages, while CORBA ties together cooperating computer applications exchanging transient data that will probably never be directly read by anyone. Neither of these technologies will replace the other, but instead they will increasingly be used together [Elenko, 1999].

From the above description of distributed object technology, we can see that it has a number of implications for the design and testing of e-commerce systems. This is discussed in the next section. Generally, advantages include:

- Rapid application development

  Object technology generally permits more rapid application development than earlier methods of programming. Obviously, this advantage is not at all unique to Internet commerce.

- Distributed applications

  Object technology permits applications to be distributed across multiple computers without the development staff necessarily having to be experts in all aspects of network communications and security.

- Flexibility of deployment

  Object technology, particularly the availability in the Web context of downloadable applets and controls, gives great flexibility to the designers of the e-commerce application in designing multitier applications, placing functionality at the servers, or migrating functionality all the way to the client desktop. Automatically moving part of the application to the desktop can provide great advantages of interactivity and interoperability, but also introduces some complex security and trust problems, since the desktop computer may not be a trustworthy computing platform[Dima,1999].

In practice, object technology will likely have a role in almost all Internet commerce projects. As the prominent distributed object framework, CORBA will likely be the main driving technology behind e-commerce application development [Andreoli, 1997].

3.5 CORBA-Based E-commerce System Architecture

A typical e-commerce system requires the collaboration of several entities (or agents) to complete an order transaction. The orderly and timely exchange of messages among those objects ensures the correct execution of a transaction. These agents are the consumer, the business, a financial institution, a payment server, a shipper and a supplier.

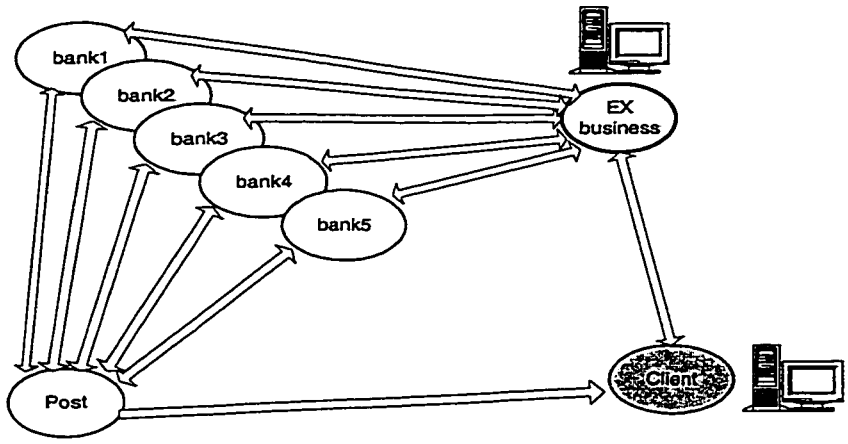


Figure 3.4 Agent Interaction Diagram

In our case study system (see section 2.6) , these agents are banks, business and post office. An agent interaction diagram is shown in Figure 3.4.

Because of the distributed nature of the application, it seems intuitive to have distributed objects (components) acting on behalf of each of the collaborating agents listed above. Using CORBA's ORB as a secure delivery platform for the application, distributed objects can collaborate to successfully complete the requested business function.

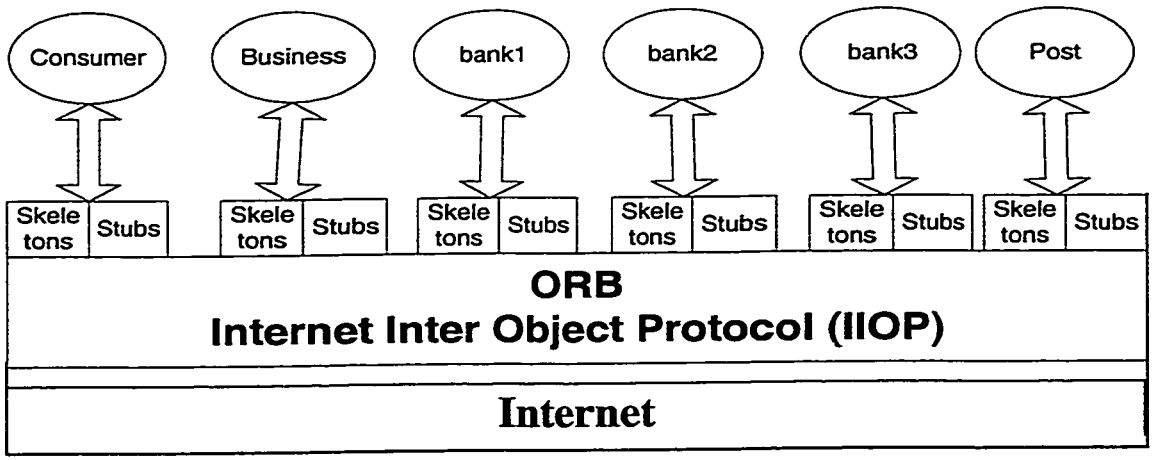


Figure 3.5 CORBA based system architecture of Online Currency Exchange

Figure 3.5 shows the CORBA-based system architecture of Online Currency Exchange. Different actors – consumer, business, banks, shippers -- in the system uses the ORB as transport mechanism.

CORBA provides the communication mechanism for application objects. The ORB uses IIOP for communication between distributed objects running on the Internet. IIOP is an application protocol that runs on top of TCP/IP.

IIOP hides the complexity of the heterogeneous networking environment, consisting of many different kinds of network hardware, topologies, routers, Ethernet, dial-up connections, and so on, from applications. As a consequence, application developers and users are insulated from the complexities of different network devices as well as from the complexities of implementing low-level network protocols.

Thus, CORBA provides simplicity to developers so that developers can concentrate on the service and functionality of the e-commerce system.

In contrast to stateless HTTP, IIOP can preserve the state of the sessions. It can also handle multiple requests per session. Once the connection is established between CORBA objects, messages do not go through the web server

In chapter 7, we give formal definition of the service provided by EX business in CORBA IDL.

There are 3 basic services: login, getQuote, placeOrder

In the IDL file, we also defined the data transferred by the EX business component, such as customer account information, the Quote, the order, etc. Here we give only the definition of EX business services.

```
module business {

  interface Business {

    // login customer's trade account in the business

    ClientAccount  login( in string customerID,

                          in String password);

    // return best quote to customer after negotiation with banks

    Quote  getQuote ( in double exchangeValue);

    // process order upon customer request

    void  placeOrder ( in ClientAccount client,

                       in Order order);

    };

};
```

Besides the simplicity, CORBA provides a lot of benefits. CORBA is platform independent, language independent, database independent, it provides the interoperability to object implement in different OO programming language, based on different platform, from different vendors. Developers can choose the most appropriate OO language, O/S, Database and software packages from different venders.

3.6 Testability of CORBA-based System

In addition to the above benefits, from a testing point of view, CORBA provides a good environment for designing highly testable distributed systems. IDL provides easily observable and controllable interfaces for all the objects in the system.

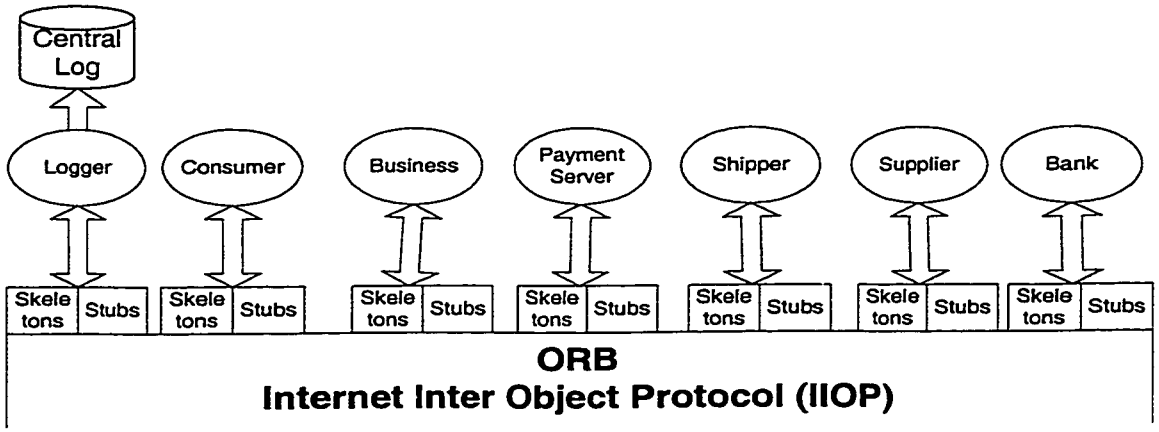


Figure 3.6 Observability of CORBA-based system

Figure 3.6 shows an example to illustrate the observability of CORBA-based system. With ORB and IDL, objects are transparent to each other regardless of their location and implementation details. We can just plug a Logger object into the ORB bus. The Logger object collects critical operational information for security, recovery, statistics, and diagnostics. Logger is just the simplest way to take advantage of high testability of CORBA-based object. We can do more complicated real-time monitoring and testing. More details are given later in this thesis.

In this chapter we discussed distributed object-oriented technology, frameworks utilizing the technology – CORBA, DCOM, XML and their benefits to distributed system development. In the next chapter, we will turn to software testing fundamentals for distributed and particularly for e-commerce systems.

Chapter 4

Distributed Testing Fundamentals

This chapter summarize the basics of software testing, including test strategy and various aspects of conformance testing issues and definitions in Conformance Testing Methodology and Framework [CTMF], the international standard for distributed testing of communications software and protocols. Followed by a detailed discussion of distributed system testing and its test architecture.

4.1 General Test Strategy

There are three well-established software test strategies available:

- Functional or Black-box testing

  Black-box tests are derived from the functional design specification, without regard to the internal program structure. It tests the product against the end user requirements, and external specifications. Black-box testing is done without any internal knowledge of the component under test (CUT). Three well known techniques of black-box testing are: Equivalence partitioning, Boundary-value analysis, and Cause-effect graph[Myers, 1979].

- Structural or White-box testing

  White-box tests require internal knowledge of the CUT to ensure its internal behavior is correct. The main techniques are Statement coverage, Decision coverage, Data Flow coverage, and State/transition coverage[Probert, 1994]

- Grey-box testing

Combines black-box testing and white-box testing. It focuses on messages import/export for the CUT and interactions between CUTs. Techniques of Grey-box testing include: C++ Data Scenarios [Turner & Robson, 1993], Modifier Sequences [Parrish et al., 1994], OOSE Testing Strategy [Jacobson et al., 1992], FREE Approach [Binder, 1994], Race Condition Analysis [Lamarche, Probert, et al].

4.2 Standard Test Methods

Since testing is an important concern in software development, a lot of standards work has been done to address various aspects of testing. For the purpose of this thesis, we are most interested in the Conformance Testing Methodology and Framework standard [CTMF]. This framework has helped the design and development of our framework for e-commerce testing in Chapter 6.

4.2.1 Conformance Testing Methodology and Framework

IS9646-OSI Conformance Testing Methodology and Framework addresses the standardization issues associated with conformance testing, such as general concepts, abstract test suite specification, TTCN (Tree and Tabular Combined Notation), test realization, etc. CTMF is the basis for developing variations of test frameworks and procedures for different application domains. In this section, we will give a brief overview of CTMF.

4.2.1.1 Basic Framework

CTMF is a basic framework for speaking about conformance and the framework for testing to determine whether equipment conform to a given specification. *Conformance requirements* of standards are deemed to fall into two categories:

- *Static conformance requirements.* This is the set(s) of requirements that are stated either in conformance clauses in the base standards or in the profile documents. It represents a sort of checklist. In fact, an implementation might be failed solely on the basis of having the wrong answer to a written question on capabilities, without ever undergoing any real live testing. Thus the static conformance requirements provide, at a broad level, the basis

for selecting the right tests to be applied to the equipment to be tested.

- *Dynamic conformance requirements.* These are the actual protocol behavioral requirements, that is, the requirements that arise from the protocol itself. They are the requirements that the implementation must exhibit the behavior permitted by, and/or required by, the protocol specification. It is the dynamic conformance requirements that are actually tested for, through the use of conformance test suites.

The *Implementation Conformance Statement* (*ICS*) is a checklist based on both the static and dynamic conformance requirements specified in the base standard. It is a summary in tabular form of the capabilities of the system to be tested, and it is completed by the party submitting the equipment for testing. It servers three purpose:

- It provides the organization which is going to perform the testing with information that enables an appropriate set of tests to be selected.
- It protects the party submitting the equipment to be tested, by ensuring that only tests appropriate to their conformance claim are actually conducted.
- It provides a basis for conducting the static conformance review.

4.2.1.2 Conformance Testing as a Process

The basis of conformance testing is the process outlined in Figure 4.1.

In the *Static Conformance Review*, the information in the *ICS* is compared against the static conformance requirements specified in the base standard. This review may immediately result in a rejection of the implementation as nonconformant.

In *Test Selection and Parameterization*, the set of tests are selected that match the characteristics of the IUT. These tests are drawn from the conformance Test Suite, which contains, in theory, a complete set of tests covering every aspect of the base standard. Based on the information peculiar to the IUT contained in the *ICS*, an appropriate subset of tests is selected for execution. IXIT(Implementation eXtra Information for Testing) contains system-specific information

outside the scope of the base standard.

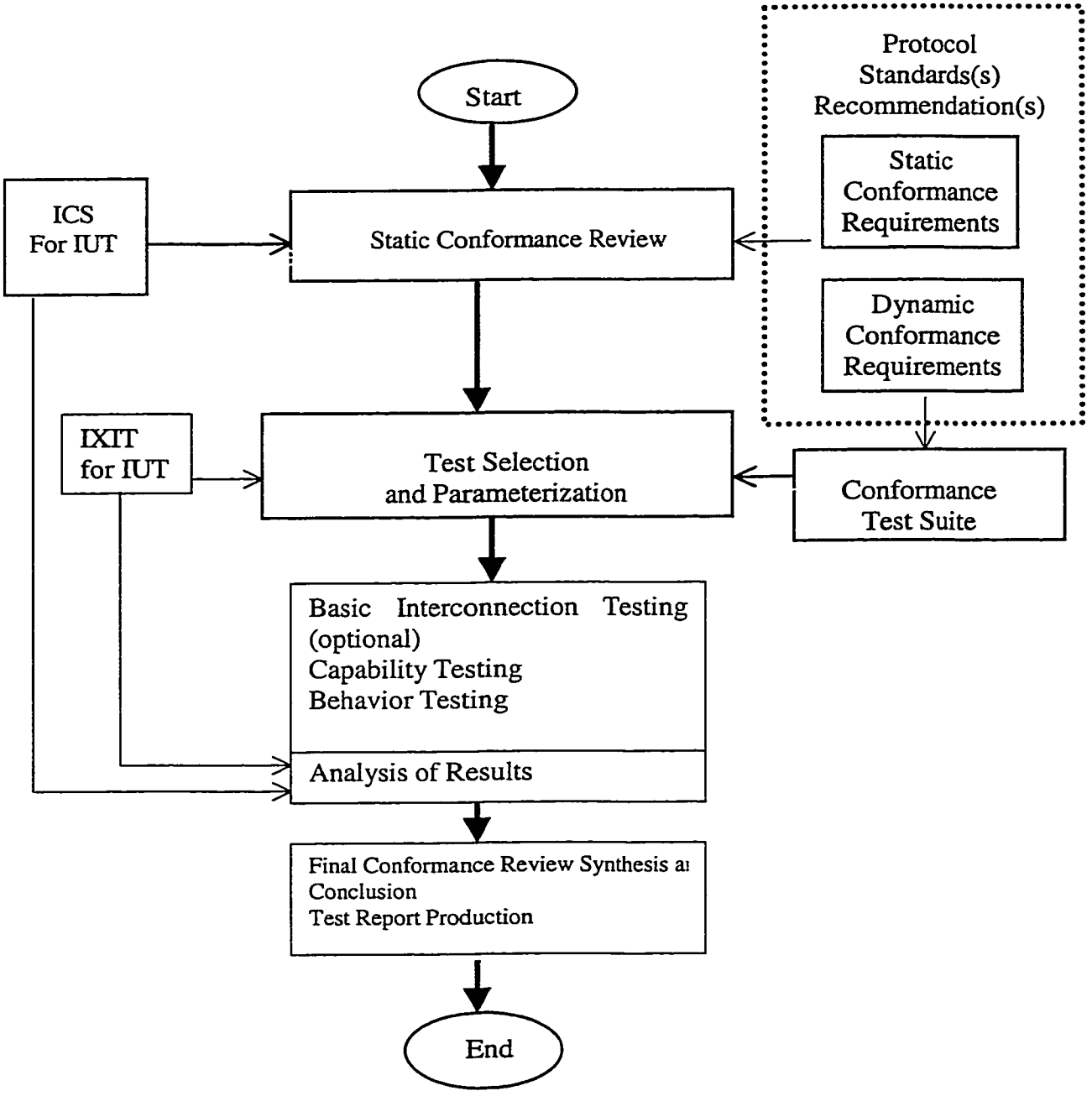


Figure 4.1 Conformance Testing Process

During *Dynamic Testing*, three sorts of tests are performed: Basic Interconnection Tests, Capability Tests, and Behavior Tests.

For every outcome of the tests, a test verdict of either pass, fail, or inconclusive will be assigned by comparing to the documented foreseen outcomes. The specific verdict is specified in the TTCN Test Suite.

4.2.2 Abstract Test Methods

OSI protocol standards define the allowed behavior of a protocol entity in terms of the Protocol Data Units (PDUs) and the Abstract Service Primitives (ASPs) at the upper and lower service boundaries. The OSI conformance testing methodology employs the testability principles of control and observation of service primitives and protocol data units at specified points and layers. This principle results in four basic test methods and their variants by which the protocol implementers may choose to have their product tested. The choice of method is important to the implementers since each test method requires a different degree of exposure of the proprietary aspects of the product.

CTMF describes the following four Abstract Test Methods to conceptually model a real system configuration (Figure 4.2, 4.3, 4.4, 4.5).

- Local Testing Method
- Distributed Test Method
- Coordinated Test Method
- Remote Test Method

Each of the test methods can be described in terms of two abstract testing functions, the *Lower Tester* (LT) and the *Upper Tester* (UT) linked by some form of *Test Co-ordination Procedures.* In each case the IUT resides on top of all previously tested protocol entities in one or more lower layers, collectively referred to as the *Service Provider.* The Lower Tester also implements the protocol entities making up the Service Provider and exerts its control and observation of test events at the IUT lower boundary through the *Point of Control and Observation* (PCO) within the Test System. A typical test event is the sending or the receiving of an ASP or a PDU. The

major testing functions comprising control and observation of the test events are performed by the Lower Tester that is always implemented within the *Test System*, external to the SUT.

This approach to specifying tests between two test entities can be generalized to the case of several lower testers or several upper testers or both. Multi-party testing (shown in Figure 4.6 for example), is under study presently, and has been observed to be particularly useful for testing ISDN systems.

The various test methods are distinguished by the nature of the Upper Tester and the associated Test Co-ordination Procedures. The type of Upper Tester determines the accessibility of the upper boundary of the IUT and, therefore, its testability.

In the *Local Test Method* (Figure 4.2), the upper tester and the test co-ordination procedures are realized within the test system. In all other test methods the upper tester in within the SUT. Furthermore, the local test method does not require that the upper boundary of the IUT be a standardized hardware interface, but leave the upper channel between test system and IUT unspecified.

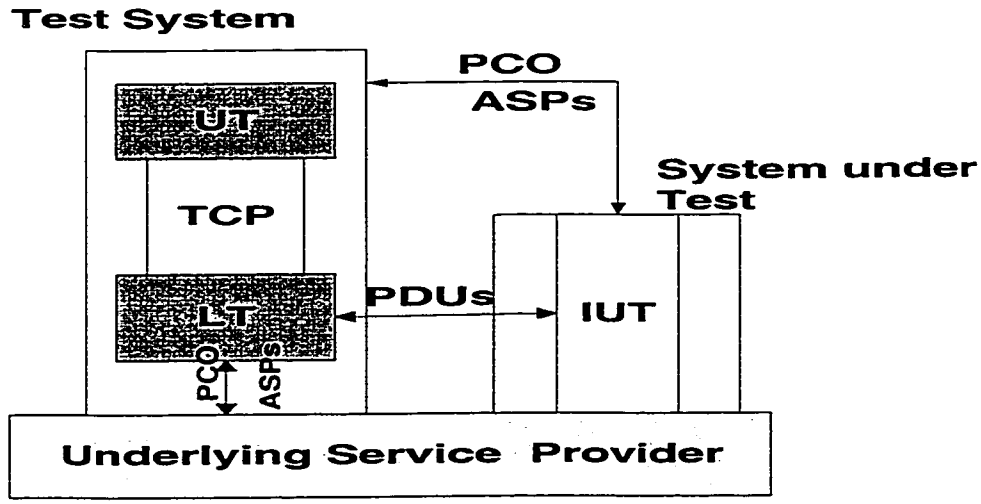


Figure 4.2 Local Test Method

Some access to the upper boundary of the IUT is assumed in both the Local and *the Distributed Test Method,* see Figure 4.3. For the distributed Test Method, the interface at this boundary may be either a human user interface or a standardized programming language interface. In both methods, the requirements on test co-ordination procedures(TCP) are specified but not their method of realization.

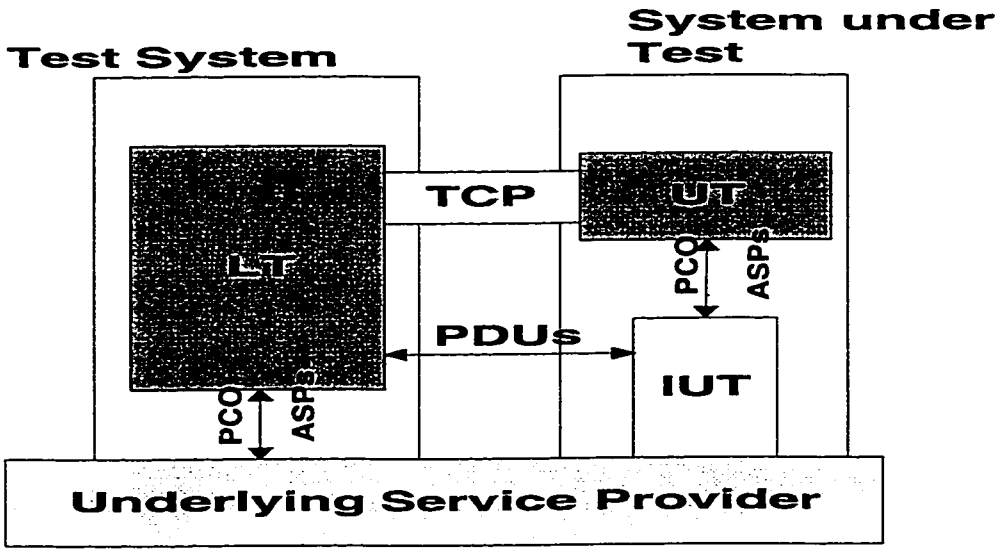


Figure 4.3 Distributed Test Method

In the *Coordinated Test Method,* see Figure 4.4, the test coordination procedures is standardized in the form of a *Test management Protocol* (TMP). The upper tester is controlled and monitored by the TMP, removing the requirement for external access to the upper boundary.

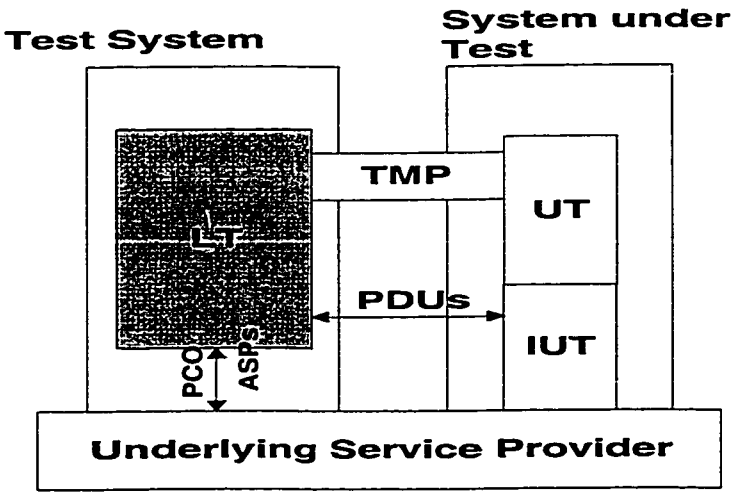


Figure 4.4 Coordinated Test Method

In the Remote Test Method (Figure 4.5), the requirements on the test co-ordination procedures are informally expressed so that only the desired effects of the test co-ordination procedures are stated in the relevant abstract test suite. The functions of the upper tester are carried out by the SUT using whatever means are appropriate to achieve the desired effects.

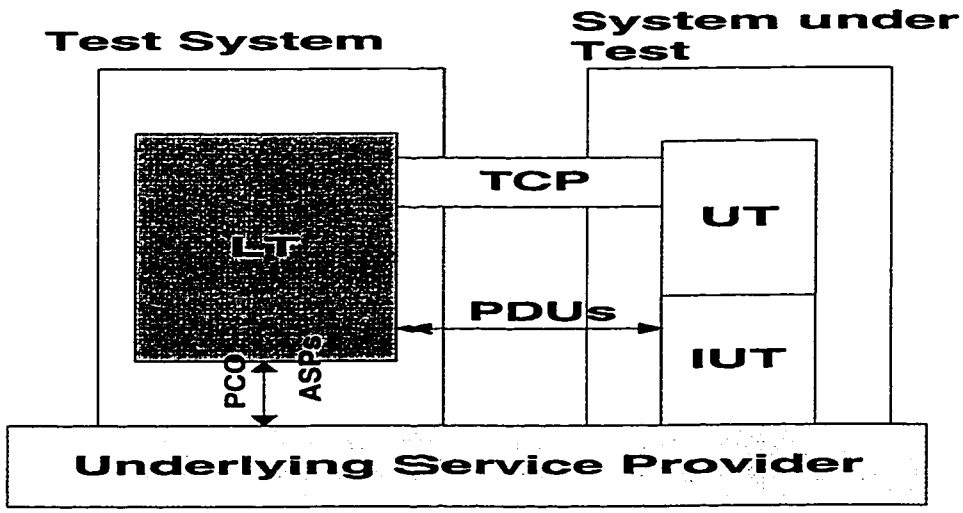


Figure 4.5 Remote Test Method

Note that the methodologies focused so far on single-layer testing, starting with the lowest layer and progressing incrementally, one layer at a time, through the seven layers. At each incremental stage it is assumed that all protocol entities below the IUT have been previously tested and found to be conforming. In practice, especially for development testing, this assumption may not hold. In practice, multi-party testing is used.

4.2.3 Multi-party Testing Context

A generalization of the above discussed abstract test methods, also known as the single-party testing context, is given by the multi-party testing context (Figure 4.6). In this setting, an IUT is supposed to communicate simultaneously with several real open systems; a network of application relays, for instance, maintains communication links with several peers at the same

time. In the multi-party testing context more than one LT, and zero, one or more UTs are active and control and observe the IUT. The coordination of LTs is performed by an entity referred to as the lower tester control function (LTCF). In particular, the LTCF is the entity that determines the final test verdict after test execution. All LTs are required to return a preliminary test result to the LTCF after they have stopped test case execution. The LTCF is also responsible for starting all LTs. Coordination of LTCF, LTs and UTs is defined in a TCP. Communication between LTCF, LTs and UTs is supported by coordination points (CP) which, like PCOs, are modeled as two FIFO queues for inputs and output.

LTs communicate with IUT and possibly with an UT as in the single-party context, the same rules for identifying points of control and observation apply.

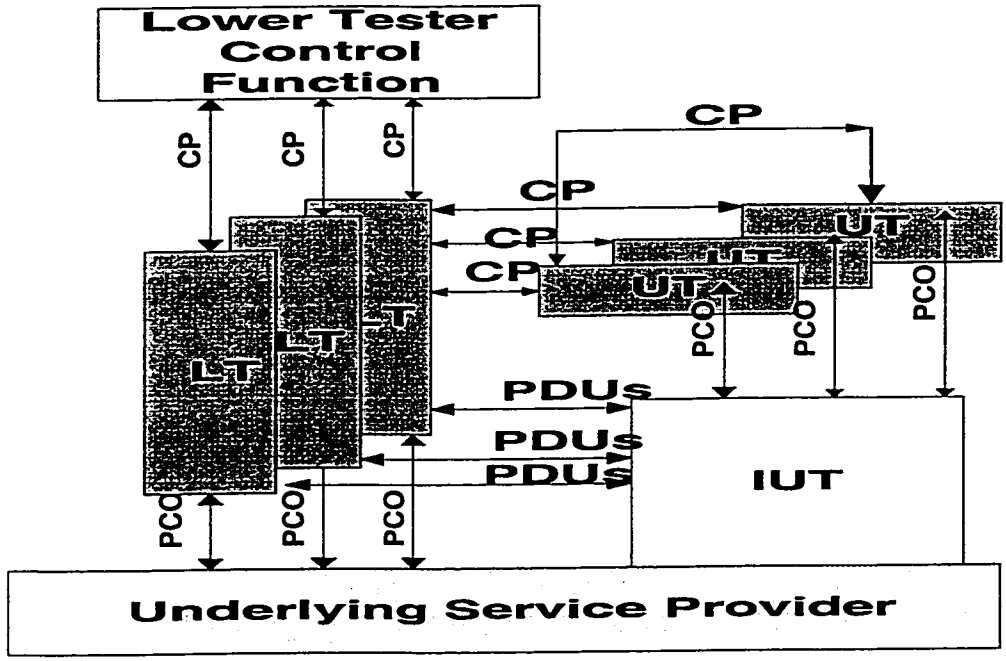


Figure 4.6 Multi-Party Testing Context

4.2.4 Test Architecture beyond Conformance

In (ISO International Standard 9646-1, 1995), the following statement can be found:" The

primary purpose of conformance testing is to increase the probability that different implementations are able to interwork.... Even if two implementations conform to the same protocol specification, they may fail to interwork fully. Trial interworking is therefore recommended." According to [CTMF2, 1993] and [Gadra, 1990], conforming implementations may fail to interwork for the following reasons:

- Protocol specifications contain options and implementations may differ in the options being supported.
- Factors outside the scope of conformance testing and OSI, e.g., performance requirements, may impact the behavior of real systems, which may not be foreseen by the specification.

In the following subsections, approaches to testing interoperability, performance and quality-of-service are discussed. Since the evolution of multimedia and e-commerce applications has started some years ago, the latter type of testing has become a significant issue.

4.2.4.1 Interoperability Testing

Interoperability testing evaluates the degree of interoperability of implementations with one another. It involves testing both the capabilities and the behavior of an implementation in an interconnected environment and checking whether an implementation can interwork with another implementation of the same or of a different type. Interoperability testing is not standardized. However, a couple of interoperability testing proposals and guidelines exist [Buehler, 1994], [Myungchul, 1996].

The two main approaches to interoperability testing in the context of OSI are passive and active testing [Gadre, 1990]. The difference is that active testing allows controlled error generation and a more detailed observation of the communication, whereas passive testing on the other side involves testing valid behavior only. OSI interoperability testing should be done by using a reference implementation (RI) of the protocol entity to be tested. Within the SUT, RI and IUT are combined and the tester functions play the role of service users. The behavior of the RI is

correct by definition. In the case that IUT and RI do not interwork, we assume the error is in the IUT.

Figure 4.7 shows a generic passive interoperability test system architecture where two implementations are interconnected, and control and observation are performed by two UTs.

Practice has shown that in most cases no RI is available. As a consequence, two IUTs are used and the communication between the IUTs is monitored or emulated.

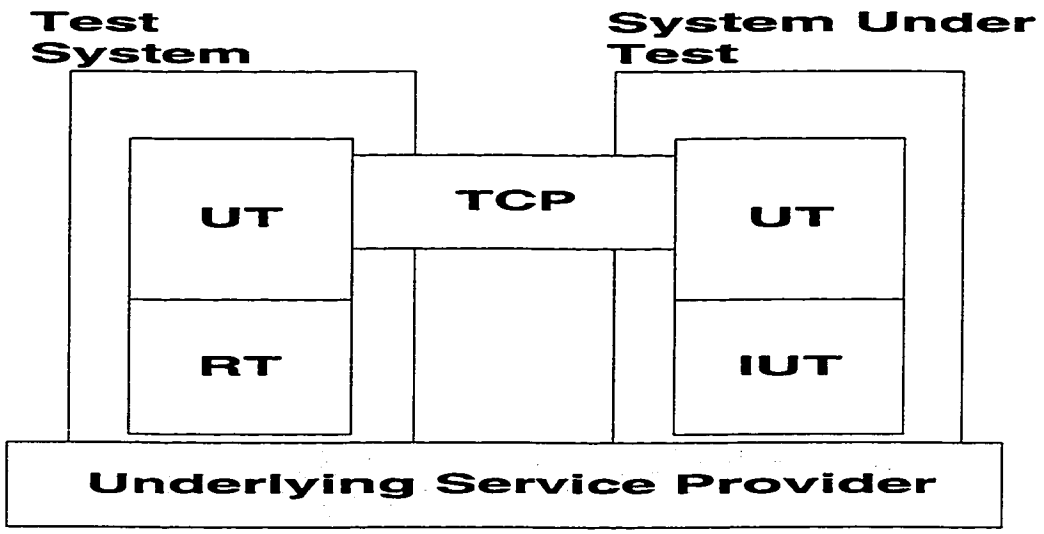


Figure 4.7 Generic Passive Interoperability Test Architecture

4.2.4.2 Performance Testing

The main objective of *performance testing* (Schieferdecker atc, 1997) is to test the performance of a network component under normal and overload situations. Performance testing identifies performance levels of the network component for different ranges of parameter settings and assesses the measured performance of the components. A performance test suite captures precisely the desired performance characteristics that have to be measured and gives procedures to capture and calculate the measurements. In addition, the performance test configuration includes the configuration of the network components, the configuration of the underlying

network, and the network load characteristics.

Depending on the characteristics of the network component under test, different types of performance test configurations are defined: end-user telecommunications application, end-to-end telecommunications service, and communications protocol (Figure 4.8). Foreground test components (FT) implement control and observation of the network under test. Background test components (BT) generate continuous streams of data to load the network component under test. Monitor components are used to monitor the real network load during the performance test. BTs do not control or observe directly the network under test but implicitly influences the network under test by putting the network into normal and overload situations.

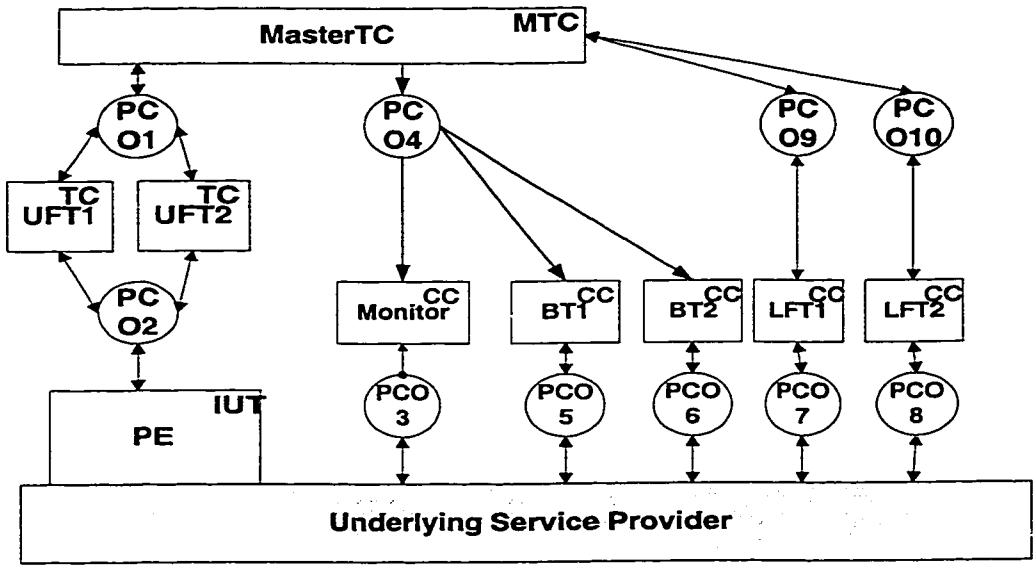


Figure 4.8 Performance Test Architecture

4.2.4.3 Quality-of-Service Testing

Quality-of-service (QoS) [ITU-T, 1989] [Danthine, 1993] refers to a set of parameters that characterize a connection between communication entities across a network. QoS parameters are performance and reliability oriented such as throughput, delay, jitter or error rates and failure probabilities. QoS parameters are negotiated by service users and the service provider at connection set-up and should be maintained during the lifetime of the connection. A QoS

semantics defines how QoS parameters are negotiated and how negotiated QoS parameters are to be processed. It is mainly the service provider who is in charge of maintaining negotiated QoS parameters.

QoS testing refers to assessing the behavior of protocol implementations performing QoS maintenance. However, it is not necessary to control and to observe the behavior of the implementation directly. It suffices if the tester can eventually observe the specified behavior according to the agreed QoS semantics.

Figure 4.9 presents a test architecture for QoS testing [Grabowshi, 1995]. As can be seen, the testing architecture is very similar to passive interoperability testing. The obvious difference is that in QoS testing the IUT is distributed. Furthermore, some QoS parameter tests, e.g., error rate tests and delay tests, require the active involvement of the network(controllability). For this, the network has to be configurable. The test architecture, therefore, provides a communication link between testers and network for the exchange of configuration information.

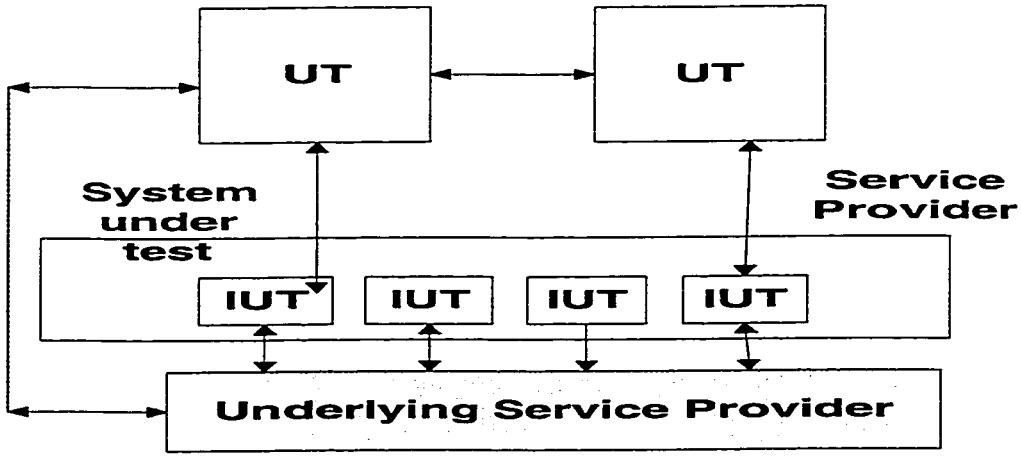


Figure 4.9 QoS Test Architecture

4.3 Generic Test Architecture for Distributed System

With the emergence of new models, architectures and middleware such as ODP, TINA and CORBA for developing open distributed systems, testing technology requires adaptation for conformance assessment in such systems. All these frameworks are object-based and aim at

creating open distributed environments supporting interworking, interoperability, and portability, in spite of heterogeneity and autonomy of the related systems.

A simplified view of distributed system architectures is given in Figure 4.10. A distributed system consists of three layers: application level objects that interact through well-defined interfaces; the middleware platform is a distributed computing environment that supports the implementation of the distributed application by offering various distribution transparencies such as access and location transparency. Example middleware platforms are OMG CORBA or OSF DCE or Microsoft DCOM; the communication network with various network nodes and end-systems offers transmission services to the middleware platform that are used to support the communication between the components of the distributed system.

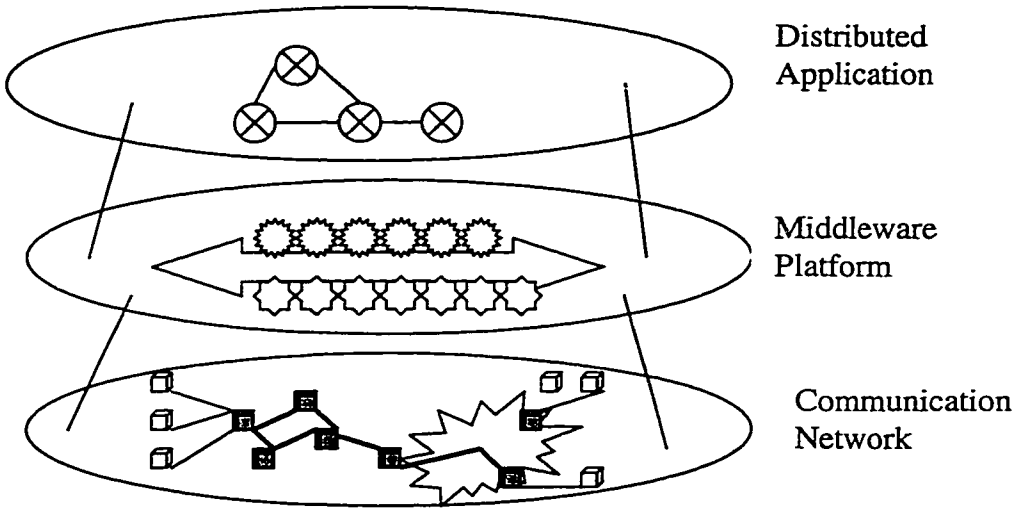


Figure 4.10 Distributed System Architecture

The basic idea for a generic test architecture is a toolbox of elements, which can be combined generically to construct a test architecture suitable for a specific application or system to be tested. The test architecture comprises several instances of different types of components [Walter, 1998]. Figure 4.11 shows the generic test architecture. The main components are:

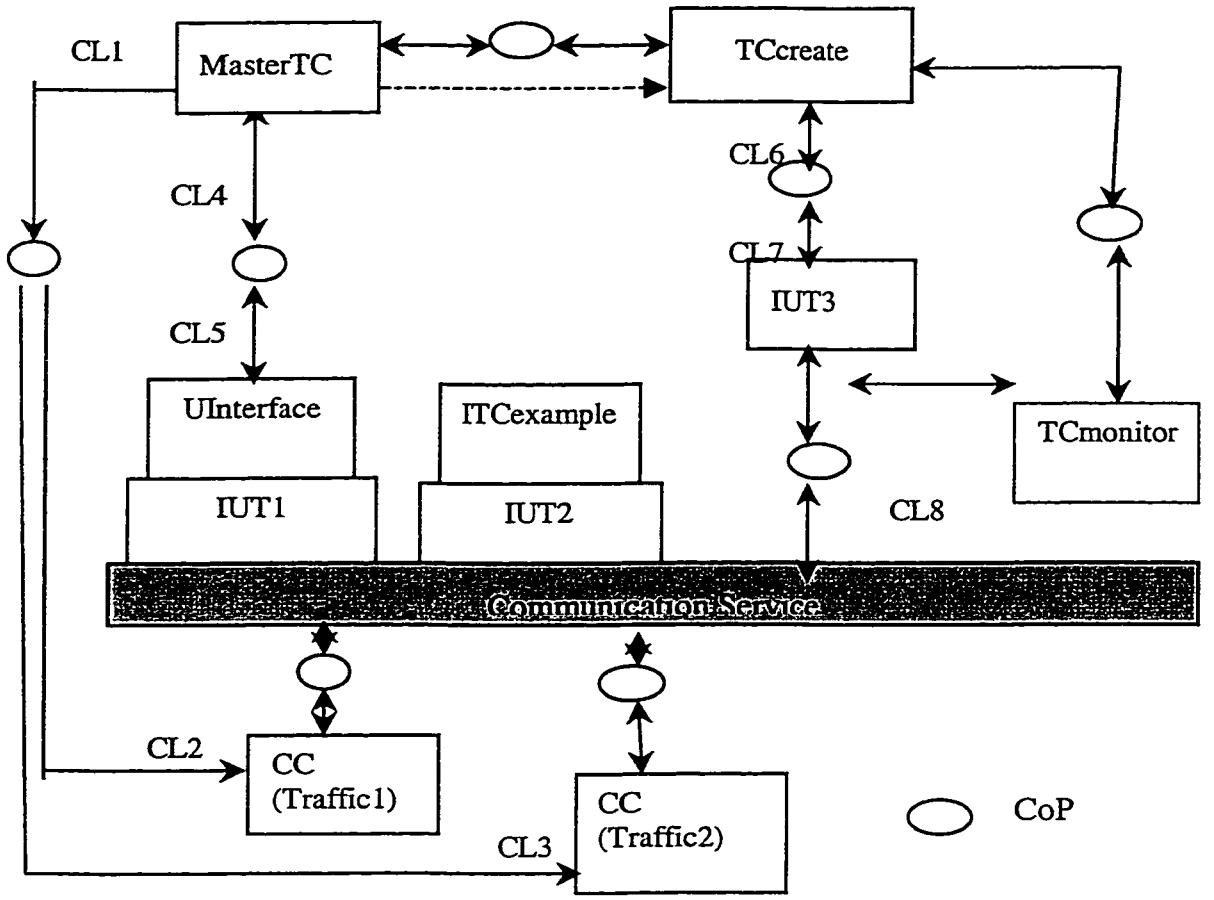


Figure 4.11 Generic Test Architecture

- Implementation Under Test.

  An IUT is a piece of software or hardware to be tested. The entire application to be tested may comprise several IUTs. The IUTs may have different functionality or be different instances of the same type.

- Test Component (TC).

  A TC is a component that drives the test. This can be done by creation of further TCs, by controlling other TCs, by contributing to the evaluation of a test run, and by controlling and observing IUTs.

There should be one Main Test Component (MTC) which starts and ends a test run. A start involves by creation and instantiation of further TCs or initiation of the first stimuli to the IUTs.

- Interface Component (IC)

  An IUT may be embedded in other applications or may be only interfaced via underlying services. The components are termed IC. An IC is not controlled by the test equipment, it is only used to interface with the IUT.

- Control Component (CC)

  A CC is used to set-up the test case specific environment or is used by TCs to control test execution. Examples of CC types are load generators for providing background load.

- Communication Point (CoP)

  A CoP in the generic test architecture corresponds to the PCOs in CTMF. It denotes a point in the test architecture where communications can be observed, controlled, and in addition to CTMF, be monitored. Several communication flows may be accessed at the same CoP.

- Communication Link (CL)

  IUTs, ICs, TCs and CCs are connected with CoPs by using CLs. A CL describes a possible communication and the kind of communication that may take place. Active communication can be classified by the kind, i.e., either synchronous or asynchronous, and the direction, i.e., either unidirectional or bi-directional. A passive CL allows monitoring communication.

- System Under Test (SUT)

  In CTMF, an IUT together with ICs is called system under test (SUT). In this generic architecture, SUT includes several IUTs.

4.4 Formal Description Techniques (FDT) for Test Automation

Testing activity has three phases as shown in Figure 4.2.

- *Executable test cases construction* – select test methods, derive test case requirements, assemble executable test cases.
- *Test Execution* – test environment setup, configuration, test case execution, test verdict, and test session control.
- *Test Result Report* – test result generation, test result analysis, and problem reporting.

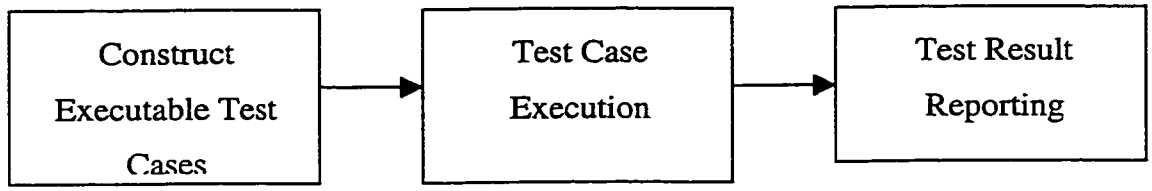


Figure 4.12 Phases of Testing

Test automation refers to the automation of test case construction (or derivation/generation) and execution. The test architecture we talked about in the previous section concerns the test execution environment. This section will focus on formal test case construction methods. Although it's not the purpose of the thesis, it will be an important part of efficient testing of e-commerce system and, it will be integrated in the test architecture in future work. There are several formal test description techniques.

4.4.1 UML

UML [Fowler, 1999] is a standardized, graphical notation used in the object-oriented analysis (OOA) and the object-oriented design (OOD) of systems. It is used for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. UML defines a number of graphical diagrams that provide different perspectives of the system under

development. In these, the characteristics of objects and classes of a system are described. Relations between objects in UML are demonstrated in the three abstractions: inheritance, aggregation and association. The processes between the objects as well as between user and system are described in UML by "use cases", or application cases. These use cases describe typical and important scenarios for the use of the system. State charts form another representation that aims to represent a behavioral view of a system.

UML serves only as the abstract description of a system - it must be complemented and implemented using other languages. These could be traditional low-level languages, such as C/C++ or Java, or a high-level language like SDL for event-driven systems. The advantage with UML is that it gives the system architect the possibility of modeling the entire system. In addition, powerful commercial tools have been developed, for example, Rational Rose for Real-Time, to generate C++ and Java code. When designing and specially for testing a system, other languages such as SDL, MSC, TTCN and others are more appropriate. A diagram illustrating MSC, SDL, and TTCN is shown in Figure 4.13.

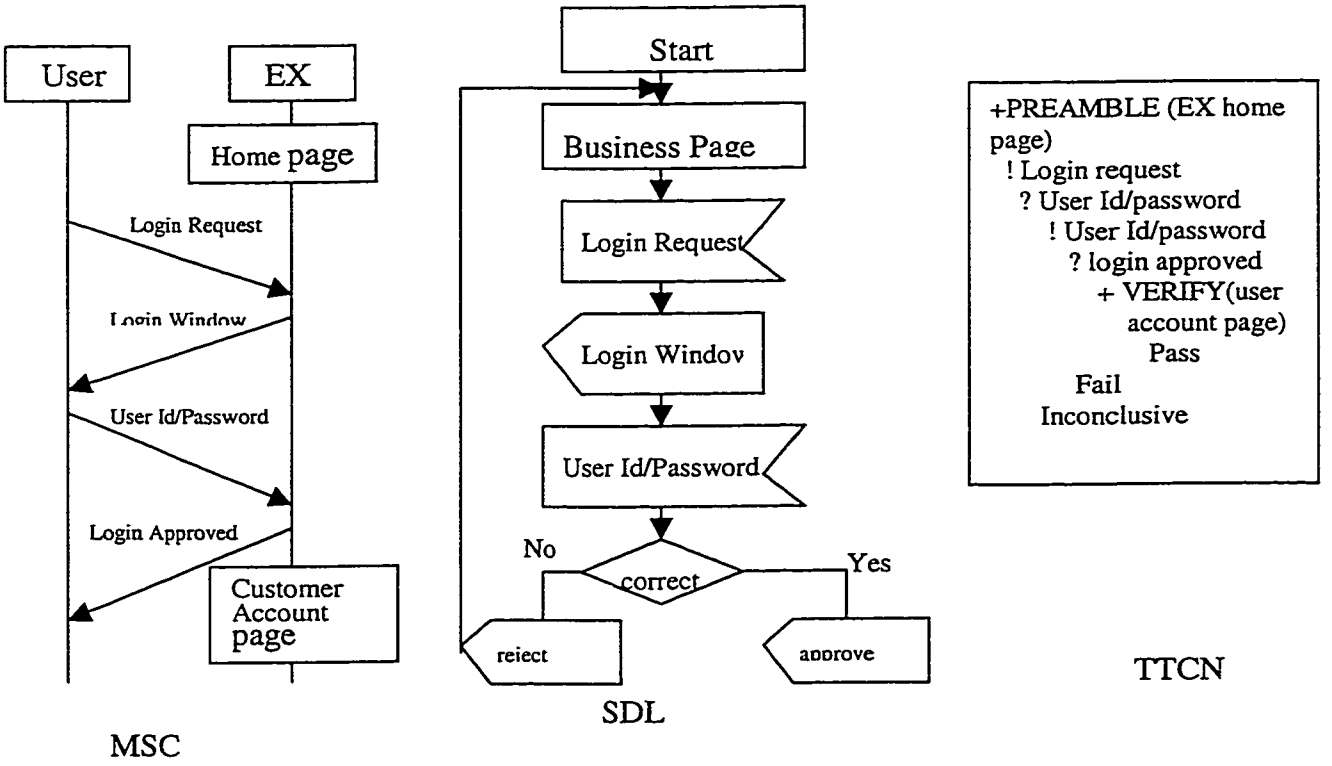


Figure 4.13 MSC, SDL, TTCN

4.4.2 MSC

Message Sequence Charts (MSC) [ITU, 1996] is a trace language for specification and description of the communication behavior of systems. A Message Sequence Chart shows chronological sequences of messages sent between system components and their environment.

Describing system interactions as in MSC is logical and intuitive, and variants of MSCs have been used in the development of communicating systems for decades. A first standard version of MSC was published by ITU-T as recommendation Z.120 in 1992.

MSC is a very powerful way of describing how messages are sent, both externally to and from a system, and internally between different system components. MSC is useful for describing the dynamic behavior of a system. Its graphical representation is well suited for presenting a complex behavior in a clear and unambiguous way that is easy to understand. The MSC language offers a powerful complement to SDL in describing the dynamic behavior of a system.

MSC can be used to produce documents that define the requirements on a system in the shape of scenarios from use cases. It facilitates the design phase, by identifying and documenting a multitude of dynamic cases. MSC also constitutes a convenient way to define test purposes. MSC also allow comparision with UML, since MSCs correspond directly to Sequence Diagram in UML.

4.4.3 SDL

SDL (Specification and Description Language) [SDL, 1999] is a modern, high-level programming language. It is object-oriented, formal, and graphical. SDL is intended for the description of complex, event-driven, real-time communicating systems. SDL is standardized by ITU-T, as standard Z.100.

Systems described in SDL consist of many processes running simultaneously which communicate with each other via signals. Each process is described by an *extended finite state machine*. The state machines are labeled extended since variables and timers can also be defined in processes.

SDL has a rich grammar that describes behavior and is unambiguous. Therefore, it is possible to build tools for the simulation of SDL systems and for the validation of formal characteristics, like deadlock avoidance. In short, this means that errors are detected at a very early stage in software life cycle.

SDL is graphical, and even non-technicians easily understand its diagrams. This translates into greatly improved communication between system designer and client, and ensures that the process from requirement capture to implementation is reliable.

The precision and formality of SDL provide the possibility for tool-supported code compilation into lower level languages, such as C/C++. Such commercial, industrial-strength tools have been developed including TAU by Telelogic and ObjectGODE by Verilog [Telelogic, 2000] This means that the SDL system can be translated into an executable application without manual coding, leading to shortened development time and increased quality. As a side effect, due to the readability of SDL diagrams, the SDL specification becomes the documentation in itself, ensuring simplified maintenance and post-development.

SDL is used worldwide for the development of all kinds of complex, communicating systems. In the telecommunications field and in internet systems, SDL is the language of choice for the development of a broad range of software and hardware.

4.4.4 TTCN

TTCN (Tree and Tabular Combined Notation) [ISO/IEC, 1995] is a formal notation for the specification of test cases.

TTCN is based on the concept of "conformance testing" discussed earlier. TTCN as a notation is designed for expressing all attributes of an abstract test suite exactly as specified in 9646-2 [CTMF2, 1993]. Thus both syntax and semantics of TTCN are tightly coupled to the draft international standards for conformance testing, and in fact, provide a concrete means of realizing those standards, sharable among different vendors of products.

TTCN was not intended to be directly executable. Rather, TTCN is intended to facilitate the

precise specification of an abstract standard conformance test suite in a manner which assures the development of a corresponding executable test suite, and which enables an audit that the executable test suite is a faithful implementation of the abstract standard conformance test suite.

As the name suggests, TTCN consists of a blend of two types of notations: a tree notation, which is used in the dynamic behavior descriptions to describe events which can occur as alternative responses to a previous event; and a tabular component, which is used to simplify the representation of all static elements, such as data types, PDU and ASP formats, verdicts associated with particular test events, and so on. A new optional format, TTCN-2000 will emulate a programming model.

TTCN is not only used in standardization work. The language is very suitable for all kinds of functional testing for real-time and communicating systems. This has led to a wide usage for all testing of communicating systems or protocols.

Given the world-wide move to using distributed platforms such CORBA to implement open distributed applications like telecommunication services, and e-commerce, the interface of each component in the system is defined in IDL. IDL is a formal interface definition language, it defines the component interface (services), it doesn't describe dynamic behavior of the component, e.g. the timeline representation of events or various executions of processing the same service request. On the other hand, TTCN provides a dynamic description of component behavior through its tree notation. So if we map IDL data types to some TTCN static elements [Mednonogov, 2000], such as one IDL *interface* maps to one TTCN PCO (Point of Control and Observation) declaration, one IDL *operation* maps to a pair of ASPs (Abstract Service Primitives), Call ASP and Reply ASP and so on, we could use TTCN to describe test cases which are consistent with the defined IDL interface and facilitate the automated test case generation from the IDL file. Thus, it is necessary to develop tool support for conversion between IDL and TTCN.

Since our research is focused on test execution phase rather than test case generation phase, and realizing a IDL/TTCM mapping gateway is a fair amount of work which is not practical for one person who also needs to implement the whole case study system, we didn't describe the test

case of our case study in TTCN and didn't implement the conversion between IDL and TTCN. Instead, we recommend it as very important follow-on work from this thesis.

In this chapter, we covered many important test issues for software system. In the next chapter, we will discuss various test requirements for e-commerce systems.

Chapter 5

E-Commerce Testing

It is widely known that testing are of the most important and the most expensive phase in the life-cycle of software development. This is especially true for e-commerce application due to well-known potential problems. Failures may have a catastrophic effect on the business reputation, financial health of both the business and the customer. This chapter will discuss the test issues for e-commerce products, outline the requirements for a good test architecture and review some related research in this area. As well, we relate the best existing test techniques described in chapter 4 to the context of into e-commerce testing.

5.1 E-commerce Test Issues

This section presents some of the types of testing e-commerce software systems.

First, What is to be tested? Like any software system, e-commerce software must be tested against its requirements specification, requirements include both functional and non-functional requirements. Non-functional requirements are especially important, because of its distributed and real-time (on-line) nature.

5.1.1 Basic (Functional) Transaction Testing

To be testable, the functional software specification must be decomposable into functions or business transactions involving multiple scenarios. Ideally, these scenarios are described formally or semiformally using use cases or Message Sequence Charts-like (MSC) representations as shown in Figure 2.12. The aim here is to test each transaction separately without interference or concurrency side effects from other transactions. Also, some of these tests can be performed locally, i.e., consumer and client can be co-located on the same machine

for testing. As well, product, system, and acceptance tests must be run in a distributed manner.

5.1.2 Robustness and Fault Tolerance Testing

For each of the transactions, we should be able to develop additional, exceptional scenarios that may or may not be explicitly included in the functional specification. Specially, we should aim at scenarios dealing with abnormal behaviors of the operating environment. Avoiding testing these aspects often leads to intolerant software, and hence, failures with severe financial consequences may result. It is important to check the error recoverability(fault tolerance) aspect of the system since it affects its reliability. Ideally, secondary scenarios will appear in the specification, and can be used to design the corresponding test cases.

5.1.3 Load and Stress Testing

The scenario shown in Figure 2.12, involves a single transaction. However, in real-life, some agents may have to deal with a substantial number of simultaneous or concurrent requests. Obviously, the business, financial institution and payment server agents must be able to deal with a heavy load of simultaneous transactions. Test cases must be designed to check how the software behaves under such operating conditions (i.e., stress testing). Ideally, in the system specification, an upper limit on the expected number of simultaneous transactions will be explicitly mentioned.

5.1.4 Quality of Service (QoS) Testing

Test cases must be designed to deal with QoS issues and parameters specified in the system specification. For example, multimedia objects' transfer delays are of importance in the interactions between consumer and business agents when browsing and downloading catalogue products.

5.1.5 Performance Testing

Test cases for the verification of some predetermined performance indicators for e-commerce systems must be developed. These indicators are obtained from the real-time system

requirements and may include various types of transfer delay and end-to-end response delay measurements.

5.1.6 Hard Real-Timeliness Testing

Hard real time requirements may be included in the specification (i.e., ideally in the use cases themselves). These requirements must be met in the system's implementation. Test cases dealing with this aspect must be designed. Moreover, these requirements have to be checked along with stress and robustness testing. For example, the real time requirements may be satisfied for a single transaction but not when considering a heavy load or when dealing with an abnormal condition, such as a failure of a component. Timeliness is important, for example, the stock purchase price is changing constantly.

5.1.7 Security / Integrity Testing

Test cases must be designed to ensure that financially critical message exchanges, like credit card numbers, passwords etc, are highly secure. For example, as shown in Figure 2.12, the messages carrying consumer bank account information and bank authorizations must be completely secured. Any attempt to modify such messages must be detected and blocked. The setup for these types of test cases must be carefully designed, for example network intrusion, data sniffing, etc.

5.1.8 Data /Database Integrity Testing

Another aspect of testing, often omitted in traditional software testing (although it is a big source of software errors), is to ensure the integrity of data flows and data in permanent data stores (i.e., databases). Test cases must be designed to check the validity of the data store before and after the execution of a transaction. These test cases must be performed under different operating contexts (i.e., under stress and abnormal conditions). For example, a transaction that was not approved by the financial institution must leave certain databases intact, except for the log file. All designed test cases must specify which data stores are affected and how.

5.1.9 Interoperability /Platform Independence Testing

All the above aspects of testing must be repeated for all supported operating environments and platforms. These environments may include operating systems, database systems and other system components. This requirement is vital for the success of e-commerce systems, since networks consist of many heterogeneous components. Applications must be compatible with a vast variety of systems to maximize market reach.

5.2 Requirements of E-commerce Test Architectures

From the last section, we can see that e-commerce testing is a complex activity, involving all three layers of the distributed system architecture as shown in Figure 4.10. Failure to address test issues of any layer will compromise the quality of the whole e-commerce system.

Different layers impose different requirements on the testing architecture. Although we list different requirements for different layers, for the purpose of this thesis, our work is focused on testing at the distributed application layer.

5.2.1 Testing of Communication Networks

The ITU/IETF activities defining communication protocols, services and mechanisms in internetworking are the driving forces in defining advanced network technologies. The following protocol-related aspects of the Internet impose particular requirements on testing network nodes as well as end-to-end services:

- a variety of group communication scenarios such as multicast with dynamic join and leave in multicast groups,
- stream support with different levels of QoS guarantees and soft-state resource reservations, like bandwidth reservation,
- a variety of routing protocols including multilayer routing approaches.

5.2.2 Testing of Middlware Platforms

Due to the need for interoperable implementations by various vendors and due to the complex nature of middleware platforms such as CORBA, there is a need for a methodology that can assess middleware platforms with respect to their compliance to the respective specifications. This is done in order to increase the likelihood that implementations can interoperate.

In essence, a middleware platform is used by a distributed application like a black box with various service access points. However, in order to test for example a CORBA ORB (this includes testing the ORB Core, the Interoperability Reference Points, CORBA Services, and CORBA Facilities), a grey-box testing approach has to be taken. In this approach, test access to ORB internal interfaces is supported [Rao, 1997].

5.2.3 Testing of Distributed Application

With the development of e-commerce systems based on middleware platforms such as CORBA, and the provision of new and complex services such as telecommunication, management and information services which may be deployed in a various distributed object systems, it is becoming increasingly important that we test the applications against the testing issues outlined in section 5.1, including to check

- Service components of the applications individually,
- Individual service components working together in a multi-service environment.

5.2.4 Requirements for Testing E-commerce Systems

As a result of the above decomposition, the following requirements for testing e-commerce systems can be identified:

- Development of distributed testing architectures with a mean for synchronizing distributed test components (see Chapter 4).
- Support for dynamically configurable and scalable test architectures;

- Ability to express test configurations for different communication scenarios;
- Support for grey-box testing with access to internal components and interfaces;
- Support of real-time, performance and QoS testing for distributed systems to test time-related aspects of distributed systems;
- Methods that support testing in the pre-deployment(release), deployment and operational phases of distributed systems.

The requirements for testing e-commerce systems are not met by CTMF[Walter, 1998]. The degree of distribution increases the complexity of testing and limits applicable test architectures. New architectures must be developed to meet these requirements.

5.3 Test strategy for e-commerce

In section 4.1, we reviewed three general software test strategies – black-box testing, white-box testing and grey-box testing. Obviously, these traditional software test techniques are very useful for building a test strategy for e-commerce. Every test strategy can be applied to testing different aspects of e-commerce system. E.g., black-box testing for functional test and performance test, grey-box testing for interoperability test and security test, and white-box testing for security test.

Our research intends to focus on black-box testing. White-box testing requires to access the system code, thus it is not addressed in this thesis. For example, we do not consider instrumentation issues related to the use of CORBA in SUT and test architecture.

5.4 Related Work

E-commerce is in its early stages of development. Testing e-commerce is even newer. Thanks to the effort devoted by pioneers of e-commerce testing, some important ideas have emerged recently to make the e-commerce application development cost effective and minimize overall time to market. These are summarized in this section.

First, in [Probert et al, 1999], we proposed a generic, cost-effective approach to optimized e-

commerce requirements capture with respect to functional coverage and customer perception of quality. The high yield techniques (techniques which will result in a "harvest of bugs") are appropriate and straightforward to be applied in the requirements capture, requirements elicitation and validation phase, as well as in the design construction and verification phase. Details of high-yield requirements capture approach will be described in chapter 6.

Given the wealth of research, tools and standards developed for protocol and distributed systems testing, [Saleh, 1998] proposed a preliminary of e-commerce test framework. The main components of the framework are:

- Guidelines on how to apply the most suitable high-yield protocol and software test activities in the context of EC software;
- Use of a test specification language to describe business transaction oriented test cases;
- Guidelines on how to describe and test real-time and quality of service requirements for e-commerce systems;
- A test architecture supporting multi-party distributed testing with high testability, observability and controllability.

Besides research into specifically e-commerce testing, we should also consider advances and results in distributed system testing research. For example, the generic test architecture for distributed systems proposed in [Walter, 1998] provides a unified approach for flexible and adaptable test architectures. The architecture is a generic toolbox of components which can be adapted to target different domains or specific applications. Details of this generic test architecture for distributed systems were presented in chapter 4.

With the emergence of new models, architectures and middleware such as CORBA for developing open distributed systems, test technology requires adaptation for functional assessment. CORBA is object-oriented and aimed at creating open distributed environments supporting internetworking, interoperability, and portability, in spite of heterogeneity and autonomy of the component systems. In this context an open distributed system may be viewed

as a system providing standardized distributed interfaces for interacting with other systems. Attaching a tester to each available interface allows testing of such systems. However problems with controllability and observability influencing fault detection during the testing process may arise if there is no coordination between the testers. [Rafiq, 1999] shows how to cope with these problems by using CTMF distributed test method and use CORBA as the test infrastructure.

[Rafiq, 1999] is not the only one to use CORBA as a distributed test platform, [Lima, 1997] aims at implementing and validating telecommunication services on top of a TINA-CORBA like platform. A simple system with one test component and one component under test is presented in [Lima, 1997].

[Vassiliou-Gioles, 1999] also presents generic tools for testing the functionality, performance, robustness and scalability of distributed systems, particular for distributed telecommunication applications, based on CORBA.

This chapter discussed some important requirements for testing of e-commerce applications, outlined some requirements for a generic e-commerce test architecture and sketched CORBA's value to e-commerce application development and testing. In the next chapter, we propose a CORBA-based e-commerce test architecture to satisfy the requirements given in this chapter.

Chapter 6

CORBA Based E-Commerce Test Architecture

While all steps in [CTMF2, 1993] correspond to what is called the Test Specification and Construction Phases, which start with the system specification and end with the generation of test suites, very little is said about test execution, i.e. how to transfer the abstract test architecture to an executable test architecture. A suitable distributed platform based architecture to test e-commerce application is needed, which can realize the elements of the generic abstract architecture to real world software development.

This chapter proposes a CORBA-based E-Commerce Test Architecture to the Generic Test Architecture Model discussed in sections 4.3 and 3.6. In so doing, we will demonstrate that this test architecture is able to satisfy the broad test requirements for e-commerce systems outlined in section 5.2.

6.1 High-yield Requirement Capture of E-commerce System

6.1.1 Introduction

As the requirement for more imaginative and successful new e-commerce systems and products arise, it is necessary to find a cost-effective, timely means to capture requirements, validate these requirements, and develop these products according to an accelerated development schedule. In fact, "time to market" is an all-important metric for successful e-commerce projects. To accelerate development, it is often necessary to assign priorities to the myriad requirements that are received from clients, market studies, and planners. Products are released in phases according to the priorities assigned. In addition, many requirements are inappropriate or incorrect with respect to real markets and real market requirements. Important requirements may be

missing, and if their absence is not discovered, this will lead to design omissions widely known to be the most serious source of costly re-work. Requirements omitted at the beginning of the development process are much more costly to find and integrate later, often after delivery has been made. Finally, in the e-commerce environment, it is extremely important to have a timely product offering which achieves the most important capabilities and arrives early in the market place (short Time to Market). However, because of widespread access by early adapters, an early offering failure is much more serious in e-commerce than in other business product domains.

Our strategy is to define key scenarios based on a customer orientation that will cost-effectively detect errors in design and development. The cost effectiveness comes by selecting types of scenarios, called **high-yield scenarios**, which are more likely to cause design errors to be manifested in design simulations or inspections and walk through. Very specifically, by **yield**, we mean the number of dangerous errors or high-risk errors that are detected by walking through, simulating or executing this scenario. By **risk** we mean the likelihood of an occurrence of the error multiplied by the cost of the consequence of the error occurring. For example, an error of very low associated cost consequence but which occurs frequently enough to annoy a customer may be considered a moderate to high-risk error. Similarly, an error which occurs very infrequently but whose associated cost is enormous (e.g., loss of life) may also be classified as moderate to high risk. Scenarios are classified according to five basic types described below [Probert et al, 1999]. The high-yield strategy is based on selecting use cases covering individual customer and business's requirement. First the normal expected behaviors are listed as scenarios, then they are systematically specialized to unexpected, risk (high-yield) scenarios. Our approach is based on the identification of high yield, moderate yield and low yield scenarios taking into consideration our classification guidelines.

6.1.2 Anatomy of Use Cases and Classification of Scenarios

A *use case* usually describes one important or core functional requirement of the software system. Each use case must have a clear purpose or goal to achieve. For example, in an electronic commerce system, core use cases would be log-in, placing an order and inquiring about an order. From a use

case, we can obtain various instances of execution traces that represent distinct possible paths through the use case. We will refer to these traces as *scenarios*. A *use case* is constructed with various scenarios. Scenarios can be divided into two categories: *primary* scenarios and *secondary* scenarios. We can also categorize scenarios according to whether or not the scenario achieves the purpose of the use case at the end of its execution. These two categorizations are not disjoint. A good reference for use cases, and *primary* and *secondary* scenarios is [Schneider, 1998].

A *primary scenario* consists of interactions that do not include any error occurrence or abnormal situations such as communication failures, storage failures, and bad input data. These scenarios are also called optimistic or "happy" scenarios of the use case (class 1). However, primary scenarios may include scenarios to describe all possible interactions that end with the user canceling or abandoning the progress of interactions, therefore, not achieving the original purpose of the use case (class 2). Class 1 and 2 scenarios are referred to as *low yield* scenarios. This designation is applied because these scenarios are usually extremely well understood by all stakeholders in the system design and development of the process, particularly customer representatives, owners, developers, designers, and testers. As a result, errors of serious misunderstandings are not at all likely to be found by such scenarios. In terminology pioneered by Glenford Myers [Myers, 1979], such scenarios are denoted "low yield". For example, in our case study Online Currency Exchange system, one of the requirement-directed use cases we identified is Online Exchange. A primary scenario of this use case (see scenario no. 1.1.1 in Table 7.1 of Section 7.2.3.1) is that the customer fills the order form with all correct information, submits it, the order is successfully processed, customer gets confirmation. In this scenario, everything works fine. This is the so-called happy scenario which is referred as low-yield because it is so well understood that no errors in design or implementation are expected.

A *secondary scenario* includes interactions involving erroneous and abnormal situations, such as Internet transmission error, user's wrong or omission input, system error, and etc. These scenarios normally involve branching out from *primary* scenarios. They may include interactions to handle the errors (i.e., fixing the bad input data) and eventually achieving the use case purpose (class 3). However, some of these scenarios may include interactions to cancel the progress explicitly or implicitly (i.e. user simply abandons the transaction) after the user or

system fails to handle the exceptional error condition (class 4). For example, for Online Exchange use case, a secondary scenario (see scenario no.2.2.2 in Table 7.1 of Section 7.2.3.1) is that customer's computer fails after customer submits the order. In this case, customer won't be informed whether the order is processed or not, and the customer will be confused with what to do next. This scenario involves an abnormal or high-risk situation which is referred as high-yield because it is no usually well understood or well documented, and therefore error-prone for designs.

In addition to scenarios of classes 3 and 4, there is another class of scenarios, namely concurrent scenarios which are also high-risk and high-yield scenarios. These scenarios describe race conditions which occur frequently in heavily utilized e-commerce environments. While somewhat difficult to design and construct, concurrent scenarios describing race conditions are extremely effective at detecting the presence of high-risk design errors. For this reason, we call these scenarios high-yield scenarios as well (class 5). Classes 3, 4 and 5 scenarios are referred to as high yield scenarios.

Table 6.1 Scenario Classifications

	Primary Scenarios - Low Yield Scenarios		Secondary Scenarios - High Yield Scenarios	
SINGLE	**Class 1** Scenarios where everything goes without any error toward achieving the UC purpose	**Class 2** Scenarios where everything goes without any error but the user selects to CANCEL before achieving the UC purpose	**Class 3** Scenarios where errors occur but corrected by the user and finally achieving the UC purpose	**Class 4** Scenarios where errors occur but user was unable to correct them or selects to CANCEL before achieving the UC purpose
CONCURRENT	**Concurrent Scenarios - High Yield Scenarios (Class 5)**			
	5.1 Multiple application of scenarios of type 1 with no conflicts and achieving UC purpose	**5.2** Multiple application of scenarios of type 2 with Canceling before achieving UC purpose	**5.2** Multiple application of scenarios of type 3 with conflicts occurring and corrected	**5.4** Multiple application of scenarios of type 4 with conflicts occurring but not corrected
	USE CASE PURPOSE ACHIEVED	USE CASE PURPOSE NOT ACHIEVED	USE CASE PURPOSE ACHIEVED	USE CASE PURPOSE NOT ACHIEVED

Table 6.1 below summarizes this categorization scheme by showing the five classes of scenarios described above. This classification of use case scenarios is a useful and generic framework with which we can obtain meaningful and representative scenarios covering all aspects of the use case interactions and the different associated risk levels [Probert et al 1999].

In the industrial practice, effectively applying high-yield strategy involves systematic activities through the whole process of system development. It also needs all the stakeholders participate in the process. The major stakeholders in e-commerce system are e-commerce software manufacturer, the business that purchases and deploys an Internet site using the e-commerce software, and the business's customers who shop at the Internet site. The e-commerce software manufacturer includes its personnel from marketing, customer interaction, system design and development, and testing. Using the high-yield strategy, stakeholders derive primary (low-yield) scenarios of the initial requirement, then walk through those scenarios, in every step of the scenarios, try to think about "what could go wrong?" and "Is there an alternative path?". Every stakeholder has his/her own perspective of interest and risk of e-commerce system. Bringing all the stakeholders into the requirement analysis sould significantly improve the risk coverage. Guided by the high-yield strategy, brainstorming, negotiating, customer trials, establishing customer operation profile and so on are common activities for stakeholders to reach a consensus of system requirements, and thus to obtain high quality of high-yield scenarios for requirement reviews and design inspections and analysis.

6.1.3 Definition of Key Testing Metrics

Since scenarios have been selected according to the likelihood of yield of high-risk errors, it is important to monitor and measure the degree of coverage of risk associated with scenarios which are selected for use cases and customer requirements from the very beginning of a project. The metrics are computed from measures which are made during the requirements analysis and high level design development process. Requirements are organized by functional area. Each functional area will be covered by scenarios. The first two measurements that are required are (a) the number of requirement areas by function and (b) the total number of scenarios.

The third primary measurement is somewhat more complex and requires reference to the classification scheme for scenarios described above. Each scenario is classified as class 1, class 2, class 3, class 4 or class 5. Scenarios of classes 1 and 2 (primary scenarios) are designated low-yield scenarios. The second category of scenarios, namely classes 3, 4 and 5 are moderate to high-risk scenarios, for many of these, the proper system reactions are unknown and certainly are not uniformly understood. Differences in interpretations of requirements, given these particular scenarios will lead to invalid design assumptions and design omissions. Thus, these types of scenarios are likely to detect the presence of such design errors. Therefore the third primary measurement is (c) the number of low-yield scenarios (classes 1 and 2). After these primary measurements have been computed, then the metrics which are listed below can be calculated and used to guide the requirements and design quality assessment process. Table 6.2 gives the formulas for calculating the coverage assessment and risk management metrics.

Table 6.2 Key Metrics Formula

Name of Metric	Formula
number of requirements for functional areas	Primary measurement (a)
total number of scenarios	Primary measurement (b)
number of low-yield scenarios	Primary measurement (c)
number of high-yield scenarios	$d = b - c$
average basic coverage	$e = b / a$
average risk coverage	$f = d / a$
degree of risk orientation	$g = d / b$
scenario risk ratio	$h = d / c$

Because secondary scenarios (class 3, 4, and 5) are more likely to detect system errors, we recommend deriving relatively large number of *high-yield* use case scenarios in scenario

selection, and keep the minimum ratio of *high-yield* scenarios to *low-yield* scenarios greater than one. In this case, more high-yield scenarios are considered during requirement capture, the likelihood of detecting system errors increased considerable; in turn, it improves test coverage and cost-effectiveness. In practice, how to select the specific value of risk coverage in depends on specific software domain, e.g. the mission critical software such as the software process customer financial information should be expected to have very high risk coverage, the software only for customer to browse the business product catalogue could have lower risk coverage.

This black-box technique is cost-effective, measurable, risk-directed and UML compatible [Probert el al, 1999]. The metrics can be adapted, improved and appropriate target values and ranges can be calibrated according to the domain and development environment. In chapter 7 (Case Study), we provide a complete table of use cases based on this high yield requirement capture approach and show how to compute and analyze the corresponding coverage assessment metrics. We present those use cases in UML format as in [Schneider, 1998].

6.2 Abstract E-Commerce Test Architecture

Figure 6.1 shows the customized generic abstract test architecture [Walter, 1998] for a generic e-commerce application. The Test Component (TC) drives the test run by creating TCs, controlling other TCs, controlling and observing the e-commerce component under test (CUT), or coordinating other TCs to contribute to the test verdict. There should be one TC acting as Master Test Component (MTC) which starts and ends a test run. A test case execution is started by creating Test Components (TCs) and by simulating the first service request to CUTs. MTC should be able to indicate when the test run is finished and assign a final verdict. In the case of reporting evaluation statistics from multiple test runs, the MTC should be able to decide whether the test run verdict contributes to the statistics, since not every test case could lead to pass or fail, some are inconclusive. Background Test Components (BTC) are also created by MTC. BTCs generate continuous load for the CUT, and TC monitors the real performance of CUT. BTCs do not control or observe CUT directly, but only implicitly influence it by putting it into normal or overload operating conditions. An examples BTC is a load generator, they are specially required in performance testing.

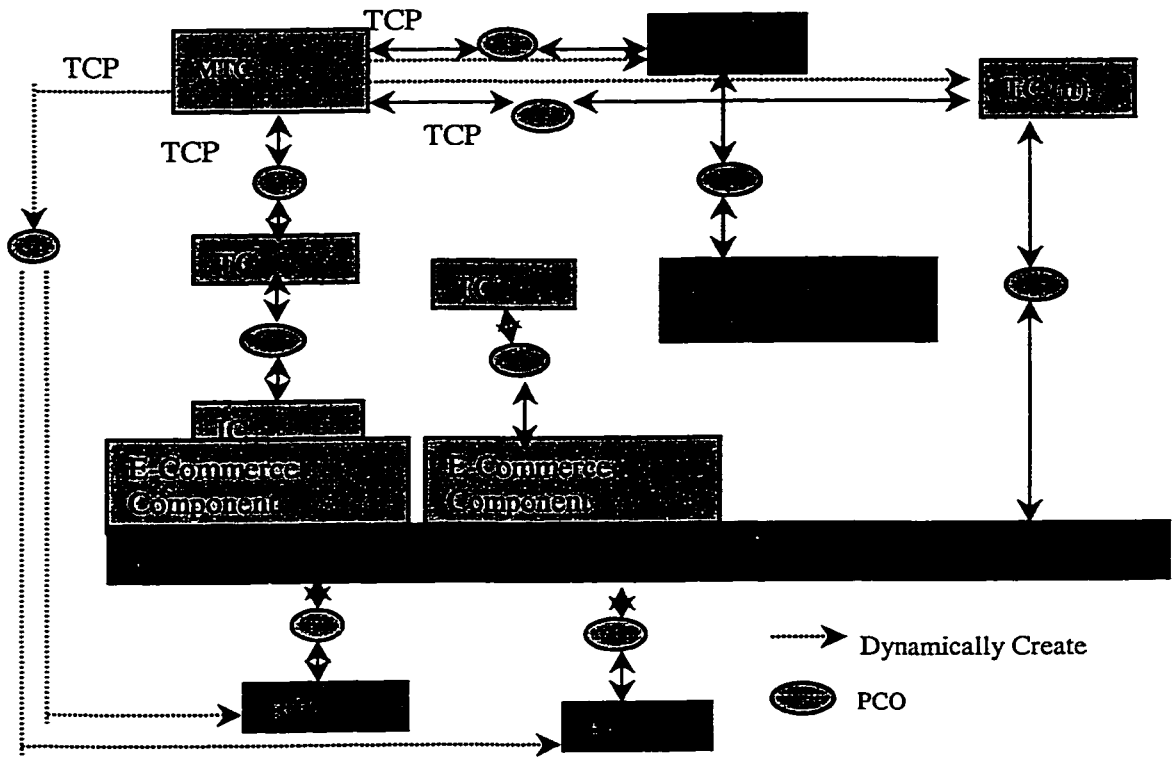


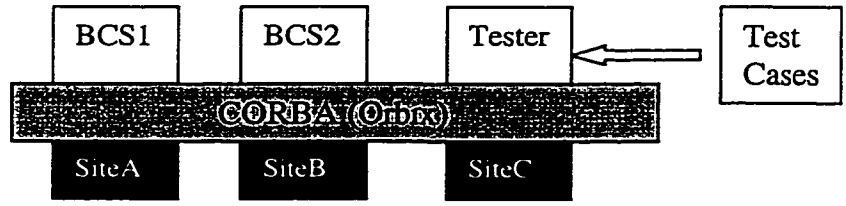
Figure 6.1. Abstract Test Architecture of E-commerce

In addition to TCs and MTC, a Point of Control and Observation (PCO) denotes a point in the test architecture where communication events or messages can be observed and controlled. The Interface component (IC) is needed for accessing CUT, e.g. an underlying service or an application above in which CUT is embedded. One extra PCO is needed between IC and CUT during performance testing since the presence of the IC could make the performance data observed by PCO less accurate than the component's real operational performance. The Test Coordination Procedure (TCP) is a component that defines the rules for the cooperation and

coordination of TCs and the MTC. This enables synchronization of test events and message handling.

6.3 CORBA-based Test Architecture

There is a preliminary step in [Lima, 1997] that tries to exploit advantages of CORBA in testing telecommunication services. A centralized tester is deployed for a system that consists of two components each providing Basic Call Service as illustrated in Figure 6.2.



BCS = telephone Basic Call Services

Figure 6.2 CORBA in Testing Telecommunication Service

From previous analyses, an e-commerce system is the collaboration of multiple distributed components (agents or objects) to complete a transaction required by a customer. To test an e-commerce application, test components need to be deployed in a distributed way and able to interoperate with each other despite of the heterogeneous environment. CORBA is just the architectural middleware to accomplish this.

Today almost all software systems are still somewhat dependent on the hardware and operating system upon which they were built. In order to migrate such systems to open distributed environments, a middleware was invented. It is located between the software application layer and the hardware layer (i.e. physical and low level programs like operating systems), assures transparency, interoperability and cooperation among all service components in spite of heterogeneous underlying systems. CORBA is a prime example and our preferred choice of such a middleware layer. CORBA here is responsible for providing API facilities to programmers, so that they do not need to cope with minute communication details.

However very little has been said about validation of distributed services, specifically, regarding distributed environments like ORB platforms. Thus, on the one hand we see a world-wide move to standardized distributed platforms to implement distributed services and on the other hand, very little effort to address the growing need for suitable methods to validate these systems.

Our work in this area focuses on implementing and validating services on top of a CORBA platform. In the following we will provide a short presentation of the implementation and validation phases that we have developed in our test process for e-commerce.

For the purpose of service validation, a tester or test driver (possibly distributed) must be designed and implemented to send stimuli to several components under test (which comprise the System Under Test - SUT) and to gather information concerning their reactions to these stimuli. Eventually, the tester will give a verdict on the observed behavior of the system. In doing so, we will be able to come up with a practical approach and process dealing with e-commerce testing:

- test construction and execution;
- distributed test execution and report generation;
- how to structure the test architecture for CORBA-based system/objects

Figure 6.3 shows the overall test architecture. The distributed platform chosen was ORBacus [OOC, 1999], which is an implementation of OMG-CORBA standard.

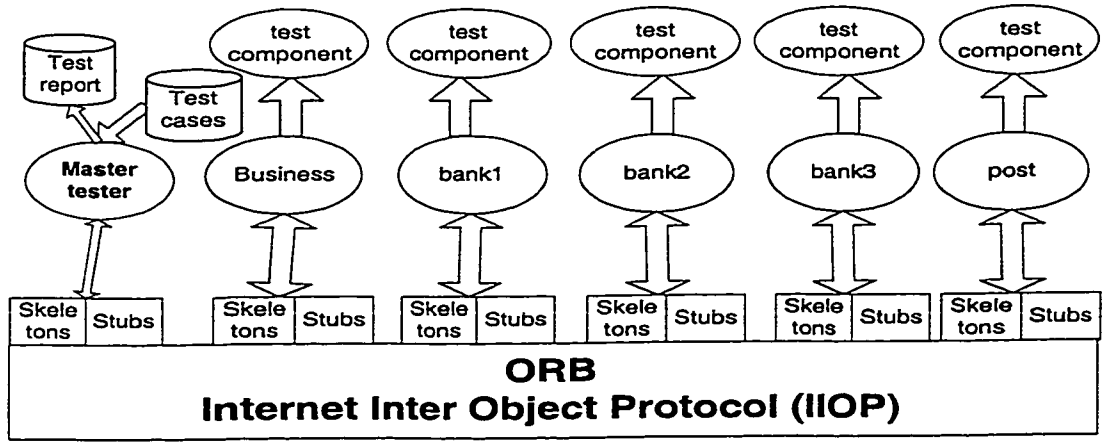


Figure 6.3 CORBA-based E-commerce Test Architecture

In this platform, all interactions (i.e. service requests) are performed by means of *channels*. Each channel is composed of a *stub* (on the client side) and a *skeleton* (on the server side) which are connected to the Object Request Broker (ORB). Channels allow a transparent communication between the client and object implementation. The ORB in turn provides the functionality required to communicate across (possibly heterogeneous) platforms.

A *service request* consists of the following information: a target object, an operation, a list of parameters and an optional request context. "One way" operations allow asynchronous message exchange amongst the various system components. It is important to point out that a client can simultaneously be a server to another object (or vice-versa). Objects must cooperate (or interoperate) in a useful way to achieve a given objective, which is exactly what CORBA models well.

Following the guidelines of classical test architectures for protocols as described in [CTMF2, 1993], we suggest that a test architecture for an open distributed object-oriented platform must have two sets of components: test components (TC) and components under test (CUT) that are possibly distributed over several sites. The TCs receive information about the object configuration (i.e. Figure 6.4 the collection of objects able to interact at their interfaces) and the test suite which is obtained through some test generation method. Simple systems (with local architecture) are usually implemented using one single TC and a CUT.

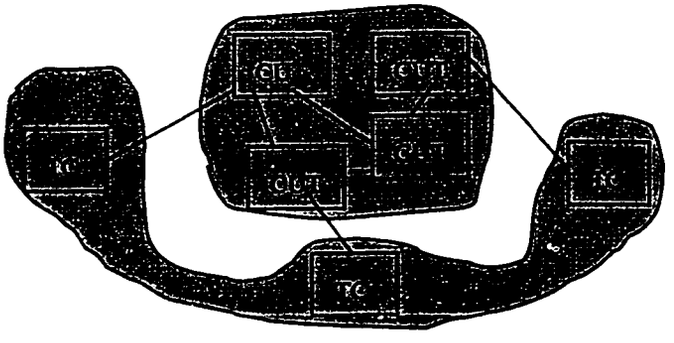


Figure 6.4. Components of the Test Architecture

The CORBA ORB serves as a channel which allows transparent communication between client and server component. It is the ideal place to deploy the PCO and TCs, which makes it very easy to observe and control the behavior of components, thus achieving high testability.

In Figure 6.3, the overall test architecture includes e-commerce CUT, TC and their interactions through ORB. Each TC deployed at PCO is resident with one CUT. The MTC may sit on a different site with CUTs, and it triggers test case execution according to the test case file. Those distributed TCs capture the behavior of the host CUT and automatically report to the MTC. The MTC analyzes the TC reports, and determines the test verdict. With the information from TCs, the MTC knows exactly what is happening at every CUT. If any problem arises, MTC can easily identify the location and time of the corresponding failure.

The communication protocol among different components is defined in IDL (Interface Definition Language). The *service request* consists of the following information:

- A target component; (e.g. business component)
- A service (operation or method) (e.g. login)
- A list of parameters; (e.g. customerID and password)
- An optional request context (e.g. ClientAccount)

The above information corresponds to typical parts of a test case for a CUT: service under test (test purpose), input and expected result, respectively. Thus CORBA IDL provides a generic test case format, suitable for representing the boundary of functional testing (black-box testing) [Probert, 1994] and a generic interface for any test component (MTC and TC).

As Figures 6.3 and 6.4 show, TCs and CUTs may be distributed over multiple sites. A tester can dynamically assign different TCs to different CUTs. Each TC observes the service request to and the responses from the CUT, then reports to MTC which derives a final verdict. Here is a sample IDL definition for a TC:

```
module test {
```

```
//test case format
  struct Step {
        string service;
        string input;
        string expectedResponse;
  };
  typedef sequence<Step> RequestList;
  struct TestCase {
        string desc;
        RequestList requestList;
  };
//test component function
  interface TestComponent {
  oneway void reportMessage ( in string sessionID,
              in string messageID,
              in string sender,
              in string receiver,
              in string timeStamp,
              in string serviceRequested,
              in string reponse);
  //for adding test component;
  void register (in TestComponent tester);
  //for extracting test component
```

```
    void unregister (in TestComponent tester);
    };
};
```

TCs interoperate through method *reportMessage*, all message exchanges are asynchronous ('oneway'). The timeStamp parameter is not only for performance testing, but also for functional testing since reasonable round trip response time of an order processing scenario is part of the functional requirement for e-commerce system.

CORBA based TCs are autonomous, self-managing and collaborative. As e-commerce systems grow, it will be easy to put together more complex testing systems by simply assembling different components. The IDL methods *register* and *unregister* are used for adding and extracting test components. Again, CORBA allows TCs and CUTs written in different languages to interoperate. These capabilities make CORBA-based testing architectures more flexible and generic than architectures based on other distributed computing platforms (see chapter 3).

Next we discuss the communication among MTC, TCs and CUT in detail for functional and performance testing.

6.4 Anatomy of CORBA-based Test Architecture

This architecture is generically defined, and can be adapted to achieve different test objectives. Here we only give two adaptations: functional testing and performance testing. In Case Study we only focus on functional testing.

6.4.1 Functional Testing

Figure 6.5 shows the CORBA-based test architecture for functional testing; it gives the detailed relationships between MTC, TCs and CUT in the test architecture. MTC is the controller of the model. It creates and collects information from TCs, queries *Test Verdict* to reach final verdict, drives *Test Report Manager* to record the test result and generate test reports. *Test Case Manager* reads the test case file and drives test case execution through the *Client Simulator* that simulates

service requests to CUT. *Test case file* contains use cases generated from the system analysis and design phase, which can be described in existing test case specification languages like MSC [ITU, 1996], and TTCN [ISO/IEC, 1995]. In the case study Online Currency Exchange system, *Test case file* is written in the format defined in TC's IDL. Use cases are obtained by using high yield requirements capturing approach described in section 6.1.

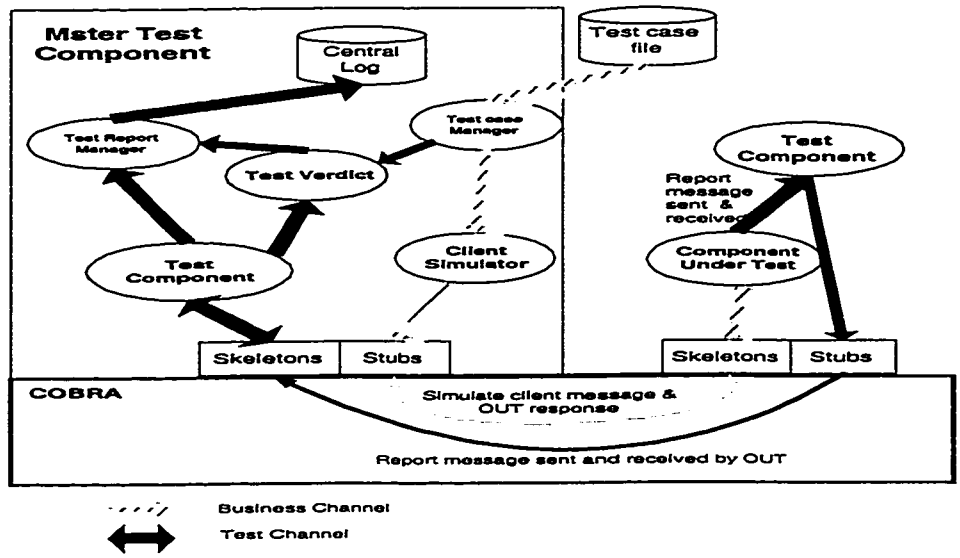


Figure 6.5 Functional Test Architecture

In this architecture, *Business Channel* (where business transaction data is communicated) and *Test Channel* (where test control and observation information is communicated) are separated. During the test execution, *Test Case Manager* drives *Client Simulator* sending requests, and supplies *Test Verdict* with the current test case. Service requests from *Client Simulator* to CUT, response from CUT and transactions among other CUTs are all confined to the *Business Channel*. Any information related to testing, such as input/output of CUT, test verdict and test report is transferred in the *Test Channel*. TCs report CUTs' behavior to MTC through method *reportMessage*, MTC forwards it to *Test Report Manager* and *Test Verdict*, *Test Verdict* checks all the messages against the expected test case messages to monitor the execution and reach the final verdict. The final verdict is passed to *Test Report Manager* to be included in the test report.

6.4.2 Performance Testing

Figure 6.6 is the CORBA-based architecture for performance testing. It is similar to functional testing, except the *Client Simulator* is replaced by *Background Tester*. *Background Tester* is driven by *Test Case Manager* to generate a continuous load, as if multiple *Client Simulators* were sending multiple service requests to CUT.

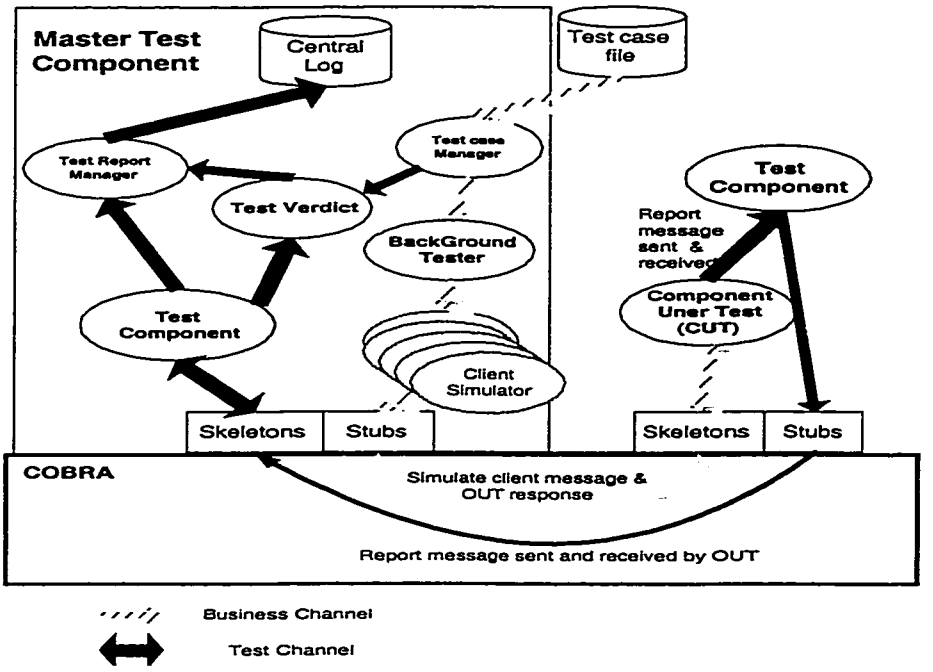


Figure 6.6 Performance Test Architecture

There are many important performance metrics for E-Commerce systems [Woodside, 1995]. For example, the system should have a reasonable round trip response time to a customer request under a reasonable workload. Here is a list of important performance metrics that we may use:

- Raw throughput between one client and one server.
- Time to complete an entire transaction
- Average response time per service request
- Maximum response time per service request

- Maximum simultaneous service requests allowed

If the TC is designed properly, it can collect the data needed during test execution to compute the above metrics.

6.5 Test Execution

Test execution can be divided into two phases:

- Test configuration set-up

  In this phase, the test architecture is initialized. First the MTC with a test case file is initialized. Then, all CUTs in the system are initialized and made ready to perform transactions upon customer request. Finally, TCs are created and allocated to CUTs. In the meantime, MTC registers the TCs being allocated with an ID that indicates the location of the CUT. In the Cast Study Online Currency Exchange system, a GUI interface is developed for tester to graphically conduct tests and review test reports (presented in the next chapter).

- Test Execution

  Now tester starts test case execution through MTC graphical user interface. MTC begins simulating clients sending signals to the various CUTs while collecting and checking their responses. This execution phase terminates with a verdict ('Pass' or 'Fail') for each test case. This is done in several steps:

  - Send service requests to the CUT and trigger a timer if a time constraint applies.
  - Collect information about CUT's behavior, i.e. responses to the service request, or issuing a 'Fail' or 'Inconclusive verdict if the timer expires before a response is received.
  - Check whether the response is correct (expected).

- Generate a report about each step in the execution and final verdict.

This chapter gave a detailed description and analysis of the CORBA based E-Commerce Test Architecture and test execution process. It also demonstrated this architecture is open, distributed and scalable to fit different test purposes, for example functional testing or performance testing. In doing so, we showed that the test architecture is able to satisfy the broad test requirements of e-commerce systems outlined in chapter 5. In the next chapter, we will go through a case study to apply the principles discussed in the chapter to a realistic e-commerce system.

Chapter 7

Assessment of the Approach by Case Study

In this chapter we present an example e-commerce system for Online Currency Exchange (EX) and show how our test architecture can be applied. Our goal is to implement and verify the service provided by the EX system on top of a CORBA platform. This system is an example of Broker Business Model discussed in Chapter 2, and has been referred to throughout this thesis. This system is of course, only an example meant to present our work. It is not intended to be a complete, commercial system. Nevertheless, it provides an adequate basis for assessment of our test approach and illustration of our methods.

7.1 Case Study Development and Assessment Plan

The case study development process proceeds as follows:

- *Step 1*: Requirements and validation by high-yield strategy for functional test design.
- *Step 2*: System design using CORBA, includes identifying system components, building the interface of each component in IDL.
- *Step 3*: System implementation, including server, client and web site.
- *Step 4*: Verification and validation process, including
    - *(a)* Designing and building CORBA-based test harness
    - *(b)* Designing high-yield test cases
    - *(c)* Performing test execution

- *(d)* Performing test results analysis

- *Step 5*: Finally we present an assessment of our approach against following criteria
    - The approach is measurable and risk driven.
    - The approach makes application construction and integration easier.
    - The approach simplifies the test process and increases testability meaning measurability, test coverage and automation of e-commerce system.
    - The approach provides high flexibility and scalability

The development starts from requirements analysis. From the customer's initial requirements which are obviously not complete and clear, some necessary assumptions are made. Then the High Yield Requirements Capture approach described in section 6.1 is applied to capture all the core use cases.

7.2 *Step 1*: High-Yield Requirement Capture

In Chapter 2, we already introduced the initial requirements and basic assumptions of the EX system. Now we apply our High-Yield Requirements Capture approach [Probert, 1999] to the system to identify and classify use case scenarios.

7.2.1 Use Cases

From the requirements specification, we can derive three core use cases: Online Exchange, Login, Get Quote.

i) The *Online Exchange* use case mainly deals with the customer going online to change Canadian dollars to U.S. dollars. The main purpose of this use case is to successfully place an order through the EX system which eventually results in the customer getting the U.S. dollars by direct deposit to his U.S. account or getting a money order delivered by the post office to his home or place of business. The equivalent amount of Canadian dollar is withdrawn from the customer's Canadian account. Figure 7.1 shows the normal (primary) scenario of this use case.

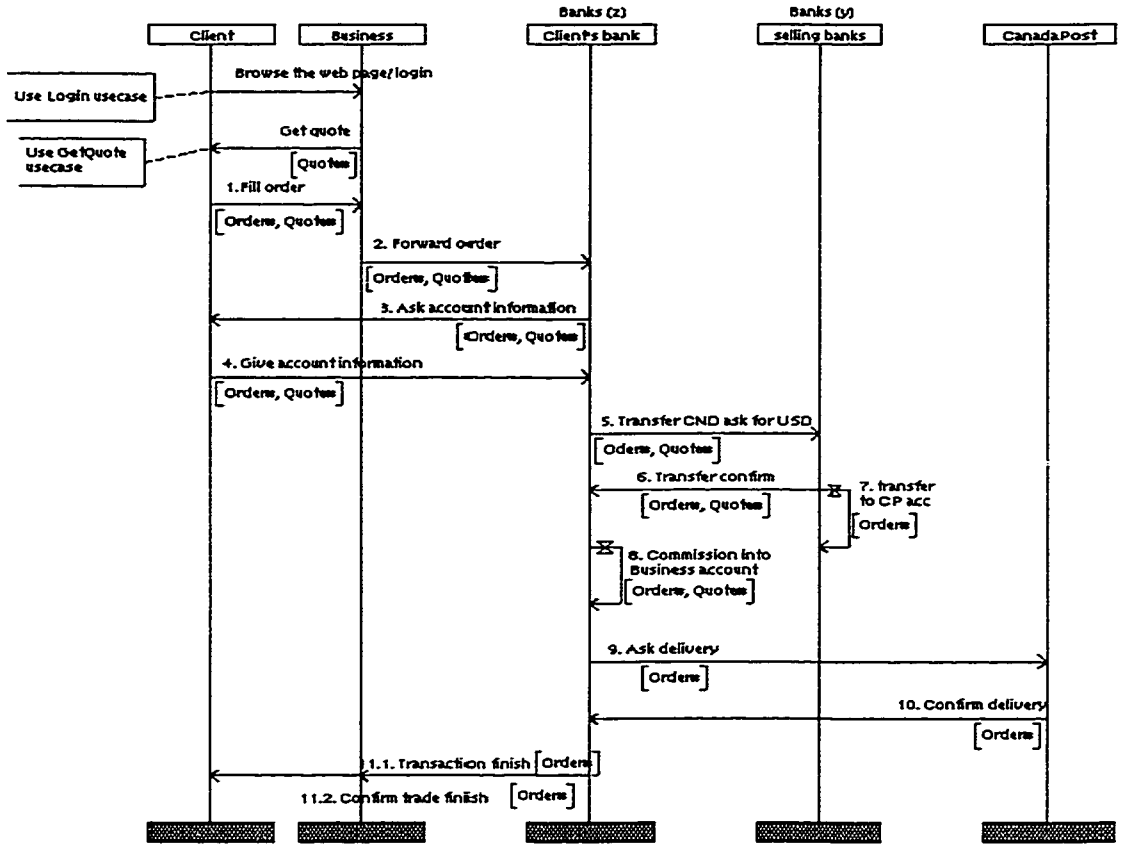


Figure 7.1 Online Exchange Normal Scenario

ii) The *Login* use case deals with the customer going to the EX web page, typing his account number and password to login to his account in EX. The main purpose of this use case is to successfully login. Figure 7.2 shows the normal (primary) scenario of the use case.

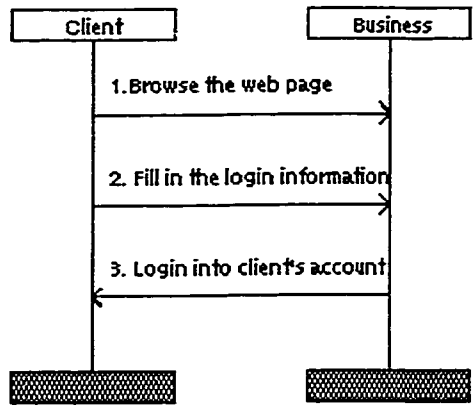


Figure 7.2 Login Normal Scenario

iii) The *Get Quote* use case deals with the customer asking the exchange rate for a certain amount of Canadian dollars to U.S. dollars. The main purpose of this use case is to successfully get the quote back to the customer. Figure 7.3 shows the normal (primary) scenario of this use case.

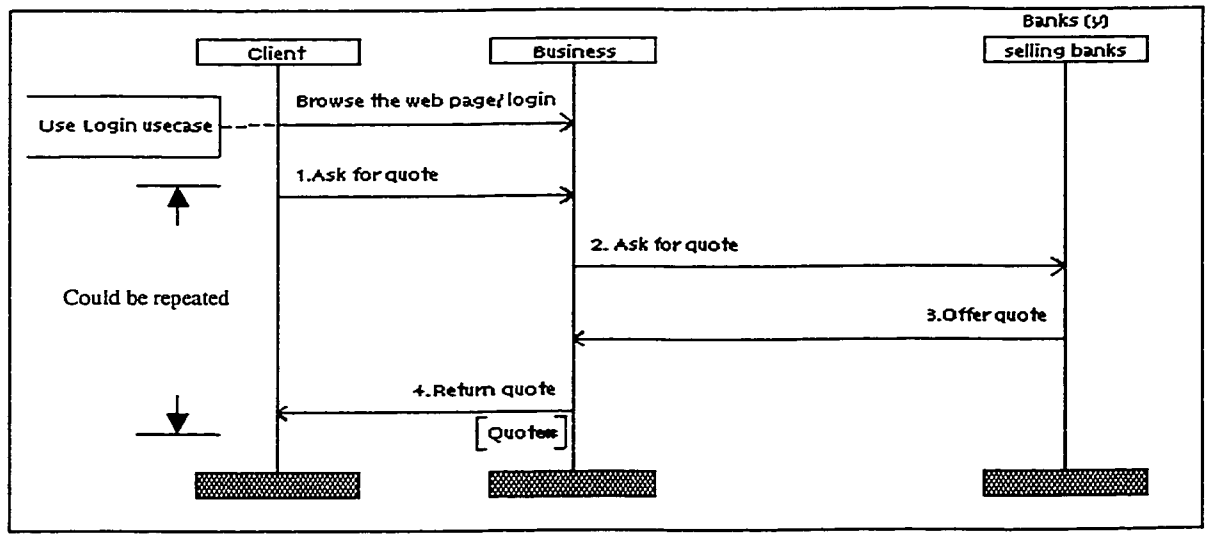


Figure 7.3 Get Quote Normal Scenario

7.2.2 High-Yield Scenario Selection Strategy

For each use case, we identified various primary, successful, basic path, low-yield (category 1) or unsuccessful, alternative paths, low-yield (category 2) scenarios including possible cancellation of the operation at various points of the use case. We have also identified secondary (high-yield) scenarios in which some errors occur but eventually are corrected by the customer, therefore resulting in a successful completion of the use case (category 3). Moreover, more high yield secondary scenarios were identified in which some errors occur, but the customer either fails to fix them or decides to cancel, therefore resulting in unsuccessful termination of the use case (category 4). Finally, we identified high-yield concurrent scenarios which some result in the successful termination of the use case, and the rest do not lead to a successful completion of the all the concurrent use cases (category 5).

As recommended in section 6.1.3, we focused on high-risk area of the system, identified relatively large number of high-yield use case scenarios. The key metrics of use case scenario (later in section 7.2.4) shows we achieved high coverage of the problem areas of the requirement specification.

7.2.3 Summary of Use Case and Their Classification.

We classify the scenarios of the 3 core EX use cases according to the classification approach [Probert, 1999].

7.2.3.1 Summary of Online Exchange use case scenarios and their classification.

In section 7.2.1, we described the core Online Exchange use case. The following table presents the list of the scenarios and their classifications as described in section 6.1.2.

In total, we identified 4 primary scenarios, 12 secondary scenarios and 5 concurrent scenarios.

Table 7.1 *Online Exchange* use case scenarios

Sc #	Ref. No.	Use Case Description	Sc. class/Yield category

		Use case: Online Exchange --Purpose: successfully place an order of exchange CND to USD	
		Actor: client, business server, client's bank, selling bank, shipping company (Canada Post)	
		Used use cases: Login, Get Quote	
		Use case start from client get the Quote	
	1.	**Primary Scenarios**	
	1.1	**Basic paths**	
1	1.1.1	Online exchange for direct deposit	1/ L
2	1.1.2	Online exchange for money order or traveler's check	1/ L
	1.2.	**Alternative paths**	
3	1.2.1	Cancel order while filling the order	2/ L
4	1.2.2	Cancel order while filling account information	2/ L
	2.	**Secondary Scenarios**	
	2.1.	**Data problems**	
5	2.1.1	Invalid order information, client cancels the order	4/ M
6	2.1.2	Invalid order information, client corrects the order	3/ H
7	2.1.3	Invalid account information, client cancels the order	4/ M
8	2.1.4	Invalid account information, client corrects the information	3/ H
9	2.1.5	Over account balance, client cancels the order	4/ M
10	2.1.6	Over account balance, client changes order selection	3/ H
	2.2	**Network/system problems**	
11	2.2.1	Client computer fails before client submits the order	4/ H

12	2.2.2	Client computer fails after client submits the order	4/ H
13	2.2.3	Client computer fails before client gives account information	4/ H
14	2.2.4	Client computer fails after client gives account information	3/ H
15	2.2.5	Business server fails before client submits the order	4/ H
16	2.2.6	Business server fails after it forwards order to client's bank	4/ H
	3.	**Concurrent Scenario**	
	3.1	**Concurrent primary scenarios**	
	3.1.1	Basic paths	
17	3.1.1.1	One client orders online from two computers at same time, no conflict #1*	5.1/ L
	3.1.2	Alternatives	
18	3.1.2.1	One client orders from 2 computers at the same time, cancel the two orders, no conflict #1*	5.2/ L
	3.2	**Concurrent secondary scenarios**	
19	3.2.1.	One client online order from two computers with conflict #1*, Client changes the orders to remove the conflict.	5.3, 5.3/ H
20	3.2.2.	One client online orders from two computers, with conflict #1*, Client cancels one order and keeps the other one to remove the conflict.	5.3, 5.4/ H
21	3.2.3	One client online orders from two computers with conflict #1*, Client cancels two orders.	5.4,5.4 H

Note: conflict #1 occurs when the client's order exceeds the credit limit.

L--low yield,

H--high yield

7.2.3.2 Summary of Login use case scenarios and their classification.

The following table presents the list of the scenarios and their classifications for the Login use case described in section 7.2.1.

In total, we identified 2 primary scenarios and 5 secondary scenarios.

Table 7.2 *Login* use case scenarios

Sc #	Ref. No.	Use Case Description	Sc. class/ Yield category
		Use case: Login --Purpose: successfully login client's account at business	
		Actor: client, business server	
		Use case start from client in business's homepage.	
	1.	**Primary Scenarios**	
	1.1	**Basic paths**	
1	1.1.1	Client login	1/ L
	1.2.	**Alternative paths**	
2	1.2.1	Client leaves business's homepage while filling login information	2/ L
	2.	**Secondary Scenarios**	
	2.1.	**Data problems**	
3	2.1.1	Invalid login information, client leaves business's homepage	4/ M
4	2.1.2	Invalid client information, client corrects the information	3/ H
	2.2	**Network/system problems**	
5	2.2.1	Client computer fails before client finishes filling login	4/ H

		information	
6	2.2.2	Client computer fails after client finishes filling login information	4/ H
7	2.2.3	Business server fails before client finishes filling login information	4/ H

7.2.3.3 Summary of Get Quote use case scenarios and their classification.

In section 7.2.1, we described the Get Quote use case. The following table presents the list of the scenarios and their classifications.

In total, we identified 3 primary scenarios and 6 secondary scenarios.

Table 7.3 *Get Quote* use case scenarios

Sc #	Ref. No.	Use case Description	Sc. class/ Yield category
		Use Case: Get Quote –Purpose: successfully get quote	
		Actor: client, business server, selling banks	
		Used use cases: Login	
		Use case starts from client entering his/her account at business server	
	1.	**Primary Scenarios**	
	1.1	**Basic paths**	
1	1.1.1	Get quote	1/ L
	1.2.	**Alternative paths**	
2	1.2.1	Cancel while filling quote request	2/ L
3	1.2.2	Logout before gets quote back	2/ L

	2.	**Secondary Scenarios**	
	2.1.	**Data problems**	
4	2.1.1	Invalid quote request (e.g. ask exchange other than CND and USD), client cancels the request	4/ M
5	2.1.2	Invalid quote request, client corrects the request	3/ H
	2.2	**Network/system problems**	
6	2.2.1	Client computer fails before client submits the quote request	4/ H
7	2.2.2	Client computer fails after client submits the quote request	4/ H
8	2.2.3	Business server fails before client submits quote request	4/ H
9	2.2.4	Business server fails during negotiation with selling banks	4/ H

7.2.4 Key Metrics for Requirement Capture

Table 7.4 shows the summarized results of applying the metric formula introduced in section 6.1.3 on the use cases described in section 7.2.2 There are in total 37 test cases. Among them, are 11 low yield test cases and 26 high yield test cases. From table 7.4, we can see how the high resulting coverage of the use cases and the concentration on high-risk, high-yield scenarios is highlighted by the last four metrics in the table.

Table 7.4 Key Metrics of Requirement Capture

Name of Metric	**Formula**	**Online Exchange**	**Login**	**Get Quote**	**Total**
Number of requirement for	a	1	1	1	3

functional areas (use cases)					
Total number of scenarios	b	21	7	9	37
Number of low-yield scenarios	c	6	2	3	11
Number of high-yield scenarios	d = b - c	15	5	6	26
Average basic coverage	e = b / a	21/1 = 21	7/1=7	9/1 = 9	37/3 = 12.3
Average risk coverage	f = d / a	15/1 = 15	5/1 = 5	6/1 = 6	26/3 = 8.7
Degree of risk orientation	g = d / b	15/21 = 0.71	5/7 = 0.71	6/9 = 0.67	26/37=0.70
Scenario risk ratio	h = d / c	15/6 = 2.5	5/2 = 2.5	6/3 = 2	26/11 = 2.4

7.3 *Step 2 (a)*: Identifying Components and Interface

The next step is to identify all the entities in the system, and the interfaces between those entities.

A typical e-commerce system requires the collaboration of several entities to complete an order transaction. The orderly and timely exchange of messages among those agents ensure the correct execution of an transaction. In EX system, every order transaction involves (as shown in Figure 7.4) the customer, the EX service provider (namely EX business), banks, shipper.

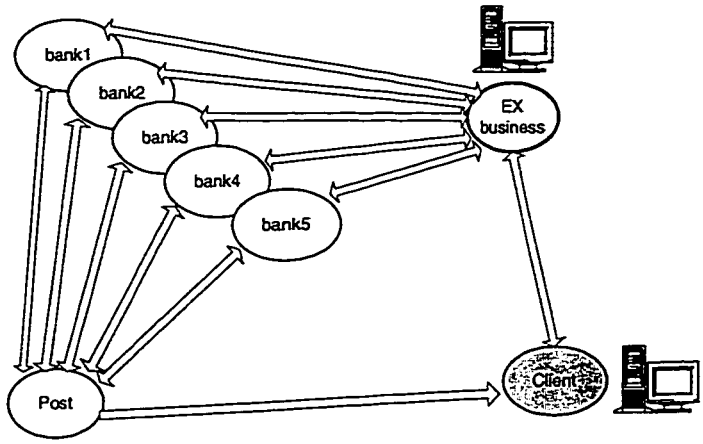


Figure 7.4 Agent Interaction Diagram

Before we show the IDL interface file for each component, we first introduce the main elements that constitute the CORBA IDL. Those elements will be seen later in IDL interface files.

Modules provide a namespace to group a set of class descriptions (or *interfaces* in OMG terminology). A module is identified by the keyword *module* followed by the name of the module. The main purpose of a module is to introduce an additional level of hierarchy in the IDL namespace.

Interfaces define a set of methods (or *operations* in OMG terminology) that a client object can invoke on the server object. Think of it as a class definition, but without the implementation section. An interface can declare one or more *exceptions* that indicate an operation did not perform successfully. An interface can be derived from one or more interfaces, which means IDL supports multiple inheritance among interfaces.

Operation is the CORBA-equivalent of a method. It denotes a service that clients can invoke. The IDL defined the operation's *signature*, which means the method's parameters and the result(s) it returns.

Data types are used to describe the accepted values of CORBA parameters, attributes,

exceptions, and return values. These data types are named CORBA objects that are used across multiple languages, operating systems, and ORBs. CORBA suppors two categories of types: *basic* and *constructed*. CORBA basic types include short, long, unsigned long, unsigned short, float, double, char, boolean, and octet. CORBA constructed types include enum, string, struct, arry, union, sequence, and any. The *struct* type is similar to a C++ structure; it lets you create any complex data type using *typedefs* (type definitions). The *sequence* type lets you pass a variable-sized array of objects. The *any* type is very useful in dynamic situations because it can represent any possible IDL data type - basic, constructed, or object reference. Each CORBA IDL data type is mapped to a native data type via the appropriate language bindings.

The CORBA IDL is very comprehensive and concise. The entire language as described in [OMG, 1995], includes the definition of the IDL grammar and all the CORBA data types.

The following are the IDL interfaces of each component.

7.3.1 EX business

The IDL file of EX business is named *business.idl*. In this file, all the *Data Types* and *Interfaces* are defined in *module business*. The data exchanged between the business, the banks and the customers are: *CustomerAcc* information, *Quote*, *Order*, *TradeHistory*, and *BusinessException*. The services provided by EX business are defined as *operations* in *interface Business*. They are: *getClientAcc* which returns customer account information after customer gives the correct login information at EX business's main web page; *getQuote* which returns best quote to customer after obtaining it from the banks; *acceptNewQuote* which updates the customer's account with a new quote after the customer accepts it; *placeOrder* which receives the customer's order and forward order to customer's bank for processing.

The complete *business.idl* is as following:

```
module business
{
//
//Date exchanged between business and bank and customer
```

```
//
        exception BusinessException
        {
                string reason;
        };


        struct Quote
        {
                string serialNumber;
                double exchangeRate;
                double value;
                string bank;
                string offerDate;
                string expireDate;
        };

        struct Order
        {
                string serialNumber;
                string date;
                Quote quote;
                double value;
                string fromBank;
                string fromAcc;
                string password;
                boolean type;
                string toBank;
                string toAcc;
        };

        struct OrderOwner
        {
                string name;
                string accNumber;
        };

//
//Data managed by business and required by customer
//
        struct Trade
        {
                string date;
                double value;
                double exchangeRate;
                string bank;
                string orderNumber;
                boolean update;
        };

        typedef sequence<Trade> TradeHistory;

        struct CustomerInfo
        {
```

```
                string name;
                string address;
                string SIN;
        };

        struct ClientAcc
        {
                string accID;
                string password;
                CustomerInfo customerInfo;
                TradeHistory history;
                Quote currentQuote;
        };

//
//Service provided by business
//
        interface Business
        {
                //
                // return client account information after client give the
correct login information at business's main web page
                //
                ClientAcc getClientAcc (in string sessionID, in string
customerID, in string password) raises (BusinessException);

                //
                //return best quote to client after acquired from bank
                //
                Quote getQuote (in string sessionID, in double value) raises
(BusinessException);

                //
                //Update client account with new quote after customer accept
the new quote
                //
                void acceptNewQuote (in ClientAcc clientAcc, in Quote quote)
raises (BusinessException);

                //
                //Recieve client's order forward order to client's bank to
process order
                //
                void placeOrder (in string sessionID, in ClientAcc client, in
Order order) raises (BusinessException);
        };

};
```

7.3.2 Banks

The IDL file of bank is named *bank.idl*. In this file, all the *Data Types* and *Interface* are defined

in *module bank*. The data exchanged between business, bank and customer are: customer *account* information, customer *Transaction*, customer *TransactionHistory* and *BankException*. The services provided by the bank are defined as *operations* in *interface SellingBank*, and *interface ClientAccBank*. Here *SellingBank* performs one specialized bank's role of providing the currency exchange service: *offerQuote* which gives EX business the latest quote upon request; *exchangeFund* which processes the transaction of currency exchange according to the request from customer's bank. *ClientAccBank* is another specialized role of providing basic bank services: *withdraw* and *deposit*.

The complete *bank.idl* is as following:

```
module bank
{
        exception BankException
        {
                string reason;
        };

//
//Data managed by bank
//
       struct Transaction
        {
                string date;
                double value;
                string transactionType;
                boolean update;
        };

        typedef sequence <Transaction> History;

        struct Account
        {
                string accountNumber;
                string password;
                string holderName;
                string holderAddress;
                string holderSIN;
                double balance;
                string currency;
                History history;
        };

//
//Using object factory to specialize bank's role
//
```

```
        //
        // one bank's role to provide currency exchange service
        //
        interface SellingBank
        {
                //
                //give business the quote
                //
                business::Quote offerQuote (in string session, in double
value) raises (BankException);

                //
                //transaction of currency exchange, according to request from
client's bank
                //
                void exchangeFund(in string sessionID, in business::Order
order) raises (BankException) ;

                //
                //bank destory
                //
                void destroy();
        };

        //
        // one bank's role to provide basic bank service: withdraw and
deposit
        //
        interface ClientAccBank
        {
                //
                // process order after get order from business
                //
                void processOrder (in string session, in business::OrderOwner
orderOwner, in business::Order order) raises (BankException);

                //
                //basic service
                //
                void withdraw (in string sessionID, in string accNumber, in
string password, in double value ) raises (BankException);
                void deposit (in string sessionID, in string accNumber, in
double value) raises (BankException);

                //
                //bank destory
                //
                void destroy();
        };

        //
        // create bank object as request
        //
        interface BankFactory
```

```
        {
                SellingBank createSellingBank();
                ClientAccBank createClientAccBank();
        };

};
```

7.3.3 POST Office (shipper)

The IDL file of post office is named *post.idl*. In this file, *Interface* are defined in *module post* The services provided by *post* are defined as *operations* in *interface Post*. For example, *delivery* receives delivery requests and sends back delivery confirmations to customers and carries out the delivery service.

The complete *post.idl* is as following:

```
module post
{
        interface Post
        {
                //
                // receive delivery request, send back delivery confirmation
and carry out the delivery
                //
                void delivery(in string sessionID, in string deliveryInfo);
        };
};
```

7.4 *Step 2 (b)*: Web Site Design and User Interface

Although web site and Graphic User Interface design is not the focus of our work, customers use the EX service and complete order transactions all though GUI and web site. It is necessary to go though the transaction flow in terms of GUI to get a look and feel for how the system works.

To simulate e-commerce in the real word, we installed Apache web server [Apache, 1999] and stores the EX business home page on that server.

When customers visit the EX business home page, they will see the page shown in Figure 7.5.

Figure 7.5 Homepage

Customer clicks the big character 'here', home page will change to service page, the first service page is the login page as shown in Figure 7.6.

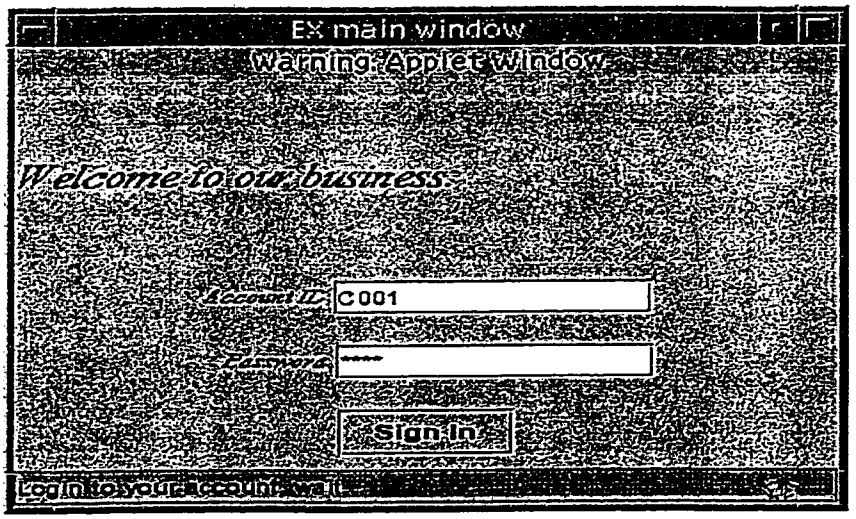


Figure 7.6 Login Window

At the login page, customer enters his/her user ID and password, which are all configured off-line when customer is applying for a user account at EX business (described in Chapter 2), then clicks 'login' button. If the user ID and password are all correct, the login page will change to customer account page, as shown in Figure 7.7.

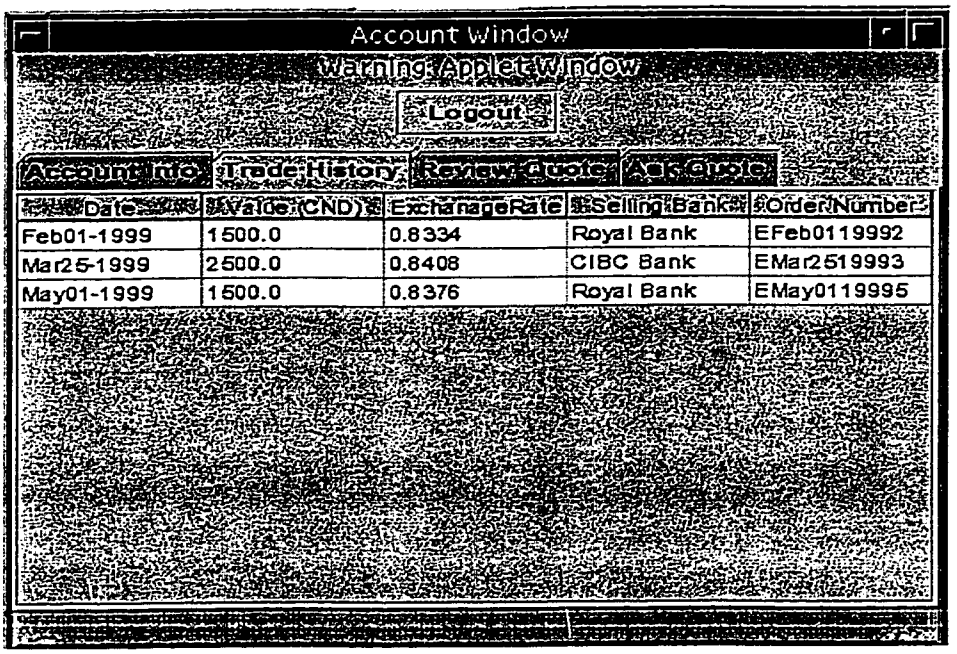


Figure 7.7 Customer Account Window

As shown in Figure 7.8, the customer can review his/her account information by choosing 'account information' tab; or review the quote by choosing 'review quote' tab; or review his/her trading history by choosing 'review history' tab; or ask for new quote by choosing 'ask new quote' tab; or simply choose 'log out' button to go back to login page.

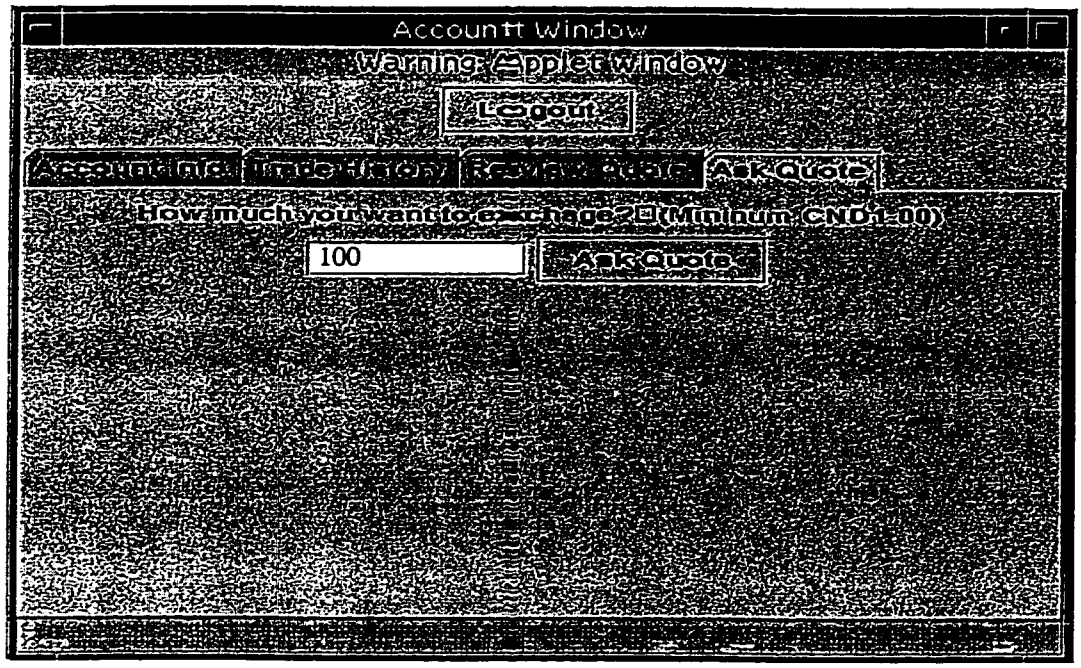


Figure 7.8 Account Info Window

When customer asks for new quote (Figure 7.8), he/she simply enters the amount of CND needed to be exchanged to USD, then click 'OK', the order form page along with the new quote will be displayed, as in Figure 7.9. Customer can choose to 'Accept Quote, Order later' or 'cancel' to go back to the customer account page, also customer can choose to fill out the order form, then click 'Submit', a message window will pop up telling the customer the order is in process.

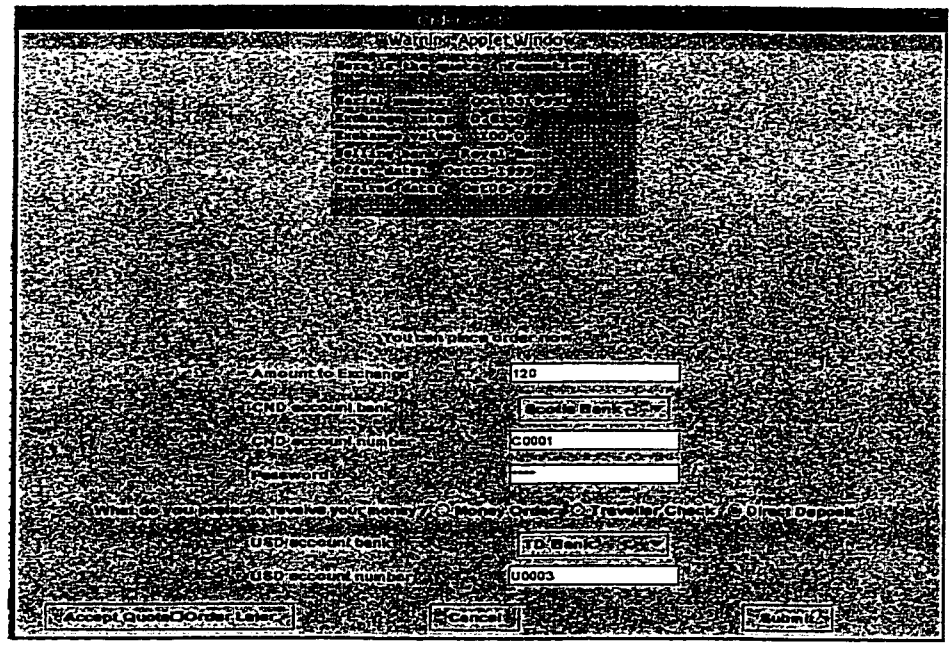


Figure 7.9 Order Form

After the order transaction is finished, the message window disappears, and a new message window pops up, as in Figure 7.10, informing customer that the order is successfully processed.

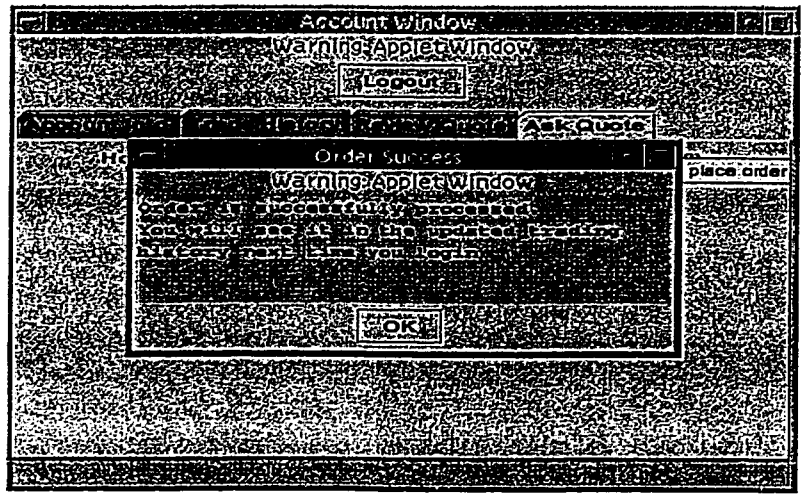


Figure 7.10 Order Finished Successfully

During the interaction between customer and GUI, any wrong input or omission of input will

cause an error message report window to pop up. This window looks like Figure 7.10. Customer can choose 'OK' to dismiss the window and go back the appropriate page to correct the input or simply cancel the operation.

7.5 *Step 3*: System Implementation

After we identified the components in the system, we then defined the interface between each component in IDL. We simply implemented all the components following the developing procedure of client/server CORBA-based application introduced in Chapter 3. We used ORBacus [OOC, 1999] CORBA implementation (ORBacus 3.0) and Java programming language (Java 2.0).

The EX system implementation has about 4000 lines of Java code, 300 lines of IDL and 400 lines of HTML. It took one person about 4 weeks to finish.

Now we can put all the components and the web server together to set up the EX system. The system architecture is shown in Figure 7.11. We note again that this is a case study system, and is not intended to be a complete commercial product.

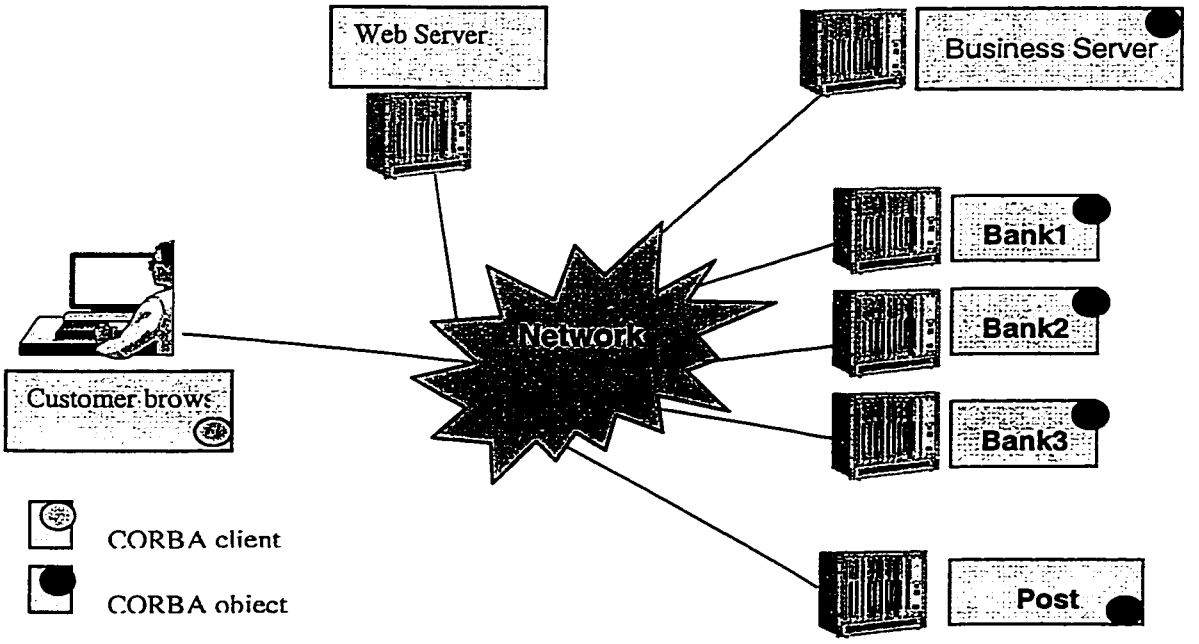


Figure 7.11 EX System Architecture

7.6 *Step 4*: Verification and Validation Process

7.6.1 *Step 4 (a)*: Design CORBA-based Test Harness

The test architecture of the system follows the test architecture defined in section 6.3, and is shown in Figure 7.12. It has about 3500 lines of Java code, 200 lines of IDL code, and took one person about 4 weeks to finish.

First, we define the interface between the test components and our system. Here is the interface of the test components defined in IDL:

```
module extester
{
        enum STATUS {ignore, finish, typeReturned, exceptionThrown, timeout
};
        enum RESULT {expected, wrong, inconclusive};
        enum VERDICT {pass, fail, inconclusive};

        struct Step
        {
                string operation;
                string input;
                string output;
        };

        struct Message
        {
                string sessionID;
                string messageID;
                string sender;
                string receiver;
                string timeStamp;
                string content;
                STATUS processStatus;
        };

        typedef sequence<Step> Process;

        struct TestCase
        {
                string desc;
                Process process;
        };

//
// test component and basic functionality
//
        interface TestComponent
        {
```

```
                //
                //Record information of message exchanged by system under
test
                //
                void serviceRequest (  in Message msg);

                //
                //register at master test component
                void register (in string testerSite);

                //
                //unregister from master test component
        void unregister (in string testerSite);
          };
};
```

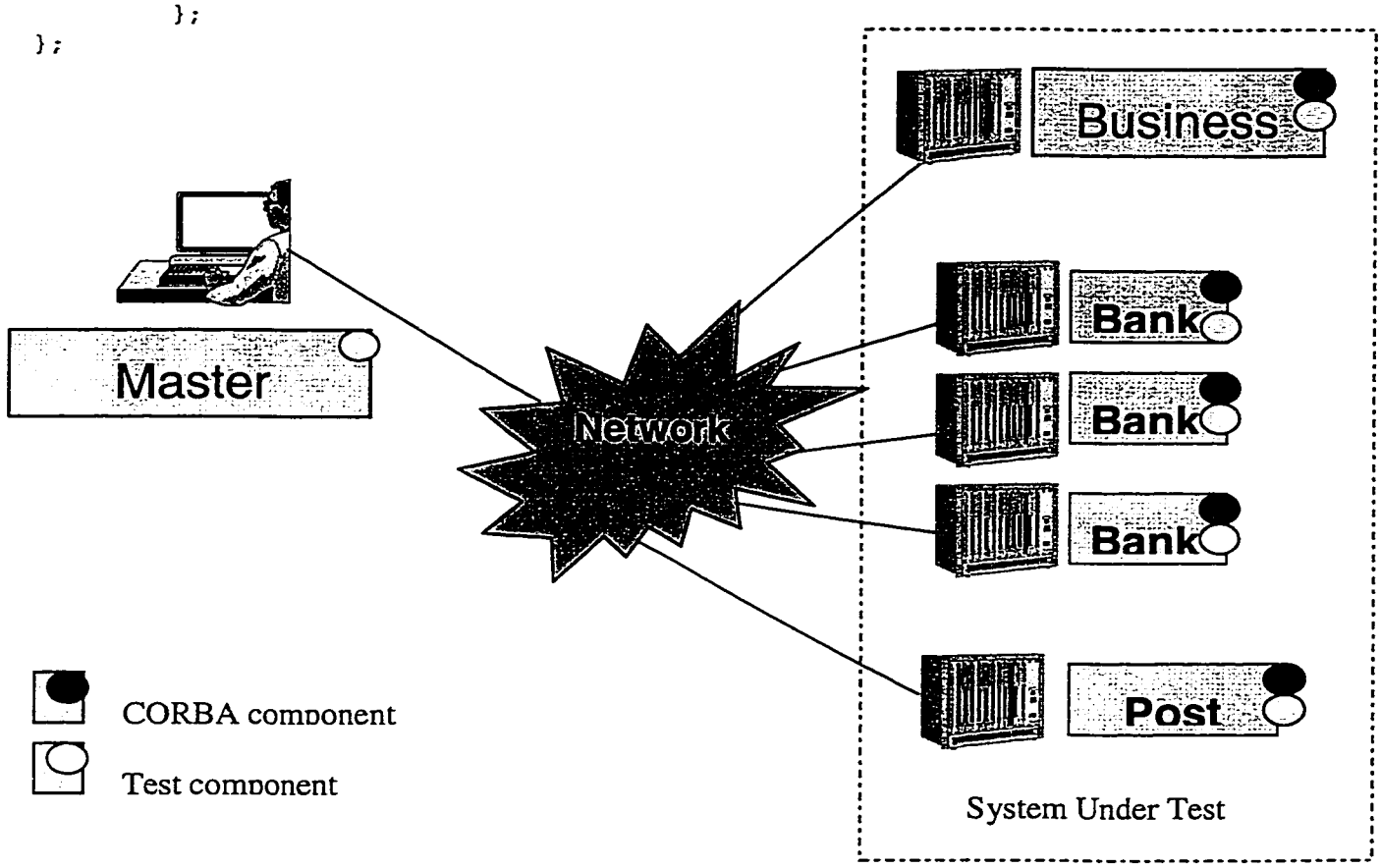


Figure 7.12 Test Architecture

MTC and TCs communicate through *serviceRequest*, *register* and *unregister* operations. The *struct TestCase* and *struct Message* are the data structure communicated between MTC and TCs.

As mentioned before, the MTC consists of Test Component, Test Case Manager, Client Simulator, Test Verdict, Test Report Manager. MTC is the central controller of the test

execution, TCs are distributed at the each CUT which is the EX business server and bank servers.

The *Test Case Manager* reads the test case from test case file, parses the test case according to the format defined *in struct TestCase*. It executes the test case step by step exactly following the *sequence<Step> Process*. In every *Step*, *Test Case Manager* passes the *Test Verdict* and *Client Simulator* the *struct Step* which contains the information of the type of service requested, the input parameter and the output type defined in *enum STATUS*.

The *Client Simulator* builds the CORBA service request according to the information in *struct Step* and send the request to EX business server and bank servers. Each server processes its request, the TC sitting at the server record the service request, the input, and actual output. According to the output, TC maps it to one of the abstract output type defined in *enum STATUS*. Then TC sends the report in format of *struct Message* through operation *serviceRequest* to MTC. In the Message, TC indicates which server (*string receiver* in *struct Message*) performed the service request, what service is request (*content*), when (*timestamp*), what is the result (*processStatus*) and test session identifier (*sessionID*) and request identifier (*messageID*).

MTC receives the *Message*, pass it to *Test Verdict*, *Test Verdict* compares the actual result with the expected result to issue the test verdict as one of the verdict result defined in *enum RESULT* and pass the test verdict to *Test Report Manager*. *Test Report Manager* simply logs every step of the test execution into a central log.

The test component fulfills all the functionality of test component outlined in section 6.4.1.

7.6.2 *Step 4 (b)*: Design High-Yield Test Cases

In an e-commerce system, the results of every step of transaction are dynamically generated and unique. For example, since every transaction has unique process ID generated on the fly, the error message is quite different from scenario to scenario. It's very hard to predict the detailed result in advance of testing. Also it's tedious work to compare every detail of the expected result with actual output. To keep our case study simple and not too labor intensive, we write test cases partially in an abstract format. The input data of test case is exactly as same as the customer real

input, the output is abstract which means we classify the output as four types:

- TypeReturned – the return result is the expected data type.
- Finish – nothing returned
- ExceptionThrown – something goes wrong, error message returned.
- Timeout – after certain amount of time, none of the above three outputs happens, this result is classified as 'timeout'.

So when the test system captures the output, it will automatically map it into one of the four categories, then compare it with the one defined in the corresponding test case.

The test cases are written based on the use cases identified in section 7.2.2 and according to the format defined in *extester.idl* in section 6.3. The example test cases are:

Primary test cases

p.1. Online exchange for direct deposit
getClientAcc
$$ C001 $$ why3
typeReturned
getQuote
$$ 100
typeReturned
placeOrder
$$ 101 $$ CIBC Bank $$ C0002 $$ why3 $$ true $$ TD Bank $$ U0003
finish

Secondary test cases

s.2. Invalid order information, client correct the order
getClientAcc
$$ C001 $$ why3
typeReturned
getQuote
$$ 100
typeReturned
placeOrder

$$ 90 $$ CIBC Bank $$ C0002 $$ why3 $$ false $$ TD Bank $$ U0003
exceptionThrown
placeOrder
$$ 100 $$ CIBC Bank $$ C0002 $$ why3 $$ false $$ TD Bank $$ U0003
finish

The MTC component will read the test cases, parse them to the test case format defined in *extester.idl* to build the executable test cases. In case study, we implement proprietary functions to parse above textual test cases. Some experiment work has been done in [Schieferdecker, 1998] which defined basic rules for mapping IDL operations, exceptions, constants, attributes and type definitions to TTCN and provided a prototype of TTCN/CORBA gateway facilitating the mapping. It set up the stage for possibly automating the test cases derivation from IDL interface. For this case study, we focus on test case execution rather than test case derivation. Future work could be done to describe test cases in formal test case description techniques such as TTCN and develop tool support of conversion between IDL and TTCN in order to fully automate all phases of testing.

Here we provide the TTCN description for above test cases to illustrate this direction.

```
p.1. Online exchange for direct deposit                                  Verdict
!loginInfo (C001, why3)
  ?typeReturned       //getClientAcc
    !exchangeAmount (100)
      ?typeReturned           //getQuote
        !orderInfo (101, CIBC Bank, C0002, why3, true, TD Bank, U0003)
          ?finish             //order process finish                   Pass
          ?*                                                         Fail
      ?*                                                             Incon.
  ?*                                                                 Incon.
```

Secondary test cases

```
s.2. Invalid order information, client correct the order
!loginInfo (C001, why3)
  ?typeReturned               // getClientAcc
    !exchangeAmount (100)
      ?typeReturned           // getQuote
        !orderInfo (90, CIBC Bank, C0002, why3, false, TD Bank, U0003)
```

```
        ?exceptionThrown          //order process error
          !orderInfo (100, CIBC Bank, C0002, why3, false, TD Bank, U0003)
            ?finish          //order process finish after correct error        Pass
            ?*                                                                 Fail
        ?*                                                                     Incon.
    ?*                                                                         Incon.
?*                                                                             Incon.
```

For reader's interest, we explain one test case, namely "p.1. Online exchange for direct deposit". First, EX's operation *getClientAcc* is called, account number "C001" and password "why3" is passed as parameters, then a *typeReturned* result (i.e. customer logins to his account successfully) is expected. Second, EX's operation *getQuote* is called, amount "100" is passed as parameter, then a *typeReturned* result (i.e. customer get the new quote successfully) is expected. Third, EX's operation *placeOrder* is called, and all the order information is passed, then the finish result (i.e. order is successfully processed) is expected. During test case execution, if the actual result of operation is as same as the expected result in all above 3 steps, the test case passed, otherwise, the test case failed or inconclusive.

7.6.3 *Step 4 (c)*: Test Case Execution

The execution process uses the same technique and follows the same procedure described in section 6.5.

We developed the GUI for the tester to display the test cases (only the description part), select test cases and choose to execute the test cases selected. The tester GUI is shown as Figure 7.13. Test case execution is done automatically.

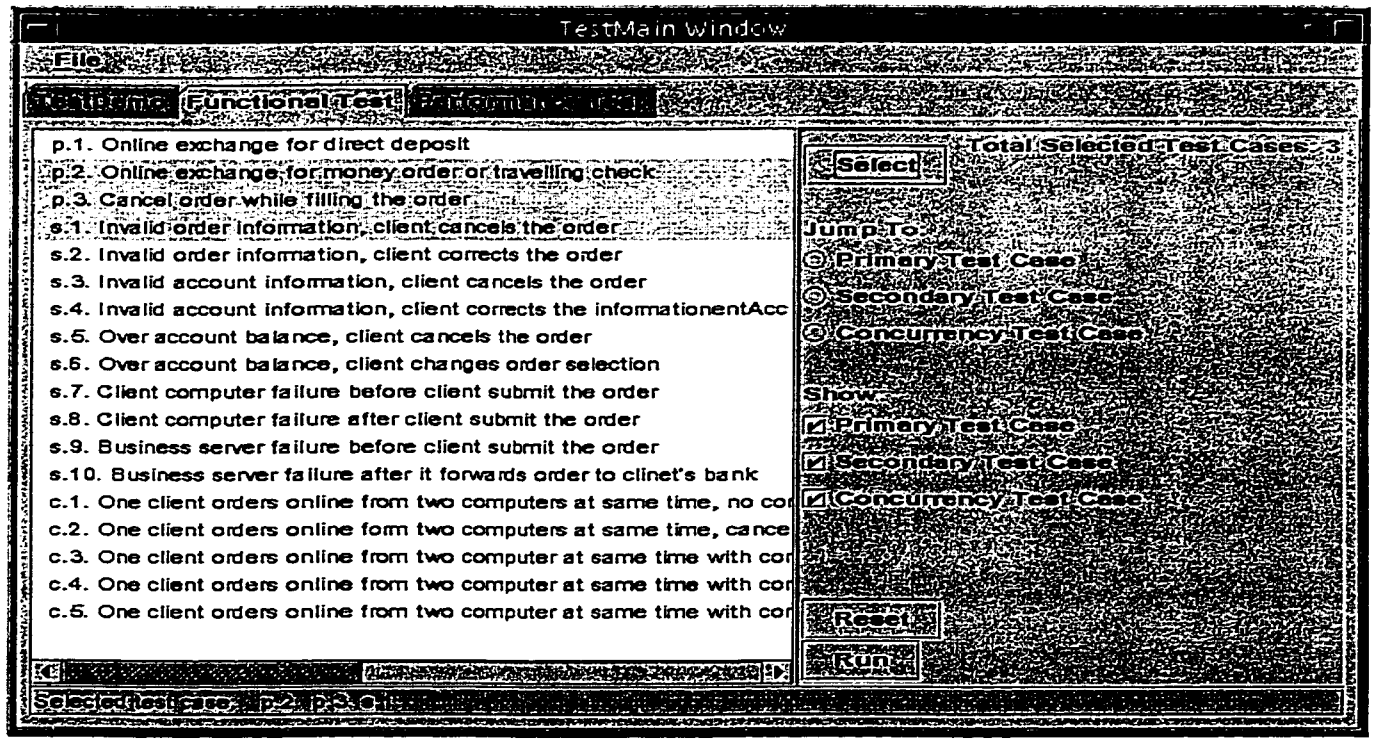


Figure 7.13 Test Control Window

7.6.4 *Step 4 (d)*: Test Results Reporting and Analysis

The test report is automatically generated during the execution of the test cases. The test system not only generates a test log file of execution of the test cases, but also logs all the customer operations. This gives us additional testing capabilities, namely capturing customer behaviour.

Following is the example of part of the test report corresponding to the test case example in section 7.6.2.

```
**********************************
******   start testing      ******
TIME :Wed Sep 08 16:41:58 GMT-04:00 1999
**********************************
SessionID -- testSession-0-0
Rational -- p.1. Online exchange for direct deposit
-- process 0
service request:  Login-getClientAcc
response:  typeReturned
time stamp:  Wed Sep 08 16:41:58 GMT-04:00 1999
```

expected response: typeReturned
result: expected
-- process 1
service request: getQuote
response: typeReturned
time stamp: Wed Sep 08 16:41:59 GMT-04:00 1999
expected response: typeReturned
result: expected
-- process 2
service request: placeOrder
response: finish
time stamp: Wed Sep 08 16:41:59 GMT-04:00 1999
expected response: finish
result: expected
verdict -- pass

SessionID -- testSession-0-2
Rational -- s.2. Invalid account information, client corrects the order
-- process 0
service request: Login-getClientAcc
response: typeReturned
time stamp: Wed Sep 08 16:42:02 GMT-04:00 1999
expected response: typeReturned
result: expected
-- process 1
service request: getQuote
response: typeReturned
time stamp: Wed Sep 08 16:42:02 GMT-04:00 1999
expected response: typeReturned
result: expected
-- process 2
service request: placeOrder
response: exceptionThrown
time stamp: Wed Sep 08 16:42:02 GMT-04:00 1999
expected response: exceptionThrown
result: expected
-- process 2
service request: placeOrder
response: finish
time stamp: Wed Sep 08 16:42:02 GMT-04:00 1999
expected response: finish
result: expected
verdict -- pass

```
************************************
******      end of test      ******
TIME :Wed Sep 08 16:42:02 GMT-04:00 1999
************************************


************************************
****** log of customer operation  ******
************************************


SessionID -- clientSession 0
-- process 0
service request:  Login-getClientAcc
response:  typeReturned
time stamp:  Wed Sep 08 16:39:04 GMT-04:00 1999
-- process 1
service request:  getQuote
response:  typeReturned
time stamp:  Wed Sep 08 16:39:30 GMT-04:00 1999
-- process 2
service request:  placeOrder
response:  finish
time stamp:  Wed Sep 08 16:40:53 GMT-04:00 1999


************************************
******       end       ******
************************************
```

This report includes the execution of two test cases and one customer operation log. The two test cases are all passed.

The first test case "Online Exchange for direct deposit" goes through the normal scenario of a business transaction. In the report, we can see from "process 0" to "process 2", customer successfully logs into his/her account, gets quote and gets the order processed. Everything goes fine in this test case, but we expect this, since this is a low-yield test case.

Note: the test case "Invalid order information, client corrects order" is one of the test cases by

which we detected an actual design errors. This case is developed as a high yield test case, it intends to test the system's behavior under abnormal conditions. When we ran this test case right after we finished implementation of the EX System, it failed. The test report showed the system didn't throw an exception when customer provided invalid order information. The case was a design omission, namely we didn't put enough guard conditions for a correct order. After we add all the guarding condition code, the test case passed as shown in the above test report.

7.7 *Step 5*: Case Study Results Assessment

The results of our work are encouraging. Implementing and validating an e-commerce service on top of a CORBA-like platform provides several advantages:

- The application demonstrates the high-yield requirements capture approach is a measurable and risk driven, systematic approach.

  From the use case table obtained in section 8.1.2, we can notice the high coverage of the use case and the concentration on high-risk, high yield scenarios. The coverage can be considered better in quality and quantity. From the metrics table shown in section 8.6.2, we apply key requirement metrics to actually measure the degree of coverage of risk associated with use case scenarios.

  We can also notice that the high yield approach is biased toward the problem areas of the requirement specifications, since the ratio of high-yield/low-yield is well over 1. The effect of the bias will not only be seen at the specification and design phases of the development process, but also, when these scenarios are used as the basis for test suite generation, more serious errors will be caught. Moreover, if these high-yield scenarios are considered during requirements capture, the likelihood of having related errors decreases considerably. We feel having more high-yield scenarios earlier in the process will improve the software quality and coverage in terms of requirements capture and testing. If we have a low ratio, i.e., more low-yield scenarios, we will have fewer test cases to cover serious errors and less time will be devoted to such test cases.

- The application construction and integration become much easier.

  E-commerce system is architecturally composed of multiple, decentralized and autonomous processing components exchanging messages across the network. For example, in the EX system, we have several distributed functional components, namely, client, EX business, banks. Those components collaborate to successfully complete the requested business function. By using CORBA's ORB as a communication framework for those e-commerce components, we are insulated from the complexities of low-level network protocols and different network devices. And by defining each component's interface in a standard language, IDL, components are allowed to be accessed and interoperated from anywhere in a distributed system. Thus adopting industrial middleware standards can achieve greater interoperability with other systems. Although we do not have any data to support it, we believe the resulting quality improvement and interoperability will help to avoid errors and rework, and thus lessen time-to-market.

- The proposed test architecture simplifies test process and increases testability of e-commerce system.

  Based on the test case format defined in IDL, the construction of the test case file becomes very straightforward. This makes it possible to automate the generation of the test case file directly from IDL.

  The execution of test cases are highly automated and distributed. The test system is able to monitor and control the entire test process of the highly distributed system from one place. It also enables us to fully automate the testing procedure, and create corresponding software tools. Test cases are automatically executed by a button 'click'; consequently, test report are automatically generated.

  By adopting this CORBA-based architecture, one can greatly improve the effectiveness of testing activities, and again shorten the time-to-market cycle.

- CORBA-based test architecture also provides high flexibility and scalability to the SUT and testing system.

Each new e-commerce service can be added as a new CORBA component. New CORBA testing component can be introduced to test this new service; since it only needs to register with the MTC as a new distributed TC. Similarly dropping a service only causes one service component and its corresponding TC to be dropped out of the test. This makes the testing architecture highly scaleable to accommodate different test goals, whether it is subsystem only testing or system-wide testing, functional testing or performance testing.

Because CORBA provides high interoporability between applications implemented on different platforms in different languages, the test system built on top of CORBA can be applied to other CORBA-based SUTs in the long run. Given the rapid development of new technologies, SUT can be ported to new platforms, transferred to more appropriate languages, etc. As long as the SUT is built on CORBA, the CORBA-based testing system will stay the same.

In this chapter we examined as a case study, an example e-commerce system and showed how our test architecture can be applied to verify the service provided by Online Currency Exchange system on top of CORBA platform. We described the entire development process: requirements capture, component identification, interface definition and construction of each component in IDL, test case generation, and finally the verification process. By means of a small, but complete case study, we demonstrated how the test architecture introduced in Chapter 7 significantly facilitates testing distributed systems.

Chapter 8

Conclusions and Future Work

In this thesis we introduced a generic test architecture of e-commerce application, and proposed a CORBA based test architecture. To demonstrate our approach, we used the development of a Online Currency Exchange system and corresponding test architecture on top of a CORBA platform as a case study. The results are very promising.

8.1 Conclusion

The main objective of this thesis is to contribute some new ideas to testing e-commerce applications focusing on the test execution phase in a distributed environment. We have achieved this objective through our analysis (section 6.3, 6.4 and 6.5) and case study (section 7.6 and 7.7) of a small but reasonably realistic e-commerce system. We demonstrated that if properly implemented, the proposed test architecture can achieve the requirements of testing e-commerce systems set out in section 5.2. Specifically, the proposed architecture

- provides a means for synchronizing distributed test components through MTC and TCs. The MTC can synchronize its child TCs by controlling test data delivery.
- supports dynamically configurable and scaleable test components through the distributed nature of CORBA platform. A new CORBA test component only needs to register with the MTC as a new distributed TC.
- is able to express test configurations for different communication scenarios.
- enables grey-box testing with access to internal components and interfaces through the definition of IDL interfaces. Since all the interfaces of SUT, TCs, etc are defined in IDL,

the implementation can decide how much internal detail it wants to expose. Furthermore the implementation may choose to expose various amounts of internal detail depending on runtime circumstance.

- supports real-time, performance and QoS testing for distributed systems to test time-related aspects through dedicated background TCs. Although these are not demonstrated in our case study, we believe that given the open, distributed, and flexible nature of the architecture, it can be configured to accommodate these goals.

- supports testing and monitoring in the pre-deployment, deployment and usage phases of distributed systems.

Thus we have demonstrated that the proposed test architecture meets many of the requirements of testing e-commerce applications and the objective of the thesis is achieved.

8.2 Future Work

We have identified the following areas that require future work.

- To improve the cost-effectiveness of testing, automation of all phases of the testing process is needed. In this thesis, we automated test execution, in the future, we can automate some aspects of test generation.

- Using existing formal description techniques to describe: the system (using SDL), the test suite obtained (using TTCN) and a test script (using MSC). In doing so, we can take advantage of commercial CASE tools, e.g SDT [Telelogic, 2000], ObjecTime [ObjectTime, 2000], to increase the automation of all phases of testing.

- As the world moves to using distributed platforms such as CORBA to implement open distributed applications, the interface of each component in the system will be defined in IDL. Thus, it is necessary to develop tool support of conversion between IDL and TTCN.

- Since IDL doesn't provide description of system dynamic behavior, e.g. different value of input and output, applying some traditional black-box techniques such as Boundary

Value Analysis to select more high-yield scenarios will help to achieve better risk coverage and more effectively verify IDL interface.

- With the rapid adoption of e-commerce, software security has become an important issue facing the industry. E-commerce security normally involves cryptographic modules, firewalls and Java Virtual Machine security feature. Each of them secures a different aspect of e-commerce transaction. If any of them is compromised, then the whole system is compromised. So software security testing has to keep pace, which means it has to do more than just determine if the system conforms to some specification or standard (conformance testing). It must also test the implementation for trap-door code, in other words, it must pinpoint if any of the system's intended functions are unintended or unauthorized. White –box testing can best satisfied this goal, but it is costly to implement and impossible for propriety third-party software. This further underlines the needs and benefits of open, standards based (such as CORBA) software architecture and its corresponding test architecture. Moreover research is needed to adapt such architectures to software security testing.

- To improve cost-effectiveness of robustness testing and reliability testing by screening out scenarios which will never occur through customer operational profile based test strategies.

- Customizing the testing architecture to other middleware, e.g. DCOM, DCE, JINI, etc

- Development tends to utilize XML for storage and manipulation of human-readable documents, and CORBA for tying together cooperating computer applications which exchange transient data. Future research is needed to find better ways of incorporating theses two technologies into one seamless software framework.

References

[Andreoli, 1997] Jean-Marc Andreoli, Francois Pacuil, and Remo Pareshi, XPECT: A Framework for Electronic Commerce, IEEE Internet Computing, p. 40-47, July/August 1997.

[Apache, 1999] http://www.apache.com

[Applegate, 1996] Applegate, L.M., Holsapple, C.W., Kalakota, R, Radermacher, F.J. and Whinston, A.B. Electronic commerce: building blocks of new business opportunity. J. Organiz. Comput. Electr. Comm. 6, 1, 1996.

[Beizer, 1990] B.Beizer, "Software Testing Techniques", second edition, Van Nostrand Reinhold, 1990.

[Bhimani, 1996] Anish Bhimani, Securing the Commercial Internet, Communications of The ACM, p. 29-34, June 1996.

[Boehm, 1976] B.W.Boehm, "Software engineering", IEEE Trans. Computer, vol. C-25, December 1976.

[Boehm, 1984] B.W.Boehm, "Verifying and validating software requirements and design specifications", IEEE Software, volume1, number1, pages 75-88, January 1984.

[Boehm, 1988] B.W.Boehm, "A spiral model of software development and enhancement", System and Software Requirements Engineering, IEEE Computer Tutorial, pages 513-527, May 1988.

[Box, 1998] Don Box, ActiveX/COM Q&A, Microsoft Systems Journal, Vol 13, No. 3, March 1998.

[Buehler, 1994] Buehler W., Introduction to ATM Forum Test Specifications, Version 1.0. ATM Forum Technical Committee, Testing Subworking Group, af-test-0022.000.

[CCITT, 1992] CCITT, Specification and Description Language, CCITT Z. 100, International Consultative Committee on Telegraphy and Telephony, Geneva, 1992

[Cerf, 1993] V. Cerf, "How the Internet Came to Be," the One –line User's Encyclopedia, ed. B. Aboba, Reading, MA: Addison-Wesley, 1993.

[Chung, 1997] P. Emerald Chung, Yennun Huang, Shalini Yajnik, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer", http://www.bell-labs.com/~emerald/dcom_corba/Paper.html , Bell Laboratories, Murray Hill, NJ, 1997.

[Cobb, 1985] R.H.Cobb, "In praise of 4GLs", Datamation, July 1985.

[COM, 1995] The Component Object Model Specification, http://www.microsoft.com/oledev/olecom/title.htm , Microsoft Corporation, Redmond, WA, 1995.

[CTMF1, 1993] Information technology – Open Systems Interconnection – Conformance Testing Methodology and Framework Parts 1: General concepts.

[CTMF2, 1993] Information technology – Open Systems Interconnection – Conformance Testing Methodology and Framework Parts 2: Abstract Test Suite specificaion.

[Dahl, 1996] Andrew Dahl, Leslie Lesmick, Internet Commerce, New Riders, 1996.

[Danthine, 1993] Danthine, A., Bonaventure, O., From Best Effort Enhanced QoS. CIO Deliverable, No. R2060/Ulg/CIO/DS/P/004, 1993.

[DCOM, 1997] "Quick Tutorial in Scalability, was Re: Sun and COM", Distributed COM-based code mailing list archive, multiple entries, http://microsoft.ease.lsoft.com/archives/dcom.html , Week 3, November 1997.

[Dima, 1999] Alden Dima, John Wack, Shakri Wakid, Raising the Bar on Software Security Testing, IT Professional, IEEE Computer Society, page 27-32, May/June 1999.

[ECFAQ, 1999] http://www.commerce.net/resources/efaq.html .

[Elenko, 1999] Mark Elenko, Mike Reinertsen, XML & CORBA. 1999

[Fowler, 1999] Martin Fowler, Kandall Scott, Carter Shanklin, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison Wesley, 1999.

[G. Winfield Treese, 1998] G. Winfield Treese, Lawrence C. Stewart, Designing Systems for Internet Commerce, Addison Wesley 1998.

[Gadre, 1990] Gadre J., Rohrer C., Summers C., Symington S., A COS Study of OIS Interoperability. Computer Standards & Interfaces, 9, 217-237, 1990.

[Gomaa, 1990] H.Gomaa, The impact of prototyping on software system engineering, George Mason University, School of Information Technology, 1990.

[Grabowski, 1995] Grabowski J., Walter T., Testing Quality-of-Service Aspects in Multimedia Application, in Protocols for Multimedia Systems 2nd Workshop, Salzberg, Austria.

[Gray, 1993] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Sam Francisco, CA: Morgan Kaufmann, 1993.

[IEEE Com.1997] Special issue on distributed object computing, IEEE Communications Magazine 35 (2), 1997.

[IDS, 1997] http://www.isc.org/dsview.cgi?domainsurvey/index.html .

[IONA, 1995] IONA Technologies Ltd. Orbix 2 Programming Guide, November 1995.

[ISD, 1995] Internet Domain Survey (IDS), Hosts Stats by County, http://www.gvu.gatech.edu/user_surveys , November. 1997.

[ITU-T, 1989] Data Communication Networks Open System Interconnection (OSI) Service Definitions Recommendations X.200 - X.219.

[ISO/IEC, 1995] ISO/IEC, Information Technology –OSI- Conformance Testing Methodology and Framework, International Standard 10746-3, ITU-T Recommendation X.903 – 1995.

[ITU, 1996] ITU-T Rec. Z.120 (1996), Message Sequence Chart (MSC), Geneva, 1996.

[Jacobson, 1992] I. Jacobson, Object-Oriented Software Engineering, Addison-Wesley, 1992.

[Jutla, 1999] Dawn Jutla, Peter Dodorik, Catherine Hajnal, Charles Davis, Making Business Sense of Electronic Commerce, IEEE Computer, March, 1999.

[Kalakota, 1996] Kalakota, R. and Whinstion, A.B. Frontiers of Electronic Commerce. Addison Wesley, Reading, Mass., 1996.

[Kamthan, 1998] Pankaj Kamthan, E-commerce on the WWW: Prospects and Concerns, Concordia, Canada, 1998.

[Lamarche, Probert, et al] R. Lamarche, R.L. Probert, V. Radonjic, B. Selic, and J. P. Corriveau, "High-Yield Strategies for Testing Object Oriented Software", In Proceedings of 2nd World Multiconference on Systemics, Cybernetics and Informatics and 4th International Conference on

Information Systems, Analysis and synthesis (SCI'98/ISAS'98), Orlando, Florida, USA, July 12-16, 1998.

[Lima, 1997] Ana R.Cavalli, Luiz Paula Lima Jr., Test execution of telecommunications services using CORBA, Report of Research, Nortelnetworks, 1997.

[Mednonogov, 2000] Mednonogov A., Kari H. H., Martikainen O., Malinen J., Conformance Testing of CORBA Service Using TTCN, Testing of Communication Systems, page 192-208, 2000

[Myers, 79] G.J.Myers, The Art of Software Testing, John Wiley & Sons, 1979.

[Myungchul, 1996] Myungchul K., Gyuhyeong K., Yoon D.C., Interoperability Testing Methodology and Guidelines. Digital Audio-Visual Council, System Integration TC, DAVIC/TC/SYS/96/006, 1996.

[OBI, 1999] http://www.openbuy.org .

[ObjectTime, 2000] http://www.objectime.com

[OMG] Object Management Group, http://www.omg.com

[OMG, 1995] Object Management Group, The Common Object Request Broker: Architecture and Specification; Revision 2.0. Framingham, Massachusetts, U.S.A., 1995.

[OOC, 1999] Object-Oriented Concepts, Inc., ORBacus for C++ and Java, the home page, http://www.occ.org , Billerica, USA, 1999.

[Orfali, 1997] Orfali R., Harkey D., Client/Server programming with Java and CORBA. Wiley, New York, 1997.

[Probert, 1992] R.L. Probert, O. Monkewich, TTCN: the international notation for specifying tests of communications systems. Computer Networks and ISDN Systems, 23, 111-126, 1992

[Probert, 1994] Robert L. Probert, Software Engineering, Ottawa, Canada, 1994.

[Probert et al, 1999] R.L. Probert, K. Saleh, W. Li, W. F. Fong, High-Yield Requirement Capture for E-commerce and its Application, International Symposium on Electronic Commerce, May 1999.

[Rao, 1997] Rao D., Behanna Ch., Sun M., Forys F., CORBA Service Test Environment. NEC

Systems Laboratory, Inc., 1997.

[Royce, 1970] W.W.Royce, "Managing the development of large software systems: concepts and techniques", in Proc. WESCON, August 1970.

[Saleh, 1999] Kassem Saleh, The Impact of Architecture on Testability and Testing for Electronic Commerce Systems, International Symposium on Electronic Commerce, May 1999.

[Schneider, 1998] G. Schneider, J. P. Winters, Applying Use Cases: A Practical Guide, Addison Wesley, 1998.

[Schieferdecker, 1997] Schieferdecker I., Stepien B., Rennoch A., PerfTTCN, a TTCN language extention for performing testing, in Testing of Communication Systems Volume 10 (ed. Myungchul Kim, Sungwon Kang, Keesoo Hong), Chapman & Hall, 1997.

[Schieferdecker, 1998] I. Schieferdecker, M. Li, A. Hoffmana, Conformance Testing of TINA Service Components – The TTCN/CORBA Gateway, Fifth International conference on Intelligence in Services and Networks IS&N 98, Antwerp, Belgium, May 25th – 28th, 1998

[SDL, 1999] http://sdl-forum.org/SDL/index.htm .

[SET, 1999] http://www.visa.com .

[Sessions, 1997] Roger Sessions, ObjectWatch Newsletter, http://www.objectwatch.com/issue6.htm , No. 6, 1997.

[Shurety, 1998] Samantha Shurety, E-Business with Net.Commerce, Prentice Hall.

[SUN, 2000] http://java.sun.com

[Telelogic, 2000] Telelogic Tau, http://www.telelogic.se

[Tennenhouse, 1997] Tennenhouse D., Smith D., Sincoskie D., Wetherall D., Minden G., A survey of active networks research. IEEE Communications Magazine, 35, 80-86, 1997

[Vassiliou-Gioles, 1999] Theofanis Vassiliou-Gioles, Ina Schieferdecker, Marc Born, Mario Winkler and Mang Li, Configuration And Execution Support For Distributed Tests, Testing of Communicating Systems Methods and Applications, IFIP TC6 12th International Workshop on Testing of Communicating System, GMD FOKUS, Berlin, 1999.

[Vinoski, 1997] http://www.cs.wustl.edu/~schmidt/corba-overview.html#Vinoski .

[Walter, 1998] T. Walter, I. Schieferdecker, j. Grabowski, Test Architecture for Distributed Systems – State of the Art and Beyond, Germany, 1998

[Woodside, 1995] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software, "IEEE Transactions on Computers", Volumn 44, papes 20-34, Jan. 1995.

[Zwass, 1996] Zwass, V. Electronic commerce: structures and issues. Intern. J. Electr. Comm. 1, 1, 3-23, 1996.

Glossary

API	Application Programming Interface
ASP	Abstract Service Primitives
ATM	Asynchronize Transfer Mode
BTC	Background Test Component
CASE	Computer Aided Software Engineering
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CTMF	Conformance Testing Methodology and Framework
CUT	Component Under Test
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DDCF	Distributed Document Component Facility
DII	Dynamic Invocation Interface
DNS	Domain Naming Service
DOC	Distributed Object Computing
DSI	Dynamic Skeleton Interface
EBBs	Electronic Bulletin Boards
E-commerce	Electronic Commerce

EDI	Electronic Date Interchange
EFT	Electronic Funds Transfer
E-mail	Electronic Mail
EX	Online Currency Exchange
FIFO	First In First Out
FTC	Foreground Test Component
FTP	File Transfer Protocol
GIOP	General Inter-ORB Protocol
GUI	Graphic User Interface
HTML	Hypertext markup Language
HTTP	Hypertext Transfer Protocol
IC	Interface Component
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
ISDN	Integrated Service Data Network
ISO	International Standard Organization
ISP	Internet Service Provider
ITU	International Telecommunication Union
IUT	Implementation Under Test

JINI	JINI is JINI, it is not acronym [SUN, 2000]
JVM	Java Virtual Machine
LT	Lower Tester
MIME	Multimedia Internet Mail Extensions
MSC	Message Sequence Chart
MTC	Mater Test Component
MTS	Microsoft Transaction Server
OBI	Open Buying on the Internet
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Object Management Group
OO	Object Oriented
OOA	Object Oriented Analysis
OOC	Object Oriented Concept
OOD	Object Oriented Design
ORB	Object Request Broker
OS	Operating System
OSF	Open System Foundation
OSI	Open System Internetworking
PC	Personal Computer

PCO	Point of Control and Observation
PDM	product Data Management
PDUs	Protocol Data Units
PICS	Protocol Implementation Conformance Statement
QoS	Quality of Service
RMI	Remote Method Invocation
SDL	Specification and Description Language
SET	Secure Electronic Transaction
SMTP	Simple Mail Transfer Protocol
SSL	Secure Sockets Layer
SUT	System Under Test
TC	Test Component
TCP	Test Coordination Procedure
TCP/IP	Transmission Control Protocol / Internet Protocol
TINA	Telecommunications Information Networking Architecture
TMP	Test Management Protocol
TTCN	Tree and Tabular Combined Notation
UDP	User Datagram Protocol
UML	Unified Modeling Language
UT	Upper Tester

V&V	Verification and Validation
WWW	World Wide Web
XML	Extensible Markup Language