A Formal Specification and Design of an Online Bazaar System

By

June He

A Thesis

Submitted to the Faculty of Graduate Studies at University of Manitoba

in Partial fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

© June He, September 2002

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

A Formal Specification and Design of an Online Bazaar System

BY

June He

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

MASTER OF SCIENCE

June He © 2003

Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilm Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

Abstract

The Bazaar System is an online application system which also provides a framework for a virtual marketplace. Different users can carry out deferent kinds of business activities through the system automatically. The design of a good model for the Bazaar System involves an integrated solution to the issues of distribution, integration, replication and security. Because of the complexity of the system, formal methods are used to specify and design the system's properties.

Object-Z, a formal specification language, is used to specify the Bazaar System's functionality. By using a formal method in the design of the system, we avoid or eliminate any ambiguities, incompleteness, inconsistency, and keeps cost down. In particular, the thesis focuses on the feasibility of applying formal techniques to the design of a Bazaar System. The resulting Bazaar System is applicable as a powerful B2B, B2C and C2C multifunction system.

Acknowledgements

I am grateful to my supervisor, Dr. Sylvanus Ehikioya, for his guidance and encouragement during my research. In particular, I thank him for his patience with me helping me to edit and structure the thesis so that it can be readable. Thanks too for the countless hours he devoted to me, even at the expense of his leisure.

I also wish to thank the members of my examining committee, Dr. Sylvanus Ehikioya, Dr. Peter Graham and Dr. Jose Rueda for agreeing to serve on my examination committee.

Finally, I wish to thank my family for their support.

Table of Contents:

Chapter 1: Introduction ---1
1.1 Benefit of the Bazaar System ---2
1.2 Description of the Bazaar System -- 4
1.3 Contributions of the Thesis --- 4
1.4 Organization of the Thesis --6

Chapter 2: Literature Review --7
2.1 Bazaar System and Online Auctions -- 7
2.2 Technical Needs of the Bazaar System --- 9
2.2.1 Three-tier Architecture ---10
2.2.2 Enterprise JavaBean™ ---11
2.3 Design Issues in the Bazaar System ---12
2.3.1 COM+ or EJB ---12
2.3.2 Why User Entity Beans to Access Databases -----------------------------13
2.4 Formal Methods ---14
2.4.1 The Unified Modeling Language --15
2.4.2 Object-Z ---16

Chapter 3: Requirements Analysis --18
3.1 Identification and Limitation ---18
3.2 Use Case Analysis --19
3.3 The Requirement Specification --27
3.3.1 The Login Subsystem ---27
3.3.2 The Order Subsystem ---33
3.3.3 The Offer Subsystem --37
3.3.4 The Supplier Subsystem --39
3.3.5 The Store Subsystem --41
3.3.6 The Product Subsystem ---43
3.3.7 The Inventory Subsystem --45

Chapter 4: The System Design Specification ---------- 49
4.1 Architecture Design of the Bazaar System ---------- 51
4.1.1 The Basic Objects of the System ---------- 51
4.1.2 The Complex Objects of the System ---------- 57
4.2 Detailed Design Document ---------- 61

Chapter 5: Implementation of the Bazaar System ---------- 68
5.1 The Implementation of EJB at Server Side ---------- 68
5.1.1 Home Interface ---------- 69
5.1.2 Remote Interface ---------- 69
5.1.3 Enterprise Bean Class ---------- 69
5.2 The Implementation of the Administration System ---------- 70
5.2.1 The Main Menu ---------- 72
5.2.2 The User Manager Menu ---------- 73
5.2.3 The Product Manager Menu ---------- 75
5.2.4 The System Manager Menu ---------- 75
5.3 The Implementation of the Front-end ---------- 76
5.3.1 Design Patterns ---------- 77
5.3.1.1 The Mode View Controller Pattern ---------- 77
5.3.1.2 The Front Component Pattern ---------- 78
5.3.1.3 The Value Object Pattern ---------- 79
5.3.2 The Front-End Interfaces ---------- 80
5.3.2.1 The Main Page of Front-End ---------- 81
5.3.2.2 The Home Page of Buyer ---------- 82
5.3.2.3 The Home Page of Store Owner ---------- 83
5.3.2.4 The Home Page of Supplier ---------- 84

Chapter 6: The Conclusions and Future Work ---------- 86
References ---------- 88
Appendix A ---------- 91

List of Figures:

Figure 2.1 High-level Architecture of Bazaar System -------------------------11

Figure 3.1 Use Case Diagram of the Bazaar System ----------------------------20

Figure 3.2 Activity Diagram for Login System ---------- -----------------------20

Figure 3.3 Collaboration Diagram for Search ----------- -----------------------22

Figure 3.4 Collaboration Diagram for Order Products or Services ------------22

Figure 3.5 Collaboration Diagram for Purchases -------------------------------23

Figure 3.6 Collaboration Diagram for Offer Products or Services ------------24

Figure 3.7 Collaboration Diagram for Bidding -------------------------------- 24

Figure 3.8 Collaboration Diagram for Modify ---------------------------------25

Figure 3.9 Collaboration Diagram for Return Products or Services ---------- 25

Figure 3.10 Collaboration Diagram for Communication ---------------------- 26

Figure 3.11 Collaboration Diagram for Watch ----------------------------------26

Figure 3.12-a Formal Specification of LogSys ------------------------------------30

Figure 3.12-b Formal Specification of AccountSys -------------------------------32

Figure 3.13-a The Internal Classes of OrderSys and their Relationship ----------33

Figure 3.13-b Formal Specification of OrderSys ----------------------------------34

Figure 3.14-a The Offer Class and its Relationship to OfferSys ------------------37

Figure 3.14-b Formal Specification of LogSys ------------------------------------38

Figure 3.15-a The Supplier Class and its Relationship to SupplierSys -----------39

Figure 3.15-b Formal Specification of SupplierSys -------------------------------40

Figure 3.16-a The Store class and its Relationship to StoreSys -------------------41

Figure 3.16-b Formal Specification of StoreSys -----------------------------------42

Figure 3.17-a The Internal Classes of ProductSys and their Relationship -------43

Figure 3.17-b Formal Specification of ProductSys --------------------------------44

Figure 3.18-a The Classes and their Relationship in Inventory Subsystem ------45

Figure 3.18-b Formal Specification of InventorySys ------------------------------47

Figure 3.18-c Formal Specification of ItemSys -------------------------------------47

Figure 4.1 The Bazaar System Architecture -------------------------------------49

Figure 4.2 Components Diagram of the Bazaar System ------------------------50
Figure 4.3 The Class Diagram of Account EJB ---------------------------------52
Figure 4.4 The Class Diagram of Mailer EJB -----------------------------------53
Figure 4.5 The Class Diagram of Offer EJB -------------------------------------54
Figure 4.6 The Class Diagram of Order EJB ------------------------------------55
Figure 4.7 The Class Diagram of Product EJB ---------------------------------56
Figure 4.8 The Class Diagram of Inventory EJB -------------------------------56
Figure 4.9 The Class Diagram of Consumer EJB -------------------------------58
Figure 4.10 The Class Diagram of Supplier EJB ---------------------------------58
Figure 4.11 The Class Diagram of Owner EJB -----------------------------------59
Figure 4.12 The Class Diagram of Catalog EJB ---------------------------------60
Figure 4.13 The Class Diagram of Cart EJB -------------------------------------60
Figure 4.14 Database Table and Relationship------------------------------------61
Figure 5.1 J2EE Environment ---68
Figure 5.2 The Administration System --71
Figure 5.3 The Main Menu of the Administration System --------------------72
Figure 5.4 The Main Menu (Pressing BazaarBar1) ---------------------------- 72
Figure 5.5 The Result of Search by Order No. ---------------------------------73
Figure 5.6 Input Boxes of the User Manager ----------------------------------74
Figure 5.7 The Menu of the User Manager -------------------------------------74
Figure 5.8 The Menu of the Product Manager ---------------------------------75
Figure 5.9 The Menu of the System Manager ----------------------------------76
Figure 5.10 The Menu of Subsystems --76
Figure 5.11 MVC in the Bazaar System -- 78
Figure 5.12 The Organization of Page for Front-End ---------------------------80
Figure 5.13 The Screen of Main Page of the Bazaar System -------------------81
Figure 5.14 The Screen of Buyer Home Page of the Bazaar System -----------82
Figure 5.15 The Order Processing ---83
Figure 5.16 The Screen of Store Owner Home Page of the Bazaar System --84
Figure 5.17 The Screen of Supplier Page of the Bazaar System ---------------85

CHAPTER 1
INTRODUCTION

The Internet and World Wide Web represent a foundation on which enterprises are working to build an information economy. Information has value similar to goods and services, and is a vital part of the market. An information economy challenges today's enterprises to re-consider the way they do business [Kas2000]. Information technology assists buyers and sellers to do product brokering, merchant brokering, negotiation, payment and delivery, and other related activities through the Internet. This business activity is called e-commerce [Lom2000].

A Bazaar System is an online application system for e-commerce. The activities of users in the Bazaar System are Internet-related. There are three types of e-commerce models: Business-to-Consumer (B2C), Business-to-Business (B2B), and Consumer-to-Consumer (C2C) [Fri2000] and a Bazaar System can function to provide any of the above three models. In a Bazaar System, sellers offer consumers products through the Internet while buyers can select products by browsing the Internet. The Bazaar System may also involve a business at both ends of a transaction. For example, through the Bazaar System, a purchase manager of a company can place orders to suppliers or a sales manager can offer finished goods to dealers. Ehikioya and Suresh [Ehi2000] defined mechanisms for correct and effective financial services necessary for e-commerce in intangible goods and services. This aspect will not be discussed in this thesis.

The Bazaar System used in this thesis operates in the B2B segment where volume buyers of products meet bulk sellers. The system can be accessed by different kinds of users via different views to do different kinds of activities in the Bazaar System. The companies that sell products or services on the Bazaar System are connected to the suppliers and buyers through the system without having to create point-to-point connections to each of them. The Bazaar System handles the task of integrating large amounts of product information or services from a variety of vendors using different systems. Using the

Bazaar System, the companies can reach new customers, and leverage information from competitors to increase sales.

1.1 Benefits of the Bazaar System

In today's highly competitive business environment, customer service and customer satisfaction are often considered as critical. Customer satisfaction and financial success are two sides of the same coin. With the guarantee of quick processing time, prompt and better customer service, and accurate management information, the Bazaar System can be used to achieve these twin goals. Additional benefits are online access, no time limits, high efficiency, no place limits, and user friendliness.

- **Online access**

The users of the Bazaar System can view information online. The system generates marketing literature, product announcements and public pricing at a high speed and greatly benefits the companies by getting rid of the costs associated with printing and distributing written information. Changes to the information and updates can be made quickly, without significant delays and cumbersome work [Ter1997].

- **No time limits**

A business website is an electronic storefront that never closes. Customers all over the world can order products and access services 365 days a year, 7 days a week, 24 hours a day. The users can access the Bazaar System at times that suit them.

- **High efficiency**

There is a vast amount of information on the Internet, and many tools available today make this information very accessible. Companies can provide all their literature and other pricing-related information on a web server, which provides customers a chance to browse and download the desired information. Companies also have to enable online ordering for their customers and dealers. To target household consumers, companies also have to enable even housewives sitting at home to order goods on the Internet. The Bazaar System fulfills these needs by satisfying the customer using fast online transaction

processing, and by being reliable and easy to use. It also benefits the companies involved by lowering their costs.

- **No place limits**

A customer can access the Bazaar System from anywhere through the same website. A site on the Internet is an inexpensive way to reach a global audience because the number of people using the Internet all over the globe is increasing at a phenomenal rate. A customer does not need to go to different physical places to get different product information.

- **Create revenue opportunity**

The Bazaar System provides automatic business processing, including online order, account management, automatic order/delivery confirmation, order tracking, and on time inventory control, which enable the stores to cut cost and increase revenue. For example, through the Bazaar System, a store manager can update product information and automatically receive related information from consumers. The store manager can watch the order line (a list of items in an order) and inventory online, and send order information to the related product suppliers at once.

- **Implement a new style of business**

Using the Internet, the Bazaar System can change the style of business. For example, traditional shopping activities require a huge effort from the consumer. The consumer spends much time in searching for products or services, comparing prices and other features of a good or service to help make an optimal purchase decision [Mou1998]. The Bazaar System stores many different kinds of information for the various users, so they can get hold of them easily and quickly.

- **User Friendly**

The Bazaar System supports different user interfaces for different kinds of users. A user of the Bazaar System can even select a language used for his/her interface.

1.2 Description of the Bazaar System

The main objective of the Bazaar System is to create an interactive online bazaar that provides a unique shopping experience for the users of the system. The bazaar consists of a number of different areas that are owned by agents or agencies. Entrance into an area can be restricted by the system based on user attributes or by the owner of the area. Each area provides different types of goods and services for sale. Users can buy commodities, custom items or just interact with each other. The bazaar is available to users 24 hours a day, 7 days a week.

The Bazaar System ensures that its users are able to conduct their activities in a safe and secure environment. The system ensures that only authorized users are allowed in the bazaar and that all online interactions between agents are secured. Agents indicate what type of activity is displayed to the other agents within the bazaar.

The Bazaar System interacts with outside systems to conduct business. All online transactions are verified with other financial systems in real-time. The systems of the sellers of goods and services also interact with the bazaar system to provide descriptions of merchandise and services sold.

The Bazaar System has powerful access, administrative and moderator functions. All of these functions are seamlessly integrated into the system for easy administration and access control.

1.3 Contributions of the Thesis

The main objective of this thesis is to investigate the feasibility of applying formal techniques to the design of a Bazaar System. Other contributions include defining a new model of bazaar systems and applying new techniques, such as J2EE (Java 2 platform, Enterprise Edition) and multi-tier architecture, to implement the system.

The Bazaar System is an online application system for e-commerce. It has powerful access, administrative and moderator functions, and it ensures that the users of the system are able to conduct their activities in a safe and secure environment. The Bazaar System guarantees quick processing time, prompt and better customer service, and accurate management information

Formal methods can facilitate specification, design, development, maintenance, reliability, and maintainability of a system, and they have been proven to be effective in developing complex systems. To our knowledge, formal methods have never been applied to a Bazaar system, and thus, this is also a contribution of the thesis. The applications of the formal methods in the thesis include:

- Using UML use case diagrams and collaboration diagrams to model the requirements analysis;
- Using Object-Z to specify the behavior of the Bazaar System;
- Using UML component diagrams and class diagrams to model the architectural design of the Bazaar System; and
- Using formal design style to document the detailed design of the Bazaar System.

A clearly defined model of the Bazaar System is presented in the thesis. The correctness of a software model has been emphasized as one of the major aspects for developing a correct software system. UML is used in modeling, requirements management, analysis and design of the Bazaar System. UML has a well-defined syntax and semantics for its notation, so it is selected for documenting the design of the Bazaar System. Based on different models, such as online market place and auction systems, the Bazaar System is modeled as a powerful B2B, B2C and C2C multifunction system.

The Bazaar System is implemented as a multi-tier Web application using J2EE technology. The J2EE platform provides a multi-tier distributed application model. The thesis emphasizes the benefits of using the new technical skills on bazaar systems.

1.4 Organization of the Thesis

The organization of the rest of the thesis is as follows: In Chapter 2, the background of bazaar systems and the technical requirements to build the system are reviewed. In Chapter 3, a requirements specification of the Bazaar System is outlined. Use case diagrams, activity diagrams and collaboration diagrams in UML are used to do the requirement analysis. A specification of the Bazaar System is given which assumes familiarity with the basic features of the Object-Z notations as given in [Ros1992] and [Duk1991]. In Chapter 4, the design specification is described. Class diagrams, state diagrams and component diagrams of UML are used to illustrate the architecture design and the design specification is elaborated in the detailed design document with design details. In Chapter 5, a prototype implementation is introduced. Chapter 6 contains the conclusion of the thesis and discusses possible future work.

CHAPTER 2
LITERATURE REVIEW

Electronic commerce has provided consumers with more options, more alternatives and more opportunities than ever before. Consumers now enjoy virtually unlimited access to goods and services, access that was heretofore unthinkable. Various online businesses are developing very fast. Consumers now have the ability to purchase goods using their computer, PDA or even a web-enabled portable phone. Generally, when we speak of electronic commerce and the individual customer, we most commonly think of business-to-consumer (B2C) transactions. But one of the most innovative developments in electronic commerce is the rapidly growing consumer-to-consumer (C2C) market.

The Bazaar System can implement business-to-business (B2B) marketplaces. The Internet and World Wide Web have emerged as a valuable networked information source that is increasingly being used for commerce. B2B marketplaces generally are Web sites where transactions between businesses occur. Some bring together a broad variety of buyers and sellers; others address narrower segments of the market, such as a certain genre of products or an individual company and all of its suppliers. The ideal business-to-business marketplace avoids collusion in favor of either buyers or sellers. It manages to get buyers to feel their needs are being served, and nudges companies making the selling as partners and not just as competitor [Fri2000]. The idea behind the Bazaar System is to help users with the negotiation step in the purchase process by making the "best possible deal" on the users' behalf [Cha1996].

2.1 Bazaar Systems and Online Auctions

Auctions are an important market mechanism, which allow selling rare and unusual goods. Auctions apply in situations where a more conventional "market" does not exist or is inappropriate. Online auctions are becoming an increasingly important channel for electronic commerce. Internet based auctions are rapidly diversifying to become products specific. There exist more than 150 online auction sites on the Internet today. The best-

known auction companies are eBay for a wide range of products, CNET for electronic goods, Priceline for air-line tickets, and E*Trade for financial products. The rapid growth of auction sites has created a robust, vibrant online shopping opportunity for online consumers which was never before available. Due to a larger number of bidders and items to be sold auctions on the web are growing fast for the reason that a sufficient match can be found. A bazaar system is also a form of an auction system.

Much work on auction design exists, but most of them concentrate on single items. But usually, multiple non-identical items are offered and a bidder's offer for an item may depend on what other items win. Nisan [Nis2000] presented the Combinatorial Auctions mechanisms. In Nisan's Model, the system allows bids on combinations of items. Compared to single item bidding, combinatorial bidding can meet the desire of the bidders' true preferences, and may lead to better allocation.

A typical B2B online auction system includes modules to manage product inventory, point of sale, ordering, purchasing, receiving, customer management, accounts receivable and payable, general ledger and online credit card processing, along with a Web storefront. Bazaar systems can provide similar services as auction systems. A successful e-commerce system should do more than merely provide an online bazaar where buyers and sellers come to exchange dollars for goods and services. The Bazaar system should be a marketplace that provides services to satisfy the customers every time they visit the site. The Bazaar System should operate as an independent agency, and the services of the system should be designed and provided independently of the people behind the business-to-business site.

Existing auction services normally rely on a central auction server. Such a centralized approach is not appropriate to meet market requirements. As the market grows, a centralized server will perform poorly, since the many users will overload the server making the whole auction process less responsive than sellers and buyers would likely consider acceptable. Further, as the auction market becomes global, the centralized approach cannot effectively cater to variation/regional market regulations and

procedures; different markets may employ their own rules, monetary regulations, payment procedures, etc. Ezhilchelvan and Morgan [Ezh2001] investigate ways of enabling widely distributed, arbitrarily large numbers of auction servers that cooperate in conducting an auction. Each auction server serves a local market and is a part of the global system. Allowing a user to bid at any one of the servers is the principal way of achieving scalability. The Bazaar System follows the same ideas and it is implemented as a distributed system.

2.2 Technical Needs of the Bazaar System

The Bazaar System is Internet-based, and will require distributed transactional applications and server-side technology. One way to meet this need is to use a multi-tier model. Normally, thin-client multi-tier applications are hard to write because they involve many lines of intricate code to handle transactions, state management, multithreading, resource pooling, and other complex low-level details.

In the competitive environment of the information economics, timing has always been a critical factor to adopting new technologies. Organizations need to quickly develop and deploy customer applications. They need ways to simply and efficiently integrate these applications with existing enterprise information system and to scale them effortlessly to meet changing demands [Kas2000]. Using the J2EE technologies [Kas2000], the Bazaar System can meet these goals.

The J2EE is a standard set of Java technologies that streamline the development, deployment, and management of enterprise applications. The J2EE platform offers additional benefits, including [Kas2000]:

- It provides a simplified architecture and development model;
- It is easier scaled to meet demand variations;
- It is easy to integrate with existing information systems;
- It offers a wide choice of servers and tools, and many components are available;
- It provides a flexible security model.

The server side technology of J2EE uses Enterprise JavaBeans™ (EJB) technology. EJBs have maintained unprecedented momentum among platform providers and enterprise development teams. EJB servers reduce the complexity of developing middleware by providing automatic support for middleware services such as transactions, security, and database connectivity. With EJB technology, distributed transactional applications become easier to write because EJBs separate the low-level details from the business logic. Thus, developers concentrate on creating the best business solution and leave the rest to the underlying architecture. We will discuss EJBs in detail in Section 2.2.2.

2.2.1 Three-tier Architecture Model

A three-tier architecture divides an application into three parts that may run on different types of computers: clients, applications servers, and data sources. The client handles the processing of personal productivity applications. Application servers process business applications such as ordering and searching, while the data is contained in a database configuration. The three-tier architecture is decomposed as follows:

- **Client Tier:**

  The user component displays information and processes graphics, handles communications, keyboard input and local applications. These provide maximum portability across different computer platforms and operating systems.

- **Application Service Tier:**

  The application service tier provides a set of sharable, multitasking components that interact with clients, peer services and the data source tier. In the Bazaar System, this tier will be responsible for enforcing business policies, keeping peer services informed of changes and notifying users of important events. It provides a controlled view to the underlying data source.

- **Data Source Tier:**

  The data source tier consists of all the databases contained in the system. The three-tier approach enables system administrators to separate the business logic from

processing logic. This modularity allows business changes to be incorporated more rapidly into applications. New software modules and program objects can be written to work with existing databases, taking advantage of the resident programming logic rather than requiring an entirely new application to be written.

There are three tiers in the *web-based Bazaar System,* the thin-client servlet (a client program that invokes business logic running on the server), the EJB server (the application server), and the database server. Figure 2.1 illustrates this architecture.

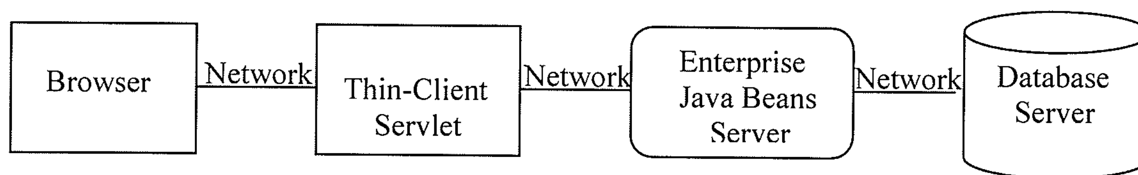


Figure 2.1 High-level Architecture of Bazaar System

2.2.2 Enterprise JavaBeans™

An Enterprise Bean is a simple class that provides two types of methods: business logic and lifecycle. A client program calls the business logic methods to interact with the data held on the server. A container calls the lifecycle methods to manage the Bean on the server. In addition to these two types of methods, an Enterprise Bean has an associated configuration file, called a deployment descriptor, which is used to configure the Bean at deployment time. A container is an entity that provides life cycle management, security, deployment, and runtime services to components. Each type of container also provides component-specific services [Kas2000].

Besides the creation and deletion of Beans, the EJB server also manages transactions, concurrency, security and data persistence. The connections between the client and server, are provided by using the RMI (Remote Method Invocation) and the JNDI (Java Naming and Directory Interface) APIs, and servers can optionally provide scalability through thread management and caching.

Entity Beans and Session Beans are two types of enterprise Beans. The former implements a business entity while the later implements a business task. Typically, an entity Bean represents one row of persistent data stored in a database table. Entity Beans are transactional and long-lived. As long as the data exists, the entity Bean can access and update that data. This does not mean you need a Bean running for every tuple. Instead, Enterprise Beans are loaded and saved as needed. A session Bean might execute database reads and writes, but it is not required. A session Bean might invoke the JDBC calls itself or it might use an entity Bean to make the call, in which case the session Bean is a client to the entity Bean. A session Bean's fields contain the state of the conversation and are transient. If the server or client crashes, the session Bean is gone. A session Bean is often used with one or more entity Beans and for complex operations on the data [Kas2000].

2.3 Design Issues in the Bazaar System

A developer is often faced with the difficult choice of selecting one technology among competing technologies. In this situation, it is important to examine the technologies thoroughly to identify their areas of strengths and weaknesses. In developing the Bazaar System, one such difficult choice is whether to use COM+ or EJB.

2.3.1 COM+ or EJB

COM+ and EJB are the two leading contenders to control the middle tier. Both provide critical support for the business logic that runs today's highly scalable e-commerce systems. COM+ and EJB can be described as Component Oriented Middleware (COMWare). Such COMWare is needed for the Bazaar System [Ses2000].

The two most important differentiating points between EJB and COM+ are programming language restrictions, and portability versus performance. Today, the languages of choice for COM+ and EJB are Visual Basic and Java, respectively. If one selects EJB one must implement the system in Java, otherwise one can use Visual Basic if the choice is COM+.

The other important difference between COM+ and EJB is the choice between portability and performance and, indirectly, the choice between portability and cost. COM+ emphasizes performance while EJB emphasizes portability, and performance is basically the reciprocal of cost. While these two approaches are irreconcilable, each offers distinct advantages. With the portability (EJB) approach, one can develop systems that run on a range of operating systems. The performance (COM+) approach allows one to dramatically reduce overall system cost.

For the Bazaar System, EJBs are the better solution. The Bazaar System is intended to be platform independent, so it can be run on any operating system. Thus, the Java language and EJBs are selected for implementation of the Bazaar System.

2.3.2 Why use Entity Beans to Access Databases?

In most examples of applying J2EE system, Entity Beans are used to access databases directly. Session Beans and other objects need to call Entity Beans to access the databases. The benefit of using Entity Beans to hold data is to maintain consistency of data in the Bazaar System. However, the drawback is that it is time consuming. In some cases, we can use session Bean to access the databases directly.

Occasionally, a distributed application will need to update multiple server objects simultaneously. But updating a large number of server objects one at a time can be very expensive, with each update requiring one or more remote method invocations. In the case of a client program accessing Entity beans, for example, the client acquires a list of primary keys from an EJB finder method, and then individually updates the enterprise beans through those Beans' remote interface. Unfortunately, the server has to create and perform transaction management for each of these instances, resulting in unacceptable overhead. One solution to this problem is to use the Batch Session Bean pattern, which encapsulates the transaction to be performed on multiple server objects into a server-side stateless session bean. This object receives a collection of objects to be used as parameters for updating the collection of Entity beans [Kas2000].

2.4 Formal Methods

The Bazaar System is a web-based, distributed system. The design of the Bazaar System is very complex because it involves several areas and many different techniques. Using formal methods to develop the Bazaar System can help to avoid ambiguities, incompleteness, inconsistency, and keep cost down. Using formal methods provides a good solution for the fundamental issues in the Bazaar System design. Formal methods ensure quality and hence increase reliability of the Bazaar System.

The strength of formal methods is that they can be used to deduce properties of a system and reason about the consistency of a design. The benefits of adapting formal techniques include [O-Z1997]: "improve the correctness of the description, reduce ambiguities and inconsistencies, and increase the readability of the description". The architecture of the metamodel is validated by a complementary technique.

A formal specification is a minimal description of a system in a mathematical notation, usually based on algebra or a calculus. For example, Z is a calculus based on set theory and first order predicate calculus, and is usually not executable [Kre1997]. Formal specifications are precise, clear, unambiguous, provable, and one can reason in formal specifications to produce inference. Formal specifications *should* be used for safety critical systems, security systems, the definition of standards, transaction processing systems. Generally, formal specifications should be used for anything that is hard, complex, or critical [Kre1997]. "Although the Bazaar System is not a safety critical system, it requires that the operations it performs be correct and that only authorized users can access critical components of this system because any error or uncontrolled access could lead to huge financial losses. In this respect, formal methods act as a quality assurance mechanism for enforcing the correctness and security requirements of the Bazaar System" [Ehi2001]. Object-oriented formal specifications can be understood at two conceptual levels, they can be viewed as a black box, whose observable properties are described by the entire specification, and as a white box containing objects whose individual properties are described by the class of the specification to which they belong.

2.4.1 The Unified Modeling Language

UML (Unified Modeling Language) is a standard notation for objected-oriented modeling. UML is a semi-formal language, and it helps users to develop software iteratively and visually model software. UML will be used in modeling, requirements management, analysis and design of the Bazaar System. UML saves a lot of time in developing a big, complex system as the Bazaar System. UML help to generate a clear document, therefore shorten the design phase.

The Unified Modeling Language (UML) is a language for modeling information systems and software artifacts. UML can be used to visualize, specify, construct, and document knowledge about software-intensive systems and their purpose at an abstract level. UML was accepted as a standard object-oriented modeling language by the OMG (Object Management Group) in 1997, and is becoming the dominant modeling language in the industry [UML1999].

In addition to its specific use as core of the analysis and design model, UML's breadth and high level of abstraction makes it an excellent base model from which other models and submodels can inherit. The syntax and semantics of notations provided in UML are defined in terms of its metamodel. In the metamodel, there are three distinct views: abstract syntax in UML class diagrams, static semantics (ensuring that all UML constructs are statically well formed) and dynamic semantics (specifying the meaning of the constructs). UML covers such concepts as types, classes, components, packages, diagrams, methods, operations, relationships, attributes, and constraints.

In UML notation, icons, 2-d symbols, paths and strings are used as graphical constructs. Additional diagram elements include mappings, names, labels, keywords, expressions and notes. The element properties, in a general sense, mean any value attached to a model element, including attributes, associations, and tagged values.

In this thesis, four UML diagrams will be used:

- Use case diagrams which are used to capture requirements;

- Collaboration diagrams which show, in a spatial view, how objects collaborate to perform use cases;
- Class diagrams representing the system structure; and
- State diagrams that describe the local behavior of classes.

Use cases help in specifying the functionality of the system. They also require the analyst to think about and define what actors are using the system. This information is used to construct the system context diagram that gives a structural overview of the system's environment.

2.4.2 Object-Z

Object-Z will be used for the formal specification of the Bazaar System. The use of Object-Z enables system designers and programmers to create, understand and maintain specifications of complex interfaces and their interaction. It gives practitioners a better understanding of the design process and fosters the usefulness of formal methods.

Object-Z is an object-oriented formal specification language. Object-Z is an extension of Z which facilitates specification in an object-oriented style. The major extension in Object-Z is the class schema, which captures the object-oriented concept of a class by encapsulating a single state schema, and its associated initial state schema, with all the operation schemas which may change its variables. Classes may be incrementally specified using Object-Z's notion of inheritance, which enables definitions from one class to be included in another via inheritance, and this feature significantly improves the clarity of large specifications [O-Z1997]. Object-Z has been applied to a range of case studies including a mobile phone system, communication protocols, a buttons console and the denotation semantics of programming languages [Ste1992].

The main features of Object-Z include class and inheritance [Ros1992]. Classes in Object-Z are the major modeling construct for specifying a system. A class has a common state structure and operations for describing an object. The state is composed of

attributes. Syntactically, a class in Object-Z is a named box composed of a visibility list, type and constant definitions, a state schema, an initial schema and operation schemas.

Using inheritance, classes in Object-Z can be used to define other classes. A class can inherit from one or several classes. The types and constant definitions of the inherited classes and those declared in the derived class are merged. Operations with the same name are conjoined, and operations can be renamed. Classes can be instantiated in other classes as attributes.

In Object-Z, instantiation is used as a mechanism for modeling relationships between objects, which in UML is modeled using association. Objects which instantiate other classes as their attributes can refer to the objects of the instantiated classes. The values of these attributes are object-identities of the referenced objects.

A system specified using an Object-Z consists of a collection of objects. The objects collectively denote some system state. A class defines the types of state variables and behavior of constituent objects. An object's value is the underlying value of that object at some point in its evolution. This comprises the value of the object's state, together with a list of operations that are enabled for this value of the object's state.

CHAPTER 3
REQUIREMENTS ANALYSIS

This chapter describes the requirements of the Bazaar System using formal methodology. UML diagrams are used to specify the requirements of the Bazaar System and Object-Z is used for the specification of the requirements.

3.1. Identification and Limitations

The Bazaar System is a complex system. It is necessary to make some limitations to implement the system within the constraints of a Master of Science thesis. The following assumptions and definitions are made:

- A user is a person who can access the Bazaar System, including Owners, Consumers (individual), Wholesale Buyers, Suppliers, and System Managers.
- Each user has a unique identifier.
- The manager of the Bazaar system assigns the rights for each type of users.
- The type of a user determines the access level of the user.
- The owner of a store specifies the rights and limits for a user to access the information of that store.

For each kind of user, the rights and limits are as follows:

Owners:

Owners are agents or agencies who own stores that are accessible via the Bazaar System. They have the highest priority (access right) level in the bazaar system. All information about the stores is accessible to the owner. They can communicate with any other user directly through the bazaar system. They have the right to indicate what type of activity is displayed to other users within the bazaar. Each owner has the right to modify information about their own store (e.g., inventory of products, product policy, etc.).

Consumers (individuals):

Consumers buy goods or services from the bazaar system. They can see related information on some kinds of goods or services, place an order; give suggestions, etc. They must be able to initiate transactions and provide details of transactions such as credit vs. cash, amount to be paid, etc.

Wholesale buyers:

Wholesale buyers are specialized consumers that must be able to buy quantities in bulk (large amounts). Wholesale buyers must declare themselves as wholesale buyers at the time of purchase. A consumer who buys a large quantity of products is not considered as a wholesale buyer unless he/she is registered with the bazaar. Registering as a wholesale buyer enables the wholesale buyer to get any special offers, discounts or privileged reductions that are generally not available for all consumers.

***Suppliers*:**

Suppliers are allowed to see information of the stores to which they supply goods. They give an offer or quotation when there is a competitor's product.

***System manager*:**

The System manager creates and assigns privileges to other users who access the bazaar. He creates the products and services information, and is also responsible for maintaining the system. All the information in the system is accessible to him.

3.2. Use Case Analysis of the Bazaar System

The Bazaar System can support many operations. Each operation is related to a use case. The use case relationships between users and functions are illustrated in Figure 3.1.

Before a user can perform any function in Figure 3.1, he must login into the Bazaar System. The login operation is illustrated in Figure 3.2.

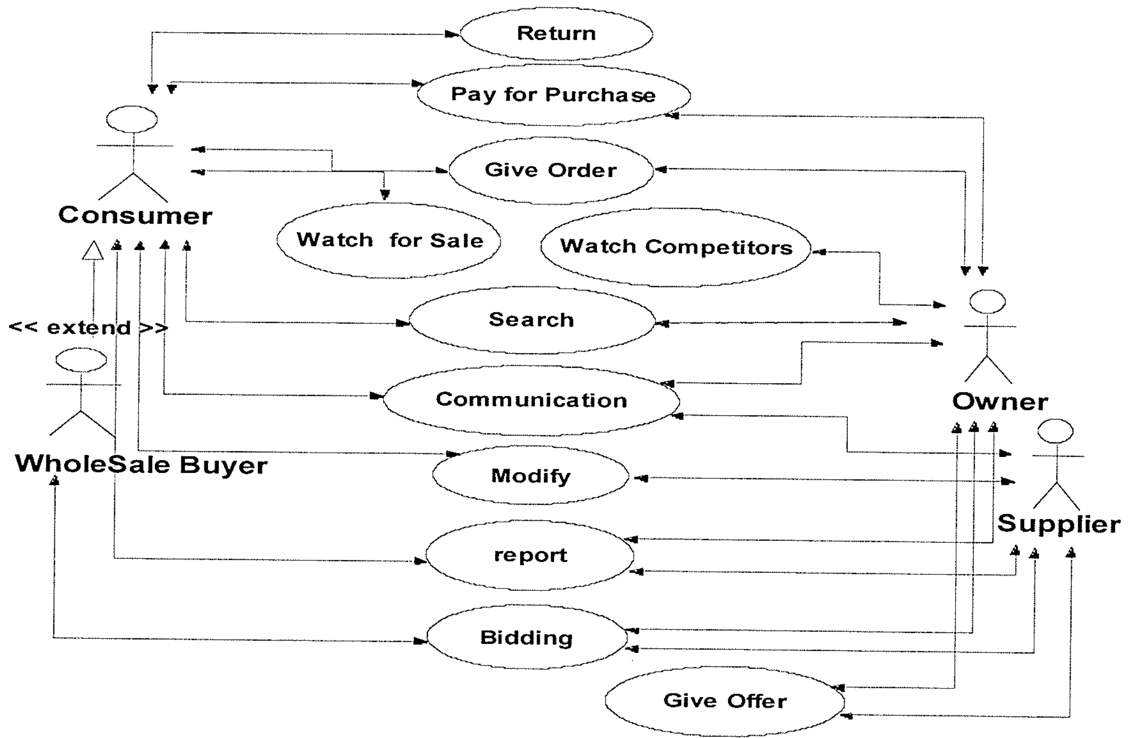


Figure 3.1 Use Case Diagram of the Bazaar System

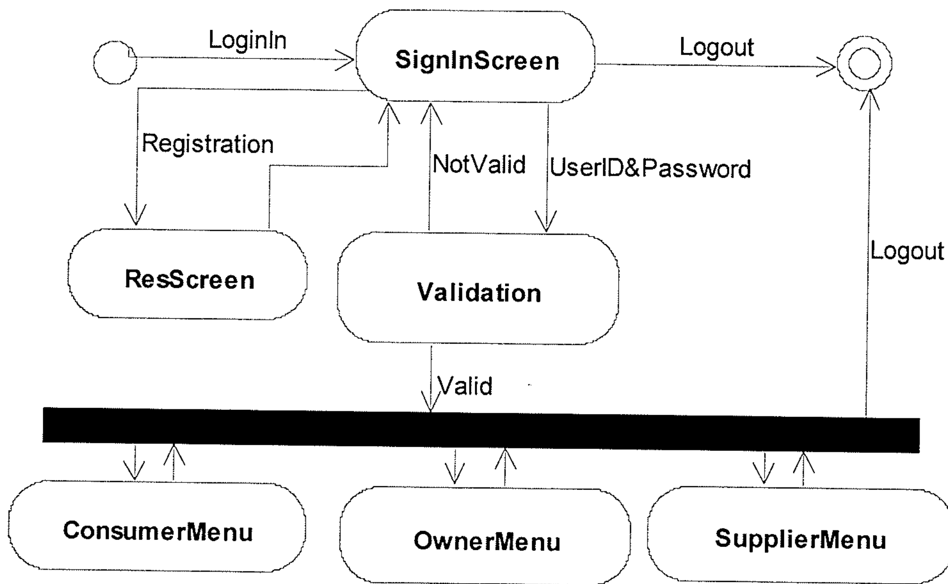


Figure 3.2 Activity Diagram for Login System

There are eight use cases for owners, including **Search**, **Modify** (access rights, inventory, store information), **Communication** (with wholesale buyers, suppliers, employees), **Watch** (competitors), **Report** (monthly income, daily income, products in store), **Order** (give order to suppliers), **Bidding** (with supplier), and **Pay for purchase** (to supplier).

There are six use cases for consumers (individual) and one extra function for wholesale buyers, including **Search**, **Watch** (for sale), **Order** (give order to suppliers), **Bidding** (for wholesale buyers only), **Return** (products), and **Pay for purchase** (to supplier).

There are six use cases for suppliers, including **Search**, **Offer** (give offers to store owners), **Modify** (product list, price), **Bidding**, **Report**, and **Communication**

In the following discussions, each case shown in Figure 3.1 is explained in detail so that a proper understanding of the operations involved in the use cases should become clear.

Case 1: **Search**

Every user can search for information in the Bazaar System that is accessible to them. The Bazaar System responds to search requests according to the type of users and the searched source. Figure 3.3. describes the search case in detail. The search case provides the following functions for a user in the system:

- Search Stores by Store name, manager name, products and prices.
- Search Products or Services by name, supplier, store name(s), and selling prices.
- Search People (employee, owner, manager, wholesale buyer) by title, address, and contact.
- Search Suppliers by supplier name, supplied products (or service), and contact.

Case 2: **Order Products or Services**

When receiving an order from a user, who is a consumer or a storeowner, the system creates an order instance, and a set of instances of order lines. The processing includes

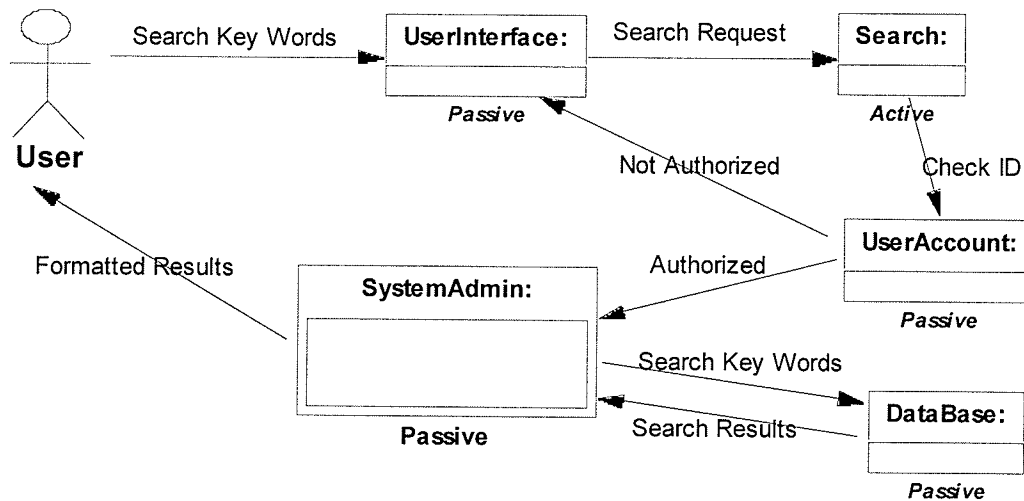


Figure 3.3 Collaboration Diagram for Search

validation checks to determine whether the customer is known or new and the composition of order lines. After an order is created, the order details, including product id and description, supplier name and store name, etc. can not be changed. The customer details (e.g., address) may be changed at any time, and the product quantity can be changed too. If an order has no order lines that can be met (that is, all store entries are zero), the order is required to be returned to the customer with some suitable covering note and the order instance is deleted. The details of the order case are described in Figure 3.4.

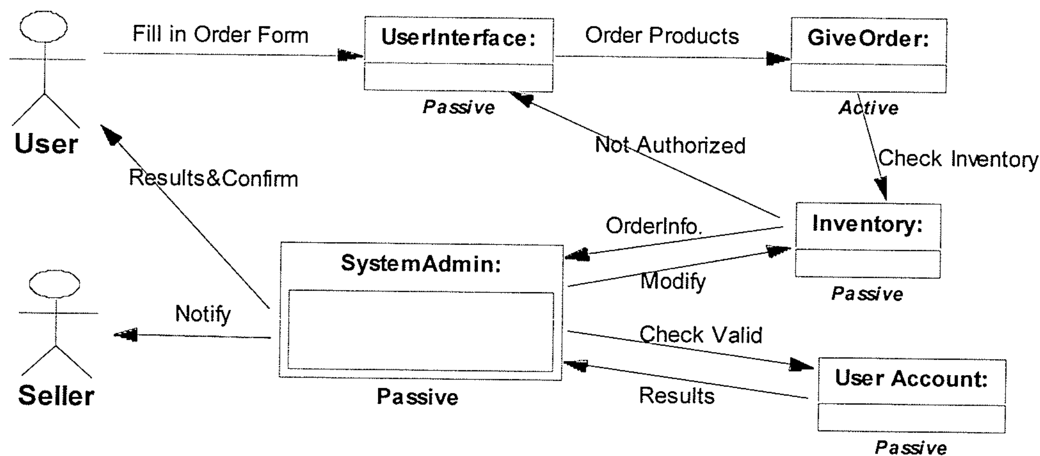


Figure 3.4 Collaboration Diagram for Order Products or Services

Case 3: **Paying for Purchases**

After an order is accepted, the buyer can select payment of amount one or more times. The buyer fills out a form including Order ID, paying amount, credit number, etc. The system creates/modifies buyer's related information. The seller will check the buyer's account and send back the balance of his/her account to the buyer. The system will then modify the status of the order and send all the necessary information to the related seller. The details of paying for purchases case are described in Figure 3.5.

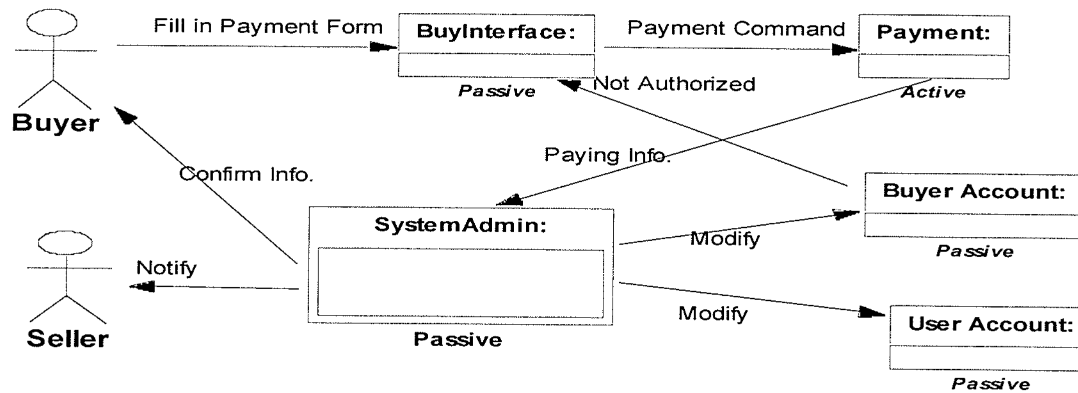


Figure 3.5 Collaboration Diagram for Pay for Purchases

Case 4: **Offer of Products or Services**

When receiving an offer from a supplier, the system creates an offer instance and sends the offer information to the store managers involved. If a store manager accepts the offer, the system creates a set of instances of offer line for the store manager. As with an order, most details can not be changed after an offer line is created, but the shipping address and the quantity can be changed at any time. The processing includes validation checks to determine whether the product is stored in the system or is new. If the product is new, the system manager must add the product first before the offer is created. The details of the offer of products or services case are described in Figure 3.6.

Case 5: **Bidding**

A buyer (a wholesale buyer or an owner) can execute a *Bidding* function. The buyer completes a bidding form (including user id, item id, quantity, and offering price), the system checks the user's preference and the policy of the product. If the bid is acceptable, the system will send the information to the buyer immediately and request the buyer to

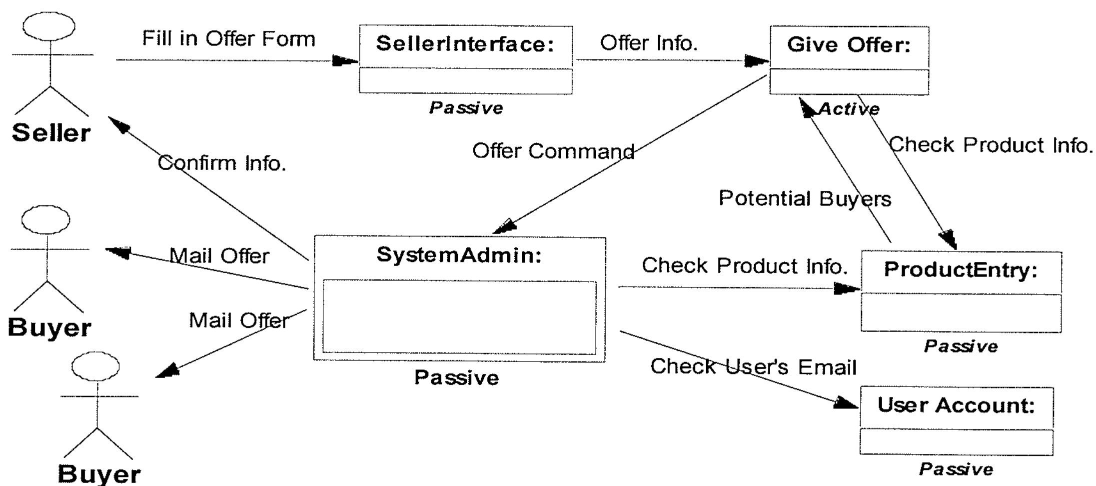


Figure 3.6 Collaboration Diagram for Offer Products or Services

fill in an order form. If the bidg could not be decided by the system, it will be sent to the store manager and the system requests the user to wait. The system will send the seller's decision to the user. The details of bidding case are described in Figure 3.7.

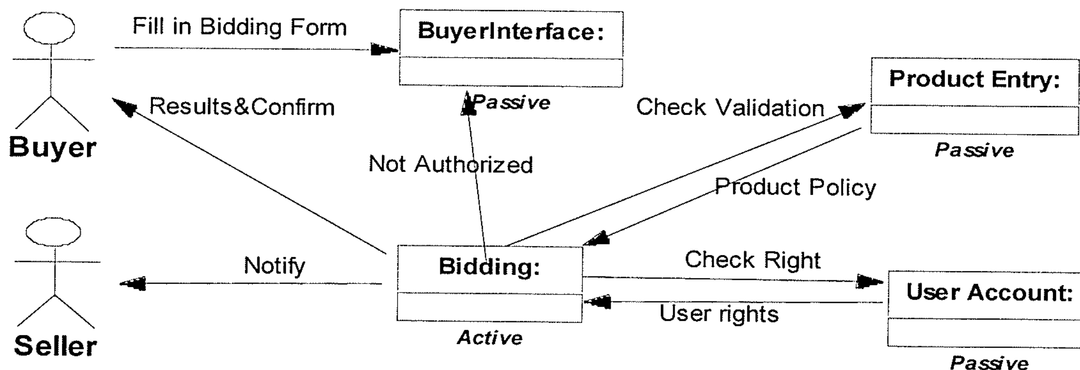


Figure 3.7 Collaboration Diagram for Bidding

Case 6: Modify Information

Owners and suppliers in the Bazaar System have the rights to modify relevant information in the System. Owners of stores can modify the prices, product list, and inventory of the products/services in their stores, and can also modify user access rights relevant to their stores. Suppliers can modify their product prices and product list. The details of modify information case are described in Figure 3.8.

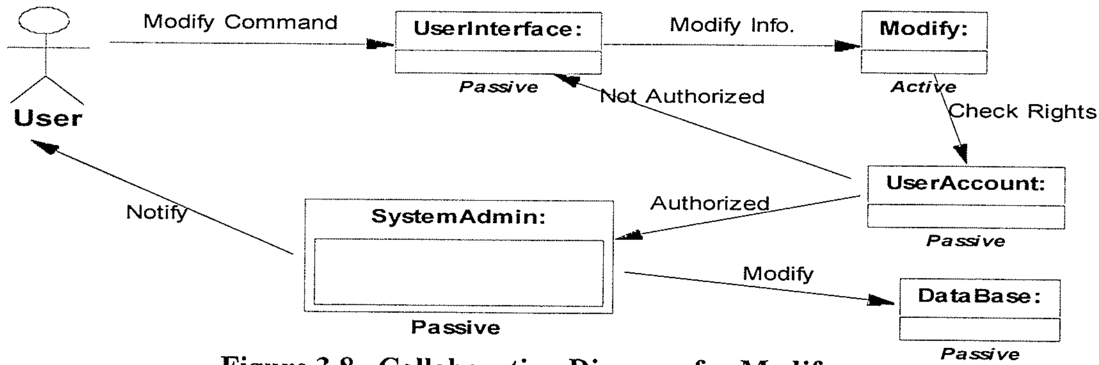


Figure 3.8 Collaboration Diagram for Modify

Case 7: **Return Products**

A user (a wholesale buyer, or a storeowner) can select the *return* function. The user fills in a form providing returned products information (including product name, data, quantity, price, order ID.). The system checks if the return is valid. If valid, a confirmation is sent to the user and to the related seller and the product inventory is modified. If not, the system informs the user that the return is not acceptable and provides an explanation. A policy (as well as other constraints) section needs to be developed, which indicates how long after purchase a buyer can return a product. The details of return product case are described in Figure 3.9.

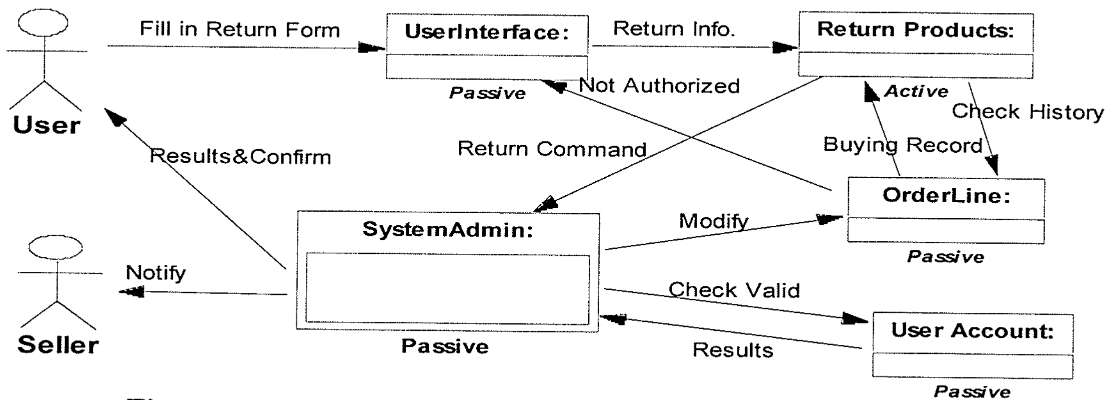
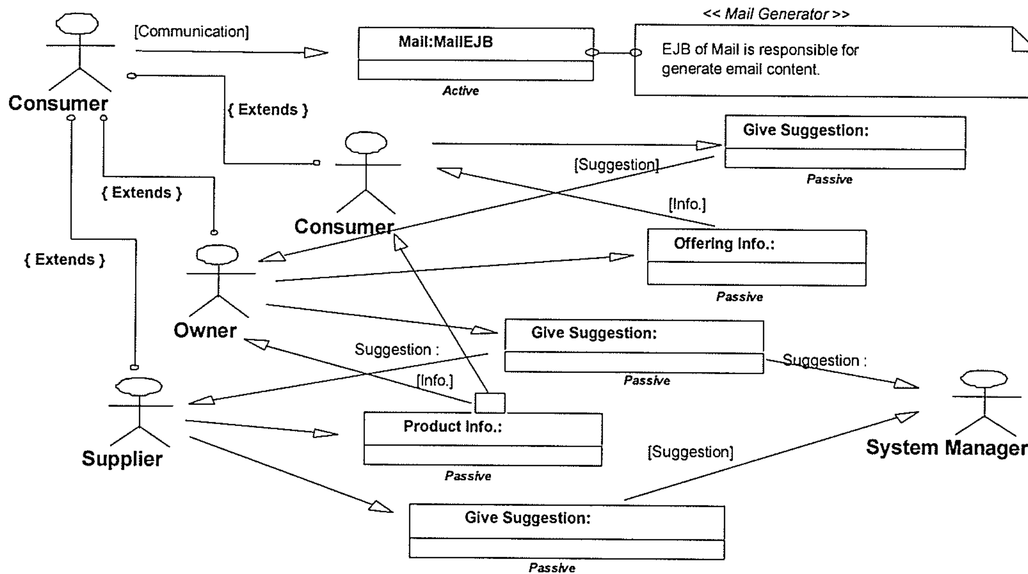


Figure 3.9 Collaboration Diagram for Return Products or Services

Case 8: **Communication**

A user in the system can communicate with another user by sending email. For example, a consumer can give suggestions or send requests to store owners or suppliers. Also, a store manager can send product information to consumers, send requests to suppliers, and send suggestions to the system manager of the Bazaar System. Similarly, a

supplier can give supplied product to store managers, and send suggestions to the system manager. The details of communication case are described in Figure 3.10.



3.10 Collaboration Diagram for Communication

Case 9: **Watch**

Owners can watch competitors through the system and consumers can watch for sales. The system collects and organizes the competitors' information for each storeowner, including the price and sale policy of the same product from other stores, and the supplier list for a designate product. The details of the watch case are described in Figure 3.11.

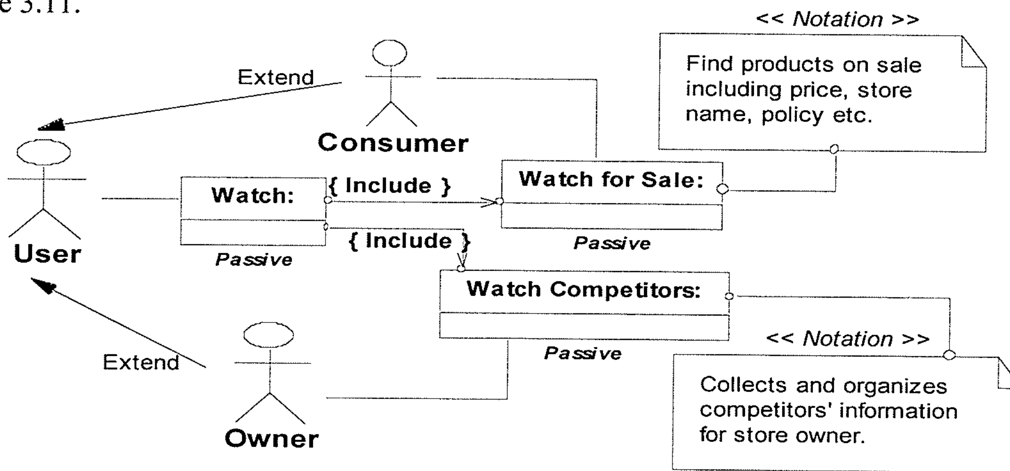


Figure 3.11 Collaboration Diagram for Watch

3.3. Specification of the Requirements

In this section, we describe the requirements of the Bazaar System using the formal specification language, Object-Z, and provide a simple relationship in UML to demonstrate the Object-Z state space of each object. Recall (see Section 2.4.2) that Object-Z is an extension of the Z language. A Z specification defines a number of state schemas, and inferring which operations may affect a particular state schema requires examining the signatures of every operation. In contrast, Object-Z associates individual operations with one state, and facilitates specification in an object-oriented style.

There are several kinds of tables in the Bazaar System, and each table has an identified name. The bazaar system is required to maintain a database of these tables. The following basic types are used in the specification:

[*Userid, UserInfo, Id, OrderE, OrderLineE, Double, String*]

Userid is used to identify a user. *Id* can be an item id, an order id, or an orderline id. *UserInfo* includes user name, contact, credit card, and other information. *OrderE* is used to hold information for an order, while *OrderLineE* is used to hold an *orderline's* information. *Double* and *String* represent the usual data types for real and string quantities respectively.

The system performs some operations according to the needs of different kinds of users. All users must 'enter the system' first, then do some operations, and finally 'exit the system'. Before a user becomes an identified user, he/she must execute the 'registration' operation except a general consumer who only browses the system for information. So, the first component of the system is the Log subsystem.

3.3.1 The Login Subsystem

A login subsystem maintains a set of accounts, one for each user of the system. Each account consists of user id, password, and user information. The 'id's of users must be unique in the system. There are several key operations in the system, such as, 'access an account', 'delete an account', 'change password', and 'change user type'.

Every user has an account, including user id, password and other user information.

$$Account \mathrel{\widehat=} [userid : Userid, passwd : String, info : UserInfo]$$

We assume every user has a unique user id. If two accounts have the same id, the two accounts are identical. The state space is described by the following schema.

$$\begin{array}{l} \textit{LogSys} \\ \hline users : \mathbb{P}\, Account \\ \hline \forall u1, u2 : Account \mid u1 \in users \land u2 \in users \bullet \\ \quad u1.userid = u2.userid \Leftrightarrow u1 = u2 \\ \hline \end{array}$$

The state invariant of *LogSys* asserts that if two accounts have the same user id, then the two accounts are identical.

There are two classes in the login subsystem, *LogSys* and *UserAccount*. The specification of an operation requirement is given in the following format:

[Name] ([input parameter(s)]) [outputparameter(s)]
Precondition
Postcondition

An operation can have one or many input parameter(s) and output parameter(s). Sometimes, it has no input or output at all (i.e., the input parameters and output parameters are optional). Each operation requirement is described by two assertions - the precondition and the postcondition. Alagar and Periyasamy [Ala1998] provide the definitions of these two assertions. The precondition for a requirement asserts what must be true before any function or operation implementing the requirement is invoked. The postcondition asserts what must be true after the function or operation that implements the requirement terminates.

The LogSys Class:

There are three operations in the *LogSys* class: login, logout, and change password.

logIn (u: Userid, p: String)

Precondition :

No other user in the system has the same user id as the user with user id (*u*) in the active accounts (*active*) and the user's password exists in the system.

Postcondition

The state space of active users is modified to include the new account, and the status of the user is set to active.

logout (u: Userid)

Precondition

The user in the system has the same user id as the user with user id (*u*) in the active accounts (*active*) and the user's password exists in the system.

Postcondition

Remove the user with user id (*u*) from the active accounts.

changePassword (oldPasswd, newPasswd, confirmPasswd: String)

Precondition

The user with *oldPasswd* exists in the system, and is in the active accounts.

Postcondition

The user's password (whose name matches the input) has been changed to the new password supplied as input (*newPasswd*).

The formal specification of the class *LogSys* is shown in Figure 3.12-a.

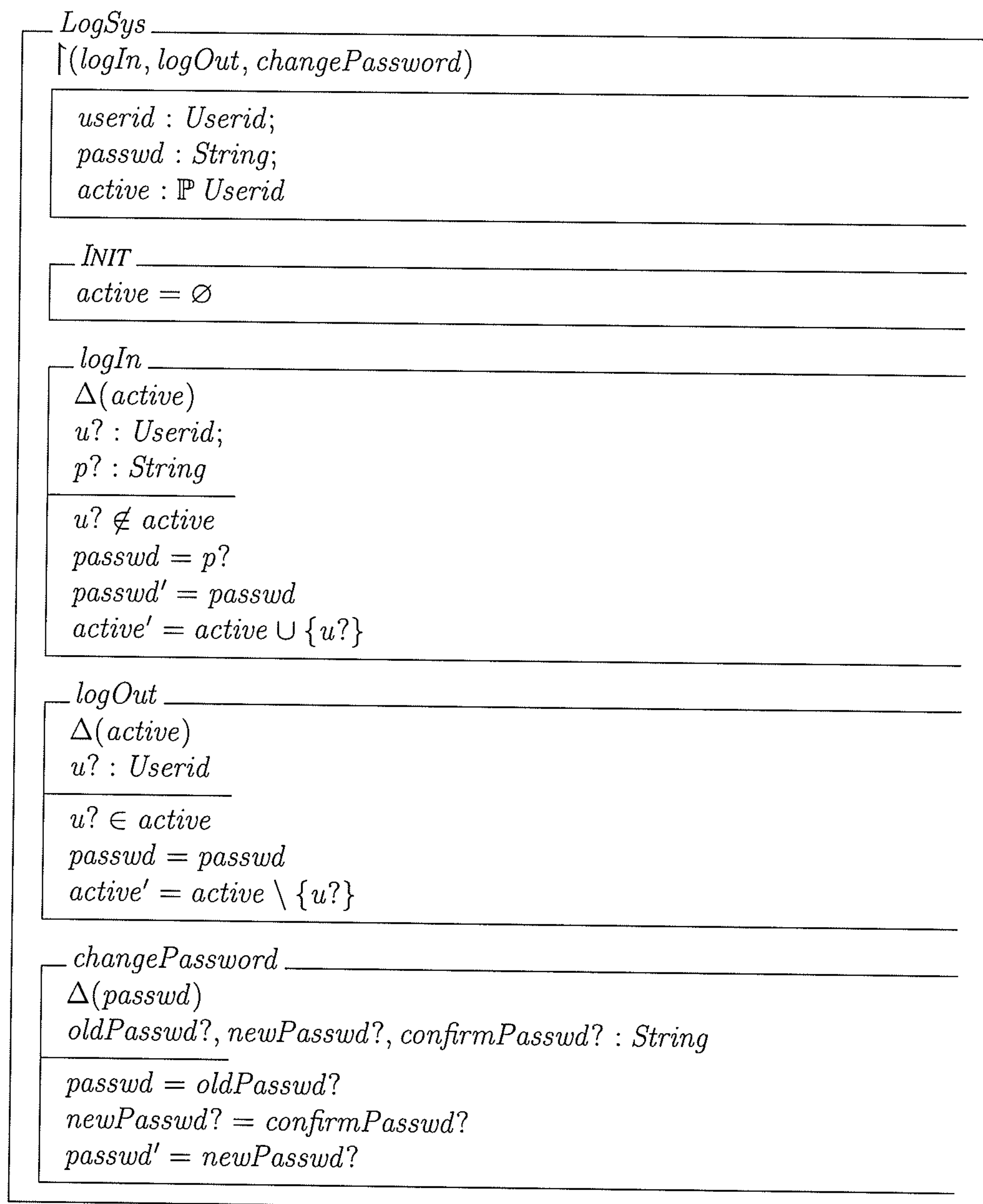


Figure 3.12-a Formal Specification of LogSys

The UserAccount Class:

The *UserAccount* class has four methods. These are to 'add an account', 'delete an account', 'change a user's information', and 'get a user's information'.

addAccount (*a*: *Account*)

Precondition

No other user in the System has the same userid as the specified user (*a.userid*).

Postcondition

The user's account is added to the accounts set (i.e., ensures that the set of accounts in the system is modified to include the new account).

DeleteAccount (*userid: Userid*)

Precondition

The account with the specified userid exists in the system.

Postcondition

The account whose userid matches the input (*userid*) is deleted from the system.

changeUserInfo (*userid: Userid, userInfo: UserInfo*)

Precondition

The user with the specified userid exists in the system.

Postcondition

The user's information in the account is changed to the new information (userInfo).

GetDetails (*userid: Userid*)

Precondition

The user with the specified userid exists in the system.

Postcondition

The user's information is returned as the output.

Figure 3.12-b shows the formal specification of the class UserAccount

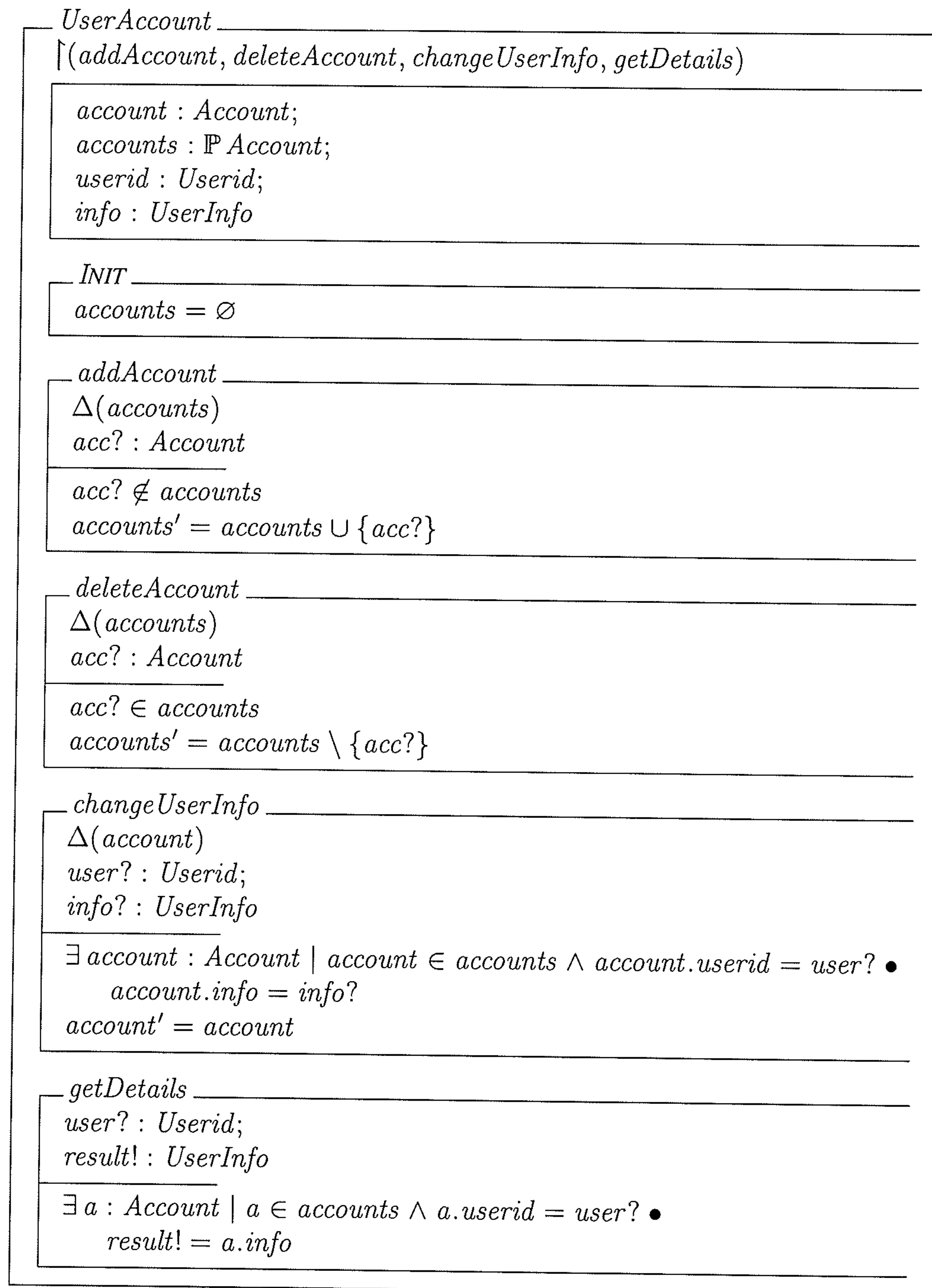


Figure 3.12-b Formal Specification of UserAccount

3.3.2 The Order subsystem

Users place order that will be processed by the system. When receiving an order from a user, the system creates an order instance, and a set of instances of order line. This processing includes validation checks to determine whether the customer exists or is new and on the composition of order lines.

After an order is created, the order details, including product id and description, supplier name and store name, etc, can not be changed. However, as before, the customer details, especially the address, may be changed at any time, and the quantity of each product can be changed too. An order instance is processed by processing each of its order lines. The details of the processing are not included here, but involve checking that at least some of the order lines can be met and that an order line is not for the same product as another order line on the same order. If an order has no order lines that can be met (that is, all store entries are zero), the order is returned to the user with a suitable message and the order instance is deleted.

The order subsytem has only one class, *OrderSys*. The *OrderSys* class contains two internal classes: Order and OrderLine (Figure 3.13-a). The operations of the class, create order, add order line, update order, remove order, and get order details, are specified in Figure 3.13-b.

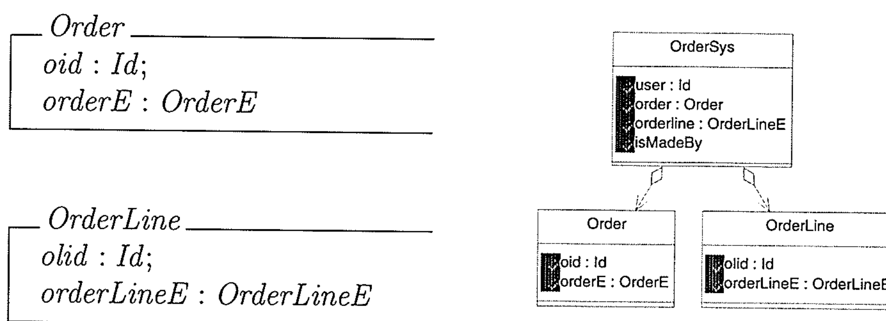


Figure 3.13-a The Internal classes and Relationship to OrderSys

OrderSys

$\upharpoonright(createOrder, addOrderLine, deleteOrderLine, updateOrder,$
$removeOrder, getDetails)$

$$
\begin{array}{l}
user : Userid; \\
oid : Id; \\
olid : Id; \\
order : Order; \\
orders : \mathbb{P}\, Order; \\
orderLine : OrderLine; \\
orderLines : \mathbb{P}\, OrderLine; \\
isMadeBy : Id \leftrightarrow Userid
\end{array}
$$

INIT

$$orders = \varnothing$$

createOrder

$$
\begin{array}{l}
\Delta(orders, orderLines) \\
order? : Order; \\
orderLine? : OrderLine; \\
user? : Userid; \\
olid! : Id; \\
oid! : Id \\
\hline
order? \notin orders \\
orders' = orders \cup \{order?\} \\
orderLine? \notin orderLines \\
orderLines' = orderLines \cup \{orderLine?\} \\
isMadeBy' = isMadeBy \cup \{oid! \mapsto user?\}
\end{array}
$$

addOrderLine

$$
\begin{array}{l}
\Delta(orderLines) \\
oid? : Id; \\
orderLine? : OrderLine; \\
olid! : Id \\
\hline
\exists\, o : order \mid o \in orders \land o.oid = oid? \bullet \\
\quad orderLine? \notin orderLines \\
orderLines' = orderLines \cup \{orderLine?\} \\
order' = order \\
isMadeBy' = isMadeBy
\end{array}
$$

deleteOrderLine

$\Delta(orderLines)$
$olid? : Id$

$\exists\, orderL : OrderLine \mid orderL \in orderLines \wedge orderL.olid = olid? \bullet$
$\quad orderLines' = orderLines \setminus \{orderL\}$

updateOrder

$\Delta(order)$
$order? : Order;$
$oid! : Id$

$order? \in orders$
$order' = order?$

removeOrder

$\Delta(orders)$
$oid? : Id;$
$user? : Userid$

$\exists\, order : Order \mid order \in orders \wedge order.oid = oid? \bullet$
$\quad orders' = orders \setminus \{order\}$
$isMadeBy' = isMadeBy \setminus \{oid? \mapsto user?\}$

getDetails

$oid? : Id;$
$result! : OrderE$

$\exists\, order : Order \mid order \in orders \wedge order.oid = oid? \bullet$
$\quad result! = order.orderE$

Figure 3.13-b The Specification of OrderSys

Data Modeling:

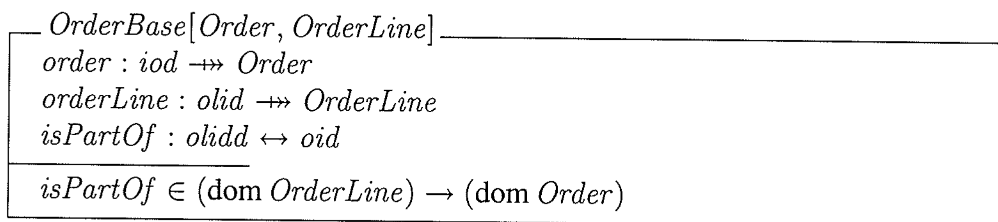
In high level Object-Z presentation, the data modeling entities that make up the state are modeled as functions from some unique identifiers to the instance. In the order subsystem, for example, we considered three entities: user entity (*UserInfo*), order entity(*OrderE*), and orderline entity(*OrderLineE*). These three entities have been defined as basic types already (see Section 3.3).

Schema of User:

The user entity (*UserInfo*) is independent of other entities. We model it by using simple schema, mapping user identifiers (*Userid*) to user instances.

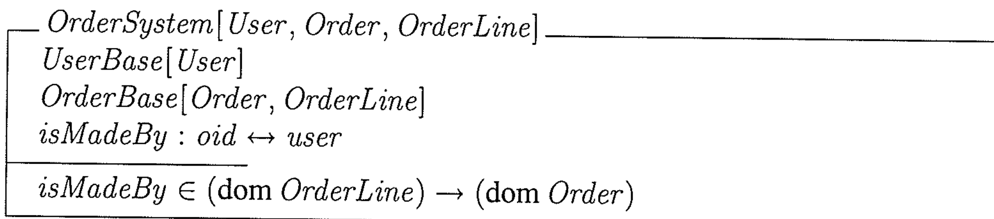
Order Schema:

An order must have some order lines; an order line must be related to one order. The order and order line entities are, therefore, dependent, and the relationship *isPartOf* exists between them.



Order System

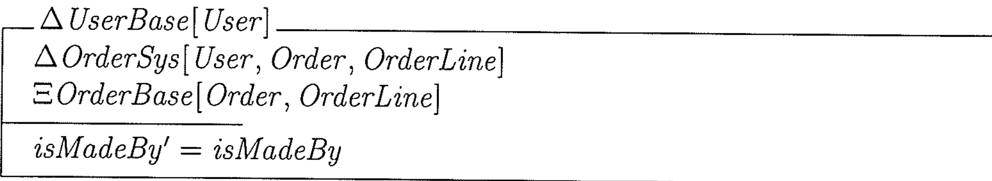
An order line is related to an order instance; each order instance is related to at least one order line. The full system state includes these two substate components, and the *isMadeBy* relationship between them.



Analysis of Operations in the Order system:

The events that affect the Order system are the receipt and deletion of orders, and the creation, modification and deletion of users. Having specified the state as independent substates, it is userful to define schemas for updating these substates. When we update on the user substates, we do not change the *isMadeBy* relation; when we update the order part, we may change this

relationship. The change can be described as:



Another operation that can be used to change the details of entity instances is to change an order entity or change an order line entity. Since the two entities are related, such an operation needs to change the two related instances of the two entities at the same time. The order's relationship does not change.

3.3.3 The Offer subsystem

A user (usually a supplier) provides an offer to supply a product or service, which will be processed by the system. The system receives events to create, modify and delete elements of tables in the database in response to processing an order. After an offer is created, the offer details, including product id and description, supplier name, and store name, can not be changed. However, it is possible to change the offer details, especially the price, at any time, and the number of the offer items can be changed too.

The offer subsystem has only one class, *OfferSys*. The *OfferSys* class contains an internal class *Offer*. Figure 3.14-a shows the *Offer* subclass and relationship to *OfferSys*.

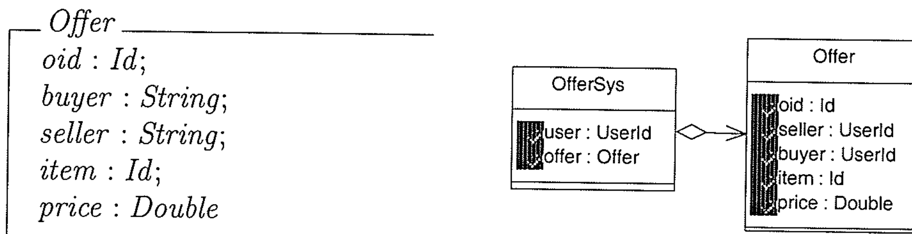
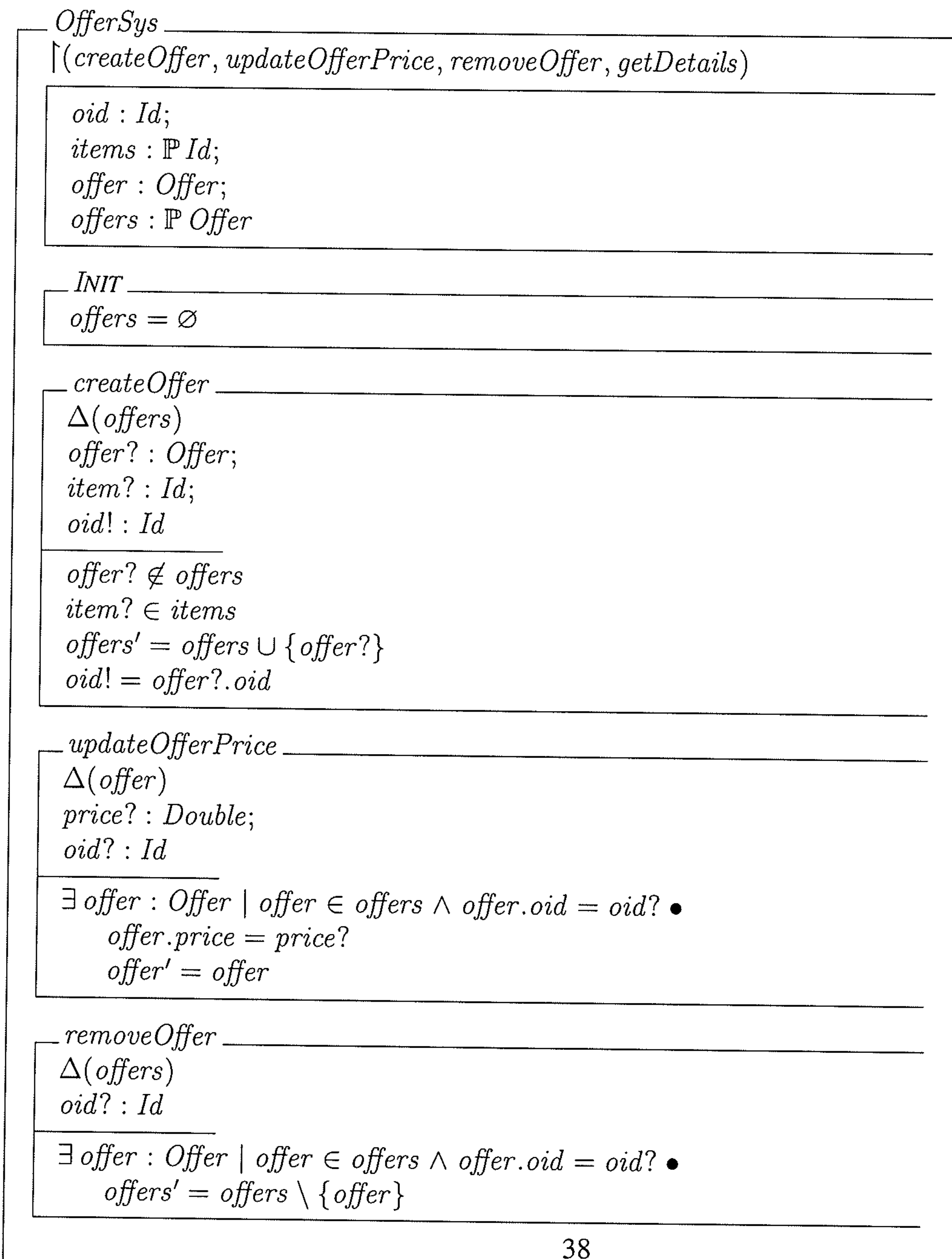


Figure 3.14-a The Offer and its Relationship to OfferSys

Figure 3.14-b describes the specification of the *OfferSys* class. There are four operations in *OfferSys* class, create an offer, update offer price, remove an offer, and get the offer's details.



OfferSys

$\upharpoonright(createOffer, updateOfferPrice, removeOffer, getDetails)$

$oid : Id;$
$items : \mathbb{P}\, Id;$
$offer : Offer;$
$offers : \mathbb{P}\, Offer$

INIT

$offers = \varnothing$

createOffer

$\Delta(offers)$
$offer? : Offer;$
$item? : Id;$
$oid! : Id$

$offer? \notin offers$
$item? \in items$
$offers' = offers \cup \{offer?\}$
$oid! = offer?.oid$

updateOfferPrice

$\Delta(offer)$
$price? : Double;$
$oid? : Id$

$\exists\, offer : Offer \mid offer \in offers \land offer.oid = oid? \bullet$
$offer.price = price?$
$offer' = offer$

removeOffer

$\Delta(offers)$
$oid? : Id$

$\exists\, offer : Offer \mid offer \in offers \land offer.oid = oid? \bullet$
$offers' = offers \setminus \{offer\}$

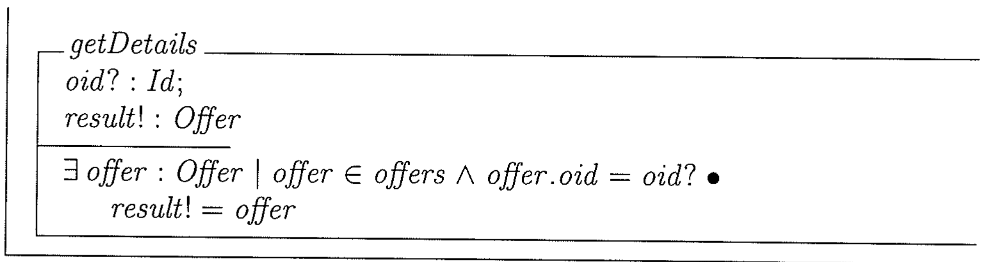


Figure 3.14-b Formal Specification of OfferSys

3.3.4 The Supplier subsystem

The supplier subsystem manages supplier information, such as, supplier address and supplier's policy, in the Bazaar System.

The supplier subsystem has only one class, *SupplierSys*. The *SupplierSys* class contains an internal class *Supplier*. Figure 3.15-a describes the class *Supplier*.

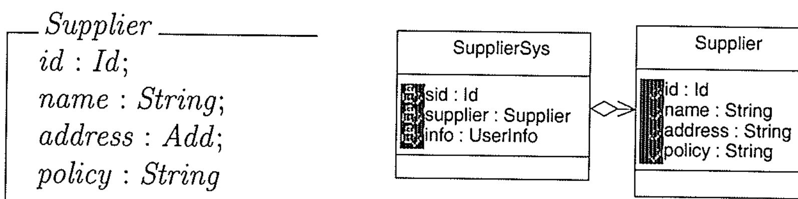


Figure 3.15-a The Supplier and Relationship to SupplierSys

The *SupplierSys* class has several operations such as: create a supplier, change policy, change address, delete a supplier, and get a supplier's details. The specification of the supplier subsystem is described in Figure 3.15-b.

SupplierSys

$\upharpoonright(createSupplier, changePolicy, changeAddress, removeSupplier, getDetails)$

$$
\begin{array}{l}
id : Id; \\
supplier : Supplier; \\
suppliers : \mathbb{P}\, Supplier
\end{array}
$$

INIT

$$suppliers = \varnothing$$

createSupplier

$$
\begin{array}{l}
\Delta(suppliers) \\
supplier? : Supplier; \\
id! : Id \\
\hline
supplier? \notin suppliers \\
suppliers' = suppliers \cup \{supplier?\} \\
id! = supplier?.id
\end{array}
$$

changePolicy

$$
\begin{array}{l}
\Delta(supplier) \\
policy? : String; \\
id? : Id \\
\hline
\exists\, supplier : Supplier \mid supplier \in suppliers \land supplier.id = id? \bullet \\
\quad supplier.policy = policy? \\
\quad supplier' = supplier
\end{array}
$$

changeAddress

$$
\begin{array}{l}
\Delta(supplier) \\
add? : Add; \\
id? : Id \\
\hline
\exists\, supplier : Supplier \mid supplier \in suppliers \land supplier.id = id? \bullet \\
\quad supplier.address = add? \\
supplier' = supplier
\end{array}
$$

removeSupplier

$$
\begin{array}{l}
\Delta(suppliers) \\
id? : Id \\
\hline
\exists\, supplier : Supplier \mid supplier \in suppliers \land supplier.id = id? \bullet \\
\quad suppliers' = suppliers \setminus \{supplier\}
\end{array}
$$

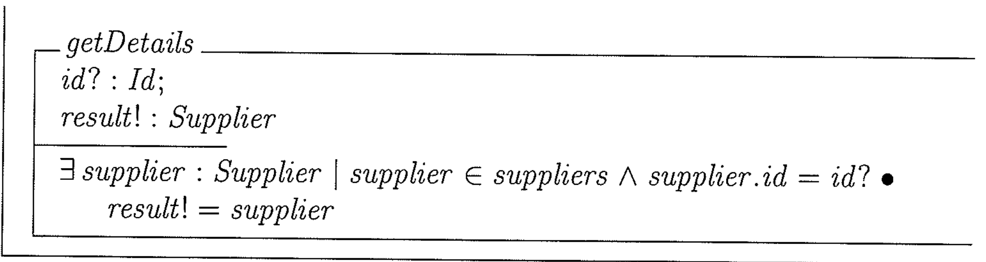


Figure 3.15-b Formal Specification of SupplierSys

3.3.5 The Store Subsystem

The store subsystem is used to manage store information, such as store name, managers, address, and policy in the Bazaar System. The store subsystem consists of only one class, *StoreSys*. The *StoreSys* class contains an internal class, *store*. Figure 3.16-a shows the *Store* class and its relationship to *StoreSys*.

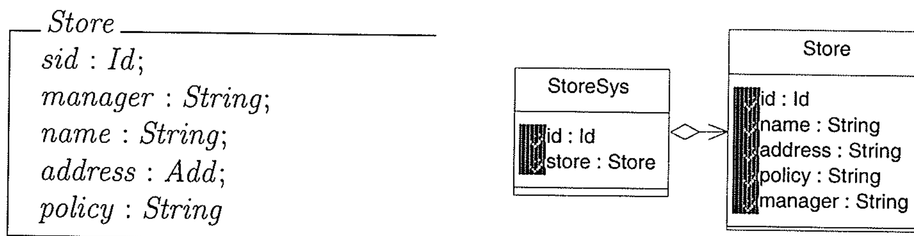
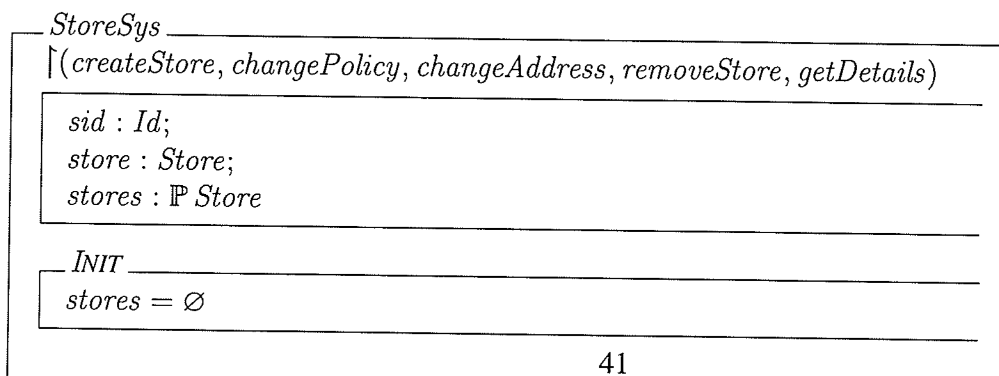


Figure 3.16-a The Store and Relationship to StoreSys

The store subsystem provides several operations including: create a store, change policy, change address, delete a store, and get a store's details. Figure 3.16-b describes the specification of the store subsystem.



createStore

$\Delta(stores)$
$store? : Store;$
$sid! : Id$

$store? \notin stores$
$stores' = stores \cup \{store?\}$
$sid! = store?.sid$

changePolicy

$\Delta(store)$
$policy? : String;$
$sid? : Id$

$\exists\, store : Store \mid store \in stores \land store.sid = sid? \bullet$
$\quad store.policy = policy?$
$store' = store$

changeAddress

$\Delta(store)$
$add? : Add;$
$sid? : Id$

$\exists\, store : Store \mid store \in stores \land store.sid = sid? \bullet$
$\quad store.address = add?$
$store' = store$

removeStore

$\Delta(stores)$
$sid? : Id$

$\exists\, store : Store \mid store \in stores \land store.sid = sid? \bullet$
$\quad stores' = stores \setminus \{store\}$

getDetails

$sid? : Id;$
$result! : Store$

$\exists\, store : Store \mid store \in stores \land store.sid = sid? \bullet$
$\quad result! = store$

Figure 3.16-b Formal Specification of StoreSys

3.3.6 The Product subsystem

The product subsystem provides functionality to manage product information, such as product category, product name, supplier and policy for dealing with returns. The product subsystem consists of one class, the *ProductSys* class. It contains two internal classes, *Category* and *Product*. Figure 3.17-a shows the two internal clases: *Category* and *Product* and illustrates their relationship to *ProductSys*.

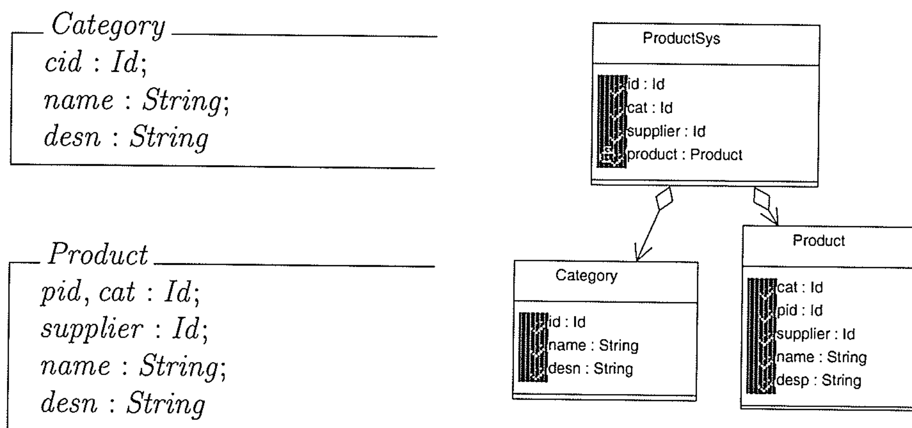
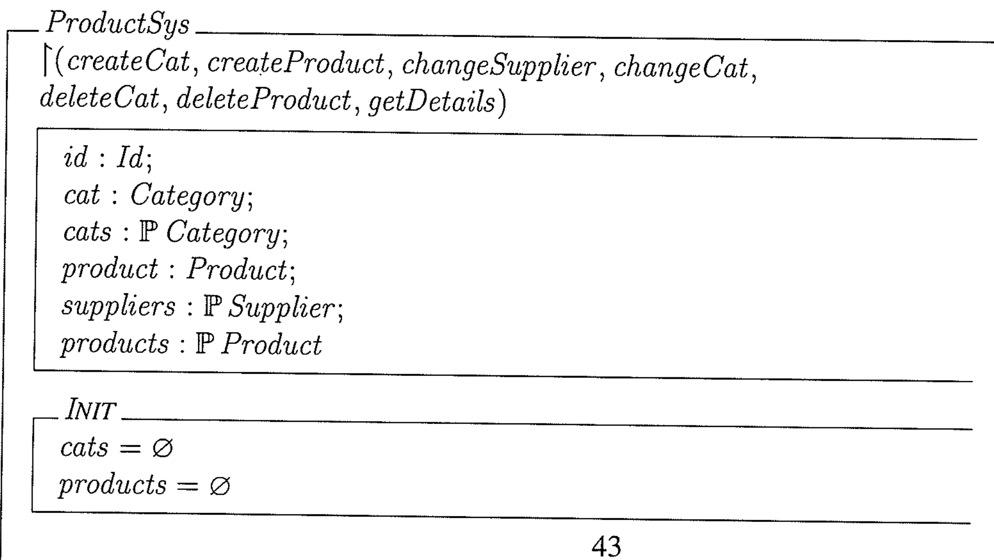


Figure 3.17-a The Internal classes of ProductSys and their Relationship

The formal specification of ProductSys is given in Figure 3.17-b. There are seven operations in the product subsystem. These operations are: create category, create product, change supplier, change category, delete product, and get details of a product.



createCat

$$\Delta(cats)$$
$$cat? : Category;$$
$$id! : Id$$

$$cat? \notin cats$$
$$id! = cat?.cid$$
$$cats' = cats \cup \{cat?\}$$

createProduct

$$\Delta(products)$$
$$cid? : Id;$$
$$product? : Product;$$
$$id! : Id$$

$$\exists\, cat : Category \mid cat \in cats \land cat.cid = cid? \bullet$$
$$\quad product? \notin products$$
$$products' = products \cup \{product?\}$$
$$id! = product?.pid$$

deleteCat

$$\Delta(cats)$$
$$id? : Id$$

$$\exists\, cat : Category \mid cat \in cats \land cat.cid = id? \bullet$$
$$\quad cats' = cats \setminus \{cat\}$$

deleteProduct

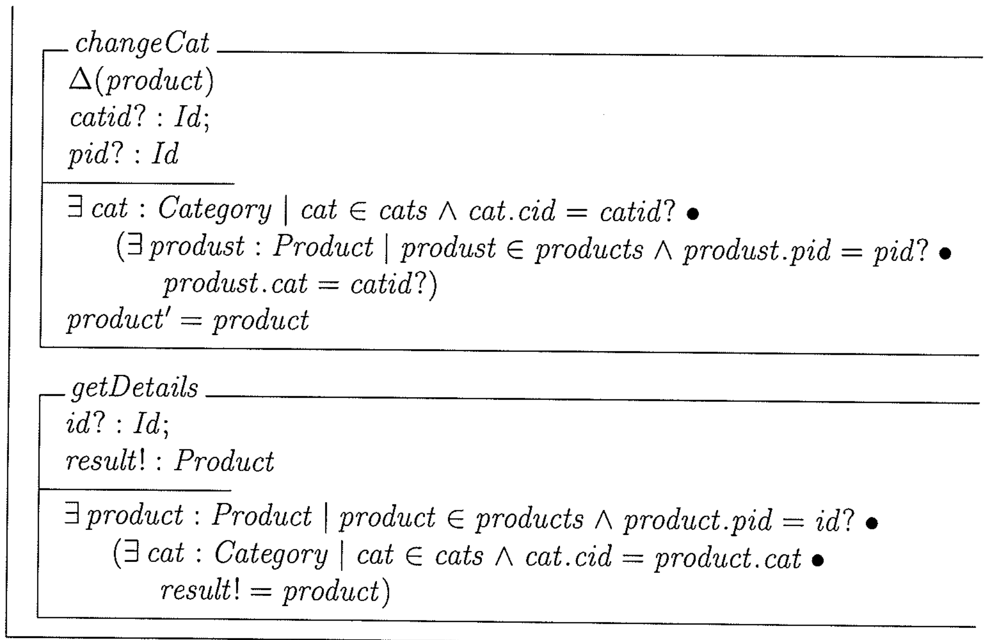
$$\Delta(products)$$
$$id? : Id$$

$$\exists\, product : Product \mid product \in products \land product.pid = id? \bullet$$
$$\quad products' = products \setminus \{product\}$$

changeSupplier

$$\Delta(product)$$
$$supid? : Id;$$
$$pid? : Id$$

$$\exists\, supplier : Supplier \mid supplier \in suppliers \land supplier.id = supid? \bullet$$
$$\quad (\exists\, produst : Product \mid produst \in products \land produst.pid = pid? \bullet$$
$$\qquad produst.supplier = supid?)$$
$$product' = product$$



changeCat

$\Delta(product)$
$catid? : Id;$
$pid? : Id$

$\exists\, cat : Category \mid cat \in cats \land cat.cid = catid? \bullet$
$\quad (\exists\, produst : Product \mid produst \in products \land produst.pid = pid? \bullet$
$\qquad produst.cat = catid?)$
$product' = product$

getDetails

$id? : Id;$
$result! : Product$

$\exists\, product : Product \mid product \in products \land product.pid = id? \bullet$
$\quad (\exists\, cat : Category \mid cat \in cats \land cat.cid = product.cat \bullet$
$\qquad result! = product)$

Figure 3.17-b Formal Specification of ProductSys

3.3.7 The Inventory subsystem

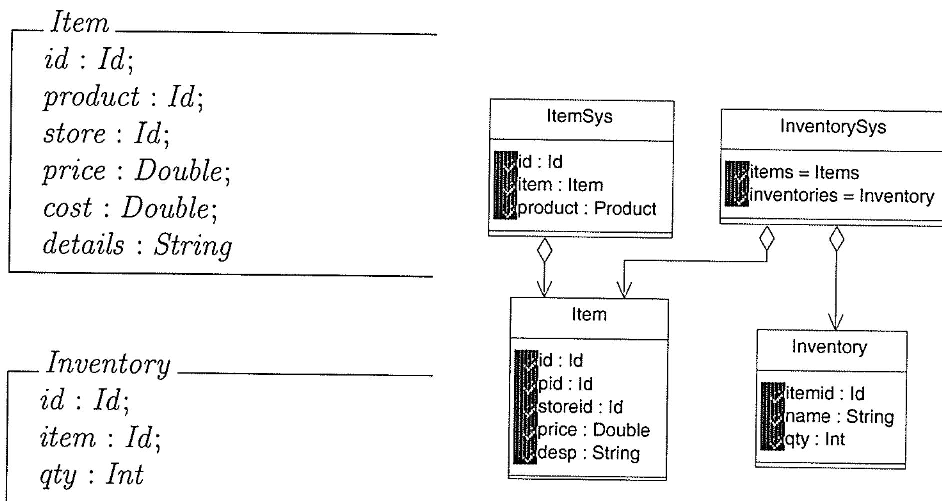
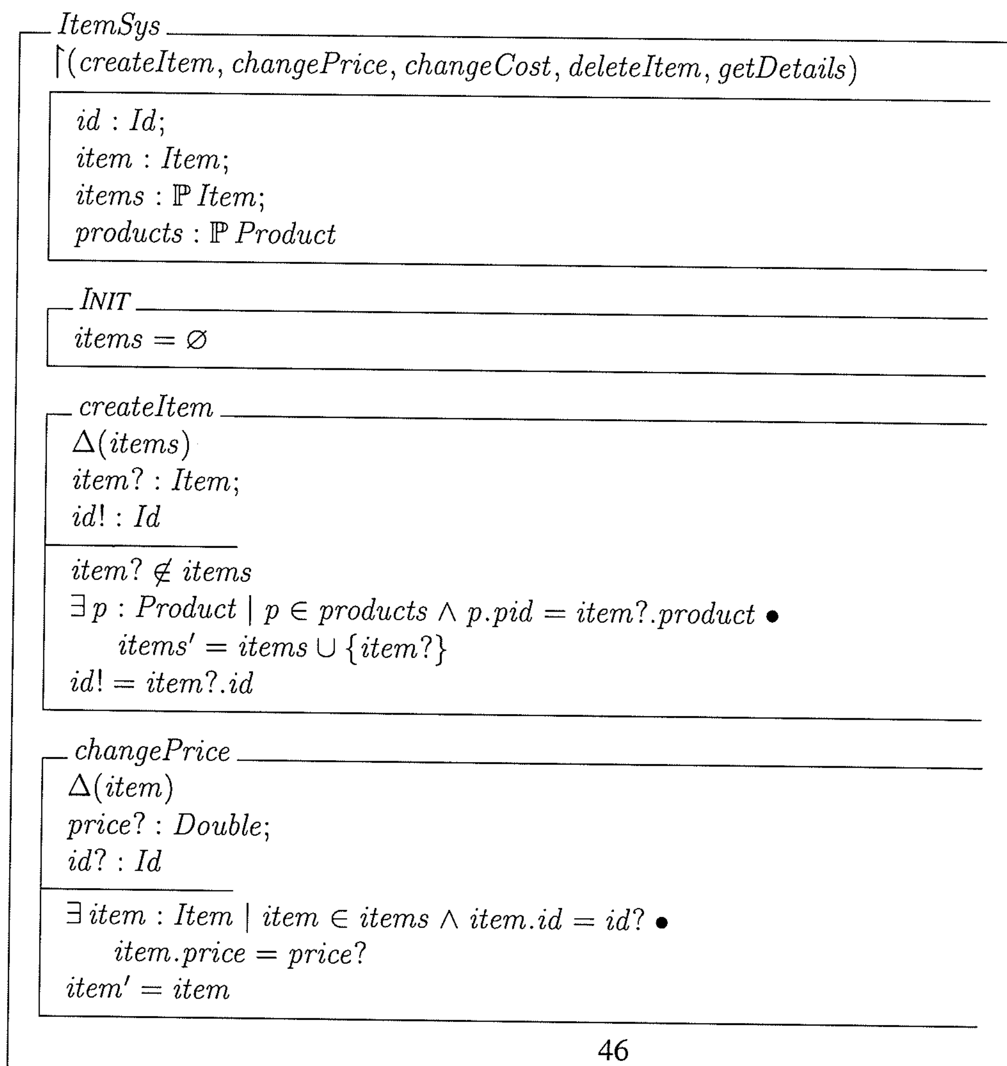


Figure 3.18-a The Two Subclasses and Relationship to *Inventory* and *Item*

The inventory subsystem is used to control and monitor the quantity of items in stores. There are several items for each product. The same product for different stores has different item ids. A buyer needs to provide *item id*, when an order for a product in the Bazaar System is made. The inventory system consists of two classes, *ItemSys* and *InventorySys*. The two internal classes, *Item* and *Inventory* also exist in the inventory subsystem.

Figure 3.18-a shows the two internal classes. The *ItemSys* class has five operations namely, create item, change price, change cost, delete items, and get details of an item. The formal specification of ItemSys is given in Figure 3.18-b.



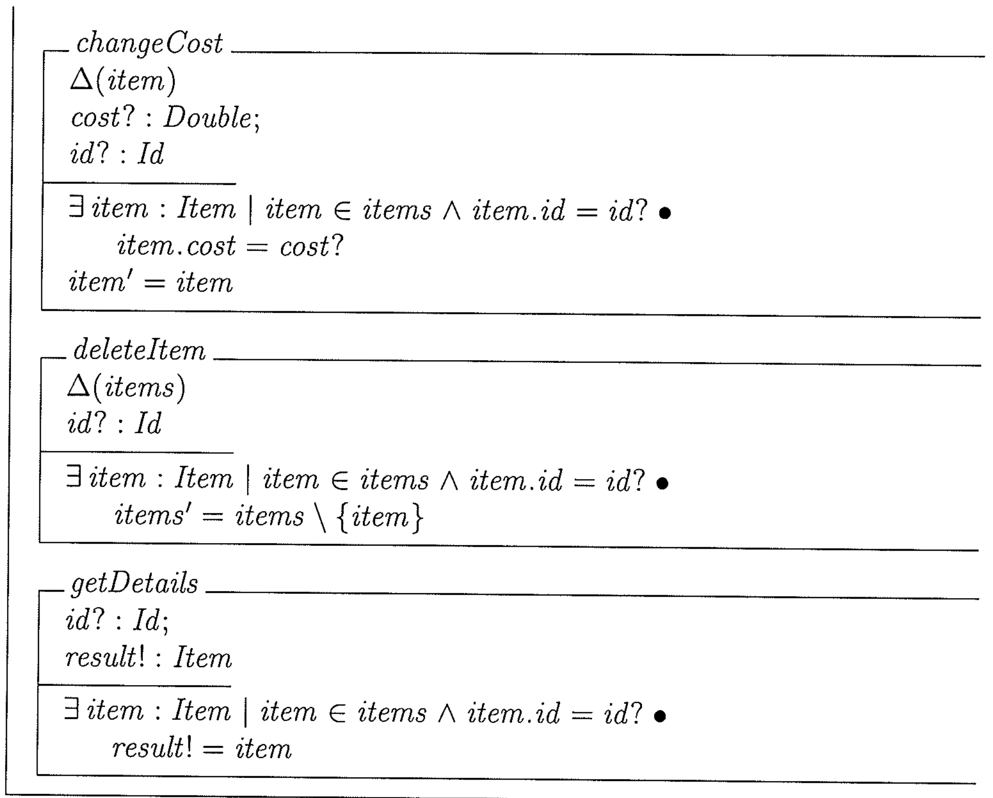
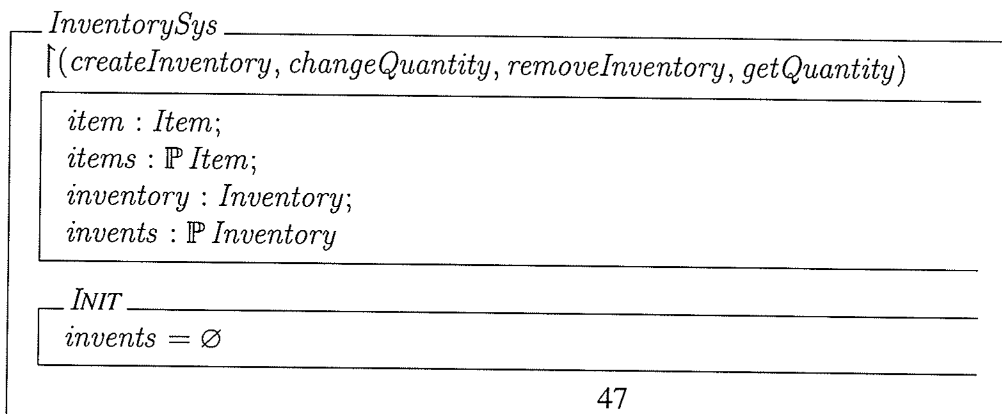


Figure 3.18-b Formal Specification of ItemSys

There are four operations provided by *InventorySys* class. The operations are create inventory, change quantity, remove inventory, and get detail. Figure 3.18-c provides a formal specification of *InventorySys*.



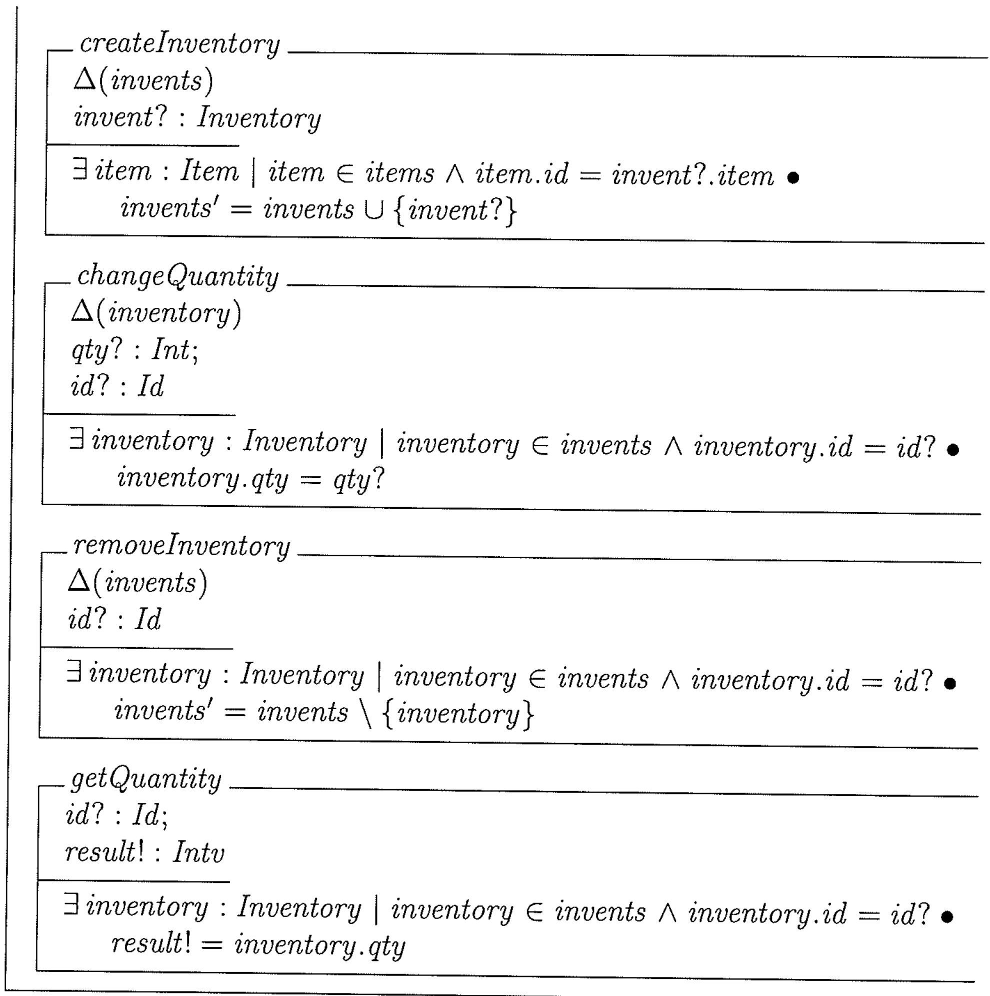


Figure 3.18-c Formal Specification of InventorySys

The Bazaar System thus consists of seven subsystems, namely: login, order, offer, supplier, store, product, and inventory. The Bazaar System can be specified as follows:

$$Bazaar \mathrel{\widehat{=}} [\,LogSys;\ OrderSys;\ OfferSys;\ SupplierSys;\ Store;\ Product;\ Inventory\,]$$

CHAPTER 4
THE SYSTEM DESIGN SPECIFICATION

Based on the requirements specification of the Bazaar System (see Chapter 3), a design document will now be created. The architectural design is described in Section 4.1. A formal, detailed design document for a subset of the Bazaar System and an example are given in Section 4.2. The entire detailed design document is attached in Appendix A.

First we present the architectural design. UML component and class diagrams (Figures 4.2 ~ 4.13) are used to describe the architectural design. The return types of some methods and the types of attributes are not specified in the architectural design. They are left to the detailed design and implementation phases of the system development.

4.1 Architectural Design of the Bazaar System

Figures 4.1 and 4.2 give a general idea of the structure of the whole system. The two figures are explained in Section 4.1.

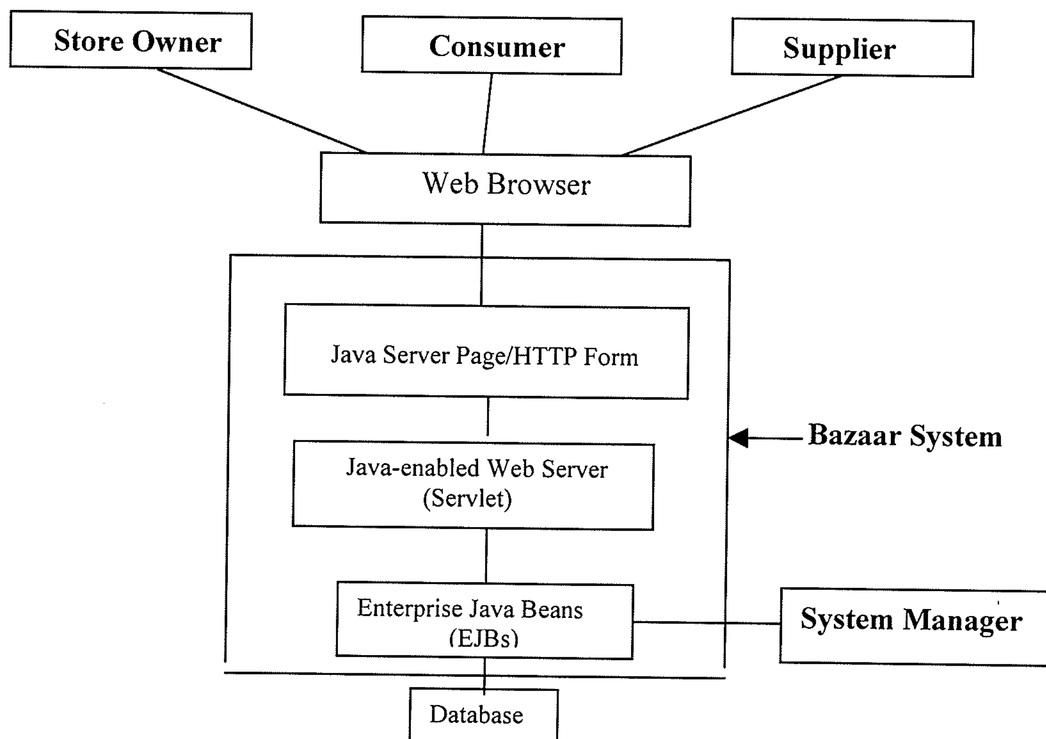


Figure 4.1 The Bazaar System Architecture

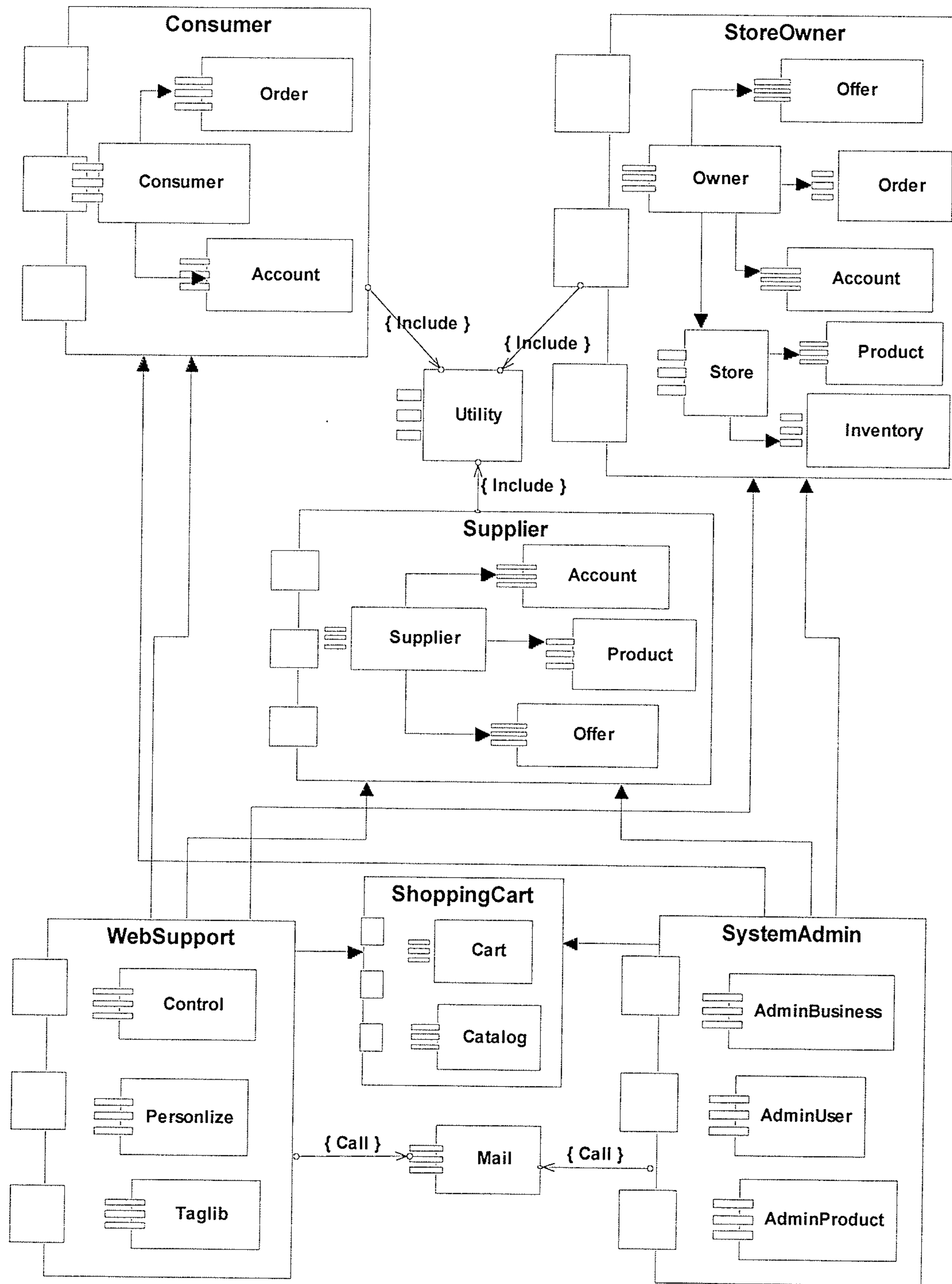


Figure 4.2 Components Diagram of the Bazaar System

From a users' point of view, the Bazaar System is described in Figure 4.1. There are four kinds of external users in the system. They are store owners, consumers, suppliers and the system manager. These external users are web-based users, and they access the Bazaar System via the Internet. The system manager is a special user, and may not access the Bazaar System through the Internet.

The Bazaar System consists of complex objects and several basic objects. Each object is related to a component on the server and can be implemented as an EJB. The relationships between, and structure of, these objects are described in Figure 4.2. There are five packages, Consumer, StoreOwner, Supplier, Websupport, ShoppingCart, and SystemAdmin. Each package is related to one complex objects. The components in a package are basic objects.

The Bazaar System is a three-tier application, which includes the server, client, and data source tier. The basic objects in the package of the Consumer, StoreOwner and the Supplier should be deployed to server. The complex objects of Consumer, StoreOwner, and Supplier are also components at the server end. An object, such as the Account object, can be deployed more than once. The components Utility, Mail, and ShoppingCart are help tools, and they also need to be deployed to the server side. The client side consists of two separate and independent systems, the Front-end system (WebSupport) and the Administrative system (SystemAdmin). The Front-end system is GUI for common users, such as the users who want to order product through the Bazaar System and the Administrative system is for the system manager to manage the Bazaar System.

4.1.1 The Basic Objects of the System

A basic object is an object on the server side implemented using a single EJB. It can be implemented as an EJB and deployed separately as a component to a J2EE container. In addition, a basic object can be deployed with other basic objects to the server side as a component to a J2EE container. For example, a Mail object is a basic object which is deployed as a component separately to a J2EE container. The Account object is also a

basic object, but it is deployed three times with other objects to J2EE containers to form three components on the server side (see Figure 4.2). Figures 4.3 - 4.8 show the class diagrams of the basic objects in the Bazaar System. Each object contains an instance of an EJB class, a HomeInterface class and a RemoteInterface class as well as other helper classes (including model class, exception handler class, and help class, etc.). The HomeInterface class is for the objects accessing the EJB at the server and the RemoteInterface class support the objects accessing the EJB at client side. The relationship between these classes is also described in the class diagrams.

Account: The Account object is implemented as an entity bean. Figure 4.3 shows the class diagram of the Account EJB. There are four other classes, Account (remote interface), AccountHome (home interface), AccountException, and AccountModel, in the Account package. The classes in the package should be deployed as a component (Account Bean) on the server side.

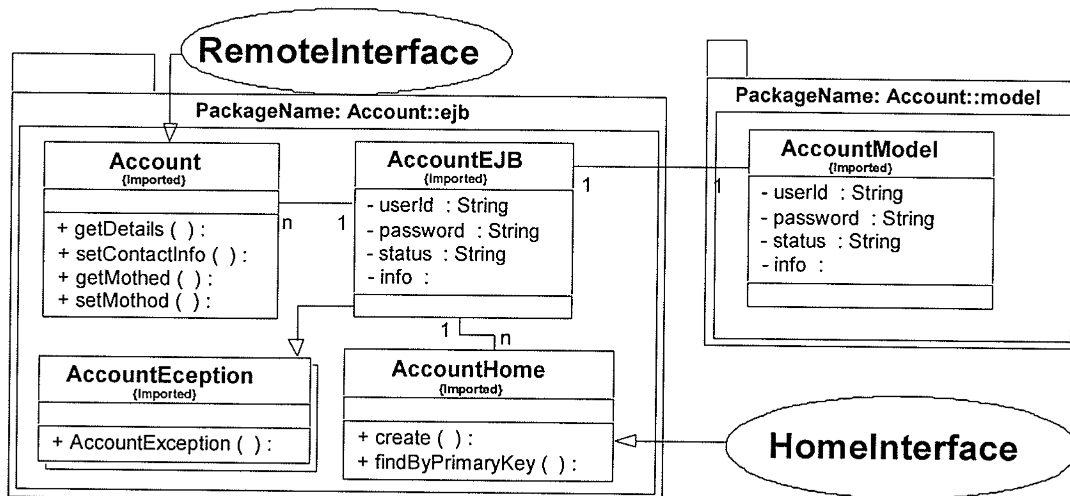


Figure 4.3 The Class Diagram of the Account EJB

The Account object holds information about users in the Bazaar System. Actions by users on client machines call the Account EJB in the server side to perform any operation related to the user's information. These operations include registration of new users, deleting an account, change an account, getting a user's information, and login/out. All these operations are mapped to business methods in the Account EJB. Additional details

of the operations are described in Account Object of "Appendix A". For example, if a user want to register in the Bazaar System, first the client inputs all the information using the client end user interface, then the information is transferred to the Account EJB at the server end. After evaluation (i.e., checking to see no such client exists in the system), the information is stored in the database.

Mailer: The mailer object is implemented as a session bean. Figure 4.4 shows the class diagram of the Mailer EJB. There are six classes: MailerEJB, Mailer (remote interface), MailerHome (home interface), sendMailException, EmailHelper, and EmailMessage, in the Mailer package. The classes in the package should be deployed as a component (Mailer Bean) on the server side.

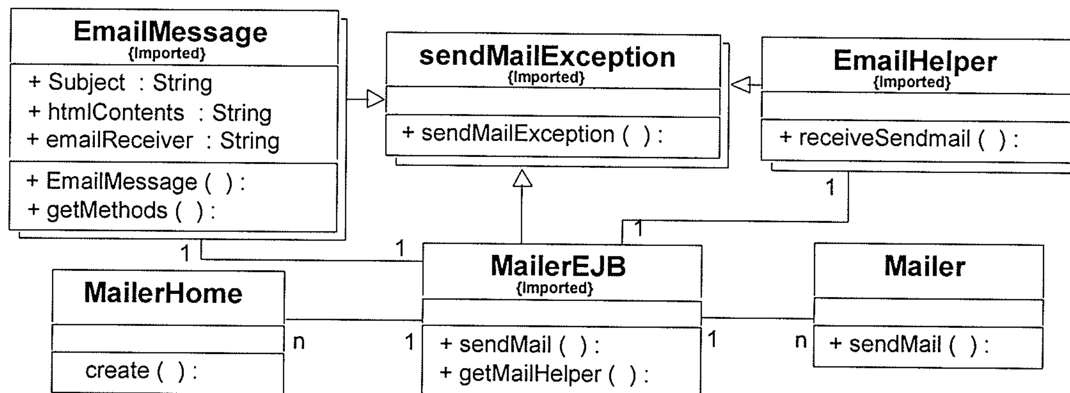


Figure 4.4 The Class Diagram of the Mailer EJB

The Mailer object is a tool to transfer information from one account to another internally or externally. The Mailer object provides several operations including: create message, change recipient, send mail, and get methods (for subject, content and recipient). The Mailer object is used to communicate among the users in the Bazaar System. The details of the operations of the Mailer object are described in Mailer Object of "Appendix A".

Offer: The Offer object is implement as a session bean. The class diagram for the Offer EJB is shown in Figure 4.5. Offer contains five classes: OfferEJB, Offer (remote interface), OfferHome (home interface), OfferLine, and OfferModel. The classes

in the package should be deployed as a component (Offer Bean) on the server side.

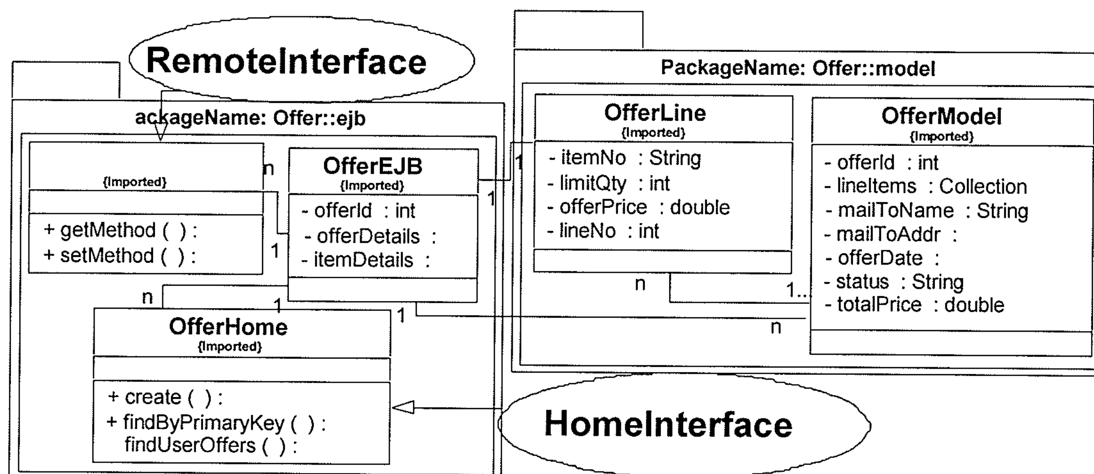


Figure 4.5 Class Diagram of the Offer EJB

A user uses the Offer object to do any operation related to offers. These include: creating an offer, deleting an offer, getting an offer's details, and changing the offer price and offer status. The details of these operations are described in Offer Object of "Appendix A". For a user to create an offer, first the user enters the offer information (i.e., product offered, quantity and price), through the remote interface at the client side; then the information is transferred to the Offer EJB at the server side; after the evaluation, the information is stored in the database and is passed to the user(s) who should receive the offer.

Order: The Order object is implemented as an entity bean. The class diagram of the Order EJB is illustrated in Figure 4.6. The Order object has five classes: OrderEJB, Order (remote interface), OrderHome (home intercace), LineItem, and OrderModel. The classes in the package should be deployed as a component (Order Bean) on the server side.

The Order object provides operations related to order actions, which include: creating an order, deleting an order, changing order information (line item, store, credit card, status etc.), and get methods for order attributes. There are two entities in the order object, line

item and order. When creating an order instance, two entities are created, an order entity and several line item entities. The order entity holds information about an order and it contains an order id, line items, user id and other order information. Each line item entity contains item id, quantity, unit price, and line item id. The order object is a component on the server side. The order information is accepted through the interface on the client side, and transferred to the server side. After evaluating all the values, the order information is stored in the database. If a user wants to search for some information about an order, first the search key is transferred to the order component at server side, then the OrderEJB retrieves the related information from the database and sends it to the client.

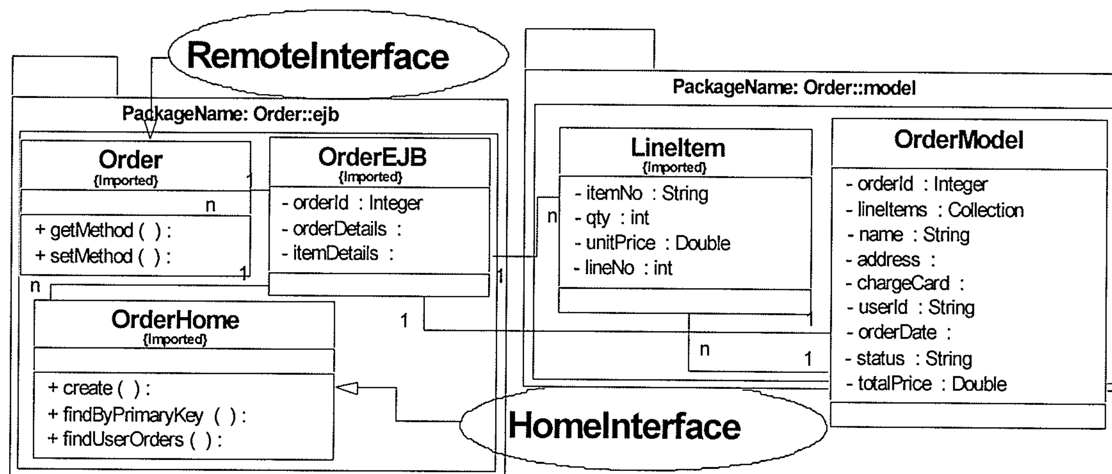


Figure 4.6 Class Diagram of the Order EJB

Product: The product object is implemented as an entity bean. The class diagram of the Product EJB is depicted in Figure 4.7. The object has six classes, ProductEJB, Product (remote interface), ProductHome (home interface), ProductException, Item, and ProductModel. The classes in the package should be deployed as a component (Product Bean) on the server side.

The product EJB is used to create a product instance and to provide operations related to products. All the object's operations, which include: creating a category, creating a product, deleting a category, deleting a product, changing product information, and get methods (for the supplier, category attributes) are based on the two entities, product

entity and item entity. The details of the Product object operations are described in Product Object of "Appendix A".

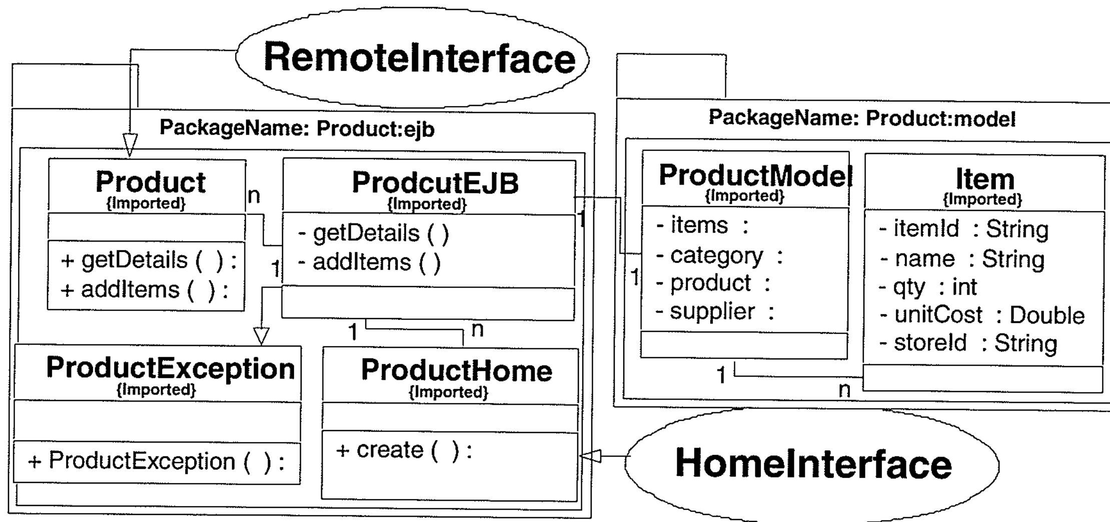


Figure 4.7 The Class Diagram of Product EJB

Inventory: The inventory object is implemented as an entity bean. The class diagram of an Inventory EJB is shown in Figure 4.8. The Inventory object has five classes, InventoryEJB, Inventory (remote interface), InventoryHome (home interface), InventoryException, and InventoryModel. The classes in the package should be deployed as a component (Inventory Bean) to the server side.

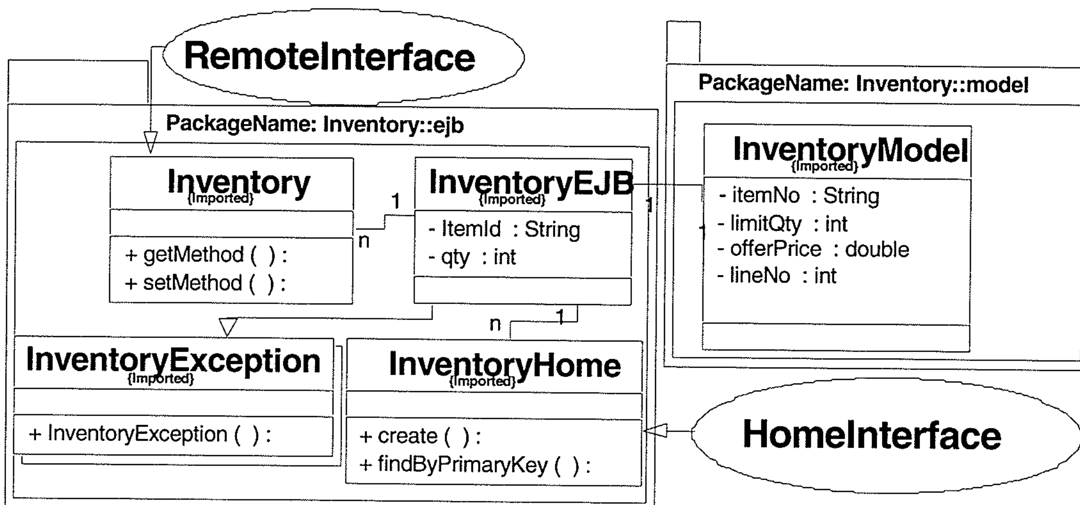


Figure 4.8 The Class Diagram of Inventory EJB

Each inventory EJB is used to create an inventory instance for a store. The operations of an Inventory object include: create an inventory instance, delete an inventory instance, get quantity, and change quantity. The details of these operations are available as Inventory Object in "Appendix A". Besides these simple objects, there are four complex objects in the system, namely Consumer, Supplier, Owner, and ShoppingCart. The next section introduces the complex objects.

4.1.2 The Complex Objects of the System

Complex objects are objects that contain more than one EJB on the server side. A complex object, such as the Consumer object, shares information with other objects (basic or complex). When a complex object is deployed to a J2EE container, the other related objects need to be deployed with it so that all the objects together will be a component at the server side. A complex object is implemented as an EJB, but it depends on other EJBs. For example, the consumer object is implemented as ConsumerEJB, but the ConsumerEJB depends on other two EJBs: OrderEJB and AccountEJB. Like basic objects, a complex object is instantiated from an EJB class, HomeIterface class and RemoteInteface class and other helper classes (including a model class, exception handler class, and help class). The complex objects in the Bazaar System include: Consumer, Supplier, Owner, and ShoppingCart. Figures 4.9 - 4.13 describe these complex objects. We briefly describe these objects below. The details of the operations of each object are described in "Appendix A".

Consumer: The consumer object is implemented as a session bean. Figure 4.9 shows the class diagram for the Consumer EJB. The Consumer EJB is dependent on Order EJB and Account EJB. Therefore, the Consumer EJB should be deployed together with these two EJBs as one component on the server side.

The consumer object handles operations for users (buyers) who have already registered with the Bazaar System. Besides Account object and Order object operations, additional operations, such as: get order reference and get account reference, add consumer's

details, and get consumer's details are also available. The consumer EJB needs to call the Account EJB or the Order EJB to process a request from a client.

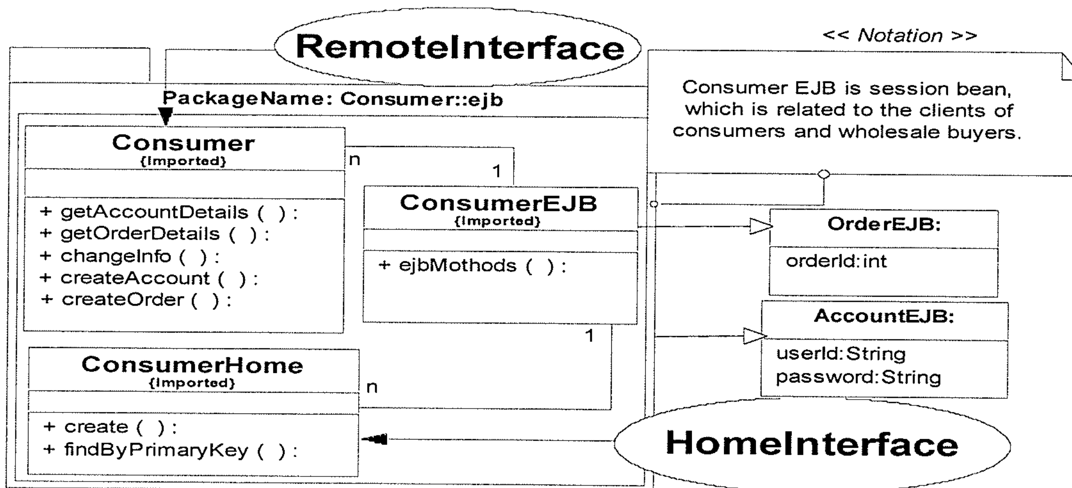


Figure 4.9 Class Diagram of the Consumer EJB

Supplier: The supplier object is implemented as a session bean. Figure 4.10 shows the class diagram for the Supplier EJB. The Supplier EJB depends on Offer EJB and Account EJB. Therefore, the Supplier EJB should be deployed together with those two EJBs as one component on the server side.

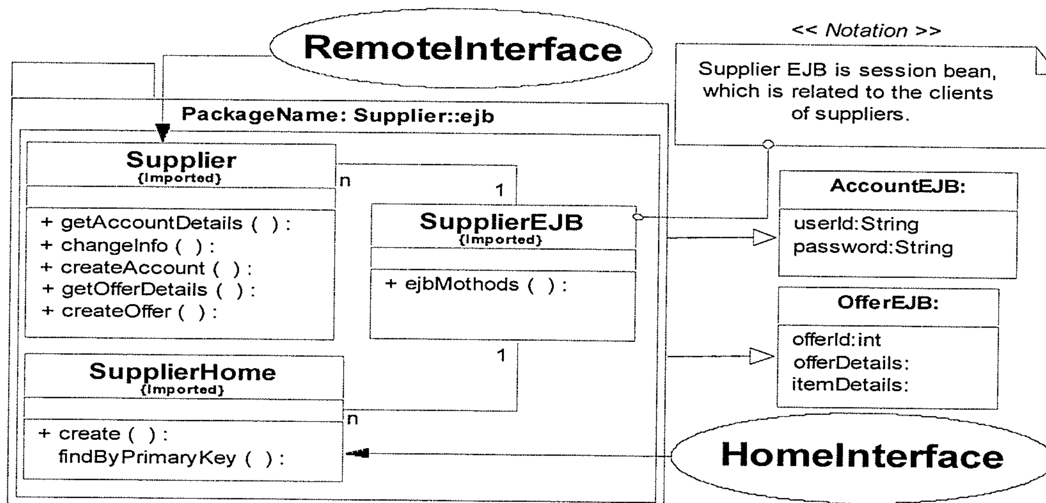


Figure 4.10 Class Diagram of the Supplier EJB

The supplier object provides all the operations required by suppliers. Besides the Account object and Offer object operations, other operations, such as create supplier, delete supplier, change supplier information, and get consumer's details are available. The supplier EJB needs to call the Account EJB or the Offer EJB to process a request from a client.

Owner: The owner object is implemented as a session bean. The class diagram of the Owner EJB is depicted in Figure 4.11. The Owner EJB depends on Order EJB, Offer EJB, and Account EJB. So the Owner EJB should be deployed together with these three EJBs as one component on the server side.

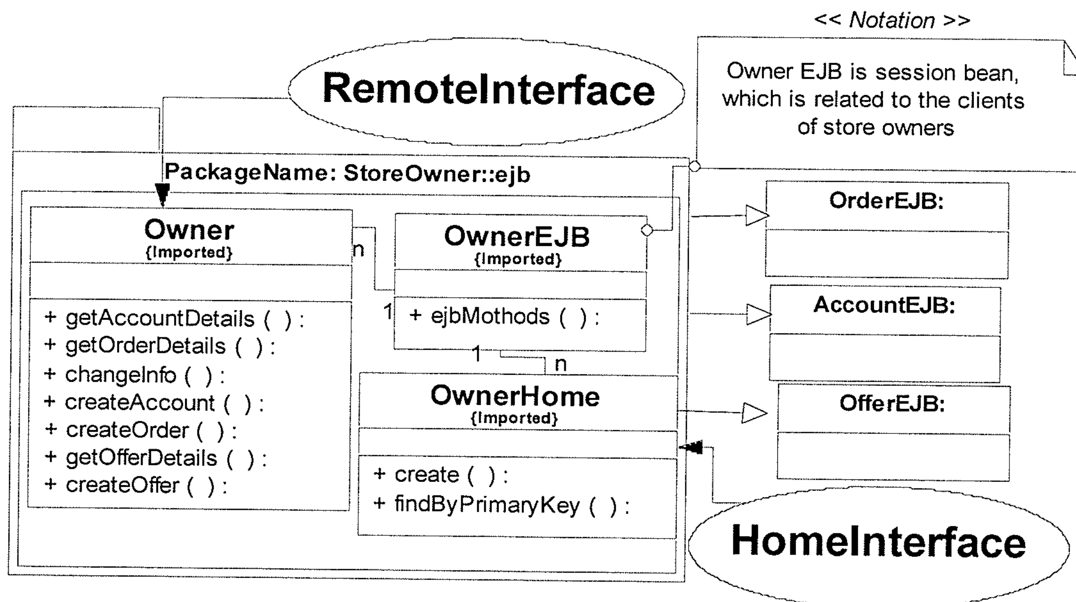


Figure 4.11 Class Diagram of the Owner EJB

The owner object provides all the necessary operations for store owners. Besides Account object, Inventory object, and Store object operations, other operations, such as: get references, change owner's information, and get owner's details are available. The owner EJB needs to call the Account EJB, the Store EJB or the Inventory EJB to process a request from a client.

ShoppingCart: The ShoppingCart object is implemented as two session beans; Cart and Catalog. Figure 4.12 shows the class diagram of the Catalog EJB while

Figure 4.13 shows the class diagram of the Cart EJB. The Catalog bean has six classes: CatalogEJB, Catalog (remote interface), CatalogHome (home interface), CatalogException, ProductMode, and CatalogDao. The Cart bean has five classes: CartEJB, Cart (remote interface), CartHome (home interface), CartModel, and CartItem.

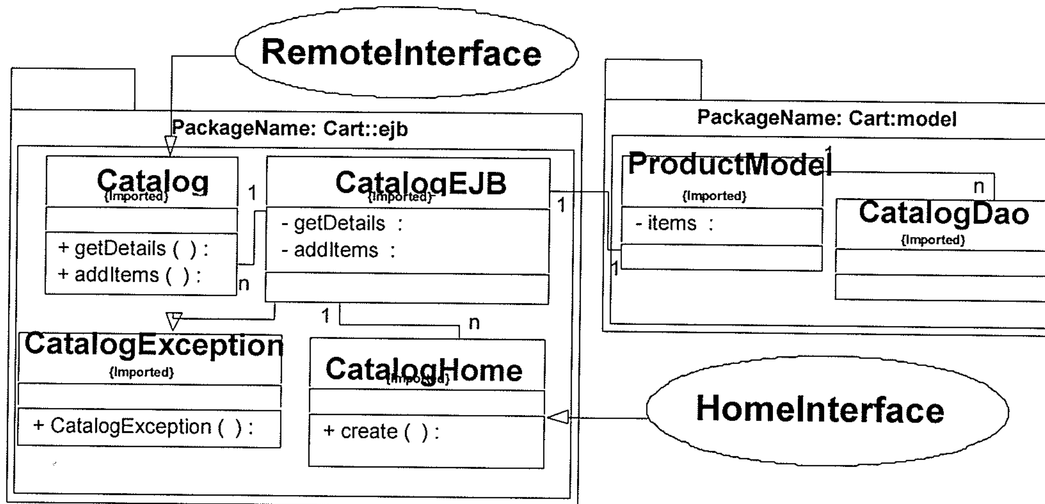


Figure 4.12 The class diagram of Catalog EJB

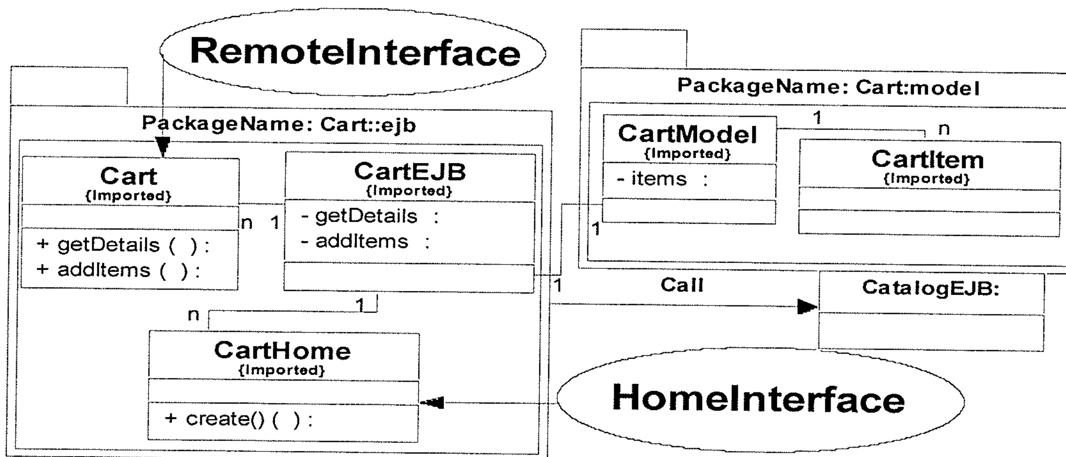


Figure 4.13 The class diagram of Cart EJB

The shopping cart object is a special object in the Bazaar System, which is used to hold selected items when a buyer browses through the catalog, searches for products, and modifies or holds an order instance temporarily.

4.2 Detailed Design Information

The requirements specification developed in Section 3.3 is now augmented with design details. The detailed design of the Bazaar System is based on seven basic objects (*Account*, *Order*, *Offer*, *Store*, *Mail*, *Item*, and *Product)* and four complex objects (*Consumer*, *Supplier*, *Owner and ShoppingCart*).

Each object's class is related to a database table or several database tables. There are twelve tables in the Bazaar System, *order*, *offer*, *category*, *product*, *item*, *itemline*, *store*, *account*, *consumer*, *owner*, *supplier*, *and inventory*. The relationship among the database tables in the Bazaar System is shown in Figure 4.14.

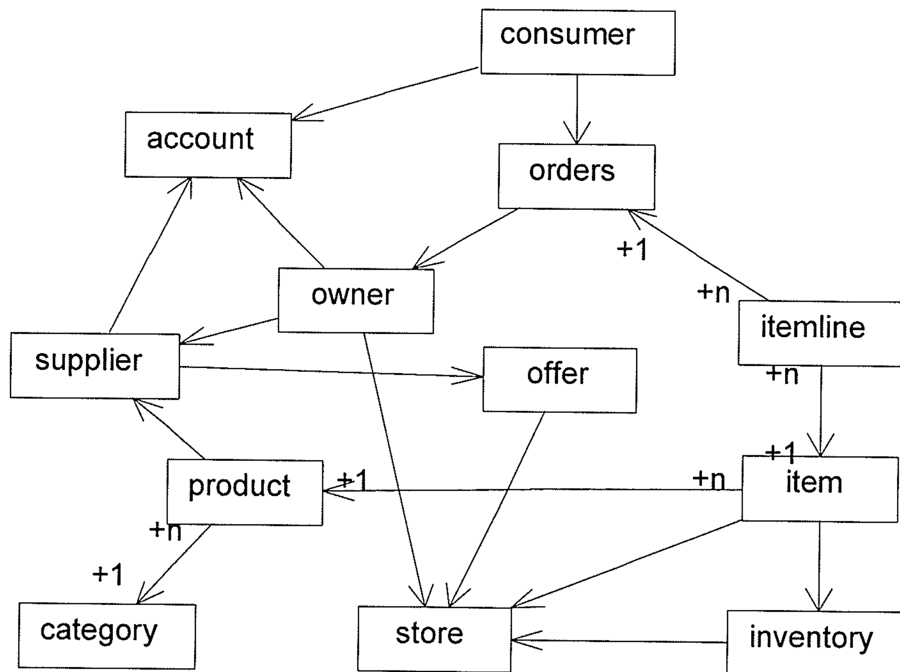


Figure 4.14 Database Tables and their Relationship

The following information is used to describe the detailed functional requirements for each class:

Index
Class Name

Inheritance
Imported Classes
Purpose
Remarks
Properties
Methods

Index is a reference assigned to this object. This reference is unique and is assigned solely for the purpose of identifying and cross-referencing this object.

Class Name is a unique name that describes the corresponding class.

Inheritance describes the inheritance relationship between this class and it's derived class(es).

Imported Classes list all the classes imported by this class.

Purpose briefly describes the objective of the class.

Remarks explains any details concerning this class, which are not captured by any of the above.

Properties describes the property of this class.

For example, the Account object is described as follows:

Account Object:

Index:	Obj-1
Class Name:	Account
Inheritance:	None
Imported classes:	None
Purpose:	Holds information for a user to access the system and to execute appropriate operations
Remarks:	Created before the user can access the system
Properties:	Entity bean

A method is described as following style:

Index
Name
Purpose
Visibility
Input parameters
Output parameters
Pseudocode
Remarks

Index is a reference assigned to the method. This reference is unique and is assigned solely for the purpose of identifying and cross-referencing this method.

Name is a unique name that describes this method.

Purpose describes the method in short prose.

Visibility describes what type the method is, whether the method is visible internal or external for the users.

Input parameters is a list of input parameters required for the method.

Output parameters lists the set of output parameters that are expected to be returned by the method. Sometimes, there may not be any output parameter for a method. It should be noted that the method may also change some global entities in the system.

Remarks explains some details concerning the method which are not captured by any of the above. For example, the relationship between the current method and other methods.

For example, the methods of the Account Object are described as follows:

Method to set the attributes:

Index: Con-1
Name: AccountEJB
Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode:

```
find datasource for connection {
        datasource =
            lookup("java:comp/en/jdbc/MyDataSource");
}
```

Remarks: No

Methods of the Object:

Index: Op-1
Name: registration
Purpose: To add a user to the system
Visibility: External; visible
Input parameters: userId; password; creditCard; email;userDetails
Output parameters: Confirm information (userId, password)
Pseudocode:

```
if userExists(userid)
    return null;
else {
    createAccount(userId, password, creditCard, email);
    addDetails(userId, userDetails);
    return (userId, password);
}
```

Remarks: No

Index: Op-2
Name: deleteAccount
Purpose: To delete an account
Visibility: External; visible
Input parameters: userId
Output parameters: None
Pseudocode:

```
if userExists(userid) {
        deleteAccount(userId);
        deleteDetail(userId);
        return true;
}
else
        return false;
```

Remarks: No

Index: Op-3
Name: login
Purpose: To login into the system
Visibility: External; visible
Input parameters: userId; password
Output parameters: A message (successfully or fail)
Pseudocode:

```
if (userId) in userArray(userId, status) {
        print("user is Already in the system");
}
else if (userExists(userid) & validP(password)) {
        add userId to userArray;
        the Status of the user is set to active;
        print("login into the system successfully");
}
else
        print("the userid or password is not correct");
```

Remarks: No

Index: Op-4
Name: changeAccount
Purpose: To change a user account's information
Visibility: External; visible
Input parameters: userId; newData
Output parameters: None
Pseudocode:

```
if userExists(userId)  {
    getDBConnection;
    modify database(UPDATE account_table WITH  newData, WHERE
    userid = userId);
}
```

Remarks: No

Index: Op-5
Name: getAccountDetails
Purpose: To retrieve a user's account details
Visibility: External; visible
Input parameters: userId
Output parameters: list of (userId, password, creditCard,balance, email)
Pseudocode:

```
if userExists(userId) {
```

```
getDBConnection;
retrieve data from database {
      SELECT (name, address, password, credit,
                                    balance, email)
            FROM account_table
            WHERE userid = userId);
   }
}
```

Remarks: No

Index: Op-6
Name: getUserDetails
Purpose: To retrieve a user's detail
Visibility: External; visible
Input parameters: userId
Output parameters: list of details
Pseudocode:

```
getDBConnection;
if userExists(userId) {
      retrieve data from database {
      SELECT (name, address, etc.)
                FROM user_table
                WHERE userid =  userId;
   }
}
```

Remarks: The table name user_table and details will change (depends on the caller consumer, supplier or owner).

Index: Op-7
Name: changeUserInfo
Purpose: To change a user's information
Visibility: External; visible
Input parameters: userId; newData
Output parameters: None
Pseudocode:

```
getDBConnection;
if userExists(userId) {
      modify database{
         UPDATE user_table WITH  newData,
               WHERE userid  =  userId;
      }
}
```

Remarks: No

Index: Op-8
Name: logout
Purpose: To exit the system
Visibility: External; visible
Input parameters: userId; password
Output parameters: None
Pseudocode:

```
if (userId, password) in userArray {
      delete (userid, password) from userArray;
      the Status of the user is set to inactive;
}
```

Remarks: No

Index: Op-9

Name: createAccount
Purpose: To create an account
Visibility: External; visible
Input parameters: userId; password; status; credit; email
Output parameters: None
Pseudocode:

```
If (isValidDate(userId, password)) {
        getDBConnection;
        add data to database {
            INSERT (userId, password, status, credit, email)
                        INTO account_table;
              return true;
        }
else
        return false;
}
```

Remarks: No

Index: Op-10
Name: addDetails
Purpose: To add user information to the system
Visibility: External; visible
Input parameters: userId; info
Output parameters: None
Pseudocode:

```
if userId in userArray
        Store info to  database;
```

Remarks: Abstract method

Methods used internally by the object

Index: In-1
Name: userExists
Purpose: To check if the user exists in the system
Visibility: Internal; invisible
Input parameters: userId
Output parameters: result (true or false)
Pseudocode:

```
getDBConnection;
retrieve data from database {
     string = SELECT userid FROM account
                  WHERE userid  = userId;
}
if (string = null)
        return false;
else
     return true;
```

Remarks: No

Index: In-2
Name: isValidData
Purpose: To check if the data is valid
Visibility: Internal; invisible
Input parameters: userId; password
Output parameters: result (true or false)
Pseudocode:

```
if ((userId = null) || (password = null))
        return false;
```

```
else
    return true;
```

Remarks: No

Index: In-3
Name: validP
Purpose: To check if the user's password is correct
Visibility: Internal; invisible
Input parameters: userId; password
Output parameters: result (true or false)
Pseudocode:

```
search database {
      string = SELECT password FROM account
                  WHERE userid = userId;
}
if (string = password)
      return true;
else
    return false;
```

Remarks: No

Index: In-4
Name: deleteAccount
Purpose: To delete an account
Visibility: Internal; invisible
Input parameters: userId
Output parameters: None
Pseudocode:

```
getDBConnection;
remove an account entry from database {
      DELETE FROM account_table
          WHERE userid = userId;
}
```

Remarks: No

Index: In-5
Name: deleteDtails
Purpose: To delete a user's detail from database
Visibility: Internal; invisible
Input parameters: userId
Output parameters: None
Pseudocode:

```
getDBConnection;
remove an account entry from database {
      DELETE FROM user_table WHERE
            userid = userId;
}
```

Remarks: No

CHAPTER 5

IMPLEMENTATION OF THE BAZAAR SYSTEM

The Bazaar System is implemented using the J2EE platform. J2EE enables developers to write reusable, and portable server-side business logic using an environment (see Figure 5.1) that provides a multi-tier distributed application model. The components at the server consist of a "Bazaar Servlet", Entity Beans, Session Beans, and Container Classes.

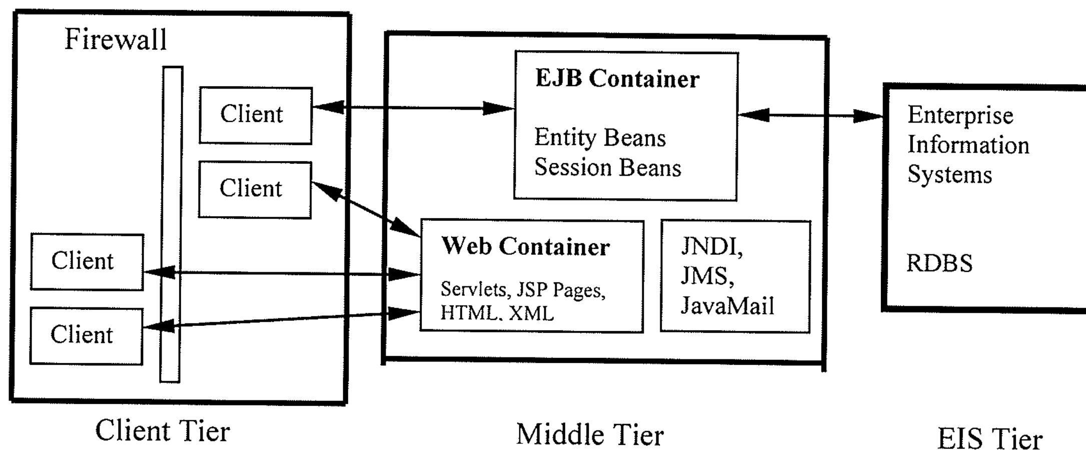


Figure 5.1 J2EE Environment (adapted from [1])

The next section describes how the Bazaar System was implemented using J2EE and JSP (Java Server Pages) technologies.

5.1 The Implementation of EJBs on the Server Side

Depending on the J2EE platform to handle complex system-level issues, Enterprise JavaBeans technology provides a distributed component model that enables developers to focus on solving business problems. First, we explain the implementation of the EJBs described earlier.

As described earlier, there are Home Interface, Remote Interface and Enterprise Bean Classes for each EJB.

5.1.1 Home Interface

The home interface provides the methods for creating and removing enterprise beans. This interface must extend *javax.ejb.EJBHome*. A client can do the following operations through the home interface:

- Create new enterprise bean instances;
- Get the metadata for an enterprise bean through the java.ejb.EJBMetaData interface;
- Remove an enterprise bean instance; and
- Obtain a handle to the home interface, which provides mechanism for persistent enterprise beans.

5.1.2 Remote Interface

The remote interface defines the client's view of an enterprise bean – the set of business methods available to the clients. This interface must extend javax.ejb.EJBObject. The remote interface supports business methods and delegates invocation of a business method to the enterprise bean instance. A remote interface defines the methods that allow clients to perform the following operations via a reference to an enterprise bean instance:

- Obtain the home interface;
- Remove the enterprise bean instance;
- Obtain a handle to the enterprise bean instance; and
- Obtain an entity bean instance's primary key.

5.1.3 Enterprise Bean Class

The enterprise bean class provides the actual implementation of the business methods of the bean. The container calls on the particular bean when the client selects the corresponding methods listed in the remote interface. The enterprise bean class must implement the *java.ejb.EntityBean* or *java.ejb.SessionBean* interface.

Both the remote interface and enterprise bean class have responsibility for two specialized categories of methods: create methods and finder methods. Create methods

provide ways to customize the bean when it is created, and finder methods provide ways to locate beans.

There are two kinds of EJBs, Entity Beans (EBs) and Session Beans (SBs). The description of these two types of EJBs was given in Chapter 2. An EB represents an object's view of business data stored in persistent storage (such as a database). The bean provides an object wrapper around the data to simplify the task of accessing and manipulating it. In our implementation, a client accesses the database via EBs, which represent specific information in the Bazaar System. An EB represents a single logical row of data in a database table. It allows shared access from multiple clients and can live past the duration of a client's session with the server (i.e. is "persistent"). Usually a table in the database is related to an EB at the server side.

EBs provide robust, long-lived persistent data management. A business object, that needs to live after a client's session with the server is over, should be modeled as an EB. EBs live even after a client's session with the server is over.

SBs are used to implement business objects that contain client-specific business logic. A Session Bean typically executes on behalf of a single client and can not be shared among multiple clients because the state of such a business object reflects its interaction with a particular client and is not for general access. SBs do not directly represent shared data in the database, although they can access and update such data. The state of a session object is non-persistent and need not be written to the database. Such state exists only for the duration of a client "session".

5.2. The Implementation of the Administration System

The administration system enables the system administrator to manage the Bazaar system. It is divided into three sub-systems: System Manager, User Manager, and Product Manager, and two Web-page Managers: AdminStorePage, and AdminSupplierPage. The activities of the sub-systems are illustrated in Figure 5.2.

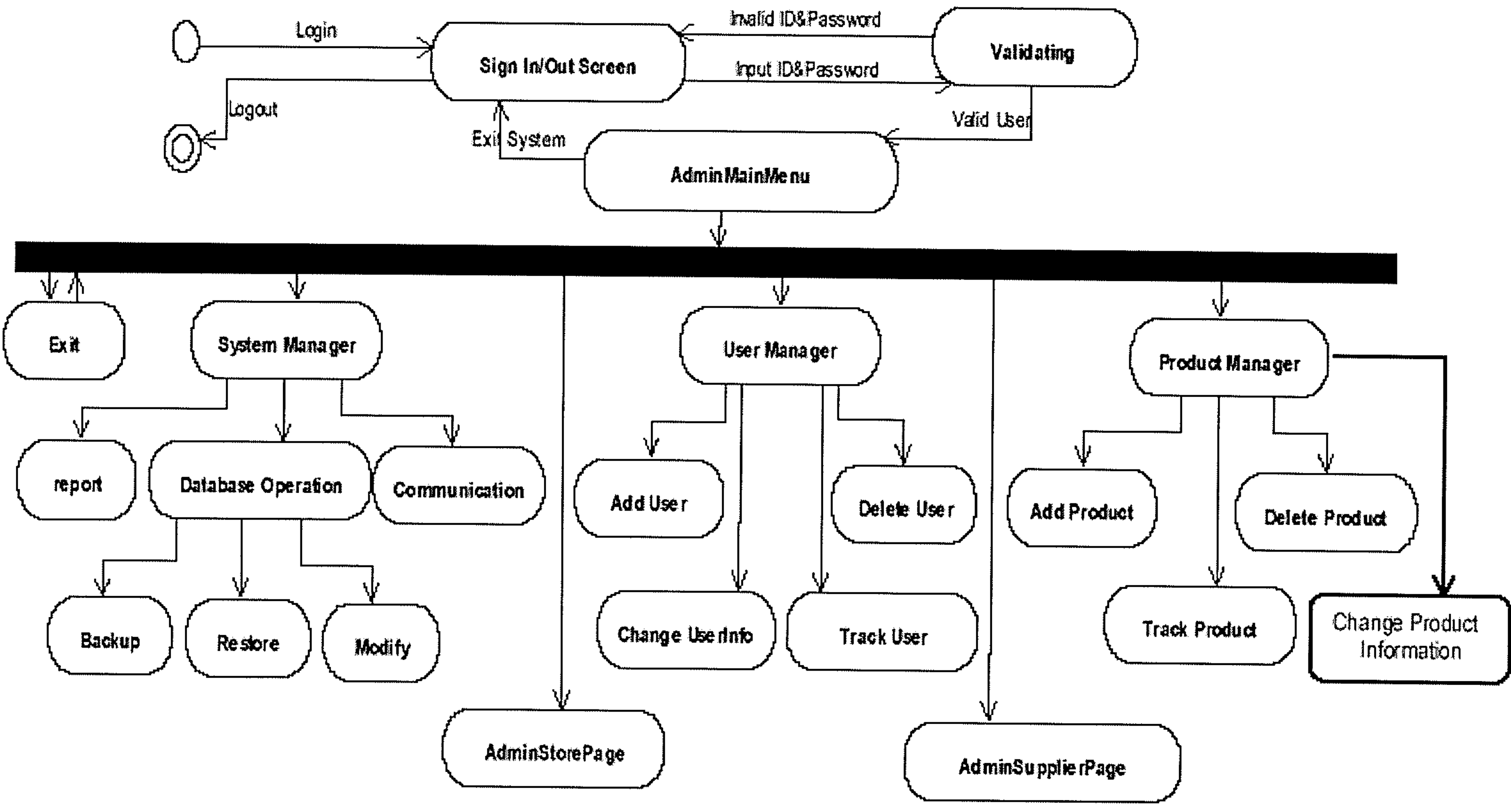


Figure 5.2 The Administration System

5.2.1 The Main Menu

The main menu of the Administration System is shown in Figure 5.3. There are two icons, which act as quick access keys, BazaarBar1 and BazaarBar2. By pressing BazaarBar1, the quick access keys, Add Consumer, Add Product, and Add Supplier will show up as shown Figure 5.4. By pressing BazaarBar2, quick search keys will appear (i.e., by product id, by purchase order number, by user id).

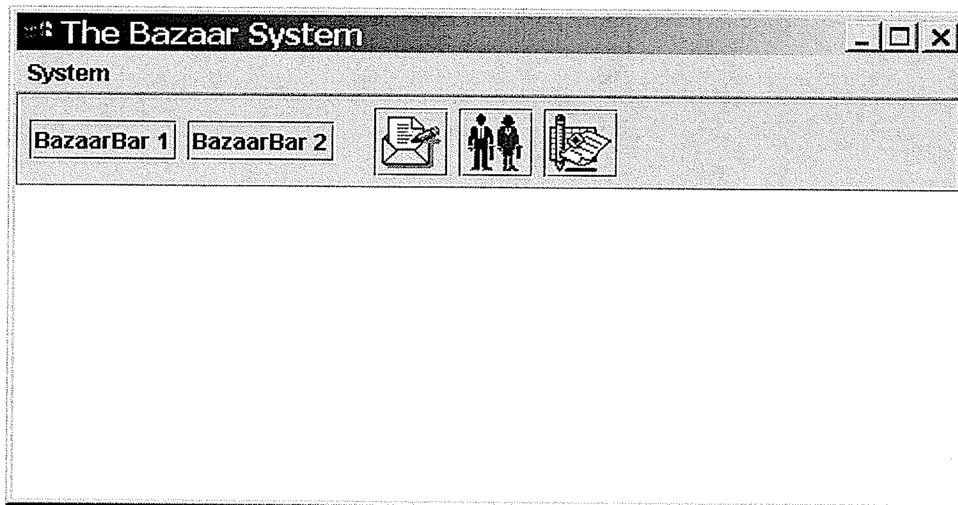


Figure 5.3 The Main Menu of the Administration System

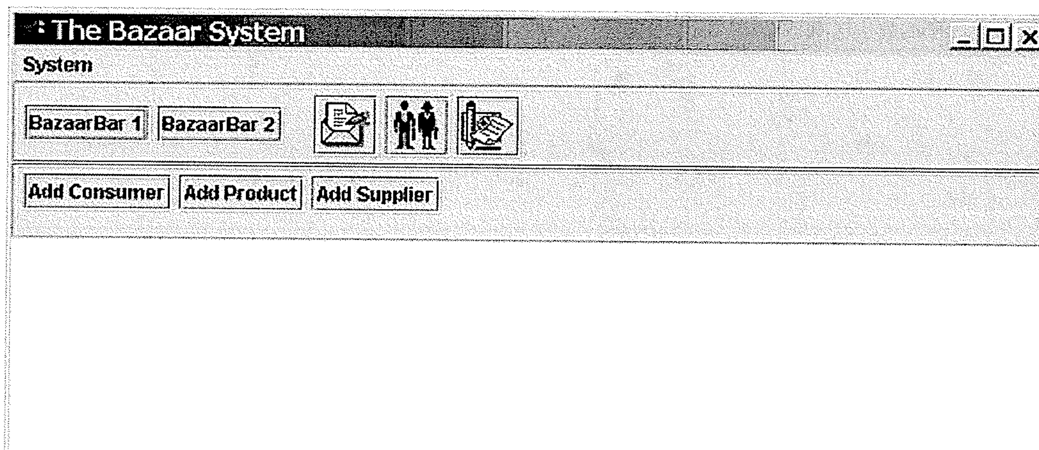


Figure 5.4 The Main Menu (Pressing BazaarBar1)

The use of quick searches is very convenient for the system administrator to search for information in the Bazaar System. Figure 5.5 shows a result of search by purchase order number.

Figure 5.5 The Result of Search by Order No.

The web page managers are situated in the main menu, they appear by clicking on "System" drop down menu on the main page. The function of the AdminStorePage is to add and modify a store's home page for the owner(s) of the store, and the AdminSupplierPage has the function of adding and modifying a supplier's home page for suppliers. Three icons (with figures in) in the main menu lead you to three sub-systems. The menu for Product Manager, User Manager and System Manager pops up in the main menu on clicking the corresponding icons.

5.2.2 The User Manager Menu

The User Manager menu is built into the main menu as an internal frame (see Figure 5.6). The User Manager Menu appears if a user clicks on the first icon (with two men). The user manager screen has a data grid for displaying all the information about user accounts. If a user clicks on a button, the related interface appears (see Figure 5.7).

There are several functions in the User Manager subsystem: add user (supports adding a consumer, adding a supplier, and adding a storeowner); delete user; and system modification (including add a store information and add supplier information).

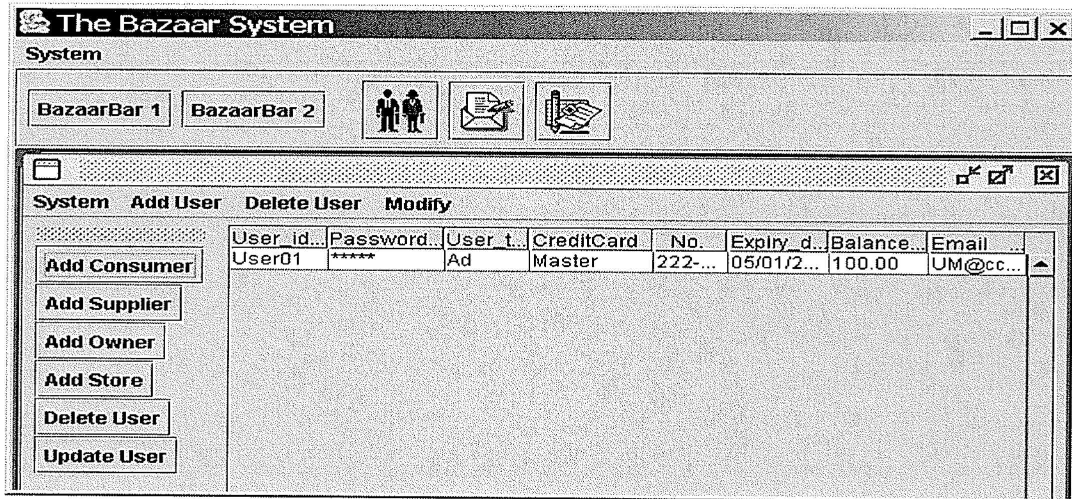


Figure 5.6 The menu of the User Manager

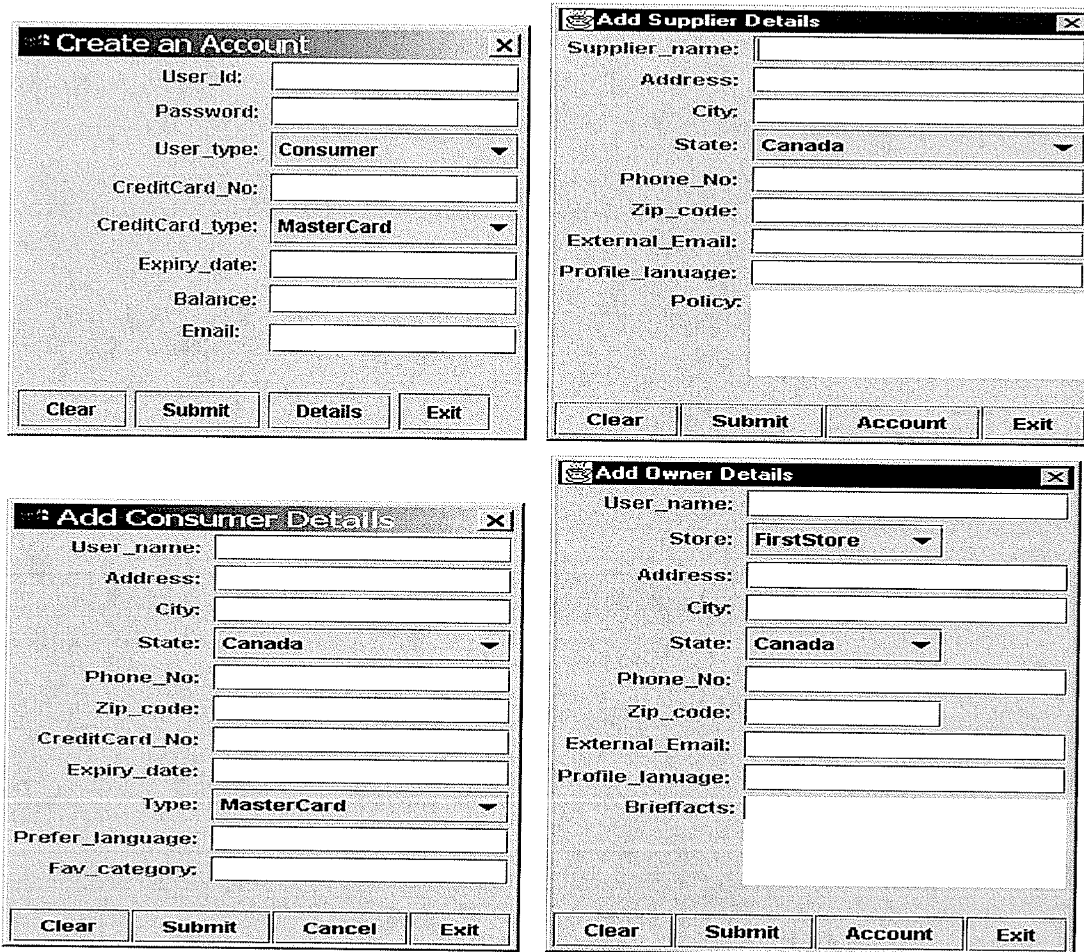


Figure 5.7 Input Boxes of the User Manager

5.2.3 The Product Manager Menu

The Product Manager menu is built into the main menu as an internal frame (see Figure 5.8). There is a table in the screen for displaying all the information related to a product. There are three items in the menu bar, "System", "Add New Product", and "Modify Product". The "System" icon is used for saving information to the Bazaar System and exiting the Product Manager sub-system. The icons in the toolbar enable the users with the functions of the Product Manager sub-system to track a product, or search for a product.

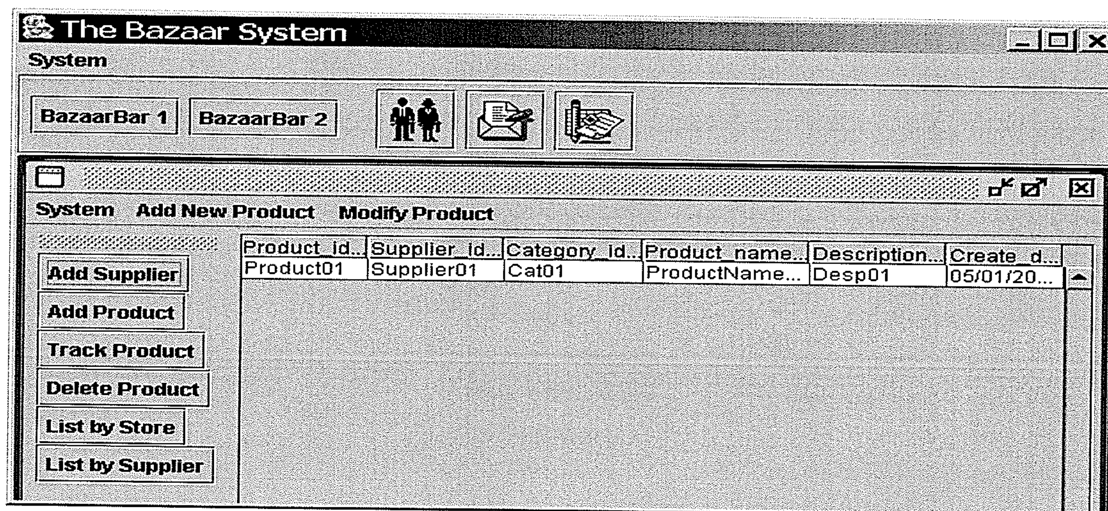


Figure 5.8 The Menu of Product Manager

5.2.4 The System Manager Menu

The menu of the System Manager sub-system is built into the main menu as an internal frame (see Figure 5.9). There are three items in the menu bar, "System", "Report", and "Database". The "System" menu is for a user to modify the system. The "Report" menu can be used to generate and display various reports for users, such as, user account structure reports, monthly new account information reports, and new product reports. The "Database" menu is used to modify the database structure (e.g., add tables), and search for information in database. Additional functions are associated with the toolbar buttons, such as the toolbar "create a home page" for a user which is used to create a home page for a store or supplier.

The user interface allows multiple windows to be tiled and/or piled up in one screen. More than one sub-system's menus can exist in the main menu as internal frames at the same time without conflicts (see Figure 5.10).

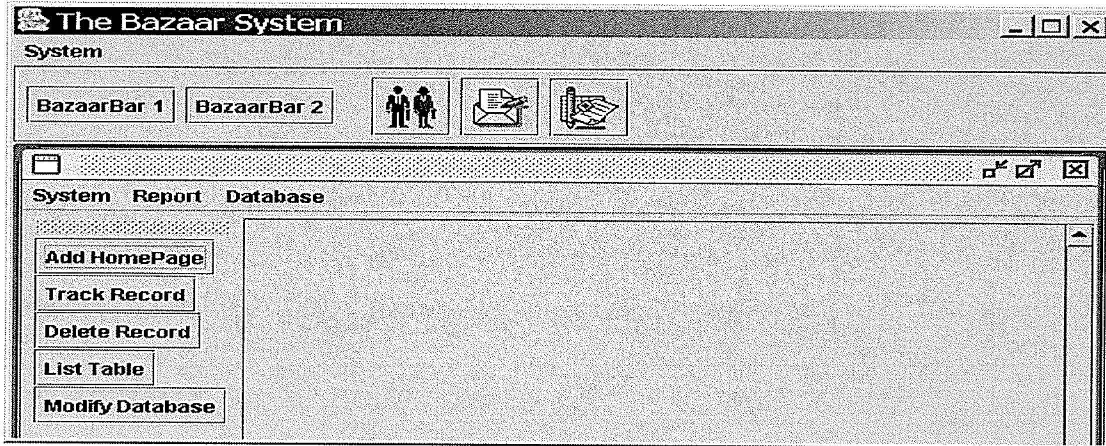


Figure 5.9 The menu of System Manager

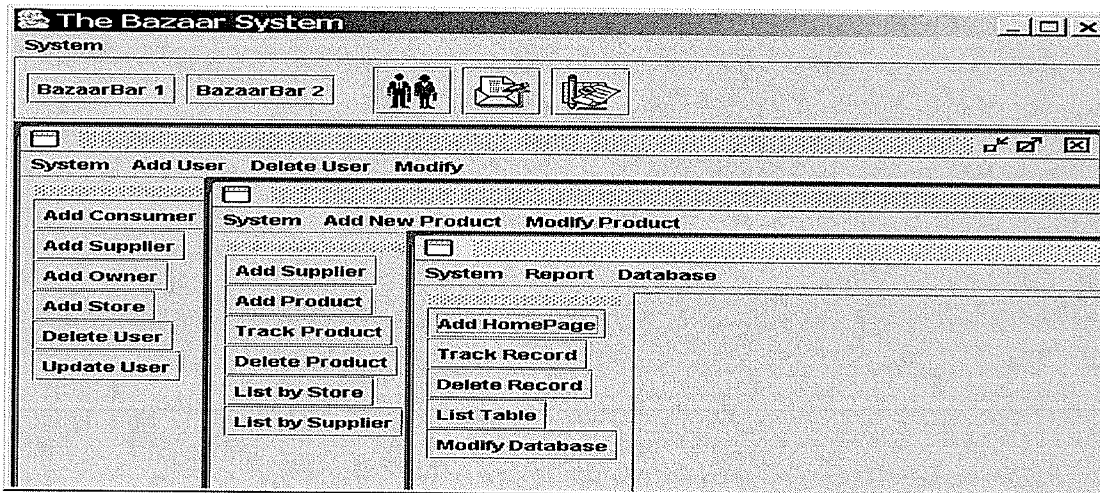


Figure 5.10 The Menus of Sub-Systems

5.3. The Implementation of the Front-End

In this section, we first introduce four design patterns [Hal2000] and how they are used to implement the Front-End components. Then give some implementation examples of Front-end interfaces in the Bazaar System.

5.3.1 The Design Patterns

JSP technology is used to implement the client side functionality. Since the client software executes on user systems, it is not possible to control every aspect of the client environment (e.g., hardware, operating system platform, and browser version). There are trade-offs to be made in partitioning application responsibility between server and client in any distributed application [Kas2000]. Using design patterns can solve those problems. So several design patterns are described in implementing the client side application. The benefits of using these design patterns are many just as formalizing the system. Four design patterns are introduced in this thesis: *Model View Controller Pattern* (MVC), *Front Component Pattern* (FCP), *Value Object Pattern* (VOP) and *Command Pattern* (CP). These patterns are described in detail below.

5.3.1.1 The Model View Controller Pattern (MVC)

Simple GUI based applications are commonly organized around event-driven user interfaces. The developer creates the graphical elements in a user interface with a tool and then writes blocks of code that execute application actions in response to user input. Some design methodologies emphasize starting with the user interface, and developing the final system around it. The result is a system organized around user interface elements and user actions on those elements, with persistent data manipulation, application functionality, and display code completely intertwined. This approach can be successful for a small system, but is often inadequate for large, complex, distributed systems, such as the Bazaar System. Sophisticated applications, that require long-term maintenance, need to be structured so that a maintainer can learn and understand them. In this approach, code is less reusable, because each component depends on other components in order to do anything.

In order to increase reusability, the MVC pattern facilitates maintenance, extensibility, and flexibility by partially decoupling data presentation, data representation, and application operations. The MVC pattern also enables multiple simultaneous data views [Kas2000].

The MVC design pattern separates the application data from both the ways the data can be viewed and accessed, and also from the mapping between system events and application behaviors. This improves distributed application design. There are three component types in the MVC pattern: the Model component, the View component, and the Controller component. The Model component encapsulates the application state, provides access to functions and notifies interested parties when data changes. The View component presents data to the user and maintains consistency with the model data. The Controller component translates user inputs into application actions and selects appropriate data displays based on user input and context.

The MVC pattern is used in the Bazaar System. Figure 5.11 shows how the MVC pattern is applied in the Bazaar System.

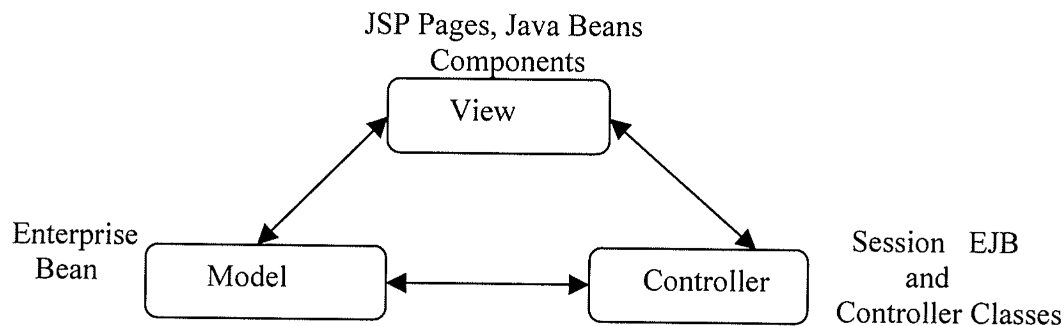


Figure 5.11 MVC in the Bazaar System

5.2.1.2 The Front Component Pattern (FCP)

To simplify implementation and maintenance of user interface presentation and workflow, the FCP is used to implement the Bazaar System.

There are many ways to present data, as interactive applications grow. A well-factored application will partition responsibility for specific tasks to various classes. But the fine-grained partitioning of responsibility between classes increases the number of objects, as well as the number of connections between objects. A class that explicitly references many other classes is less reusable. Additionally, the interface presented by any particular

class becomes arbitrarily complex, making the system difficult to maintain and extend. These problems can be overcome by applying the front component pattern.

Many Web applications are structured as highly interdependent system. The collection of Web pages is very brittle, a change in any of which may affect many other pages, and the changes may be difficult to trace. The FCP can avoid this problem by centralizing the dispatch of HTTP service requests within a single Servlet. The Servlet class maps user requests to application model operations and determines which user views to present next.

In the Bazaar System, the Front Component is the single point of entry for HTTP requests to Controller in the MVC design (i.e., a component to which all requests for application URLs are delivered). The Front Component, *Main.jsp*, processes these requests and delegates the generation of the response to the template page.

5.3.1.3 The Value Object Pattern (VOP)

The data for an address object can be retrieved once, sent to the client from the server, and instantiated on the client. The local copy of the address can serve as a proxy. Subsequent accesses to the copy of an address object's state are local without communicating to the server. Such a client-side copy of an object is called a value object. Accessing the value object can reduce network traffic and improve response time [Kas2000].

The VOP has three types of components: *LocalEntityObject*, *EntityObject* and *ValueObject*. The *LocalEntityObject* presents a local interface to a remote object and represents one or more properties of the remote object. The *EntityObject* has meaningful identity; it represents one or more properties as immutable objects. The *ValueObject* represents the value of a property of some other object, provides access to its state via property assessors.

The Value Object Pattern is also used to implement the Bazaar System.

5.3.1.4 The Command Pattern (CP)

As mentioned above, the complexity of interconnections between components is encapsulated in the Front Component. If the Front Component does not have a clear, consistent, and extensible mechanism for making these connections, it will become a monolithic chunk of code that is difficult maintain, understand, and extend. If the FCP pattern is used in the system, the Command pattern is needed to manage Front Component complexity. The Command pattern is used to manage the potential complexity of the Front Component. With the command pattern, extensions to the front component functionality do not change the Front Component's API. Each extension simply defines a new command handled by the command pattern and a way to handle the command.

5.3.2 Several Interfaces of the Front-End

The client in the Bazaar System is the user's browser. It displays HTML from the Front Component, and posts user input to the Front Component. The Front Component in the Bazaar System is implemented by the servlet, *MainServlet.java*, which is mapped to the single URL namespace. *MainServlet* uses two classes, *Processor.java* and *Request_to_Event_Translator.java* to map user actions to model actions, and the class *ScreenFlowManager.java* to select and create the next presentation (i.e. HTML page). The Front Component transmits HTML or JSP pages to the client.

The HTML and JSP pages are the main part of the implementation of the Front-End. The organization of each page is depicted in Figure 5.12.

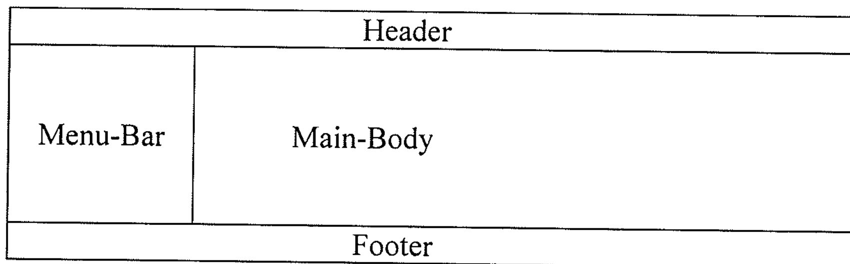


Figure 5.12 The Organization of Page for Front-End

5.3.2.1 The Main Page of the Front-End

The main page in the Bazaar system is shown in Figure 5.13. Recall that there are four different kinds of users in the Bazaar System, store owners, suppliers, registered users and buyers. An unregistered user can only search public information and do regular shopping. Before registration, a buyer can search products and put them into the shopping cart, and buy the product(s) using a credit card. Without registering in the Bazaar System the buyer can not receive any internal discounts. There are only four items in the menu-bar area, *Information*, *OnSale*, *Shopping*, and *Registration.* An unregistered user can check public information, such as, the stores and the kinds of products in the system, and find product information for a store. An unregistered user can also check on sale information, but only if the information is public. A registered user can login to the system by entering his/her user name and password directly from the main page.

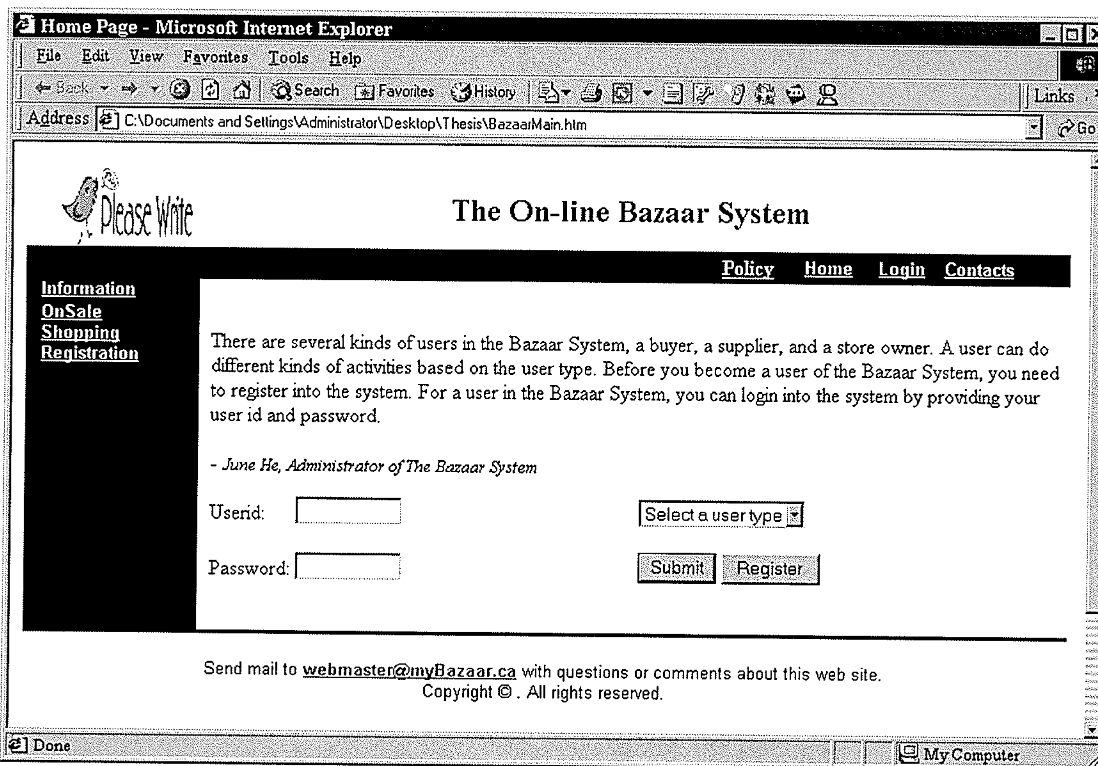


Figure 5.13 The Main Page of the Bazaar System

5.3.2.2 The Home Page for a Buyer

After registration, a buyer can do all shopping activities. Figure 5.14 shows the buyer home page in the Bazaar System. The items in Menu-Bar area (see Figure 5.12) list the operations for registered buyers, including searching information related to a product, a store, or a supplier, finding all sale products (internal or public). *ShoppingCart* can also be used to hold all selected items for a user in the Bazaar System even after the user logout from the system. There is an account for each registered user, and users can access and modify the information in their accounts.

Figure 5.15 shows how an order activity is implemented in the Bazaar System. Four objects on the server are involved in the order activity. They are *ShoppingCart*, *Order*, *Inventory*, and *Store*. A buyer searches the system to find the products he/she wants to buy, then puts them in the shopping cart. After some research, the buyer makes his/her order for the items in the shopping cart. The order information is passed to the Order object, then all items are checked for availability in the inventory.

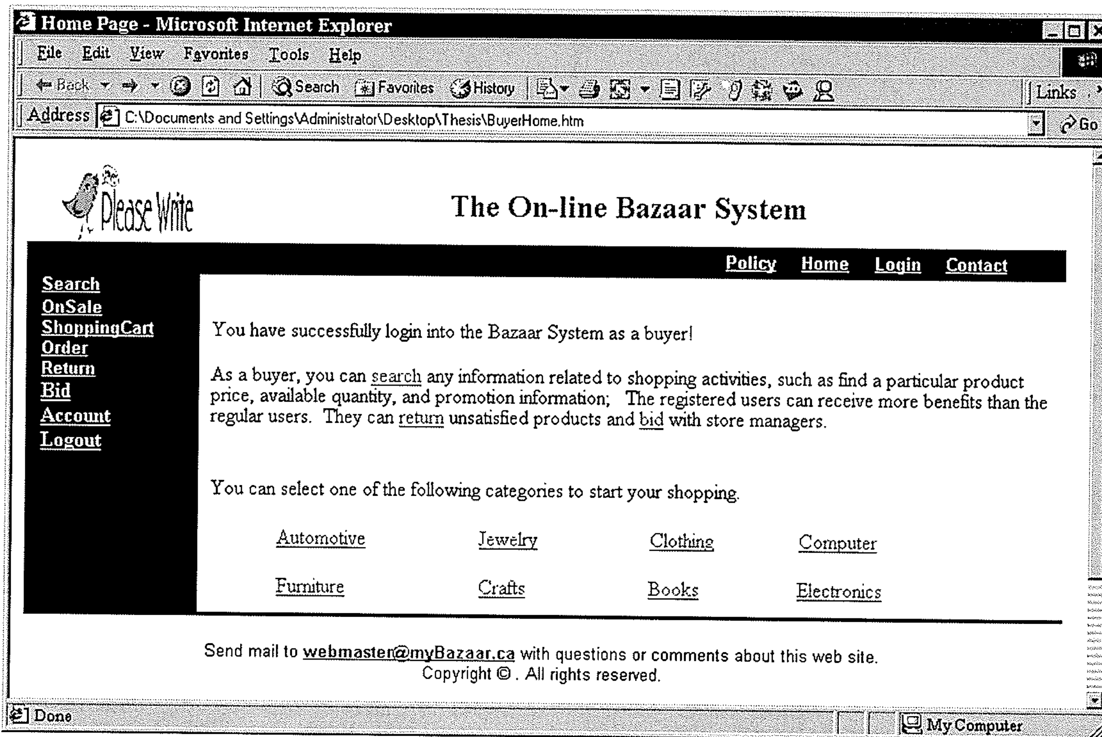


Figure 5.14 The screen of Buyer Home Page of the Bazaar System

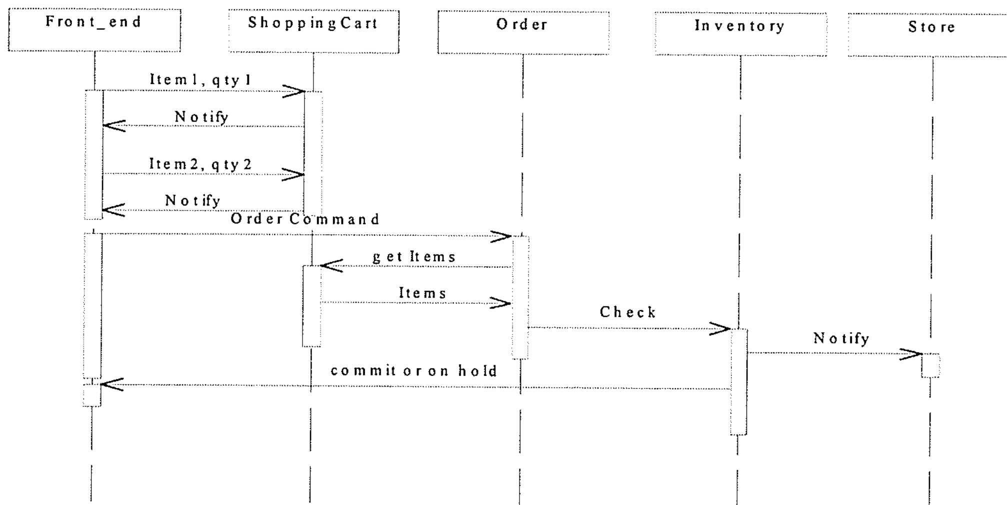


Figure 5.15 The Order Processing

5.3.2.3 The Home Page for a Store Owner

After a user logs in to the Bazaar System as a store owner, the home page for a store owner appears (as shown in Figure 5.16). The main functions includeing *Search*, *Order*, *Modify*, *Bid*, and *Report*, are listed in the Menu-Bar area (see Figure 5.12). An owner can search all the information in the system (except some stores' internal information), order products or services from suppliers and bargain with them, modify the rights and limits for a user to access the owner's store and other product or user information of the store.

The store policy is accessible by clicking the *Policy* item. A store owner's home page in the Bazaar System is used internally in the Bazaar System to list the operations for the particular store owner(s). The home page of the store can be accessed by clicking the *Home* menu item. Figure 5.16 also lists and explains some important operations an owner can perform in the Main-Body of the window. For example, an owner can click the *check inventory* item to check the related store's inventory or click the *products list* item to find the products available at the store.

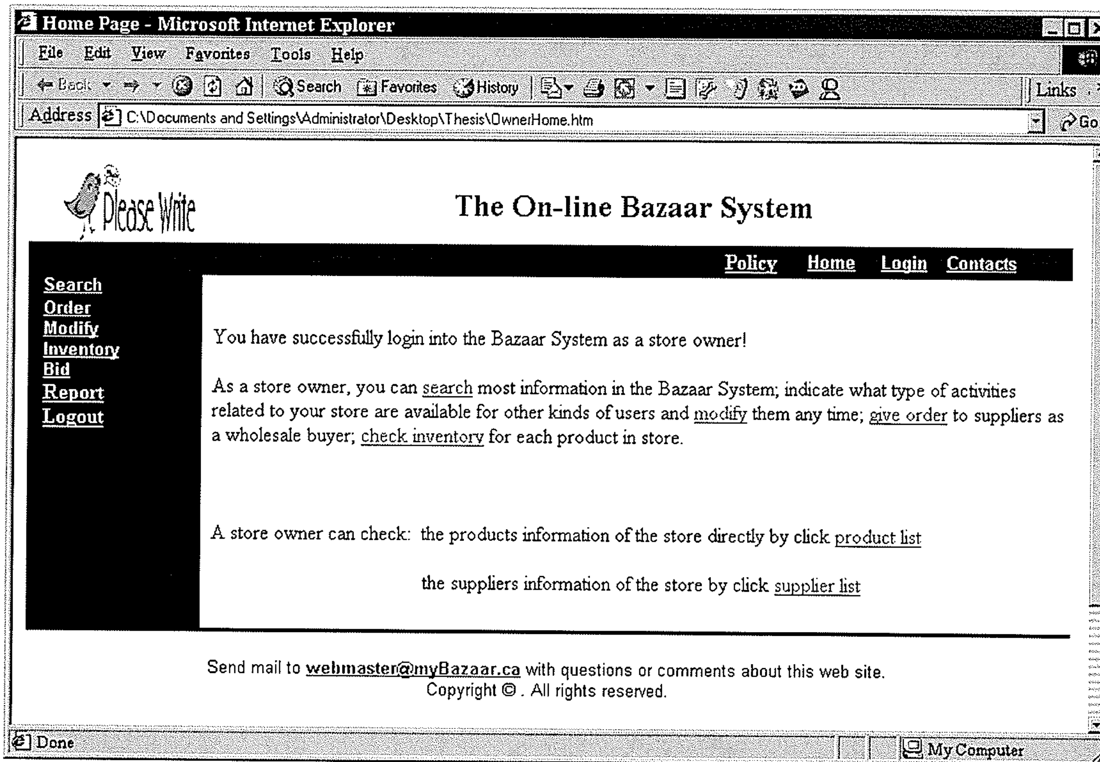


Figure 5.16 The Screen of Store Owner Home Page of the Bazaar System

5.3.2.4 The Home Page for a Supplier

After a user logs in to the Bazaar System as a supplier, the window shown in Figure 5.17 appears. A supplier can access his/her own home page by clicking the *Home* item, and can find the supplier's policy by clicking the *Policy* item. The main operations provided to a supplier are listed in the Menu-Bar area and include: Search, Report, Check Inventory, Give Order, and Bid. A supplier can access and modify his/her account by clicking the Account item.

The Main-Body in Figure 5.17 contains a description of most operations a supplier can do in the Bazaar System.

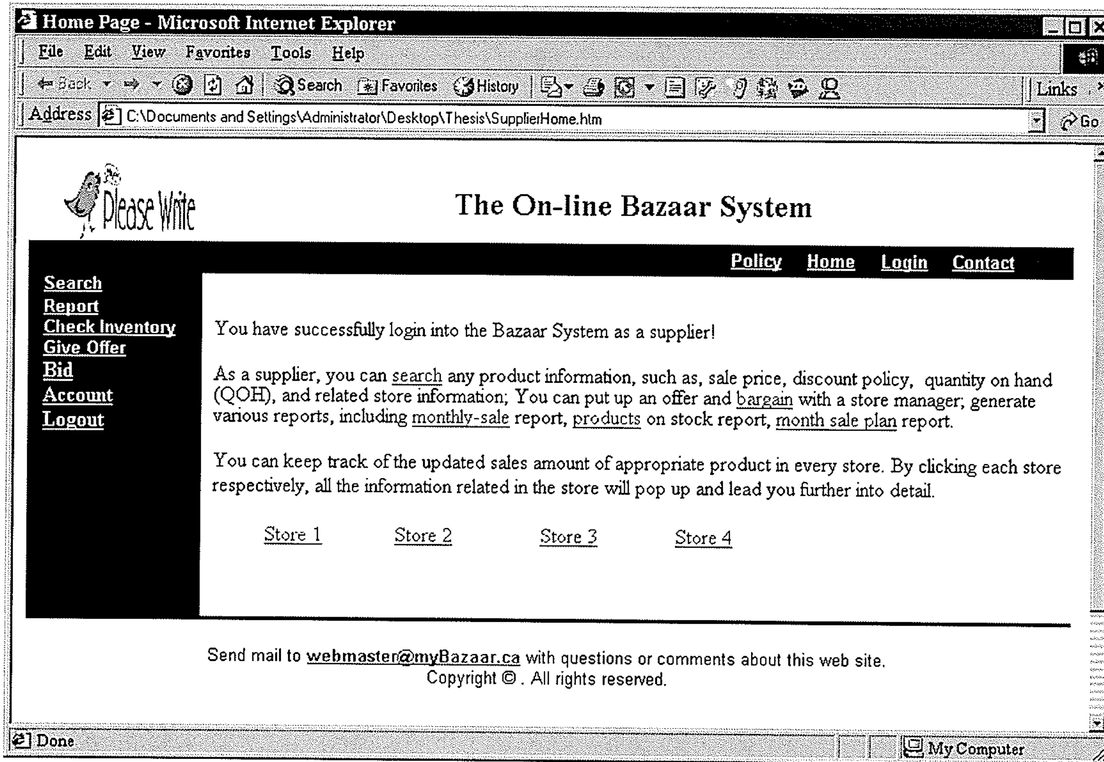


Figure 5.17 The Screen of Supplier Home Page of the Bazaar System

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

The thesis focused on the specification of an online Bazaar System. It investigated the development of the Bazaar System using formal methods. Formal methods were used to specify the software requirements and design. Two kinds of formal languages, UML and Object-Z, were used for the thesis.

UML was used for the requirements analysis and to model the architecture design of the Bazaar System. It is known nowadays that an object-oriented language is necessary but insufficient to create object systems.

The formal specification of the Bazaar System was done using the Object-Z formalism. A type checking tool, wizard [Joh1996], was used to check the correctness of the Object-Z specification document. In this thesis, Object-Z was used in the functional specification of requirement analysis.

Object-Z was used to specify and design the essential functionalities of the Bazaar System. Object-Z was not used for the formal specification of all the methods implemented in the Bazaar System. Future effort needs to address using Object-Z to specify and design the implementation details of all the methods in the Bazaar System.

The thesis used J2EE technologies to implement the Bazaar System. The J2EE platform provides system services that simplify the work that application objects need to perform. The thesis emphasized the business functions implementation of the Bazaar System. For a real application system, system security and its implementation are major issues. In future, we still need to do some performance analysis, such as determining how robust and trustworthy is the Bazaar System using performance tools. Predicting application performance when deploying e-commerce applications and performance analysis should be considered.

In the future, we will need to explore the possibility of using UML for software testing to address the entire testing lifecycle in detail and do performance analysis of the Bazaar System. The formal integration tests using UML collaboration could be future research interest and it is beyond the discussion of this thesis. Also, we need to use other formalisms to capture design specifications, such as, finite state machines and I/O automata, to specify the requirements of the Bazaar System.

REFERENCES

[Ala1998] V. S. Alagar and K. Periyasamy, *Specification of Software Systems*, Spring-Verlag, 1998.

[Bar1992] Rosalind Barden, Susan Stepney, and David Cooper, "The Use of Z", J. E. Nicholls (Editor), *Proceedings of the 6th Z User Meeting*, York 1991, Workshops in Computing, 99-124. Springer-Verlag, 1992.

[But2001] Michael Butler, "Introductory Notes on Specification with Z", Dept. of Electronics and Computer Science, University of Southampton, March 2001.

[Cha1996] Anthony Chavez and Pattie Maes, "Kasbah: An Agent Marketplace for Buying and Selling Goods", *Conference on Practical Applications of Intelligent Agents and Multi-Agent Technology*, April 1996.

[Duk1991] Roger Duke, P. King, and Graeme Rose and Graeme Smith, "The Object-Z Specification Language", *Technology of Object-Oriented Language and System* (TOOLS), Prentice-Hall, 1991, Page 465-483.

[Duk1994] Roger Duke, Gordon Rose and Graeme Smith, "Object-Z: a Specification Language Advocated for the Description of Standards", *Technical Report No. 94-45*, Software Verification Research Centre Deparment of Computer Science, Queensland 4072, Australia, December 1994.

[Ehi2000] Ehikioya S. A. and Suresh J., "Electronic Commerce for Services and Intangible Goods", *First international Conference on Internet Computing*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 26-29, 2000.

[Ehi2001] Ehikioya S. A.; "A Formal Specification of Auction Systems using Z Notation", *International Journal of Computer Science and Information Systems*, March 2001, (Submitted).

[EZH2001] P. Ezhilchelvan and G. Morgan, "A Dependable Distributed Auction System: Architecture and an Implementation Framework", *The Fifth International Symposium on Autonomous Decentralized Systems (with Emphasis on Electronic Commerce)*, March 26-28, 2001, Dallas, Texas.

[Fri2000] John Friend, "Be more than a bazaar, business Web sites told", *Business*, November 16th, 2000.
(available at http://inq.philly.com/content/inquirer/2000/11/16/business/B2B16.htm)

[Hal2000] Marty Hall, *Servlet and JavaServer Pages*, Sun Microsystem Press, 2000.

[Joh1996] Wendy Johnston, "A Type Checker for Object-Z", *Technical Report No. 96-24*, Software Verification Research Center Department of Computer Science, Queensland 4072, Australia, September 1996.

[Kas2000] Nicholas Kassem and The Enterprise Team, *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition*, Sun Microsystem Laboratories, Aug 2000.

[Kre1997] Rob Kremer, "SENG 611 Requirements Engineering, Formal Specification", *Software engineering Research Network,* University of Calgary, Alberta, Canada, 1997.

[Lom2000] Alessio R. Lomucio, Michael Woodridge and Nicholas R. Jennings, "A Classification Scheme for Negotiation in Electronic Commerce", *European Perspective on Agent-Mediated Electronic Commerce, (eds, C. Sierra and Dignum)*, Springer Verlag, March 2000.

[Mor2000] Joan Moris, Peter Ree and Pattie Maes, "Sardine: Dynamic Seller Strategies in an Auction Marketplace", *Proceedings of the Conference on Electronic Commerce (EC '00)*, Minneapolis, MN, October 17-20, 2000.

[Mou1998] Alexandros Moukas, Robert Guttman, and Pattie Maes, "Agent-mediated Electronic Commerce: An MIT Media Laboratory Perspective", *Proceedings of the International Conference on electronic Commerce*, Korea, 1998.

[Nis2000] Noam Nisan, "Bidding and Allocation in Combinatorial Auctions", *ACM Conference on Electronic Commerce*, April 17, 2000.

[O-Z1997] Alena Griffiths, "Modular Reasoning in Object-Z", *Technical Report No. 97-28*, Software Verification Research Center Department of Computer Science, Queensland 4072, Australia, August 1997.

[Pol1999] Fiona Polack and Susan Stepney, "Systems Development using Z Generics", *FM '99: World Congress on Formal Methods*, Toulouse, France, 1999, Proceedings Volume II. Volume 1709 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pages 1048-1067.

[Ses2000] Roger Sessions, "EJB vs. COM+: It comes down to language, performance", *Objectwatch Inc.*, 2000, (Available on http://www.objectwatch.com).

[Ros1992] Graeme Rose, "Object-Z", in S. Stepney, R. Barden and D. Gooper, editors, *Object Orientation in Z, Workshops in Computing*, Springer-Verlag, 1992, page 59-77.

[Ste1992] Susan Stepney, Rosalind Barden, and David Cooper, "Object Orientation in Z", *Workshops in Computing Series,* Springer-Verlag, 1992.

[Ter1997] Ioannis S. Terpsidis, Alexandtos Moukas, Bill pergioudakis, Georgios Doukids and Pattie Maes, "The Potential of Electronic Commerce in Re-engineering Consumer-Related Relationships Through Intelligent Agents", *Advances in Information Technologies: The Business Challenge,* IOS Press, 1997.

[UML1999] Unified Modeling Language (UML) Specification, version 1.3a, Object Management Group, March 1999. (available at: http://www.rational.com/uml)

[UML2000] Soon-Kyeong Kim and David Carrington, "UML Metamodel Formalization with Object-Z: The State Machine Package", *Technical Report No. 00-29,* Software Verification Research Center Department of Computer Science, Queensland 4072, Australia, August 2000.

APPENDIX A

DETAILED DESIGN

The functional requirements of the following objects are now described in greater detail.

Account Object:

Index: Obj-1
Object Name: Account
Inheritance: None
Imported Objects: None
Purpose: Holds information for a user to access the system and to execute appropriate operations
Remarks: Created before the user can access the system
Properties: Entity bean
Methods:

Method to set the attributes:

Index: Con-1
Name: AccountEJB
Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection
Remarks: No

Methods of the Object:
Index: Op-1
Name: registration
Purpose: To add an user to the system
Visibility: External; visible
Input parameters: userId; password; creditCard; email; userDetails
Output parameters: Confirm information (userId, password)
Pseudocode: createAccount(userId, password, creditCard, email);
addDetails(userId, userDetails);
return (userId, password);
Remarks: No

Index: Op-2
Name: deleteAccount

Purpose: To delete an account
Visibility: External; visible
Input parameters: userId
Output parameters: None
Pseudocode:

```
if userExists(userid) {
        deleteAccount(userId);
        deleteDetail(userId);
        return true;
}
else
        return false;
```

Remarks: No

Index: Op-3
Name: login
Purpose: To login into the system
Visibility: External; visible
Input parameters: userId; password
Output parameters: A message (successfully or fail)
Pseudocode:

```
if (userId) in userArray(userId, status) {
        print("user is Already in the system");
}
else if (userExists(userid) & validP(password)) {
        add userId to userArray;
        the Status of the user is set to active;
        print("login into the system successfully");
}
else
        print("the userid or password is not correct");
```

Remarks: No

Index: Op-4
Name: changeAccount
Purpose: To change a user account's information
Visibility: External; visible
Input parameters: userId; newData
Output parameters: None
Pseudocode:

```
if userExists(userId)
        modify database(UPDATE account_table WITH
        newData, WHERE userid = userId);
```

Remarks: No

Index: Op-5
Name: getAccountDetails
Purpose: To retrieve a user's account details

Visibility: External; visible
Input parameters: userId
Output parameters: list of (userId, password, creditCard,balance, email)
Pseudocode: if userExists(userId)
retrieve data from database (SELECT (*name, address, password, credit, balance, email*) FROM account_table WHERE *userid* = userId);
Remarks: No

Index: Op-6
Name: getUserDetails
Purpose: To retrieve a user's detail
Visibility: External; visible
Input parameters: userId
Output parameters: list of details
Pseudocode: if userExists(userId)
retrieve data from database (SELECT (*name*, *address*, etc.) FROM user_table WHERE *userid* = userId);
Remarks: The table name user_table and details will chang depends on the caller consumer, supplier or owner).

Index: Op-7
Name: changeUserInfo
Purpose: To change a user's information
Visibility: External; visible
Input parameters: userId; newData
Output parameters: None
Pseudocode: if userExists(userId)
modify database(UPDATE user_table WITH newData, WHERE *userid* = userId);
Remarks: No

Index: Op-8
Name: logout
Purpose: To exit the system
Visibility: External; visible
Input parameters: userId; password
Output parameters: None
Pseudocode: if (userId, password) in userArray
delete (*userid*, *password*) from userArray;
the Status of the user is set to inactive;
Remarks: No

Index: Op-9
Name: createAccount

Purpose: To create an account
Visibility: External; visible
Input parameters: userId; password; status; credit; email
Output parameters: None
Pseudocode:
```
If (isValidDate(userId, password)) {
        INSERT (userId, password, status,credit, email)
        INTO account_table;
        return true;
}
else
        return false;
```
Remarks: No

Index: Op-10
Name: addDetails
Purpose: To add user information to the system
Visibility: External; visible
Input parameters: userId; info
Output parameters: None
Pseudocode:
```
if userId in userArray
        Store info to database;
```
Remarks: No

Methods used internally by the object

Index: In-1
Name: userExists
Purpose: To check if the user exists in the system
Visibility: Internal; invisible
Input parameters: userId
Output parameters: result (true or false)
Pseudocode:
```
string = SELECT userid FROM account
        WHERE userid = userId;
if (string = null)
        return false;
else
        return true;
```
Remarks: No

Index: In-2
Name: isValidData
Purpose: To check if the data is valid
Visibility: Internal; invisible
Input parameters: userId; password

Output parameters: result (true or false)

Pseudocode:

```
if ((userId = null) || (password = null))
        return false;
else
        return true;
```

Remarks: No

Index: In-3
Name: validP
Purpose: To check if the user's password is correct
Visibility: Internal; invisible
Input parameters: userId; password
Output parameters: result (true or false)
Pseudocode:

```
search database
string = SELECT password FROM account
        WHERE userid = userId;
if (string = password)
        return true;
else
        return false;
```

Remarks: No

Index: In-4
Name: deleteAccount
Purpose: To delete an account
Visibility: Internal; invisible
Input parameters: userId
Output parameters: None
Pseudocode:

```
remove an account entry from database {
        DELETE FROM account_table
        WHERE userid = userId;
}
```

Remarks: No

Index: In-5
Name: deleteDtails
Purpose: To delete an user's detail from database
Visibility: Internal; invisible
Input parameters: userId
Output parameters: None
Pseudocode:

```
remove an account entry from database {
        DELETE FROM user_table WHERE
        userid = userId;
}
```

Remarks: No

Order Object:

Index: Obj-2
Object Name: Order
Inheritance: None
Imported Objects: None
Purpose: To create purchase order instance
Remarks: After an order is created, the order details including user's information and orderline items can be changed at any time before the order's status is set to be "ACTIVE"
Properties: Entity bean
Methods:

Method to set the attributes:

Index: Con-1
Name: OrderEJB
Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode:

```
find datasource connection;
orderId = 0;     total price = 0;
orderDate = Clendar.getInstance();
```

Remarks: No

Methods of the Object:
Index: Op-1
Name: createOrder
Purpose: To create a purchase order for a user
Visibility: External; visible
Input parameters: lineItems; storeId; userId; creditCard; carrier; totalPrice
Output parameters: No
Pseudocode:

```
if (isValidData(lineItems; storeId; userId; creditCard;
                        carrier; totalPrice))    {
        orderId = getUniqueOrderId();
        orderDate = currentDate();
        updateDate = orderDate;
        status = "PENDING";
        insertOrder(orderId, storeId, userId, creditCard,
                carrier, orderDate, status, totalPrice);
        insertLineItems(orderId, lineItems);
```

```
        insertOrderStatus(orderId, updateDate, deliverDate,
                status)
        return true;
}
else
        return false;
```

Remarks: No

Index: Op-2
Name: deleteOrder
Purpose: To delete an order instance
Visibility: External; visible
Input parameters: orderId
Output parameters: Confirm information (successful or fail)
Pseudocode:

```
if orderExists(orderId) {
        deleteOrder(orderId);
        deleteLineItems(orderId);
        deleteOrderStatus(orderId);
        return true;
}
else
        return false;
```

Remarks: No

Index: Op-3
Name: changeOrder
Purpose: To change an order's information
Visibility: External; visible
Input parameters: orderId; newOrder(storeId,userId, creditCard, carrier, orderDate, status, totalPrice)
Output parameters: None
Pseudocode:

```
if orderExists(orderId)
        UPDATE order_table SET
        order newOrder, WHERE
        orderid = orderId;
```

Remarks: No

Index: Op-4
Name: changeOrderStatus
Purpose: To change an order's status
Visibility: External; visible
Input parameters: orderId; newStatus
Output parameters: None
Pseudocode:

```
if orderExists(orderId)
   Modify database
   {
```

```
        (if status = "READY" {
            deliverDate = currentDate();
            UPDATE order_status_table SET
            deliverdate = deliverDate,
            status = newStatus, WHERE
            orderid  =  orderId;
        }
        else if {
            updateDate = currentDate();
            UPDATE order_status_table SET
            updatedate = updateDate,
            status = newStatus, WHERE
            orderid  =  orderId;
        }
    }
```

Remarks: Only store owners and system manager can modify order status.

Index: Op-5
Name: changeLineItems
Purpose: To change an order's lineItems(add, delete or update item)
Visibility: External; visible
Input parameters: orderId; newLineItems
Output parameters: None
Pseudocode:

```
if orderExists(orderId)
delete(newLineItems);
insert(newLineItems);
```

Remarks: None

Index: Op-6
Name: findUserOrders
Purpose: To retrieve a user's all orders' Ids from database
Visibility: External; visible
Input parameters: userId
Output parameters: list of orderIds
Pseudocode:

```
orderIdList = SELECT orderid FROM order_table
        WHERE userid =  userId);
return orderIdList;
```

Remarks: No

Index: Op-7
Name: findStoreOrders
Purpose: To retrieve a store's all orders' Ids from database
Visibility: External; visible
Input parameters: storeId
Output parameters: list of orderIds

Pseudocode: orderIdList = SELECT *orderid* FROM order_table
WHERE *storeid* = storeId);
return orderIdList;
Remarks: None

Index: Op-8
Name: getOrderDetails
Purpose: To retrieve an order's information from database
Visibility: External; visible
Input parameters: orderId
Output parameters: list of (orderId, storeId, userId, creditCard, carrier, orderDate, status, totalPrice)
Pseudocode: list = SELECT (*orderid*, *storeid*, *userid*, *creditcard*, *carrier*, *orderdate*, *status*, *totalprice*)
FROM order_table
WHERE *orderid* = orderId);
return list;
Remarks: None

Index: Op-9
Name: getOrderStatus
Purpose: To retrieve an order's status from database
Visibility: External; visible
Input parameters: orderId
Output parameters: list of (orderId, updateDate, deliverDate, status)
Pseudocode: list = SELECT (*orderid*, *updatedate*, *deliverdate*, *status*)
FROM order_status_table
WHERE *orderid* = orderId);
return list;
Remarks: None

Index: Op-10
Name: getLineItems
Purpose: To retrieve an order's items
Visibility: External; visible
Input parameters: orderId
Output parameters: list of items
Pseudocode: list = SELECT (*orderid*, *itemid*, *quantity*, *uniprice*)
FROM line_item_table WHERE *orderid* = orderId);
return list;
Remarks: None

Index: Op-11
Name: getPendingOrders
Purpose: To retrieve all pending orders' Ids from database
Visibility: External; visible

Input parameters: None
Output parameters: list of orderIds
Pseudocode: orderIdList = SELECT *orderid* FROM order_table
WHERE *status* = "P");
return orderIdList;
Remarks: None

Methods used internally by the object

Index: In-1
Name: orderExists
Purpose: To check if the oder exists
Visibility: Internal; invisible
Input parameters: orderId
Output parameters: result (true or false)
Pseudocode:

```
string = SELECT orderid FROM order_table
        WHERE orderrid = orderId;
if (string = null)
        return false;
else
        return true;
```

Remarks: None

Index: In-2
Name: getUniqueOrderId
Purpose: To get a unique oderId
Visibility: Internal; invisible
Input parameters: None
Output parameters: orderId
Pseudocode:

```
orderId = nextSeqNum(order_table)
return orderId;
```

Remarks: None

Index: In-3
Name: isValidData
Purpose: To check if the data is valid
Visibility: Internal; invisible
Input parameters: data
Output parameters: result (true or false)
Pseudocode:

```
if any input data is invalid
        return false;
else
        return true;
```

Remarks: None

Index: In-4
Name: insertOrder
Purpose: To add an order entry to database
Visibility: Internal; invisible
Input parameters: orderId; storeId; userId; creditCard; carrier, orderDate; status; totalPrice
Output parameters: None
Pseudocode:

```
INSERT INTO order_table VALUS
(orderId; storeId; userId; creditCard; carrier,
 orderDate; status; totalPrice);
```

Remarks: None

Index: In-5
Name: insertOrderStatus
Purpose: To add an order_status entry to database
Visibility: Internal; invisible
Input parameters: orderId; updateDate; deliverDate; status
Output parameters: None
Pseudocode:

```
INSERT INTO order_status_table VALUS
(orderId, updateDate, deliverDate, status);
```

Remarks: None

Index: In-6
Name: insertLineItems
Purpose: To add an order_status entry to database
Visibility: Internal; invisible
Input parameters: list of item(orderId, itemId, qty, uniPrice)
Output parameters: None
Pseudocode:

```
if array.haveNext() {
    add an item entry to database {
        INSERT INTO line_item_table VALUS
        (orderId, itemId, qty, uniPrice);
    }
}
```

Remarks: None

Index: In-7
Name: deleteOrder
Purpose: To remove an order entry from database
Visibility: Internal; invisible
Input parameters: orderId
Output parameters: None
Pseudocode:

```
DELETE FROM order_table WHERE
        orderid = orderId;
```

Remarks: None

Index: In-8
Name: deleteOrderStatus
Purpose: To remove an order entry from database
Visibility: Internal; invisible
Input parameters: orderId
Output parameters: None
Pseudocode: DELETE FROM order_table WHERE
orderid = orderId;
Remarks: None

Index: In-9
Name: deleteLineItems
Purpose: To remove all lineItems for an order from database
Visibility: Internal; invisible
Input parameters: orderId
Output parameters: None
Pseudocode: DELETE FROM line_item_table WHERE
orderid = orderId;
Remarks: None

Offer Object:

Index: Obj-3
Object Name: Offer
Inheritance: None
Imported Objects: None
Purpose: To create an offer.
Remarks: None
Properties: Entity bean
Methods:

Method to set the attributes:
Index: Con-1
Name: OfferEJB
Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection
offerId = 0; offerDate = Calendar.getInstance();
Remarks: None

Methods of the Object:
Index: Op-1

Name: createOffer
Purpose: To create an offer
Visibility: External; visible
Input parameters: sellerId; buyerId; itemId; offeringPrice
Output parameters: No
Pseudocode:

```
if (isValidData(sellerId, buyerId, itemId,
                offeringPrice)){
        offerId = getUniqueOfferId();
        offerDate = Calendar.getInstance();
        status = "inactive";
        INSERT INTO offer_table VALUES
        (offerId, sellerId, buyerId,  itemId, offerDate,
        offeringPrice, status);
        return true;
}
else
        return false;
```

Remarks: An active offering price will be used in order processing

Index: Op-2
Name: DeleteOffer
Purpose: To delete an offer instance
Visibility: External; visible
Input parameters: offerId
Output parameters: Confirm information (successful or fail)
Pseudocode:

```
if offerExists(offerId) {
        DELETE FROM offer_table WHERE
        offerid = offerId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-3
Name: getOfferDetails
Purpose: To retrieve an offer details from database
Visibility: External; visible
Input parameters: offerId
Output parameters: list of (sellerId,buyerId,itemId,offeringPrice)
Pseudocode:

```
if offerExists(orderId) {
        list = SELECT (sellerId,buyerId,
                itemId,offeringPrice)
                FROM offer_table
                WHERE  offerid = offerId;
        return list;
```

```
}
else
        return null;
```

Remarks: None

Index: Op-4
Name: changeOfferingPrice
Purpose: To update offering price
Visibility: External; visible
Input parameters: offerId; price
Output parameters: None
Pseudocode:

```
if offerExists(offerId) {
        UPDATE  offer_table
        SET price = price
        WHERE offerid = offerId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-5
Name: changeOfferStatus
Purpose: To update offer status
Visibility: External; visible
Input parameters: offerId; newStatus
Output parameters: None
Pseudocode:

```
if offerExists(orderId) {
        UPDATE  offer_table
        SET status = newStatus
        WHERE offerid = offerId;
        return true;
}
else
        return false;
```

Remarks: None

Methods used internally by the object

Index: In-1
Name: offerExists
Purpose: To check if the offer exists
Visibility: Internal; invisible
Input parameters: offerId
Output parameters: result (true or false)

Pseudocode: string = SELECT *offerid* FROM offer_table
WHERE *offerrid* = offerId;
if (string = null)
return false;
else
return true;

Remarks: None

Index: In-2
Name: getUniqueOfferId
Purpose: To get a unique offerId
Visibility: Internal; invisible
Input parameters: None
Output parameters: offerId
Pseudocode:

```
offerId = nextSeqNum(offer_table)
return offerId;
```

Remarks: None

Index: In-3
Name: isValidData
Purpose: To check if the data is valid
Visibility: Internal; invisible
Input parameters: data
Output parameters: result (true or false)
Pseudocode:

```
if any input data is invalid
        return false;
else
        return true;
```

Remarks: None

Store Object:

Index: Obj-4
Object Name: Store
Inheritance: None
Imported Objects: None
Purpose: To create a store and provide store information
Remarks: None
Properties: Entity bean
Methods:

Method to set the attributes:
Index: Con-1
Name: StoreEJB

Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection
Remarks: None

Methods of the Object:
Index: Op-1
Name: createStore
Purpose: To create an store and add store information to the system
Visibility: External; visible
Input parameters: storeId; storeName; managerName; address; PhoneNo; itemCode; email; policy
Output parameters: No
Pseudocode:

```
if (isValidData(storeId, storeName, managerName,
        address, phoneNo, itemCode, email, policy)){
        add an entry to database {
        INSERT INTO store_table VALUES
        (storeId, storeName, managerName, Address,
        PhoneNo, itemCode, email, policy);
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-2
Name: DeleteStore
Purpose: To delete an store instance
Visibility: External; visible
Input parameters: storeId
Output parameters: Confirm information (successful or fail)
Pseudocode:

```
if storeExists(storeId) {
        DELETE FROM store_table WHERE
        storeid = storeId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-3
Name: getStoreDetails
Purpose: To retrieve an store details from database
Visibility: External; visible

Input parameters: storeId
Output parameters: list of (storeId, storeName, managerName, address, phoneNo, itemCode, email, policy)
Pseudocode:

```
if storeExists(orderId) {
        list = SELECT (storeid, storename, managername,
                address, phoneno, itemcode, email, policy)
                WHERE storeid = storeId;
        return list;
}
else
        return null;
```

Remarks: None

Index: Op-4
Name: changeStorePolicy
Purpose: To update a store policy
Visibility: External; visible
Input parameters: storeId; policy
Output parameters: None
Pseudocode:

```
if storeExists(storeId) {
        UPDATE  store_table
        SET policy = policy
        WHERE storeid = storeId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-5
Name: changeStoreAdress
Purpose: To update a store's address
Visibility: External; visible
Input parameters: storeId; newAddress
Output parameters: None
Pseudocode:

```
if storeExists(storeId) {
        UPDATE  store_table
        SET address = newAddress
        WHERE storeid = storeId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-6

Name: changeStoreEmail
Purpose: To update a store's email
Visibility: External; visible
Input parameters: storeId; newEmail
Output parameters: None
Pseudocode:

```
if storeExists(storeId) {
        UPDATE  store_table
        SET email = newEmail
        WHERE storeid = storeId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-7
Name: getItemCode
Purpose: To retrieve a store's item code
Visibility: External; visible
Input parameters: storeId
Output parameters: itemCode
Pseudocode:

```
if storeExists(storeId) {
        itemCode = SELECT itemcode
                   FROM  store_table
                   WHERE storeid = storeId;
        return itemCode;
}
else
        return null;
```

Remarks: None

Methods used internally by the object

Index: In-1
Name: storeExists
Purpose: To check if the store exists
Visibility: Internal; invisible
Input parameters: storeId
Output parameters: result (true or false)
Pseudocode:

```
string = SELECT storeid FROM store_table
        WHERE storeid  =  storeId;
if (string = null)
        return false;
```

else
    return true;

Remarks: None

Index: In-2
Name: isValidData
Purpose: To check if the data is valid
Visibility: Internal; invisible
Input parameters: data
Output parameters: result (true or false)
Pseudocode:

```
if any input data is invalid
        return false;
else
        return true;
```

Remarks: None

Item Object:

Index: Obj-5
Object Name: Item
Inheritance: None
Imported Objects: None
Purpose: To create an item
Remarks: None
Properties: Entity bean
Methods:

Method to set the attributes:
Index: Con-1
Name: ItemEJB
Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection
Remarks: None

Methods of the Object:
Index: Op-1
Name: createItem
Purpose: To add an item to the system
Visibility: External; visible
Input parameters: itemId; productId; listPrice; unitCost; storeId; status; attr

Output parameters: No

Pseudocode:

```
if (productExists(productId) &
        isValidData(itemId, productId, listPrice,
                unitCost, storeId, status, attr)) {
        INSERT INTO item_table VALUES
                (itemId, productId, listPrice,
                unitCost, storeId, status, attr);
        return true;
}
else {
        print("itemId already existed!");
        return false;
}
```

Remarks: None

Index: Op-2
Name: deleteItem
Purpose: To delete an item instance
Visibility: External; visible
Input parameters: itemId
Output parameters: Confirm information (successful or fail)
Pseudocode:

```
if itemExists(itemId) {
        DELETE FROM item_table WHERE
        itemid = itemId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-3
Name: getItemDetails
Purpose: To retrieve an item details from database
Visibility: External; visible
Input parameters: itemId
Output parameters: list of (itemId, productId, listPrice,
unitCost, storeId, status, attr);

Pseudocode:

```
if itemExists(itemId) {
        list = SELECT (itemid, productid, listprice,
                unitcost, storeid, status, attr)
                FROM item_table
                WHERE iteid = itemId;
        return list;
}
else
```

```
        return null;
```

Remarks: None

Index: Op-4
Name: getPruductId
Purpose: To retrieve an item's productId
Visibility: External; visible
Input parameters: itemId
Output parameters: PruductId
Pseudocode:

```
if itemExists(itemId) {
        productId = SELECT productid
                    FROM  item_table
                    WHERE itemid = itemId;
        return productId;
}
else
        return null;
```

Remarks: None

Index: Op-5
Name: getItemPrice
Purpose: To retrieve an item's list price
Visibility: External; visible
Input parameters: itemId
Output parameters: listPrice
Pseudocode:

```
if itemExists(itemId) {
        listPrice = SELECT listprice
                    FROM  item_table
                    WHERE itemid = itemId;
        return listPrice;
}
else
        return null;
```

Remarks: None

Index: Op-6
Name: getItemCost
Purpose: To retrieve an item's unit cost
Visibility: External; visible
Input parameters: itemId
Output parameters: unitCost
Pseudocode:

```
if itemExists(itemId) {
        unitCost = SELECT unitcost
                   FROM  item_table
                   WHERE itemid = itemId;
        return unitCost;
```

```
}
else
        return null;
```

Remarks: None

Index: Op-7
Name: getItemStatus
Purpose: To retrieve an item's status
Visibility: External; visible
Input parameters: itemId
Output parameters: status
Pseudocode:

```
if itemExists(itemId) {
        status = SELECT status
                FROM  item_table
                WHERE itemid = itemId;
        return unitCost;
}
else
        return null;
```

Remarks: None

Index: Op-8
Name: changeItemStatus
Purpose: To update an item's status
Visibility: External; visible
Input parameters: itemId; newStatus
Output parameters: None
Pseudocode:

```
if itemExists(itemId) {
        UPDATE  item_table SET
        status = newStatus WHERE
        itemid = itemId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-9
Name: changeItemPrice
Purpose: To update an item's price
Visibility: External; visible
Input parameters: itemId; newPrice
Output parameters: None
Pseudocode:

```
if itemExists(itemId) {
        UPDATE  item_table
        SET listprice = newPrice
```

```
        WHERE itemid = itemId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-10
Name: changeItemCost
Purpose: To update an item's status
Visibility: External; visible
Input parameters: itemId; newCost
Output parameters: None
Pseudocode:

```
if itemExists(itemId) {
        UPDATE  item_table
        SET unitcost = newCost
        WHERE itemid = itemId;
        return true;
}
else
        return false;
```

Remarks: None

Methods used internally by the object

Index: In-1
Name: itemExists
Purpose: To check if the item exists
Visibility: Internal; invisible
Input parameters: itemId
Output parameters: result (true or false)
Pseudocode:

```
string = SELECT itemid FROM item_table
        WHERE itemid  =  itemId;
if (string = null)
        return false;
else
        return true;
```

Remarks: None

Index: In-2
Name: productExists
Purpose: To check if the productId exists
Visibility: Internal; invisible
Input parameters: productId
Output parameters: result (true or false)

Pseudocode:

```
string = SELECT productid FROM product_table
          WHERE productid = productId;
if (string = null)
        return false;
else
        return true;
```

Remarks: None

Index: In-3
Name: isValidData
Purpose: To check if the data is valid
Visibility: Internal; invisible
Input parameters: data
Output parameters: result (true or false)
Pseudocode:

```
if any input data is invalid
        return false;
else
        return true;
```

Remarks: None

Product Object:

Index: Obj-6
Object Name: Product
Inheritance: None
Imported Objects: None
Purpose: To create a product instance
Remarks: None
Properties: Entity bean
Methods:

Method to set the attributes:
Index: Con-1
Name: ProductEJB
Purpose: To initialize the attributes of the object
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection
Remarks: None

Methods of the Object:
Index: Op-1
Name: createCategory

Purpose: To add an category instance to the system
Visibility: External; visible
Input parameters: categoryId; name; desc
Output parameters: No
Pseudocode:

```
if (isValidData(categoryId, name, desc)) {
        INSERT INTO category_table VALUES
                (categoryId, name, desc);
        return true;
}
else {
        return false;
}
```

Remarks: None

Index: Op-1
Name: createProduct
Purpose: To add an product instance to the system
Visibility: External; visible
Input parameters: productId; supplierId; catagoryId; name; desc storeId; status; attr
Output parameters: No
Pseudocode:

```
if (categoryExists(categoryId) &
        isValidData(productId, supplierId,
                categoryId, name, desc)) {
        INSERT INTO item_table
        VALUES (productId, supplierId, catagoryId,
                        name, desc);
        return true;
}
else {
        return false;
}
```

Remarks: None

Index: Op-3
Name: deleteCategory
Purpose: To delete an gategory instance
Visibility: External; visible
Input parameters: categoryId
Output parameters: Confirm information (successful or fail)
Pseudocode:

```
if categoryExists(itemId) {
        DELETE FROM category_table
        WHERE catid = categoryId;
        return true;
}
else
```

	return false;
Remarks:	None

Index:	Op-4
Name:	deleteProduct
Purpose:	To delete an product instance
Visibility:	External; visible
Input parameters:	productId
Output parameters:	Confirm information (successful or fail)
Pseudocode:	

```
if productExists(productId) {
        DELETE FROM product_table
        WHERE productid = productId;
        return true;
}
else
        return false;
```

Remarks:	None

Index:	Op-5
Name:	getCategoryDetails
Purpose:	To retrieve an category details from database
Visibility:	External; visible
Input parameters:	catId
Output parameters:	list of (catId, name, desc);
Pseudocode:	

```
if categoryExists(catId) {
        list =    SELECT (cateId, name, desc)
                  WHERE catid = catId;
        return list;
}
else
        return null;
```

Remarks:	None

Index:	Op-6
Name:	getProductDetails
Purpose:	To retrieve an product details from database
Visibility:	External; visible
Input parameters:	productId
Output parameters:	list of (catId, supplier, category, name, desc);
Pseudocode:	

```
if productExists(productId) {
        list = SELECT (productid, supplier, catgory,
                                    name, desc)
                FROM product_table
                WHERE productid = productId;
        return list;
}
```

```
else
        return null;
```

Remarks: None

Index: Op-7
Name: getSupplier
Purpose: To retrieve a product's supplier
Visibility: External; visible
Input parameters: productId
Output parameters: supplier
Pseudocode:

```
if productExists(productId) {
        supplier = SELECT supplier
                FROM  product_table
                WHERE productid = productId;
        return supplier;
}
else
        return null;
```

Remarks: None

Index: Op-8
Name: getCategory
Purpose: To retrieve a product's category
Visibility: External; visible
Input parameters: productId
Output parameters: category
Pseudocode:

```
if productExists(productId) {
        category = SELECT category
                FROM  product_table
                WHERE productid = productId;
        return category;
}
else
        return null;
```

Remarks: None

Index: Op-9
Name: changeSupplier
Purpose: To update a product's supplier
Visibility: External; visible
Input parameters: productId; newSupplier
Output parameters: None
Pseudocode:

```
if productExists(productId) {
        UPDATE  product_table SET
        supplier = newSupplier
        WHERE productid = productId;
```

```
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-10
Name: changeCategory
Purpose: To update a category's name and description
Visibility: External; visible
Input parameters: catId; newName; newDesc
Output parameters: None
Pseudocode:

```
if categoryExists(catId) {
        UPDATE category_table SET
        name = newName, desc = newDesc
        WHERE itemid = itemId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-11
Name: changeProduct
Purpose: To update a product's name or description
Visibility: External; visible
Input parameters: productId; newName; newdesc
Output parameters: None
Pseudocode:

```
if productExists(productId) {
        UPDATE product_table SET
        name = newName, desc = new Desc
        WHERE productid = productId;
        return true;
}
else
        return false;
```

Remarks: None

Methods used internally by the object

Index: In-1
Name: categoryExists
Purpose: To check if the item exists
Visibility: Internal; invisible
Input parameters: catId

Output parameters:	result (true or false)

Pseudocode:	string = SELECT cat*id* FROM category_table
	WHERE *catid* = catId;
	if (string = null)
	return false;
	else
	return true;

Remarks:	None

Index:	In-2
Name:	productExists
Purpose:	To check if the productId exists
Visibility:	Internal; invisible
Input parameters:	productId
Output parameters:	result (true or false)

Pseudocode:	string = SELECT *productid* FROM product_table
	WHERE *productid* = productId;
	if (string = null)
	return false;
	else
	return true;

Remarks:	None

Index:	In-3
Name:	isValidData
Purpose:	To check if the data is valid
Visibility:	Internal; invisible
Input parameters:	data
Output parameters:	result (true or false)
Pseudocode:	if any input data is invalid return false; else return true;
Remarks:	None

Inventory Object:

Index:	Obj-7
Object Name:	Inventory
Inheritance:	None
Imported Objects:	None
Purpose:	To create an item inventory
Remarks:	None
Properties:	Entity bean

Methods:

Method to set the attributes:

Index:	Con-1
Name:	InventoryEJB
Purpose:	To initialize the attributes of the object
Visibility:	External; visible
Input parameters:	None
Output parameters:	None
Pseudocode:	find datasource connection
Remarks:	None

Methods of the Object:

Index: Op-1

Name: createInventory

Purpose: To add an item inventory to the system

Visibility: External; visible

Input parameters: itemId; qty

Output parameters: None

Pseudocode:

```
if itemExits(itemId) {
        INSERT INTO inventory_table VALUES
        (itemId, qty);
        return true;
}
else {
        print("no such item exists, create the item first");
        return false;
}
```

Remarks: None

Index: Op-2

Name: deleteInventory

Purpose: To delete an item inventory instance

Visibility: External; visible

Input parameters: itemId

Output parameters: Confirm information (successful or fail)

Pseudocode:

```
if itemExists(itemId) {
        DELETE FROM inventory_table
        WHERE itemid = itemId;
        return true;
}
else
        return false;
```

Remarks: None

Index: Op-3

Name: getQuantity
Purpose: To retrieve an item quantity from inventory
Visibility: External; visible
Input parameters: itemId
Output parameters: qty
Pseudocode:

```
if itemIdExists(itemId) {
        qty =   SELECT quqntity
                FROM inventory_table
                WHERE itemid = itemId;
        Return qty;
}
else
        return -1;
```

Remarks: None

Index: Op-4
Name: reduceQuantity
Purpose: To reduce an item inventory
Visibility: External; visible
Input parameters: itemId; num
Output parameters: qty
Pseudocode:

```
if itemIdExists(itemId) {
        oldQty = getQty(itemId);
        qty = oldQty – num;
        if (qty > 0) {
                createInventory(itemId, qty);
                return qty;
        }
}
else
        return -1;
```

Remarks: None

Index: Op-5
Name: addQuantity
Purpose: To increase an item inventory
Visibility: External; visible
Input parameters: itemId; num
Output parameters: qty
Pseudocode:

```
if itemIdExists(itemId) {
        oldQty = getQty(itemId);
        qty = oldQty + num;
        createInventory(itemId, qty);
        return qty;
}
else
```

```
        return -1;
```

Remarks: None

Methods used internally by the object

Index: In-1
Name: itemExists
Purpose: To check if the item exists
Visibility: Internal; invisible
Input parameters: itemId
Output parameters: result (true or false)
Pseudocode:

```
string = SELECT itemid
         FROM inventory_table
         WHERE itemid = itemId;
if (string = null)
        return false;
else
        return true;
```

Remarks: None

CatalogDao Object:

Index: Obj-8
Object Name: CatalogDao
Inheritance: None
Imported Objects: None
Purpose: To create an object for other object to retrieve category, product and item information from database by search keys
Remarks: The object do not modify data
Properties: Java class that encapsulates SQL calls
Methods:

Method to set the attributes:
Index: Con-1
Name: CatalogDao
Purpose: To initialize the attributes of the class
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection

Methods of the Object:

Index: Op-1
Name: getCategory
Purpose: To retrieve a category details
Visibility: External; visible
Input parameters: catId
Output parameters: list of (catId, name, desc)
Pseudocode:

```
 list = SELECT (catId, name, desc)
          FROM category_table
          WHERE catid = catId;
return list;
```

Remarks: None

Index: Op-2
Name: getProduct
Purpose: To retrieve a product details from database
Visibility: External; visible
Input parameters: productId
Output parameters: list of (catId, supplier, category, name, desc);
Pseudocode:

```
 list = SELECT (productid, supplier, catgory,
               name, desc)
          FROM product_table
          WHERE productid = productId;
return list;
```

Remarks: None

Index: Op-3
Name: getItem
Purpose: To retrieve an item details from database
Visibility: External; visible
Input parameters: itemId
Output parameters: list of (itemId, productId, listPrice, unitCost, storeId, status, attr);
Pseudocode:

```
 list =  SELECT (itemid, productid, listprice,
                  unitcost, storeid, status, attr)
          FROM item_table
          WHERE iteid = itemId;
return list;
```

Remarks: None

Index: Op-4
Name: getCategories
Purpose: To retrieve all category
Visibility: External; visible
Input parameters: None
Output parameters: list of catIds

Pseudocode: list = SELECT *catid* FROM category_table;
return list;

Remarks: None

Index: Op-5
Name: getProducts
Purpose: To retrieve the products of a category from database
Visibility: External; visible
Input parameters: catId
Output parameters: list of supplierIds
Pseudocode:

```
list =  SELECT productid
        FROM product_table
        WHERE catid = catId;
return list;
```

Remarks: None

Index: Op-6
Name: getItems
Purpose: To retrieve an item details from database
Visibility: External; visible
Input parameters: productId
Output parameters: list of itemIds
Pseudocode:

```
list =  SELECT itemid
        FROM item_table
        WHERE iteid = itemId;
return list;
```

Remarks: None

Index: Op-7
Name: searchProductsByName
Purpose: To retrieve products by product name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of (itemId, listPrice, store, productId, productName, Supplier, productDesc)
Pseudocode:

```
list =  SELECT (itemid, listprice, store, a.productid,
                name, supplier, desc)
        FROM item_table a, product_table b
        WHERE a.productId = b.productId and
              b.productid in (SELECT product from
                                     product_table
                     WHERE lower(name) like keyWord);
return list;
```

Remarks: None

Index: Op-8
Name: searchProductsByCatName

Purpose: To retrieve products by category name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of (itemId, listPrice, store, productId, productName, Supplier, productDesc)
Pseudocode:

```
list =  SELECT (itemid, listprice, store, a.productid,
                        name,supplier, desc)
        FROM item_table a, product_table b
        WHERE a.productId = b.productId and
              b.productid in (SELECT product from
                     product_table c, category_table d
              WHERE c.category = d.catid and
                     lower(d.name) like keyWord);
return list;
```

Remarks: None

Index: Op-9
Name: searchProductsBySupplier
Purpose: To retrieve products by supplier name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of (itemId, listPrice, store, productId, productName, Supplier, productDesc)
Pseudocode:

```
list =  SELECT (itemid, listprice, store, a.productid,
                        name, supplier, desc)
        FROM item_table a, product_table b
      WHERE a.productId = b.productId and b.productid
           in (SELECT productid from product_table c,
                        supplier_table d
              WHERE c.productid = d.productid  and
                      lower(d.name) like keyWord);
return list;
```

Remarks: None

Index: Op-10
Name: searchProductsByStore
Purpose: To retrieve products by store name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of (itemId, listPrice, store, productId, productName, Supplier, productDesc)
Pseudocode:

```
list = SELECT (itemid, listprice, store, a.productid, name,
         supplier, desc) FROM item_table a, product_table b
       WHERE a.productId = b.productId and a.itemid
         in (SELECT itemid from item_table c, store_table d
           WHERE c.itemid = d.itemid  and
                   lower(d.name) like keyWord);
```

return list;

Remarks: None

UserDao Object:

Index: Obj-9
Object Name: UserDao
Inheritance: None
Imported Objects: None
Purpose: To create an object for other object to retrieve users' (consumers, suppliers, and store owners) information from database
Remarks: The object do not modify data
Properties: Java class that encapsulates SQL calls
Methods:

Method to set the attributes:
Index: Con-1
Name: UserDao
Purpose: To initialize the attributes of the class
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode: find datasource connection

Methods of the Object:
Index: Op-1
Name: getCosumer
Purpose: To retrieve a consumer's details
Visibility: External; visible
Input parameters: userId
Output parameters: list of (userId, name, address, phone, email, level, languagePref, favCategory)
Pseudocode:

list = SELECT (*userid, name, address, phone, email, level, languagepref, favcategory*)
FROM consumer_table
WHERE *userid* = userId;
return list;

Remarks: None

Index: Op-2
Name: getSupplier
Purpose: To retrieve a supplier details from database
Visibility: External; visible

Input parameters: supplierId
Output parameters: list of (supplierId, name, address, phone, email,policy)
Pseudocode:

list = SELECT (*supplierId, name, address, phone, email,languagepref, policy*)
FROM supplier_table
WHERE *supplierid* = supplierId;
return list;

Remarks: None

Index: Op-3
Name: getOwner
Purpose: To retrieve a store owner's details from database
Visibility: External; visible
Input parameters: userId
Output parameters: list of (userId, name, address, phone, email,policy)
Pseudocode:

list = SELECT (*userid, name, storeid,shares, address, phone, email, brieffacts, languagepref*)
FROM owner_table
WHERE *userid* = userId;
return list;

Remarks: None

Index: Op-4
Name: getStore
Purpose: To retrieve a store's details from database
Visibility: External; visible
Input parameters: storeId
Output parameters: list of (storeId, storeName, managerName, address, phone, email, policy, itemCode)
Pseudocode:

list = SELECT (*storeid*, *storename*, *managername*, *ddress*, *phone*, *email*, *policy, itemcode*)
FROM store_table
WHERE *storeid* = storeId;
return list;

Remarks: None

Index: Op-5
Name: getCosumers
Purpose: To retrieve the consumers at same favorite category
Visibility: External; visible
Input parameters: catId
Output parameters: list of userIds
Pseudocode:

list = SELECT *userid*
FROM consumer_table
WHERE *favcategory* = catId;
return list;

Remarks: None

Index: Op-6
Name: getOwners
Purpose: To retrieve a store owners from database
Visibility: External; visible
Input parameters: storeId
Output parameters: list of ownerIds
Pseudocode:

```
list =  SELECT userid
        FROM owner_table
        WHERE storeid = storeId;
return list;
```

Remarks: None

Index: Op-7
Name: searchConsumersByName
Purpose: To retrieve consumer details by name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of (userId, name, email, favCategory)
Pseudocode:

```
list =  ELECT (userid, name, email, favcat)
        FROM consumer_table
        WHERE lower(name) like keyWord;
return list;
```

Remarks: None

Index: Op-8
Name: searchSupplierByName
Purpose: To retrieve supplier details by name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of supplierIds
Pseudocode:

```
list =  SELECT supplierid
        FROM supplier_table
        WHERE lower(name) like keyWord);
```

Remarks: None

Index: Op-9
Name: searchSupplierByProduct
Purpose: To retrieve supplier by product
Visibility: External; visible
Input parameters: productId
Output parameters: list supplierIds
Pseudocode:

```
list =  SELECT supplierid
        FROM supplier_table a, product_table b
        WHERE a.supplierid = b.supplier and
              b.productid = productId;
return list;
```

Remarks: None

Index: Op-10
Name: searchStoreByItem
Purpose: To retrieve store by item
Visibility: External; visible
Input parameters: itemId
Output parameters: list of storeIds
Pseudocode:

```
list =  SELECT storeid
        FROM store_table a, item_table b
        WHERE a.storeid = b.store and
              b .itemid = itemId;
return list;
```

Remarks: None

Index: Op-11
Name: searchOwnersByName
Purpose: To retrieve consumer details by name
Visibility: External; visible
Input parameters: keyWords
Output parameters: list of (userId, name, email, favCategory)
Pseudocode:

```
list =  SELECT (userid, name, email, store)
        FROM owner_table a, store_table b
        WHERE lower(a.name) like keyWord;
return list;
```

Remarks: None

Consumer Object:

Index: Obj-10
Object Name: Consumer
Inheritance: None
Imported Objects: Account; Order;
Purpose: To create an object for consumer
Remarks: Holds main operations for consumer
Properties: Stateless session bean
Methods:

Methods of the Object:

Methods from Account:

registration
login

logout
getAccountDetails
changeAccount

Methods from Order:

createOrder
deleteOrder
changeOrder
changeLineItes
getOrderDetails
getLineItems
find userOrders

Index: Op-1
Name: getReferences
Purpose: To get account and order references
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode:

```
if (accountReference = null) {
    obRef = lookup the reference of account;
    accountHomeRef = portableRemoteObject.
                        narrow(obRef, AccountHome.class);
}
if (orderReference = null) {
    obRef = lookup the reference of order;
    orderHomeRef = portableRemoteObject.
                        narrow(obRef, OrderHome.class);
}
```

Remarks: None

Index: Op-2
Name: addConsumer
Purpose: To add a consumer's information to the system
Visibility: External; visible
Input parameters: name; address; phone; email; level; languagePref; favCategory
Output parameters: None
Pseudocode:

```
userId = context.getCallerPrincipal().getName();
INSERT INTO consumer_table
VALUES (userId, name, address, phoneNo, email, level,
            langusgePref, favCategory)
```

Remarks: None

Index: Op-3
Name: getCosumerdetails
Purpose: To retrieve a consumer's details

Visibility: External; visible
Input parameters: None
Output parameters: list of (userId, name, address, phone, email, level, languagePref, favCategory)
Pseudocode:

list = SELECT (*userid, name, address, phone, email, level, languagepref, favcategory*)
FROM consumer_table WHERE
userid = context.getCallerPrincipal().getName();
return list;

Remarks: None

Index: Op-4
Name: changeConsumer
Purpose: To change a consumer's information
Visibility: External; visible
Input parameters: newConsumer(name, address, phone, email, level, languagePref, favCategory)
Output parameters: None
Pseudocode:

UPDATE consumer_table
SET *name* = name, *address* = address, *phone* = phoneNo,
email = email, *level* = level, *favcat* = favCategory,
languagepref = languagePref
WHERE *userid* = context.getCallerPrincipal().getName();

Remarks: None

Owner Object:

Index: Obj-11
Object Name: Owner
Inheritance: None
Imported Objects: Account; Order; Store; Inventory
Purpose: To create an object for owner
Remarks: Holds main operations for owner
Properties: Stateless session bean
Methods:

Methods of the Object:

Methods from Account:

registration
login
logout
getAccountDetails

changeAccount

Methods from Order:

createOrder
deleteOrder
changeOrderStatus
getOrderDetails
getLineItems
findStoreOrders

Methods from Store:

getStoreDetails
changePolicy
changeAddress
changeEmail

Methods from Inventory: addInventory
reduceInventory

Methods from Item:

getItemDetails
getPruductId
changeItemCost
changeItemPrice

Index: Op-1
Name: getReferences
Purpose: To get account and order references
Visibility: External; visible
Input parameters: None
Output parameters: None
Pseudocode:

```
if (accountReference = null) {
    obRef = lookup the reference of account;
    accountHomeRef = portableRemoteObject.
                    narrow(obRef, AccountHome.class);
}
if (orderReference = null) {
    obRef = lookup the reference of order;
    orderHomeRef = portableRemoteObject.
                    narrow(obRef, OrderHome.class);
}
if (storeReference = null) {
    obRef = lookup the reference of store;
    storeHomeRef = portableRemoteObject.
                    narrow(obRef, StoreHome.class);
}
if (inventoryReference = null) {
```

```
    obRef = lookup the reference of inventory;
    inventoryHomeRef = portableRemoteObject.
                narrow(obRef, InventoryHome.class);
}
if (itemReference = null) {
    obRef = lookup the reference of item;
    itemHomeRef = portableRemoteObject.
                narrow(obRef, ItemHome.class);
}
```

Remarks: None
Index: Op-2
Name: addOwner
Purpose: To add an owner's information to the system
Visibility: External; visible
Input parameters: name; store; shares; address; phoneNo; email; briefFact; languagePref
Output parameters: None
Pseudocode:

```
userId = context.getCallerPrincipal().getName();
INSERT INTO consumer_table
VALUES (userId, name, store, shares, address, phoneNo,
        email, languagePref)
```

Remarks: None

Index: Op-3
Name: getOwnerdetails
Purpose: To retrieve a consumer's details
Visibility: External; visible
Input parameters: None
Output parameters: list of (userId, name, store, shares, address, phoneNo, email, briefFacts, languagePref)
Pseudocode:

```
list = SELECT (userid, name, store, shares, address,
               phone, email, level, languagepref, favcat)
       FROM owner_table WHERE
       userid = context.getCallerPrincipal().getName();
return list;
```

Remarks: None

Index: Op-4
Name: changeOwner
Purpose: To change an owner's information
Visibility: External; visible
Input parameters: newOwner(name, store, shares, address, phoneNo, email, briefFact, languagePref)
Output parameters: None
Pseudocode:

```
UPDATE consumer_table
```

```
SET     name = name, store = store, shares = shares,
        address = address, phone = phoneNo,
        email = email, languagepref = languagePref,
WHERE userid = context.getCallerPrincipal().getName();
```

Remarks: None

Index: Op-5
Name: getItems
Purpose: To retrieve items of a store
Visibility: External; visible
Input parameters: None
Output parameters: list of itemIds
Pseudocode:

```
userid = context.getCallerPrincipal().getName();
storeId = SELECT storeid
            FROM owner_table
            WHERE userid = userid;
list =  SELECT (itemid, productid, listprice,
                unitcost, storeid, status, attr)
        FROM item_table
        WHERE storeid = storeId;
return list;
```

Remarks: None

ShoppingCart Object:

Index: Obj-12
Object Name: ShoppingCart
Inheritance: None
Imported Objects: None
Purpose: To browse through the catalog or search for products and to modify and hold an order instance temporally
Remarks: None
Properties: Stateless session bean
Methods:

Method to set the attributes:
Index: Con-1
Name: ShoppingCartEJB
Purpose: To initialize the attributes of the class
Visibility: External; visible
Input parameters: None
Output parameters: cart

Pseudocode: cart = new HashMap();

Methods of the Object:

Index: Op-1
Name: createShoppingCart
Purpose: To add items to a cart
Visibility: External; visible
Input parameters: cart1
Output parameters: HashMap cart
Pseudocode: create a cart (with or without items) for an order
cart = cart1.clone()
Remarks: None

Index: Op-2
Name: getDetails
Purpose: To retrieve all item details in a cart
Visibility: External; visible
Input parameters: cart
Output parameters: list of cartItems
Pseudocode:

```
if cart.hasNext() {
    itemId = cart.next();
    qtyNeeded = cart.get(itemId);
    item = catalogDao.getItem(itemId);
    productId = item.getProductId();
    product = catalogDao.getProduct(productId);
    productName = product.getName();
    cartItem = new CartItem(itemId, productId,
                    productName, item.getAttribute(),
                    qtyNeeded, item.getListCost());
   items.add(cartItem);
}
return items;
```

Remarks: None

Index: Op-3
Name: addItem
Purpose: To add an item to a cart
Visibility: External; visible
Input parameters: itemId; qty
Output parameters: None
Pseudocode: cart.put(itemId, qty);
Remarks: None

Index: Op-4
Name: deleteItem
Purpose: To delete an item from a cart

Visibility:	External; visible
Input parameters:	itemId
Output parameters:	None
Pseudocode:	cart.remove(itemId);
Remarks:	None

Index:	Op-5
Name:	updateItemQty
Purpose:	To update the quantity of an item in a cart
Visibility:	External; visible
Input parameters:	itemId; qty
Output parameters:	None
Pseudocode:	cart.remove(itemId); cart.put(itemId, qty);
Remarks:	None

Index:	Op-6
Name:	empty
Purpose:	To clear a cart
Visibility:	External; visible
Input parameters:	cart
Output parameters:	None
Pseudocode:	cart.clear();
Remarks:	None