An Object-Based Middleware Framework for E-Commerce Transactions

By

Hongjun Shen

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

An Object-Based Middleware Framework for E-Commerce Transactions

By

Hongjun Shen

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

0-494-08969-5

Your file Votre référence
ISBN:
Our file Notre reférence
ISBN:

An Object-Based Middleware Framework for E-Commerce Transactions

by

Hongjun Shen

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

of

Master of Science

Abstract

The goal of this research is to design an object-based middleware framework, which will facilitate the development of business-to-consumer (B2C) e-commerce applications. With the rapid development of the Internet and the World Wide Web, e-commerce applications have become prevalent. However, the development of e-commerce applications today is considered expensive and risky because of the lack of large-scale reusability and interoperability of e-commerce applications. Many standard middleware frameworks exist, such as COM+, EJBs, and CORBA. However, they are too generic to support services for B2C e-commerce. To address this problem, I propose an object-based middleware framework, which will facilitate the implementation of B2C e-commerce applications by providing some of the main services that typically have to be implemented from scratch, such as customer information management, payment management, and order processing. The proposed middleware framework involves the design of the common services and the implementation of a subset of the design to validate the design. Reuse of the proposed middleware framework can improve programmers' productivity, as well as enhance the quality, performance, reliability and interoperability of e-commerce applications. I expect that the proposed object-based middleware framework will provide a foundation for building reliable B2C e-commerce transaction systems.

Acknowledgements

First, I would like to thank my supervisor Dr. Vojislav Misic, who spent a lot of time editing my thesis and gave me a great number of valuable suggestions. I am also deeply grateful to my pre-supervisor, Dr. Sylvanus Ehikioya, for his guidance and encouragement during my research. Without his help and guidance, this work would have been impossible. I would also like to thank all the members of my thesis committee: Dr. Bob Travica, and Dr. Ellen Liu. Finally, I would also like to thank my family for their love, understanding, encouragement and support.

Table of Contents

List of Figures

Tables

Chapter 1

Introduction

E-commerce is a new way in which people conduct business using the Internet and the World Wide Web. Since the emergence of e-commerce in the last decade, online business transactions have experienced a spectacular growth and this tendency will continue for the next few years. E-commerce makes a wide range of products accessible to customers, and can be accessed from anywhere at anytime. People can easily surf the Internet shopping, banking, investing, and being entertained without leaving their comfortable homes or offices. Businesses leverage e-commerce to deliver more products and services to a global market without being restricted by geography, time, or cultural boundaries. Therefore, e-commerce has the potential of bringing many benefits to organizations and many comprehensive services to customers. E-commerce also provides organizations with more accurate, extensive information, a wider range of choices, and also enables fair competition. In this manner, e-commerce is gradually changing the way people conduct business, the relationship between businesses and customers, and people's life styles.

There are three common e-commerce types in practice today, namely business-to-consumer (B2C), business-to-business (B2B), and consumer-to-consumer (C2C). The B2C e-commerce focuses on selling products to individual consumers by a business. The B2B e-commerce focuses on selling products to other businesses. The C2C e-commerce provides a mechanism for individual consumers to sell to or buy from one another directly. The different e-commerce types have different business logics and different implementation requirements. In this research, I focused on B2C e-commerce type.

In B2C e-commerce, customers need to be able to view a list of all available products, including a description, image, and price for each product. Just as in the traditional store, customers will have the ability to add items to their shopping cart, view the contents of the shopping cart at any time, remove items from the shopping cart and purchase the items in the shopping cart. In order to ship purchased items to customers, customers will have an account containing their personal information such as name, address, credit card information, and purchasing history. For companies that provide e-commerce services to their customers, they need to keep an order list, which shows all the orders have been made through their e-commerce systems. They should also be able to track on the inventory to provide sufficient supplies of products.

The improvements in telecommunication network infrastructures have made all these operations possible. However, companies that wants to take the advantage of Internet to make more benefits face many challenges. These challenges include how to build e-commerce systems rapidly, which should save both money and time for the company, how to provide functionalities that satisfy both customers and the company's requirement, how to build e-commerce systems that are flexible enough to face future challenges, how to leverage the legacy systems, and how to manage complexity of e-commerce systems. To alleviate these problems, I proposed an object-based middleware framework in this thesis to facilitate the development of online store applications and to ease the burden of managing the complexity involved from software developers.

1.1 E-Commerce System Architecture

To deliver successful e-commerce transactions, an e-commerce system must be stable, running around the clock, year after year, and should be easily upgraded. To meet these expectations from both customers and e-commerce service providers, we require a reliable, scalable, and flexible e-commerce infrastructure, which includes servers, networks, operating systems, middleware, etc. Therefore, the n-tier client/server architecture is usually adopted as shown in Figure 1-1.

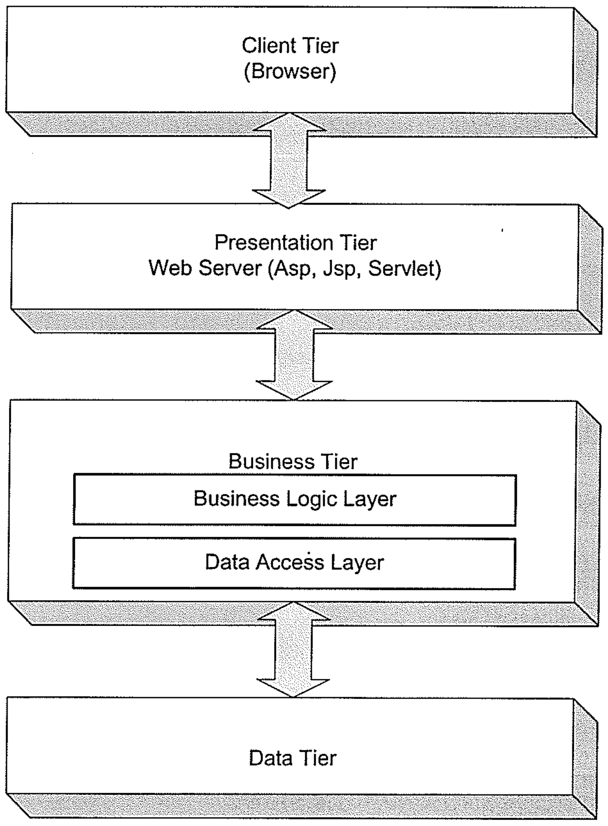


Figure 1-1 N-Tier Architecture

The *n*-tier architecture is a client/server architecture that breaks up applications into *n* logical, functional layers, where $n>2$. Typically, the *n*-tier architecture includes the client tier, the presentation tier, the business tier, and the database tier. The client tier is simply a web browser (e.g., Netscape or Internet Explorer). The presentation tier implements the user interface. For an e-commerce application, the presentation tier is not only including the web forms but also including all the classes which help to present data from business layer. The business tier manages the business logic and allows user to share and control the business logic. This tier could be separate into multiple layers. Typically, one layer is concentrate on describing business logic. The other layer is responsible for data access from the database tier. In the most complex case, business logic layer could be multiple layers. The database tier manages the backend database.

Presentation tier is where the web pages are implemented. Typically, JSP [26], ASP [27], or Servlets [26] technology is used to implement this tier. Business tier is to implement the complex business logic and to manipulate data coming from the database tier. COM+, CORBA, or EJBs technology is usually used to implement this tier.

An *n*-tier architecture has many advantages over the traditional 2-tier client/server architecture. First, through the clear separation of presentation tier and business tier, a system can easily scale up because the hardware and software for each tier can be increased independently. Second, developers can modify a specific tier rather than have to rewrite the entire application. Third, it facilitates the development of e-commerce application through reuse of pre-built business logic components if available. In this research, I focused on the business tier and provided a middleware framework, which facilitates the development of B2C e-commerce applications.

1.2 Motivation

To build an e-commerce system that can accommodate many customers concurrently and be always available is a great challenge to the developers. Since an e-commerce system is inherently large, complex and distributed [30], therefore, it must be carefully designed to meet current necessities and future updates. As a consequence, the development of e-commerce application today is considered expensive and risky.

The fast growth of e-commerce has motivated the search for faster and easier ways to build e-commerce applications. For example, many companies and organizations have proposed standard middleware framework, such as the Object Management Group's Common Object Request Broker Architecture (CORBA) [17], Microsoft's Common Object Model (COM+) [15], and Sun Microsystem's Enterprise JavaBeans (EJBs) [24]. These middleware frameworks provide extensive services that are necessary for building large scale, distributed applications and hide many of the common implementation details of these services from application developers. However, these services are too generic to easily support the building of e-commerce applications. Therefore, a middleware framework for the e-commerce domain, which has the great potential to increase system quality and decrease the development effort, is important.

Aleksy et al. [1] have proposed a CORBA-based middleware framework for e-auction applications providing services such as bidder, auctioneer, and auction manager, among others. Ripper et al. [21] presented a middleware framework for building agent-based e-commerce systems providing services like multiple communication protocols and multiple negotiation strategies. Both of these systems are specially designed for supporting the C2C business model. Bichler and Segev [3] described an object-based framework for supporting brokerage in the B2B business domain.

Online store application is common today such as Microsoft's Pet Shop application and Sun Microsystem's Pet Store application. Both of these applications used an n-tier architecture design and identified some of the basic functions that a pet store should support. However, the Microsoft's Pet Shop application is created to illustrate how to use the .NET framework to build multi-tier distribute applications and the Sun Microsystem's Pet Store is created to help developers and architects understand how to use J2EE technologies.

Therefore, a middleware framework for the B2C e-commerce domain is still unavailable. Because all the common services provided by e-commerce application such as catalog management, customer information management, order processing and so on have to be built from scratch. It is not only a waste of time and also error prone.

To address this problem, I identified all the common services that are necessary for building n-tier online store applications in this study. I encapsulated all these services in a set of components and provided services through the components' interface. These services works as middleware framework and can be reused to build similar online store applications.

Software reuse is an essential goal in software engineering because of the potential benefits it brings, such as increase in productivity, improvement of software product quality, and reduction of maintenance costs. Many reuse attempts have been made in practice with different successes, such as code reuse and design reuse. In general, the higher the level of abstraction, the more efficient reuse can be. In B2C e-commerce domain, an elaborately designed middleware framework, which will be able to capture the common functions of an e-commerce system, is beneficial. First, the framework design could be reused by multiple e-commerce applications and only small changes might be needed to accommodate special requirements of many diverse applications. Second, the implementation of this framework, presented by a set of components, could be reused directly as building blocks to implement e-commerce applications. Therefore, a component's reusability can be easily justified. However, designing for reuse is difficult, needs extensive analysis and careful design, so it usually takes more time at the design stage.

1.3 Contribution of the Thesis

In this research, I will design an object-based middleware framework for the B2C business model. In the proposed middleware framework, I will identify all the common services necessary for e-commerce transactions, encapsulate the common services into a set of components, and define these services using the components' interfaces. Therefore, the foci of this thesis are:

- To provide a reliable and correct design of all common services required by e-commerce transactions using the unified modeling language (UML) [4, 19] and pseudo code.
- To implement a subset of the design, such as customer, order, supplier, and payment, through which the correctness of the design can be validated.

The benefits of the proposed framework are twofold. First, the framework design, which is independent of any programming language, is reusable for a family of B2C applications. Second, the implementation code, which provides services through the interfaces of a set of components, can also be reused. In addition, I will improve the framework's reusability, maintainability, and performance by providing a proper size for each components and reducing communications between components. I expect that the proposed object-based middleware framework would greatly enhance the process of building reliable B2C e-commerce applications.

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 gives the background of e-commerce system architecture and object-based middleware framework, reviews some technologies used to build middlewares of e-commerce applications and some existing middleware frameworks in e-commerce domain. Chapter 3 presents the detailed design of my proposed object-based middleware framework for B2C e-commerce. In Chapter 4, I describe the implementation of the proposed middleware framework, gives an example application that derived from the proposed middleware framework, and depicts the example application's evaluation results. Chapter 5 makes some concluding comments and suggests directions for future research.

Chapter 2

Literature Review

Before formally presenting the design of the middleware framework for B2C e-commerce transaction, (in Chapter 3), an overview of framework, middleware, and object-based and component-based development are provided. Some related works and design tools that are relevant to the thesis are also described.

2.1 Background

2.1.1 Object-based and Component-based Development

The object concept was first emerged in the programming language *Simula* in 1960's. An object is an instantiation unit which encapsulates its state and behavior [13]. In object-oriented paradigm everything is treated as an object. Objects interact with each other through messages that contain information used in invoking operations on the appropriate objects. Systems implemented using object-oriented techniques are usually including features like inheritance, encapsulation and polymorphism [31].

In object-based paradigm, objects are complete packages. Everything that describes the implementation of the object is self-contained. Therefore, system implemented using object-based techniques should be easy to change and to extend. However, not all development environments (and their respective programming languages) support all of the object oriented features like inheritance, overloading, and overriding.

In component-based development, components are used as building blocks to implement a final solution. A component is an encapsulated unit of software with one or more interfaces that provide clients with access to its services [13]. Generally, a component is only available in binary form, (i.e., the component is precompiled). Therefore, the implementation details are completely hidden. The functionality provided by a component is available through its interface. In such a way, component-based approach is more likely to achieve reusability and ease the maintenance of a software product. Therefore, a component-based approach was used in this work.

Internally, a component may be implemented using an object-oriented paradigm, an object-based paradigm or even using traditional procedure paradigm. Object oriented approach has been successfully applied in building graphic user interface framework and distributed applications. However, object are usually not independent entities, therefore, changes to one object may require changes to several other objects and hamper system evolution. And since the object-based paradigm provides more reliability and reusability than the procedure-based paradigm [18], I used the object-based paradigm for designing and implementing each component.

2.1.2 Middleware

In the client/server architecture, the clients (which usually have graphical user interface) request and obtain services from the servers. Middleware is, then, a layer of software that enables and facilitates this client/server interaction. It consists of a set of services that allow multiple processes running on one or more machines to interact across a network and therefore make building distributed enterprise applications possible. Middleware can ease the design, programming and managing distributed applications by providing consistent and integrated distributed programming language environment and shield the heterogeneity and complexity of the underlying machine architecture, operating system and network technologies from developers.

Middleware includes but is not limited to database middleware such as ODBC, SQL, and Oracle Glue, Internet middleware such as HTTP and Secure Socket Layer (SSL), object middleware such as CORBA and COM+, and domain specific middleware.

Object middleware such as CORBA and COM+ provide fundamental services for building distributed enterprise applications but services that are required for a specific domain are not provided. Therefore domain specific middleware can extend the capabilities of object middleware and serve for a more specialized purpose. In this thesis, I provided a middleware for B2C e-commerce domain, which facilitates the building of similar e-commerce applications.

2.1.3 Framework

A framework is a reusable design, which is expressed by a set of classes and the collaboration among instances of these classes [8]. These classes can be tailored by developers to fit a particular application and can be reused by a series of similar applications. Frameworks include, but are not limited to, GUI frameworks, middleware frameworks, application frameworks, and domain frameworks. A GUI framework can be utilized to build user interfaces. Examples of this category of framework are JAVA AWT [28] and Swing [22]. A middleware framework provides services that are needed for building the middle layer of an application, such as

COM+, CORBA, and EJBs. An application framework is used for building a complete application: from user interface, transaction, and concurrency control to database connection. The JAVA API (Application Programming Interface) and Microsoft's .NET [25], which are both class libraries with lists of classes, interfaces used to build applications, components, or controls, belong to this category. A domain framework is a collection of classes that can be reused in a particular domain, like OFFER [3] for the B2B business domain.

Johnson and Foote [41] classified framework into white-box frameworks and black-box frameworks. The classes of a white-box framework are transparent to developers. Developers have to understand the classes before they can use them. Therefore, white-box frameworks are normally hard to learn and hard to reuse. On the contrary, a black-box framework is only accessible through its external interfaces, the detailed implementation is invisible to developers; therefore, it is easy to reuse but it has less flexibility.

The middleware framework I proposed in this research was a black box framework for the e-commerce domain focusing on the B2C business model.

2.2 Related Work

2.2.1 Standard Middleware Frameworks

To ease the development of middleware components, many standard middleware frameworks have been proposed. Three of the main frameworks are Microsoft's Common Object Model (COM+), Sun Microsystem's Enterprise JavaBeans (EJBs), and the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA).

COM+ is an object model provided by Microsoft. COM+ supports many standard services like security, transactions, and garbage collection. It can run on operating systems other than Windows 2000 by using third party components. The main feature distinguishing COM+ from other approaches is that COM+ has achieved binary encapsulation and binary compatibility [6], which are lacking in both CORBA and EJB. Binary encapsulation means that the client objects do not have to be re-compiled if the server objects change, while binary compatibility means the client and server objects can be developed in different environments and using different languages [6]. Microsoft achieved this feature using a mechanism that separates interfaces from implementation. Thus, COM+ supports a broad range of implementation languages: C++, Visual Basic, and Visual J++, etc. Furthermore, compared to other middleware standards, COM+ is not only fast and easy to build with, but it is also more robust [6]. For all these reasons, I choose COM+ as the middleware platform for the implementation of my proposed middleware framework for B2C e-commerce transaction.

Enterprise JavaBeans (EJBs) is a server component architecture provided by Sun Microsystem. EJB's standard component framework provides services for transactions, database connections, security, and replication, which are important features necessary to create component-based, distributed business applications. The deployment environment could be J2EE, IBM's WebSpere, or BEA's WebLogic, etc. However, Enterprise JavaBeans using CORBA or RMI network protocol are sometimes slow compare to COM+ [10]. In addition, Java is the only implementation language allowed, which also hampers the usefulness of EJB.

CORBA is an object-based distributed architecture defined by the Object Management Group (OMG). CORBA specifies an Interface Definition Language (IDL), which specifies details about the different distributed objects and gives a common interface hiding the implementation details thereby providing support for different programming languages. Communication between distributed and heterogeneous objects is achieved through an Object Request Broker (ORB), which is responsible for managing communication and data exchange between objects. CORBA also supports standard services, such as persistence, transactions and concurrency control, etc.

COM+, EJB, and CORBA are generic object models designed for support services like transactions, security, and replication. However, they do not provide services that are needed for a specific domain. Domain developers have to take care of everything from middleware design to implementation. E-commerce domains are not an exception. Most e-commerce systems share some common functionality, such as customer information management, payment management, and order processing. Usually, these functionalities have to be implemented from scratch by each of them. Therefore, proposing such a middleware framework, composed of a set of components that provide such functionality, will contribute both to software reuse, ease of development and software quality.

2.2.2 Reusable Domain Frameworks

Software reuse has long been an active research area due to its potential benefits, which include increased productivity and product quality and decreased development cost and time. Many approaches have been proposed in the e-commerce domain generally.

Aleksy et al. [1] proposed a CORBA-based architecture for e-auction applications. By analyzing the general process of an electronic auction (or e-auction), Aleksy et al. identified two specific kinds of communication techniques that are required by e-auction applications and which are not available in CORBA. They defined these two communication techniques using CORBA's Interface Definition Language (IDL) as their core architecture. Besides this core architecture, they also described additional components that are needed for the implementation of complete e-auction applications. The main focus of their architecture is to provide flexibility so that it can support various kinds of auctions and to provide interoperability, which is important to exchange information with existing systems, such as Enterprise Resource Planning (ERP) systems.

Anido et al. [2] proposed a component-based architecture for building web-based learning systems. Their work focused on the identification of a common set of services that are needed by web-based learning systems and the definition of these common services using open software interfaces. Since their proposed architecture is independent of any underlying infrastructure and programming language, it can be implemented on any middleware platform. As an example, Anido et al. described an implementation using EJB technology. Although this architecture was developed for the e-learning domain, the development methodology (e.g., functional requirement captured from use cases) used in this work could be a guide for the development of my middleware framework for the B2C e-commerce domain.

V-Market [21] is an object-oriented framework proposed by Ripper et al. for building agent-based e-commerce systems. An agent is a piece of software that can work on behalf of users to buy, sell, and find specific goods and services. Any user of this kind of system could be a potential buyer, and a seller. Therefore, the proposed framework was mainly designed for building agent-based e-commerce applications. Ripper et al. used a software engineering approach to design the framework and used extended UML diagrams to model their design.

Bichler and Segev [3] proposed an object-based framework called OFFER. The main objective of the OFFER project was to specify and design components that support brokerage in the business-to-business e-commerce domain. Thus the applications derived support functions from the framework like customer registration, supplier propagation, customer interests expression, etc.

Laudon and Taver [34] provided a systematic approach to build successful e-commerce system. They discussed two-tier and multi-tier e-commerce architecture and technologies that can be used to build e-commerce applications. Furthermore, they identified the data flow in a typical online store application and discussed the challenges in building e-commerce applications.

However, a middleware framework for building online store application is not available according to our knowledge. To address this problem, I propose an object-based middleware framework in this research.

2.2.3 Component-base Software Development

Providing a middleware framework for e-commerce transactions is a complex process involving a set of core phases of component-based software development, including requirements analysis, requirements specification, design, implementation, quality evaluation, etc. Much research work has been done to improve the quality of each of these phases.

Progovac [20] provides some guidelines for doing requirement analysis. These guidelines include the duality principle in data collection and the lead principle in use case design. The duality principle means that one should always analyze a system's requirement from both the users' and designers' view points. The lead principle means that one can identify a system's core use cases first, and then use these core use cases to find the others. Using these guidelines, the correctness of requirement analysis and design can be predicted. Jang et al. [9] summarize the existing component specification methods and propose a set of formal activities, which specify the requirements of components and verify the correctness of component specification using the formal specification language Z. The proposed activities are able to improve the quality of components and reduce the cost of design and implementation by guaranteeing the correctness of component specifications.

MiSook et al. [5] introduce an efficient component identification method based on use cases, which includes three phases. The first phase identifies one component for each use case. The second phase separates the common classes from the components identified from the first phase. Common classes are classes that belong to two or more components. The last phase further separates the large components into small size components that only contain highly related functions. By using this method, in practice, the reusability and maintenance of components can be predicted.

Similarly, Kim et al. [11] propose an efficient framework generating method based on UML diagrams. In their method, frameworks are identified by analyzing use case diagrams[1], class diagrams[2], and sequence diagrams[3]. In addition, Yang et al. [29] propose a practical object-oriented framework development process, which consists of four typical software development phases: analysis, design, implementation, and testing. The proposed process is also based on UML diagrams. Using these methods in practice improves the productivity of the framework development.

Coupling is the degree to which components depend on one another [7]. Cohesion is the extent to which the individual components are needed to perform the same task [7]. High cohesion and low coupling have long been design goals in software components design because components with low coupling and high cohesion are more likely to have high quality such as being easy to reuse, simple to understand, and easy to maintain.

The methods used to measure cohesion are usually by measuring the interrelationship of constituent parts of a component. Misic [16] suggests a new method to measure cohesion, (i.e., measure the external usage pattern of a component without considering the component's internal structure). Misic uses the term coherence instead of cohesion to isolate his method from the traditional ones. However, coherence and cohesion are same concept. Cohesion and coupling analysis could help the identification of interface of components and make decision concerning the sizes of components.

Kim et al. [12] provide a framework to improve the quality of components in component-based software development. Their framework has four phases: quality

[1] Use case diagram: Use case diagrams model the functionality of a system using actors and use cases. Use cases are services or functions provided by the system to its users.
[2] Class diagram: They describe the static structure of a system.
[3] Sequence diagram: Sequence diagrams describe interactions among classes in terms of an

specification, quality planning, quality control, and quality evaluation. Developers can apply their framework in various phases of component development to improve the quality of components.

2.3 Design Methods

2.3.1 UML

UML (Unified Modeling Language) is an object-oriented modeling language standardized by the Object Management Group (OMG) mainly for software systems development. UML combines three different modeling methods (i.e., OMT, Booch, and OOSE) to specify, visualize, construct, and document the artifacts of software systems from the most abstract description of the system behavior, through the system architecture, down to the level of detailed design. UML is becoming the dominant modeling language in object-oriented analysis and design community [19].

UML provides multiple diagrams to model a system from several perspectives or at multiple level of abstraction. These diagrams are use case diagram, class diagram, state diagram, activity diagram, sequence diagram, collaboration diagram, package diagram, component diagram, and deployment diagram.

Use case diagrams specify the interaction of the users and the response of the system. By analyzing these interactions, we can capture the functionalities that a system should provide. These analyses also could guide the design and the implementation of the system and guide the making of testing plan. Class diagrams show the classes of the system and the interrelationships among these classes. However, class diagrams are static diagram, they cannot show how the classes

exchange of messages over time.

interact to each other. State diagrams, collaboration diagrams, sequence diagrams, package diagram, and activity diagrams can present the dynamic aspects of a system. Each of these diagrams has their own advantages and disadvantages and therefore can be used in different situations. Component diagrams show grouped modules of a system and deployment diagrams identify the system configuration at a given running time.

UML consists of a variety of notations, which have made UML a popular modeling language in multiple application domains for system documentation and specification, for capturing user requirements and defining initial software architecture. UML notation is basically consisted of four kinds of graphical constructs (i.e., icons, 2-d symbols, paths, and strings) and some additional diagram elements, which include mappings, names, labels, keywords, expressions and notes. Moreover, UML has extension facilities (stereotypes, tagged values and constraints) that allow semantically meaningful versions of UML to be built for specific application domains.

UML modeling language is essential in software development. First, UML is an efficient tool for communication among people. Designers need to communicate with potential users at requirement analysis stage. Designers and developers need to communicate at design or develop stage. The visualized, easy understand UML diagrams can perform as a media helping them to understand each other. Second, UML helps to manage complexity of software development. UML diagram can separate the whole system into different parts. By understanding and managing each part, we can solve the problems of the whole system. Third, UML provides the capability of software design reuse. By identify the similarity of different applications in the same domain, we can find that some the design parts are actually reusable.

In this thesis, I will use several different types of UML diagrams to present different aspects of the proposed framework design.

- Use case diagrams which are used to capture the functional requirements;
- Collaboration diagrams which show how objects collaborate to perform use cases;
- Class diagrams and component diagrams that present the system structure;
- State diagrams that show the dynamic behavior of objects;
- Deployment diagrams that describe the deployment of distribute objects.

2.3.2 Pseudo Code

Pseudo code is a detailed description of what a computer program or an algorithm must do using a natural language rather than a programming language. It cannot be compiled nor executed. But it is very easy to be converted into the final code since most of notion used in pseudo code are borrowed from programming languages such as C, Lisp, or Fortran.

Pseudo code is a very efficient tool to describe software design because it has many advantages. First of all, it allows designers to express the design in great detail and focus more on the logic aspects of a computer program without being bothered by the correctness of implementation code. Secondly, it provides programmers a detailed template for the next step of writing code in a specific language. Pseudo code can be easily translated into program code because the structured natural language description is very similar to the real structure of program code. In addition, reading pseudo code is much easier than reading another person's code. Thirdly, it is easy to inspect that whether the implementation actually match the design because it allows the easy communication among designers and programmers. Catching errors at the pseudo code stage is cheap than catching them later in the development process. Therefore, I will use pseudo code to express the detailed design of this middleware framework.

Chapter 3

Design

This chapter first discusses the design issues concerning the middleware framework for B2C e-commerce transactions and then presents the middleware framework design. The design includes requirements analysis, architecture design and detailed design. UML class and component diagrams are used to describe the architectural design, use case and collaboration diagrams are used to capture the requirements, state diagrams are extensively used to depict the behavior of objects and pseudo code is used for detailed design.

3.1 Middleware Framework Design Issues

Due to the lack of a middleware framework for B2C e-commerce applications, each application has to be developed from scratch. Such e-commerce systems are not only expensive to create but are also error prone. Therefore, a middleware framework, which is reusable both in its design and implementation, is desirable. However, a middleware framework has to be easy to reuse, to maintain, and to understand. It also must provide high performance; otherwise it will not be accepted by users and will not survive. To achieve these design goals, I will examine components' granularity, communication, coordination, and concurrency issues during the design and implementation stages.

3.1.1 Component Granularity

Component granularity determines the number of functions performed by a component. Component granularity plays an important role in finding a correct balance between performance and maintenance. If the functional partition (i.e., the way the application logic is divided across components according to function) is too coarse, generally, components' performance will be good, but they will be hard to reuse, debug, and maintain. If the partition is too fine-grained, they will be easy to reuse and maintain, but there will be additional network communication overhead. Coupling, cohesion, and coherence are metrics which can be used as partitioning criteria in making function partition decisions. Components with low coupling will have low dependency on other components and they will have low change impact by others. Therefore, they are more likely to be reused, simple to understand, and easy to maintain. Components with high cohesion have relatively few methods with highly related functionalities, which also mean they are much easier to reuse, maintain, and understand. By performing coherence, cohesion, and coupling analysis on each component, one can potentially predict its future performance, reusability, and maintenance requirements.

3.1.2 Communication

Communications between distributed objects require different forms to satisfy the needs of non-functional requirements like system reliability and high performance. Some of the communication forms are provided by standard middleware platforms, such as synchronized communication [6]. In synchronized communication, the client object is blocked while the server object executes the requested operation. However, some communication forms, such as asynchronous communication (i.e., server object gives control back to the client object immediately after receiving the client object's request) are often not provided. Application developers could use threads or message queues to implement these services that are not provided by standard middleware platforms. However, achieving a high degree of reliability while using a message queue or transactions is both time- and space-intensive, so in the design stage the designer should evaluate the gains and trade-offs, and choose an appropriate communication method to achieve reasonable performance and reliability.

3.1.3 Concurrency

Many customers may visit an e-commerce site concurrently. It might happen that these customers' operations are modifying the same server objects. In this circumstance, two problems might occur: an update may be lost or the state of the object may become inconsistent [6]. To solve these problems, we have to control concurrency in such a way that two transactions must be executed one after the other (i.e., they must be serializable). To control concurrency, respective standard object models have various implementation methods. For example, COM+ handles concurrency by spawning concurrent threads as long as developers make a synchronization configuration by using the component services administrative tool. CORBA uses locking, both two-phase and hierarchical. However, a designer has to locate the objects that need concurrency control and specify them in the design stage.

3.1.4 Consistency

When designing object-based middleware for e-commerce transactions, object consistency is also a very important issue that needs to be carefully addressed. For instance, in an order check out transaction, we must ensure that the same number of items purchased by a customer is removed from the inventory. Another example is that if a debit operation would leave a negative balance from a customer's credit card account, the object should always reject this execution. To achieve object state consistency, not only should we capture the consistency in the design stage, but also implement the methods used to keep consistency in the implementation stage.

Replication is a mechanism often used to support availability. Replicas, which are distributed on different computers, require state consistency. If the replicas fail to keep a consistent state with the original object, then their function will similarly differ. Hence, system consistency cannot be assured. To meet these consistency constrains, the standard object models (COM+, EJB, CORBA) provide different mechanisms, but the implementation details of each object are still the responsibility of designers.

3.1.5 Performance

E-commerce applications make high demands on a system's performance. Therefore, we need a deep understanding of our middleware framework, which is intended to facilitate the development of an e-commerce application. How will the components interact with each other? What will be the system's bottlenecks? How will the system's workload affect its performance? All of these questions need to be considered at the design stage.

3.2 Issues That Will Be Examined

The granularity of each component must be carefully selected to ensure the performance, quality and reusability. I will use combined metrics, namely coupling, cohesion, and coherence, to analyze each component and to confirm its expected performance and reusability at the design stage. To achieve this design goal, I will use both traditional and modern methods to measure the coupling and cohesion of components. The traditional method provided by Steven et al., [23] is simply to maximize relationships among elements in the same component and minimize the relationships among elements in different components. Consequently, components will be more likely to have low coupling and high cohesion. The modern method, which is provided by Misic [16], is to measure the coherence of components. Instead of measuring the interrelationships of a component's constituent parts, coherence measures the external usage pattern of a component without considering the component's internal structure. By using mixed metrics to analyze each component, I will be able to properly select component granularity, thereby improving component reusability and maintainability and ensuring a high performance for the proposed middleware framework.

The concurrency issue will also be examined. At the design stage, I will identify all of the objects that need concurrency control, such as product, customer, shopping cart, and inventory. However, implementation methods may vary according to different programming languages. For example, one could use threads and synchronization primitives to implement concurrency in JAVA or use other concurrency techniques provided by different middleware platforms. In the implementation stage, I will use the concurrency control techniques provided by COM+, because it takes care of concurrency issue as long as designers make a synchronization configuration.

Consistency is another issue that needs to be examined both at the design and implementation stages. For example, if a customer has paid for a number of products in his/her basket, then the same number of products should be decreased from the inventory table through the inventory object. Otherwise, another customer might buy a product that does not exist in the inventory. All such situations will be identified at the design stage and indicated in pseudo code. Additionally, I will use the transaction mechanism provided by standard middleware frameworks to keep components consistent.

3.3 Design Methodology

The complexity of an e-commerce system requires good design methods to face challenges such as flexibility, scalability, and reusability. The MVC design pattern was used throughout the design of the proposed middleware framework.

3.3.1 The Model View Controller Patterns (MVC)

Model-View-Controller is a design pattern that enforces the separation between the input, processing, and output of an application. An application based on an MVC design pattern is separated into model, view, and controller components. Each of these components handle a set of tasks. The model component represents an application's data and business rules. The view component specifies how the data should be presented and provides an interface to accept user input. The controller accepts user requests and translates each request into actions to be performed by the model. Using the MVC design pattern, the three components' reusability, flexibility, and maintainability can be improved since each is self-contained and the inner implementations are hidden from one another. One can easily change any of the three components without impacting the others.

E-commerce applications are typical *n*-tier architecture applications, which require scalability, flexibility, and reusability. Therefore, it is very important to follow the MVC design pattern in the development of this framework. With respect to the MVC design pattern, an e-commerce application can be modeled as presentation component (i.e. View), model component and controller component. The View component is a logically self-contained layer, which are the web pages used to accept user input and to display the data coming from the model component. The model component is represented by a set of business objects, such as product, customer, and supplier. These objects implement actual data processing and business rules. The model component feeds data to the view component without worrying about the actual data formatting. The controller component is represented by a set of control classes, which is responsible for notice of action and commands model and view component to change. Figure 3-1 shows how the MVC design pattern is applied to the e-commerce system.

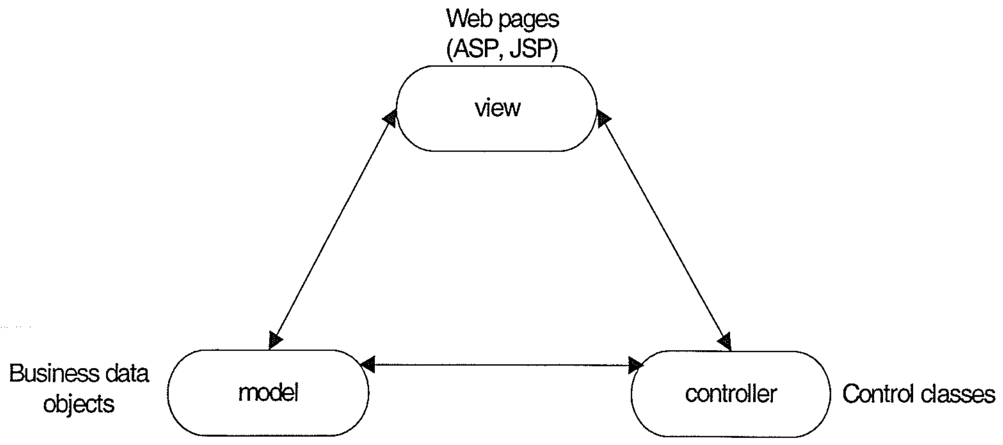


Figure 3-1 MVC Design Pattern in E-Commerce

3.4 Component Identification Method

The functionality of my proposed middleware framework is provided by a set of components, as component-based development has been the most promising way of improving software's reusability, maintainability, and productivity. Therefore, an efficient component identification method is essential for the success of this thesis work.

A systematic component identification method, which combines MiSook et al.'s (2001) component identification method based on use case, and Misic's (2001) cohesion measurement method, is used in this thesis. Component reusability and maintainability can be predicted since the components are derived conforming to the high cohesion and low coupling principle.

This combined method includes three phases. The first phase identifies one component for each use case. The second phase is quite an involved phase, which depends on the sequence, class, or collaboration diagrams that are built according to the event flow of each use case. Analyzing the sequence, class, or collaboration diagrams, we can identify the classes that are required by each use case. Accordingly, we can identify the common classes that belong to two or more components (i.e., those identified in the first phase). Determining which component the common classes should belong to requires coupling analysis. A class's coupling number is calculated by the number of association relationship it has with other classes. The association relationship includes composition, aggregation, inheritance, and association. A common class should be firstly placed in the component in which it has higher coupling with other classes. When it has high coupling in both of the components and the coupling number reaches three or more, we should combine the two components in phase two.

In phase three, some of the large components found in phase two are separated into smaller components by performing cohesion analysis. Suppose component *A* is a large component consisting of components A_1, A_2, ...A_m (i.e., A_1, A_2, ...A_m are components identified in phase one). Component *A* is so large that it should be separated into several smaller components. First, all classes which have relations of composition, aggregation, and inheritance in component *A* are grouped together. We may get several grouped classes C_1, C_2, ... C_k. Then we put each of the grouped class into each consistent component A_1, A_2, ...A_m forming m*k architectures $\{\{A_1, C_1\}, A_2, \ldots A_m\}_1$, $\{A_1, \{A_2, C_1\} \ldots A_m\}_2$, $\{A_1, A_2, \ldots \{A_m, C_1\}\}_3$, $\{\{A_1, C_2\}, A_2, \ldots A_m\}_4$, $\{A_1, \{A_2, C_2\} \ldots A_m\}$, $\{A_1, A_2, \ldots \{A_m, C_2\}\}$, $\ldots \{\{A_1, C_k\}, A_2, \ldots A_m\}$, $\{A_1, \{A_2, C_k\} \ldots A_m\}$, $\{A_1, A_2, \ldots \{A_m, C_k\}\}_{m*k}$. Second, we estimate each component's cohesion in a given architecture. Then we calculate the average cohesion of each of the architectures and choose the one with the highest average cohesion as our final components. Cohesion is estimated using the metric provided by Misic. Let S stand for the component in question, and let R(S) denote its client component. Let S_w denote the subset of the set S used to write its clients. And let $S_w(x)$ denote the part of that subset which is actually used to write the client x. Then the cohesion of S is:

$$\Psi(S)= \sum(\#S(x)-1)/ \sum(\#S-1)$$

where #S stands for the number of classes in component S.

3.5 Requirements of E-Commerce Middleware Framework

Requirements of an e-commerce middleware framework include functional requirements and non-functional requirements. Functional requirements are specified using UML use case diagrams and collaboration diagrams. Each use case is analyzed in section 3.5.1. Nonfunctional requirements are expressed using metrics and each metric is discussed in section 3.5.2.

3.5.1 Functional Requirements

The proposed middleware framework, which is reusable by a family of similar applications, should provide functions that are common from application to application.

Figure 3.2 presents the common use cases of such an e-commerce system. In order to identify the functions, I will analyze each use case scenario and document the scenarios using UML collaboration diagrams shown from Figure 3-3 to Figure 3-12. The notation for collaboration diagrams is referenced from [35].

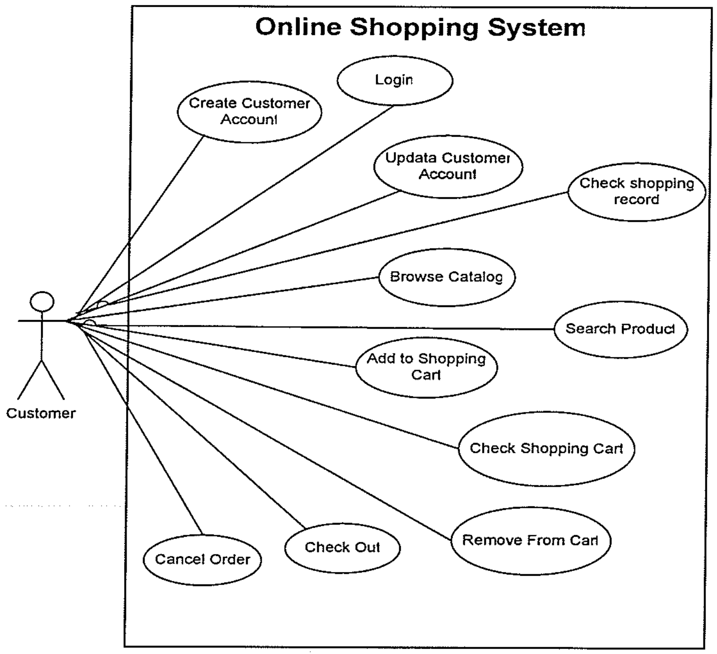


Figure 3.2 Use Case Diagrams for B2C E-Commerce

Case 1. Create a Customer Account

Any user who wishes to purchase products via an e-commerce system must provide his/her personal information to the system to become a registered customer. Figure 3.3 presents the detailed process of creating a new customer account. A user provides his/her detailed personal information to the system, such as name, address, password and email address. The e-commerce system will check the database to see whether the e-mail and password exist. If so, the system will deny the customer's registration and request the user to input new values for these fields. Otherwise, the system will create a unique customer ID for the user and notify him/her that the account has been successfully created.

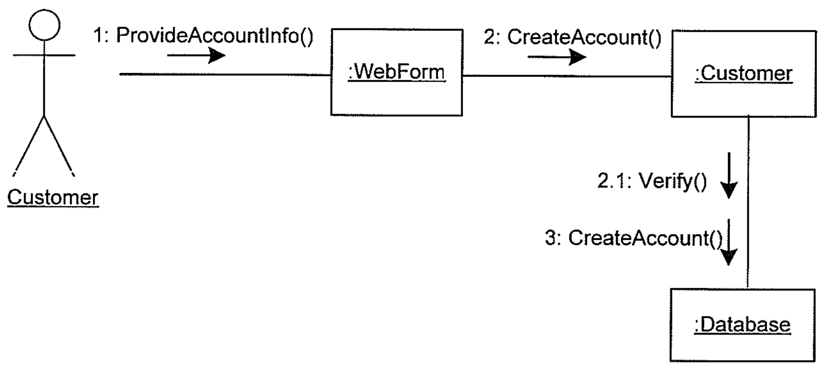


Figure 3.3 Collaboration Diagram for Creating a Customer Account

Case 2. Update a Customer Account

When customers return to an e-commerce system, they are allowed to modify their personal information as long as the change request occurs at any time other than check out. The user will be shown an update interface to collect new information. The system is responsible for verifying the modified user information. If it is valid, new information will be written into the database and customers will be notified that information has been changed. Otherwise, the system will ask customers to modify the invalid information. The detailed collaboration diagram is shown in Figure 3.4.

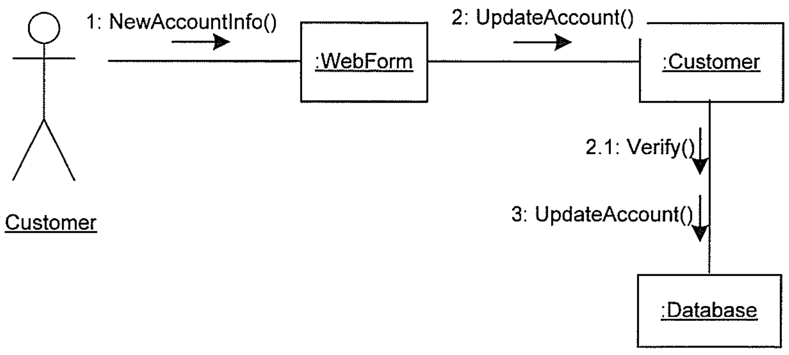


Figure 3.4 Collaboration Diagram for Updating Customer Account

Case 3. Login

Customers can logon to an e-commerce system using their e-mail addresses and passwords, which were specified in the Create a Customer Account use case. When the e-commerce system recognizes returned customers, the system will allow user to check their shopping records, update their accounts, and check out products. However, if the e-mail addresses or the passwords are invalid, customers will be informed to try again, up to three times. The login use case is shown in Figure 3.5.

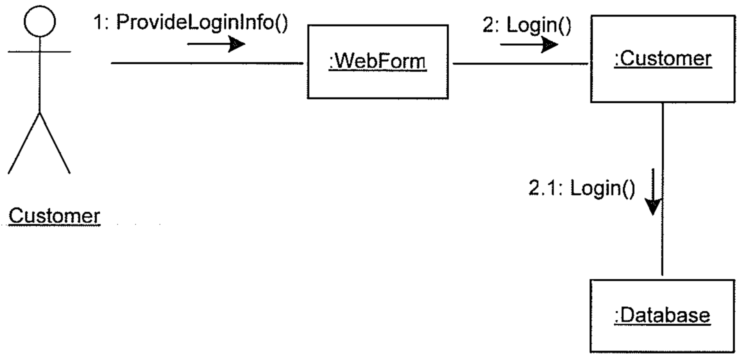


Figure 3.5 Collaboration Diagram for Login

Case 4. Check Shopping Record

Figure 3.6 shows that returned customers can check their shopping records on an e-commerce system. When a customer requests checking shopping records, the e-commerce system will show an email and password input interface. After email and password have passed validation, the e-commerce system will retrieve the customer's shopping data out of database and display them in user interface.

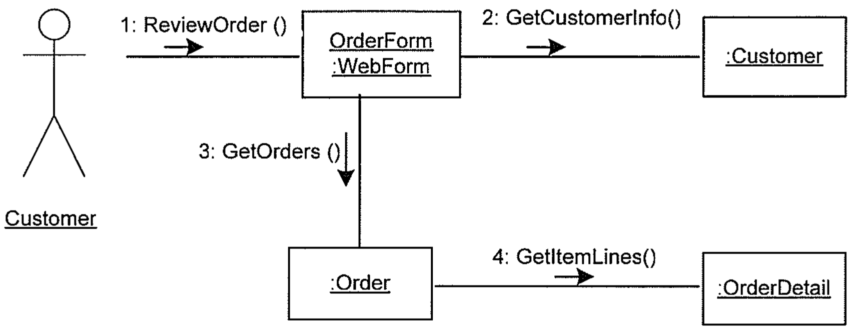


Figure 3.6 Collaboration Diagram for Checking Shopping Record

Case 5. Browse Catalog

Any user of an e-commerce system can browse the system's catalog. The catalog consists of different departments, each of which includes various products. After logon, a user can browse all departments and all of the featured products in each department. Then they can move to any particular department by selecting the corresponding hyper link. If a user is interested in a product, she may browse the detailed information. The catalog browsing diagram is shown in Figure 3.7.

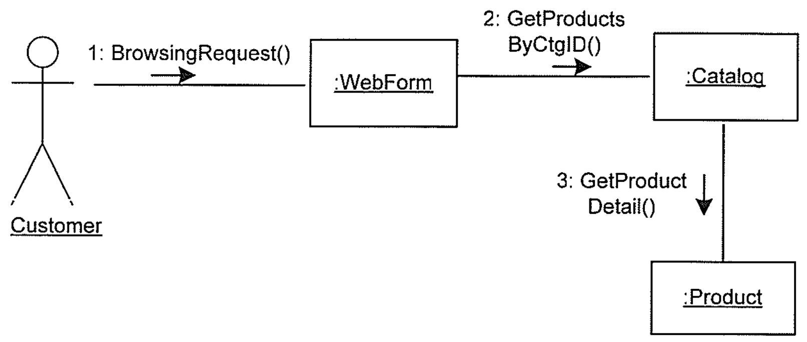


Figure 3.7 Collaboration Diagram for Catalog Browsing

Case 6. Search Product

Figure 3.8 describes how any user of an e-commerce system can search products by providing a product name, supplier name, or product price. If the search results are not empty, the e-commerce system will display the result using a formatted user interface. Otherwise, the system will display acknowledge information.

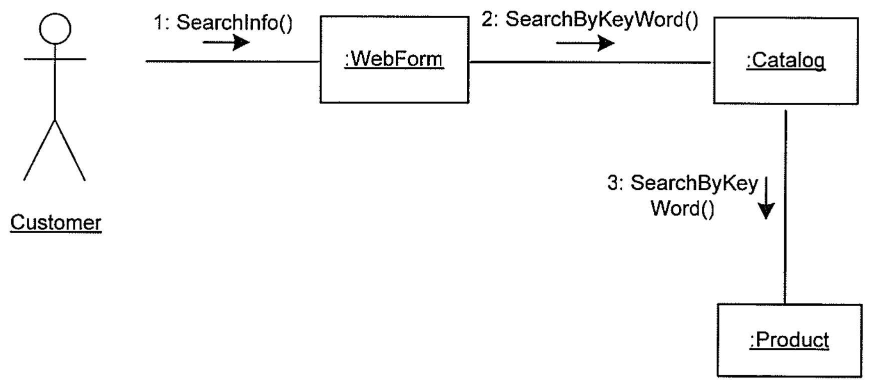


Figure 3.8 Collaboration Diagram for Searching Product

Case 7. Add Product to Cart

After browsing the catalog, customers may be interested in specific products. If the customer attempts to put a product in his/her shopping carts, the e-commerce system must check the inventory first before add the product into the customer's shopping cart. If the product that is added already exists in the shopping cart, the system will increase the product's quantity instead of adding the product into the shopping cart. Figure 3.9 describes the adding to cart use case.

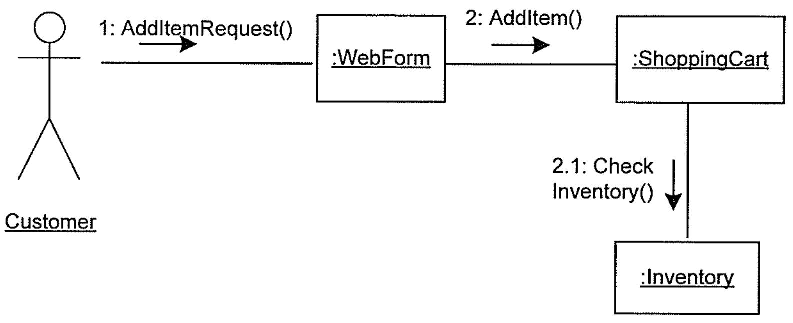


Figure 3.9 Collaboration Diagram for Adding to Cart

Case 8. Remove Product from Cart

Figure 3.10 shows how customers can remove products from their shopping carts. When the user interface shows the contents of a shopping cart, the customer can request to remove products and the system responds by showing the remaining contents of the shopping cart until the shopping cart is empty.

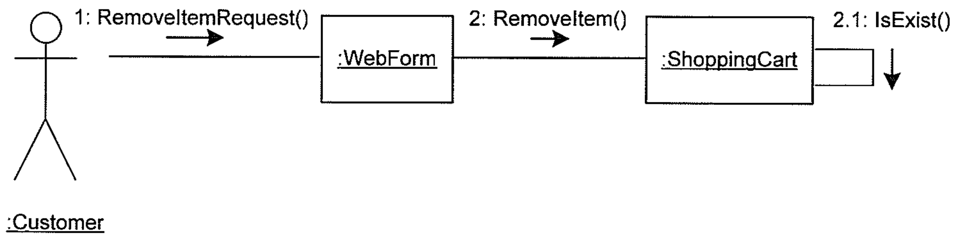


Figure 3.10 Collaboration Diagram for Removing From Cart

Case 9. Check Out

After browsing the catalog and adding products to in their shopping carts, customers can request to check out. At this moment, the system asks customers to input their payment method, detailed credit card information, shipping address, and so forth. If customers can provide all this information correctly, the system will take all the items in the shopping carts, write them to the order detail table, and create complete orders. Then the system will debit the order totals from customers' credit cards. Finally, the system will remove all the items on the orders from inventory. The detailed description is shown in Figure 3.11.

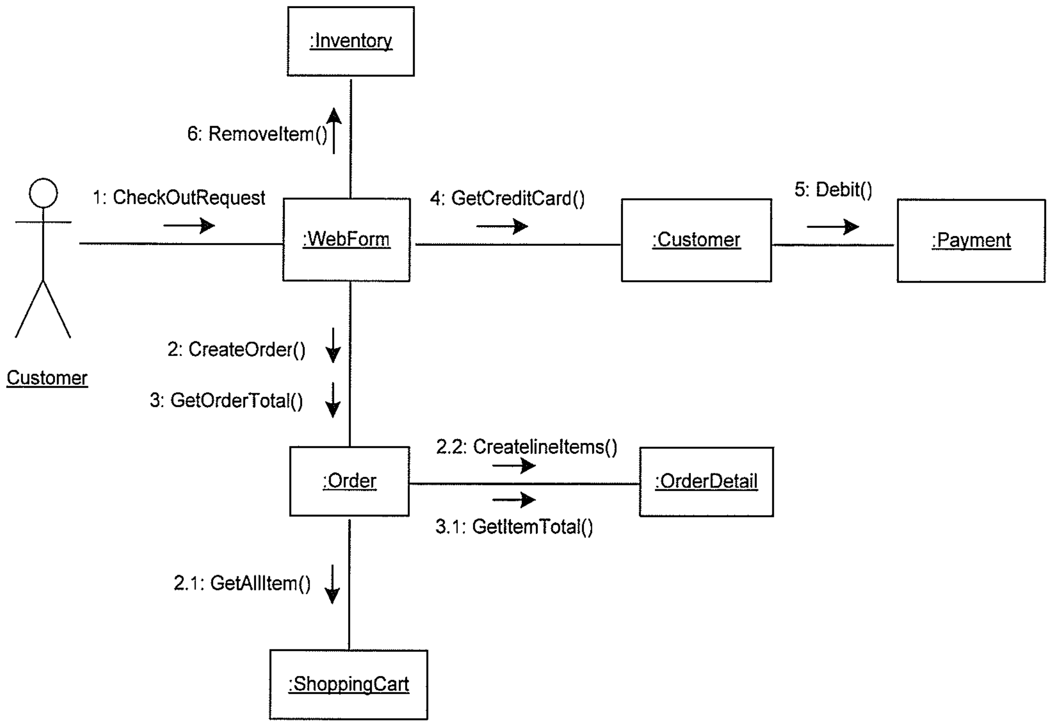


Figure 3.11 Collaboration Diagram for Checking Out

Case 10. Cancel Order

Figure 3.12 shows how customers can cancel their orders at any time before finishing the check out process. The system simply empties their shopping carts and customers can continue shopping or logout.

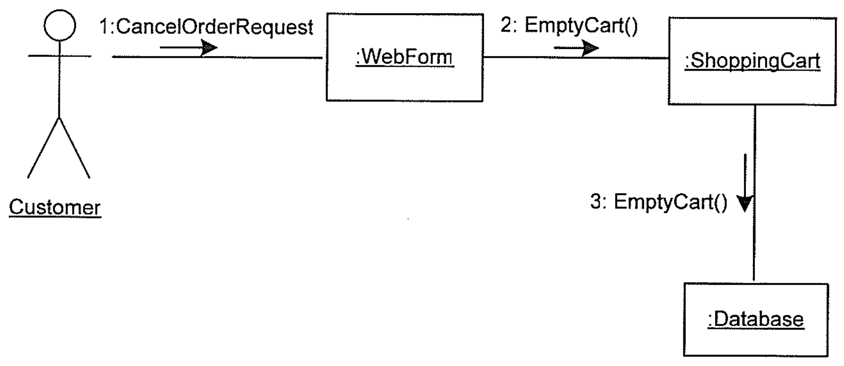


Figure 3.12 Collaboration Diagram for Canceling Order

3.5.2 Nonfunctional Requirements

A middleware framework is used for deriving a family of similar applications. Reuse of a well-designed middleware framework can improve programmers' productivity as well as enhance the quality, performance, and reliability of applications. To meet all these promises, a framework itself has to have properties such as high availability, integrity, security and performance.

- **Availability**

E-commerce is getting more and more prevalent. One of the most important reasons is that it can be accessed seven days a week and twenty four hours a day. E-commerce systems that derive from the proposed middleware framework should have as little down time as possible. The maximum down time is two minutes per day.

- **Integrity**

Integrity is the ability to determine the correctness and accuracy of data. An e-commerce system requires that all the transactions should be executed 100% accurately and all data transferred correctly over the network (e.g., customers are buying the products that they intend to buy).

- **Performance**

A middleware that promises to deliver successful B2C e-commerce must provide high performance because the competition is fierce in virtual markets. If one cannot successfully access a required web page, a user may choose another e-commerce web site, since so many of them are available. The average response time of each web page, which includes the time spent on the server handling the request, communicating over the network, and processing on the client machine (e.g., formatting the response), should be less than 5 seconds.

- **Security**

Security issues are key concerns for B2C e-commerce systems because of the open nature of the Internet. Customers' sensitive information should be protected from malicious users by leveraging several security protocols such as the Secure Socket Layer (SSL), encryption by secure HTTP (S-HTTP), and browser level authentication. We must ensure that only authorized administrators can view customer accounts and personal information, update department, and product information. Additionally, only verified customers should be able to view their shopping records.

3.6 Middleware Framework Design

3.6.1 Architecture Design

Based on the use case diagram captured in the requirements analysis, and using the component identification method discussed in 3.4, I identified seven basic components needed to deliver successful B2C e-commerce transactions, as shown in figure 3.13:

- Customer: the customer component is required because most e-commerce systems need to retain their customers' personal information, like e-mail, name and address. When customers return to the system, the system will recognize them and retrieve personal information. This component includes several classes that are used to support creating a new customer, updating an existing customers' information, and retrieving existing customers' information, etc.
- Supplier: This component contains classes that are used to provide an online catalog, hold a specific product's detailed information, manage inventory, and to manage supplier information. Customers can browse an e-commerce system's catalog, and review detailed products information. System administrators could also add new products or update old product information using this component.
- Shipment: This component maintains the classes that support different shipping policies. By using this component, existing shipping methods can be retrieved and new methods can be added to an e-commerce system.
- Shopping cart: This component contains classes that are used for adding, removing, increasing, and decreasing products in a customer's shopping basket.
- Payment: This component maintains classes that manage customers' payment methods. Customers can use this component to choose a payment method or to create a new one.
- Transaction: This component contains classes that are used for handling the checkout process. When a customer sends a checkout request, this component will write all the product information in the shopping cart into the order line table,

create an order record in the order table, and perform all the necessary transaction processing.

- Third party: This component maintains classes that are responsible for performing credit card authentication and payment processing (i.e., debit from the customer's account and credit on the supplier's account).

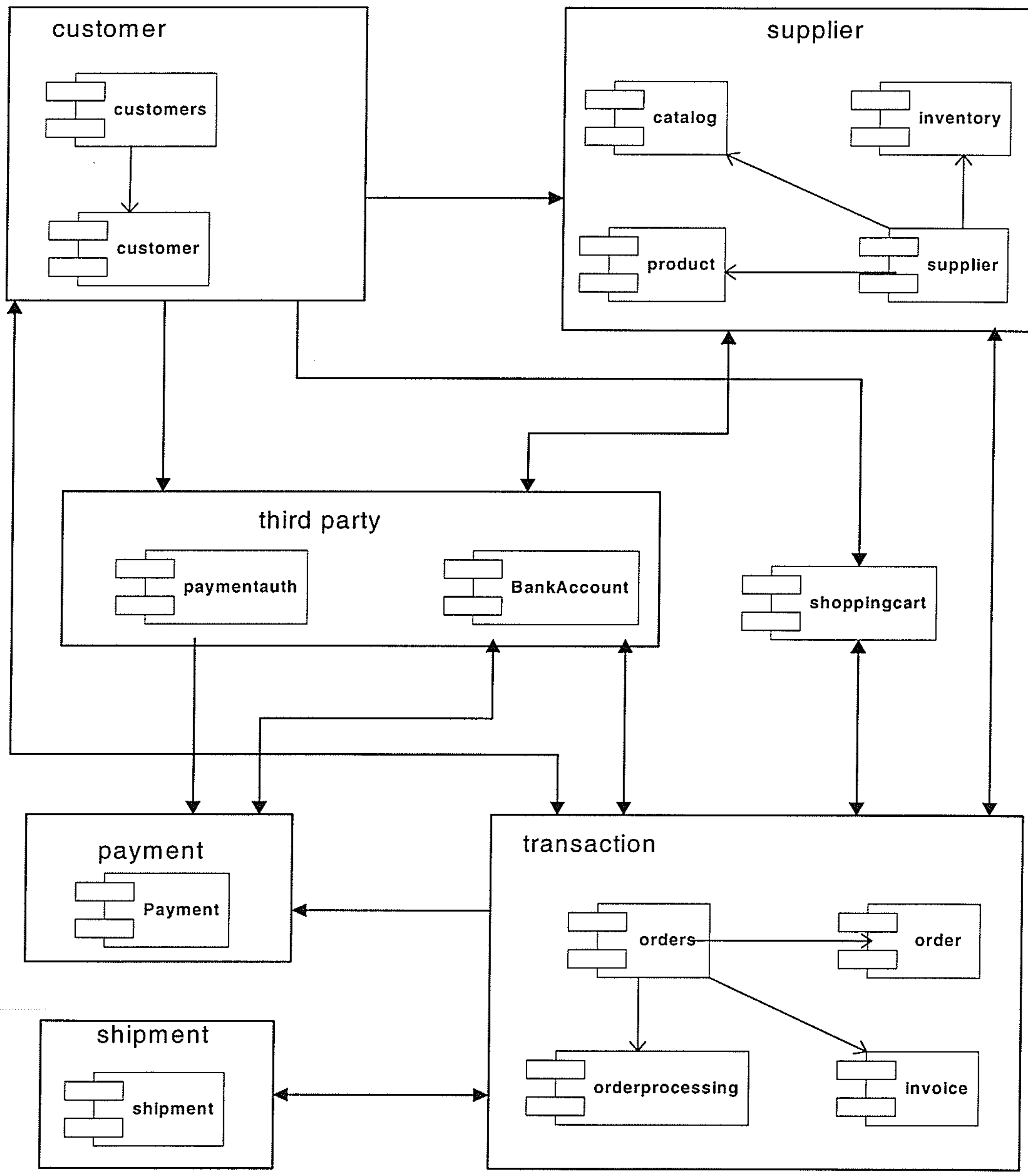


Figure 3.13 Package Diagram for E-Commerce Middleware

3.6.2 Dynamic Behavior of Objects

In this section, I use UML state chart diagrams to capture the dynamic behavior of the main objects of an e-commerce system. Analyzing these dynamic behaviors helps to identify the attributes and methods that a class should have.

1. Cart

Figure 3.14 shows that when a customer logs onto an e-commerce system, the system will create a shopping cart for him/her with an initial state of "empty". After browsing the catalog and product information, the customer might put products in his/her shopping cart and the shopping cart's state will change to "not empty". At this time, if the customer places a check out request, the system will show the contents of the shopping cart and allow the customer to make a decision either to check out or to cancel the order. If the customer chooses to pay for the products, the system will release the shopping cart after the customer paid for his/her selected products. If the customer chooses to cancel the order, then the shopping cart's state changes to "not empty" again. When the customer logs off the system, the shopping cart will be automatically released.

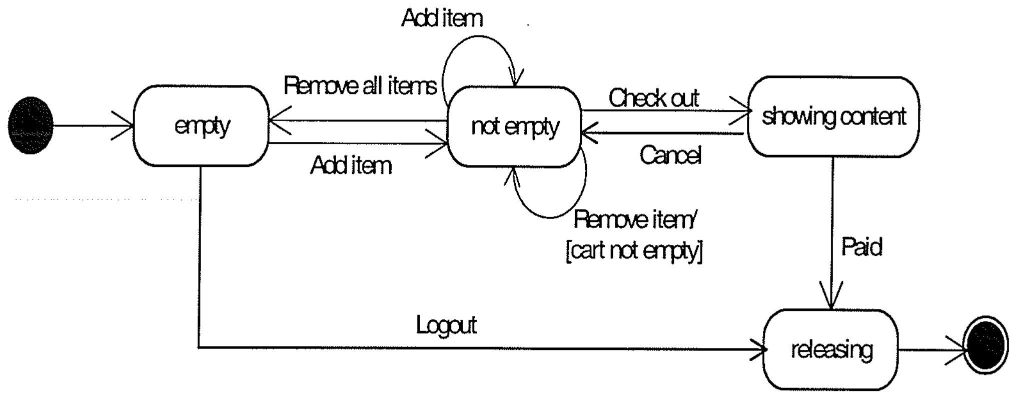


Figure 3.14 State Chart Diagram for Shopping Cart

2. Order

Figure 3.15 illustrates the order object's state changes during the shopping process. At first, the order is in an "incomplete" state. After browsing the catalog and adding items to the shopping cart, the customer may request to check out. If the check out process succeeds, an order will be created and the orders' state will change to "completed". Then, the order will enter the "processing" state, which involves stock checking, payment making, and customer notification, etc. If every step in "processing" is processed successfully, the order will enter a "fulfilled" state. If any step fails in "processing", then the order will change to the "postponed" state. When the postponement exceeds 24 hours, the order will be automatically canceled.

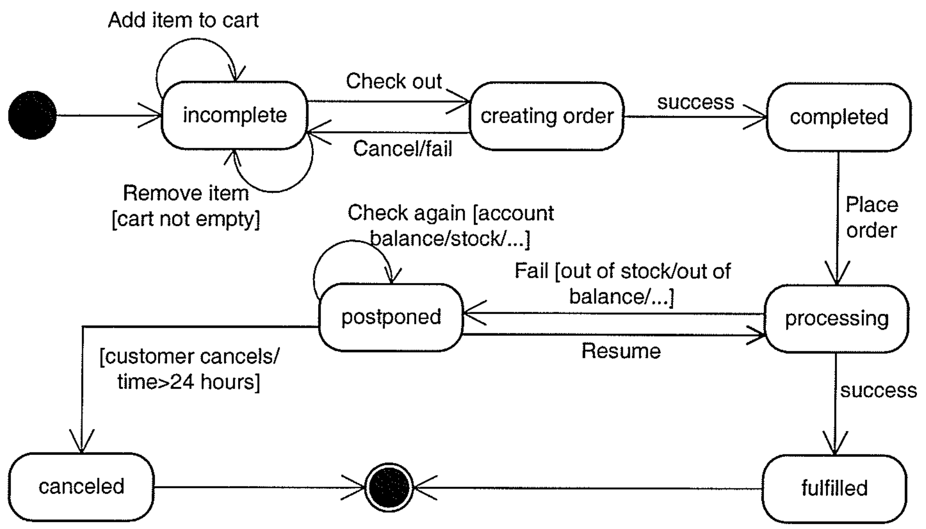

Add item to cart
incomplete
Check out
Cancel/fail
creating order
success
completed
Remove item [cart not empty]
Check again [account balance/stock/...]
Place order
Fail [out of stock/out of balance/...]
postponed
processing
Resume
[customer cancels/ time>24 hours]
success
canceled
fulfilled

Figure 3.15 State Chart Diagram for Order

3. Order Processing

Once an order has been successfully created, it will move to an "order processing" state. In this state, the e-commerce system will first check the inventory. If the inventory has the requested products in stock, the order moves to the "payment processing" state, which involves several sub steps, such as credit card verification, customer account debiting, and supplier account crediting. If every sub step proceeds successfully, the system will move to notify the shipper and the customer. In the event that any of the previous steps fail, the order will be postponed. After 24 hours of postponement, the order will be cancelled. Figure 3.16 details this state.

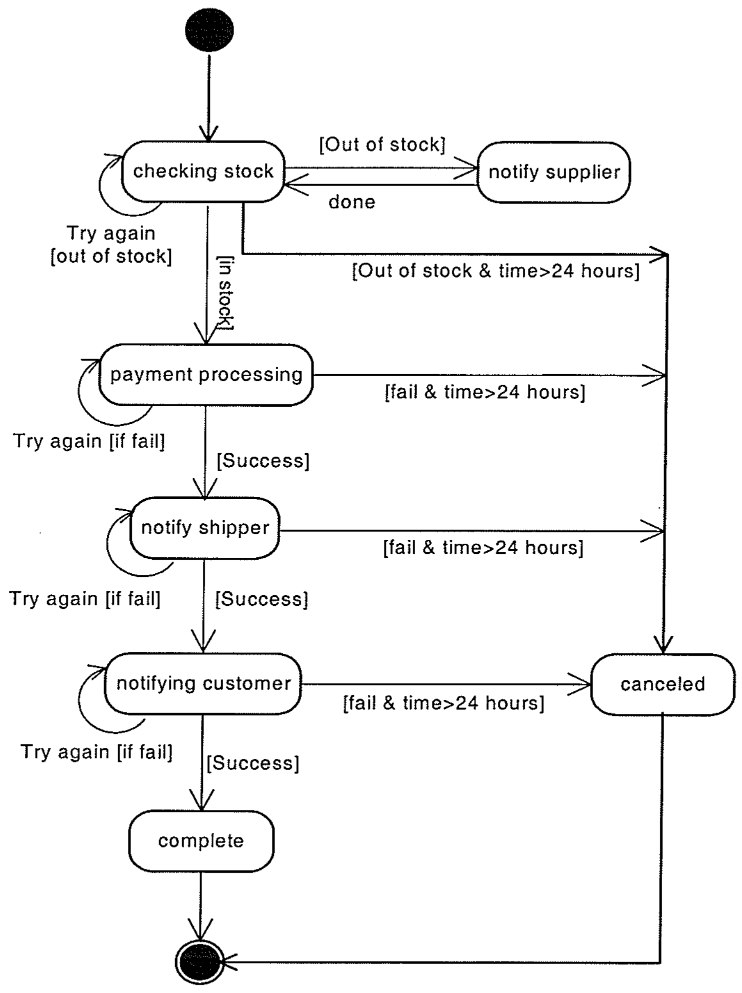


Figure 3.16 State Chart Diagram for Order Processing

4. Inventory

Figure 3.17 shows that the inventory object may be used to add new products into the database, modify existing products, and to check products stocks. When a customer submits a check out request, the inventory object will be used to check stock. If there are enough products in stock, the same number of products purchased by the customer will be removed from inventory. Otherwise, the customer will be notified that there are not enough products in stock. At the same time, the suppliers will be notified to replenish product stocks.

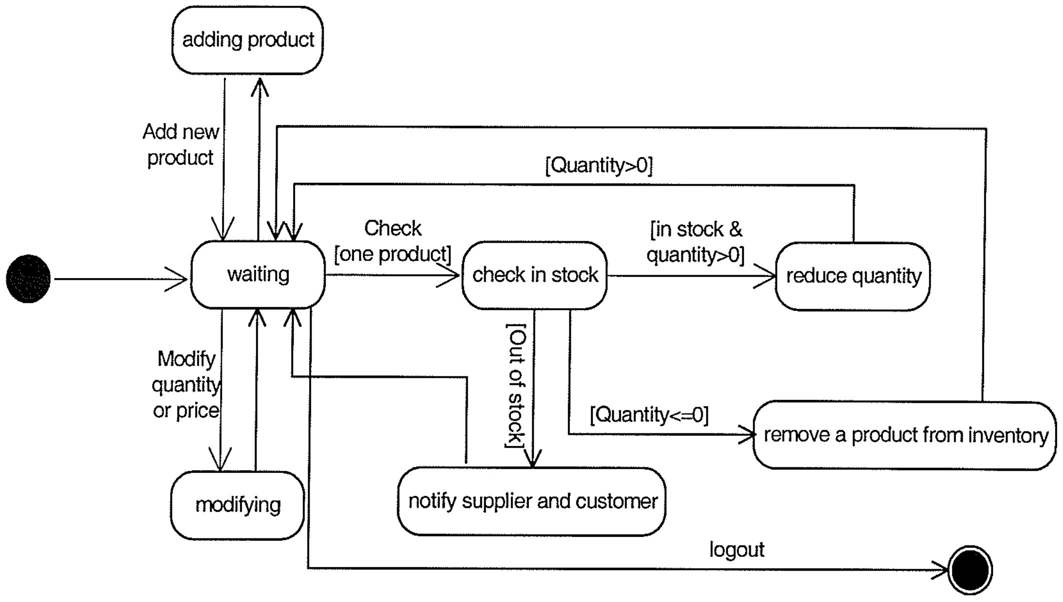


Figure 3.17 State Chart Diagram for Inventory

5. Customer

Figure 3.18 illustrates the state changes of the customer object. When customers come to an e-commerce system, they may create accounts to become registered customers, browse the catalog, and search products. If they are registered customers, individuals can check their shopping records, modify personal information, and purchase products. After they have paid for the products, customers will receive an email from the system.

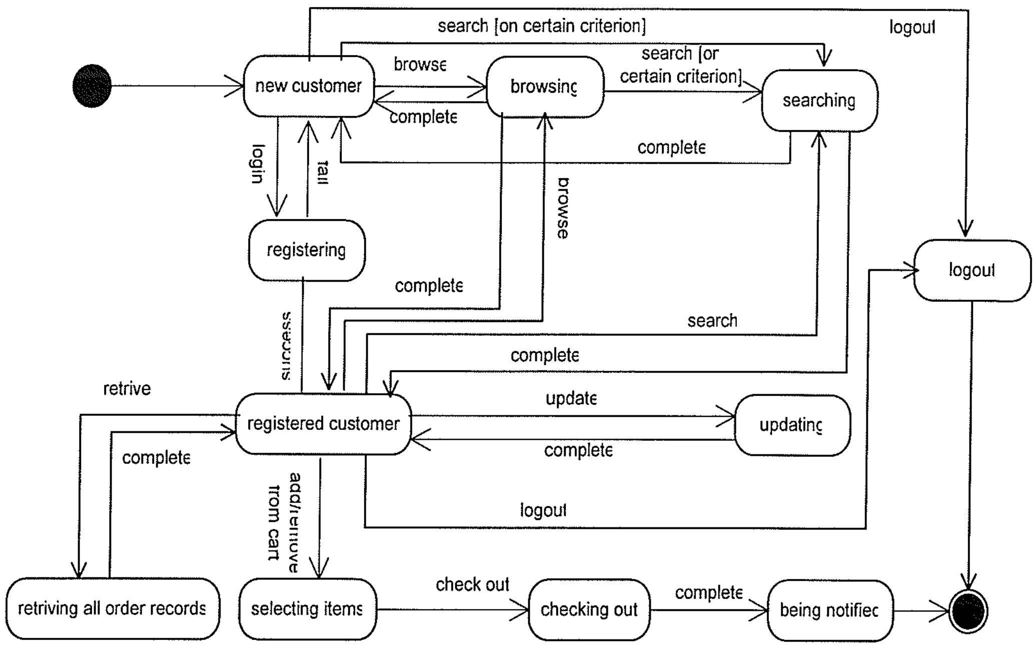


Figure 3.18 State Chart Diagram for Customer

5. Catalog

In Figure 3.19, the catalog object starts in an "initial" state, which may then proceed to the "retrieving" state. If the retrieval is successful, catalog object enables the system to show the contents of the catalog. If the retrieval fails, the e-commerce system must encounter some errors. From the "showing content" state the catalog object can change to the "modifying" state and the "browsing" state. After the customer logs out, the catalog object will be released.

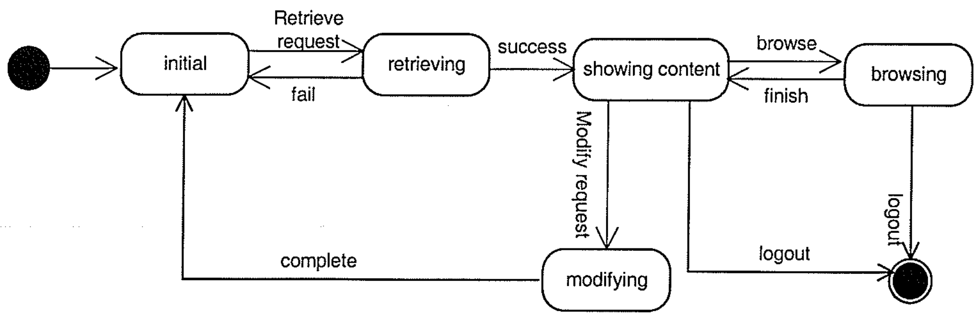


Figure 3.19 State Chart Diagram for Catalog

6. Product

Figure 3.20 shows that a product can start in the "adding to inventory" state. Once a product exists in inventory, it can be browsed and added to a shopping cart. After being checked out, a product may be in shipping and receiving states. When a product's quantity drops to zero, the product will be removed from the inventory. In addition, a product's property such as name, price and description, may be modified.

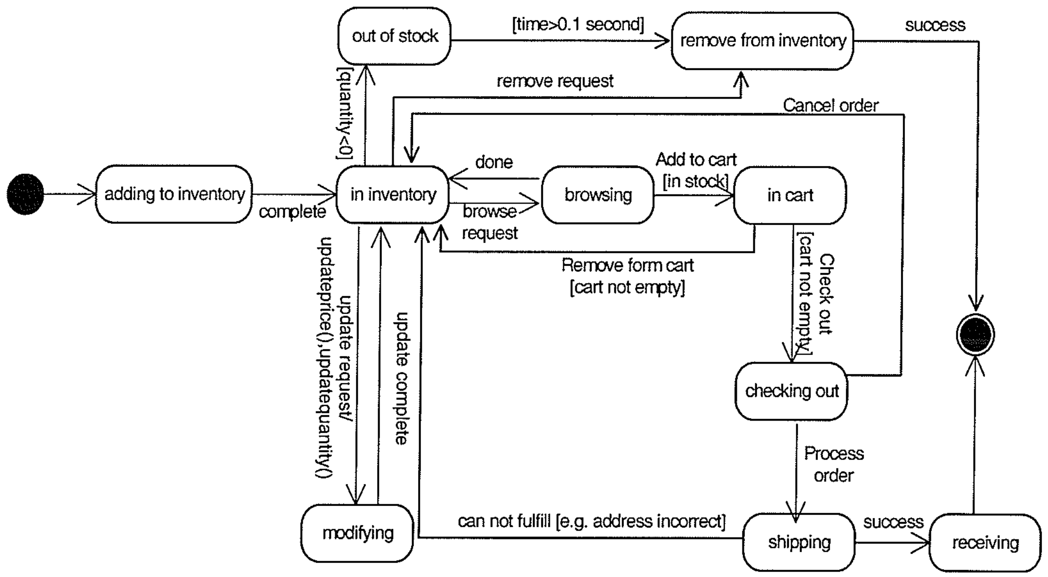


Figure 3.20 State Chart Diagram for Product

7 Payment

Figure 3.21 demonstrates that a payment process may start with collecting a customer's credit card information. Once the collection is complete, the credit card information will be encrypted and sent to a third party for verification. If the credit card information is approved, the total value of the customer's order will be debited from the customer's account and credited to the supplier's account. Otherwise, if the credit card information is rejected, the e-commerce system will ask the customer to correct his/her information until the customer cancels the order or the credit card information is approved.

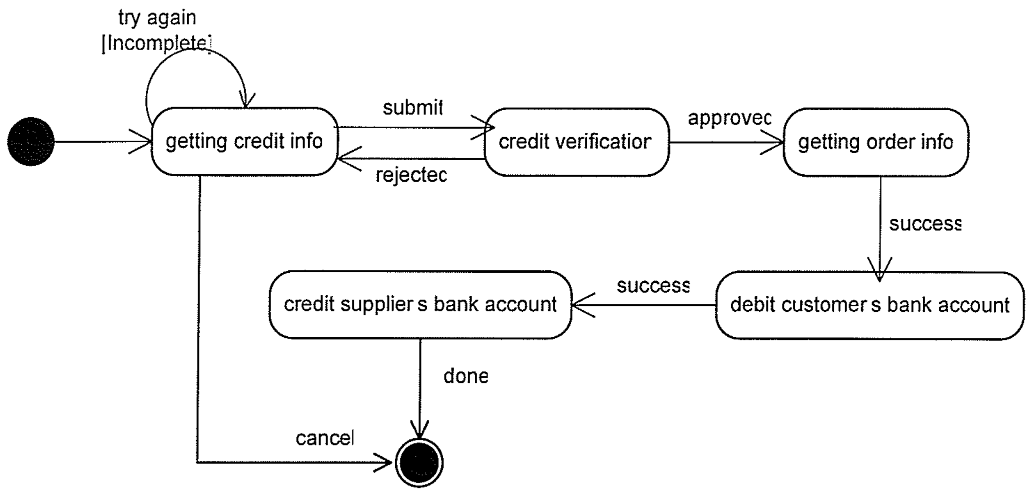


Figure 3.21 State Chart Diagram for Payment

3.6.3 Classes and Relationships among Classes

Based on the analysis of functional requirements of the proposed middleware framework and dynamic behaviors of objects, I identified the major classes and the relationships among classes shown in Figure 3.22.

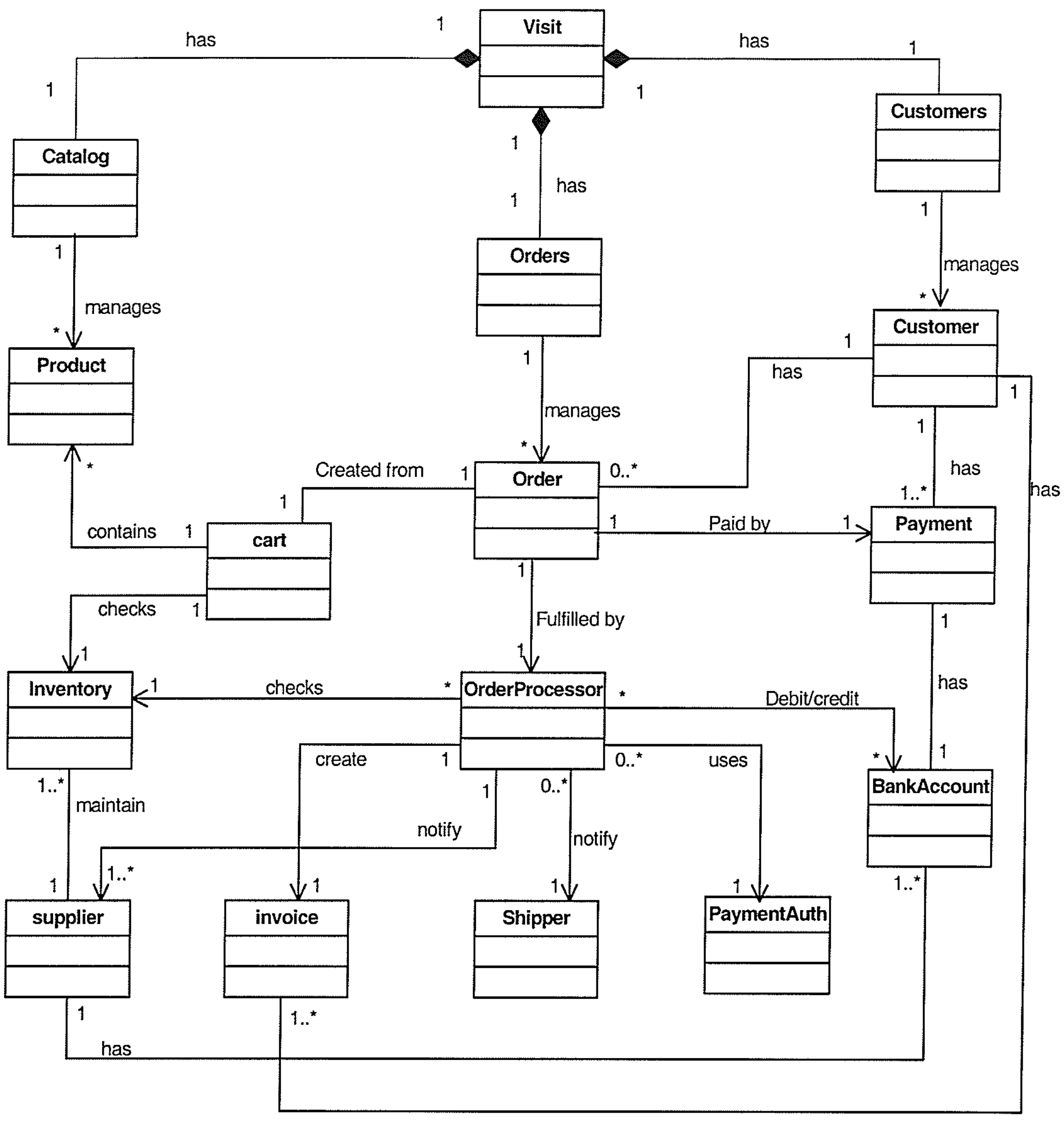


Figure 3.22 Class Diagram of E-Commerce System

1. Class Diagram for Supplier Component

The supplier component includes four main classes: catalog, product, supplier, and inventory. A system administrator may use this component to accomplish tasks, such as adding new departments to the catalog, updating product prices, and removing products from inventory, etc. A customer uses this component to browse the catalog, or retrieve a product's detailed information. A basic class database and a public interface are used in order to allow the four classes to share the same database connection object.

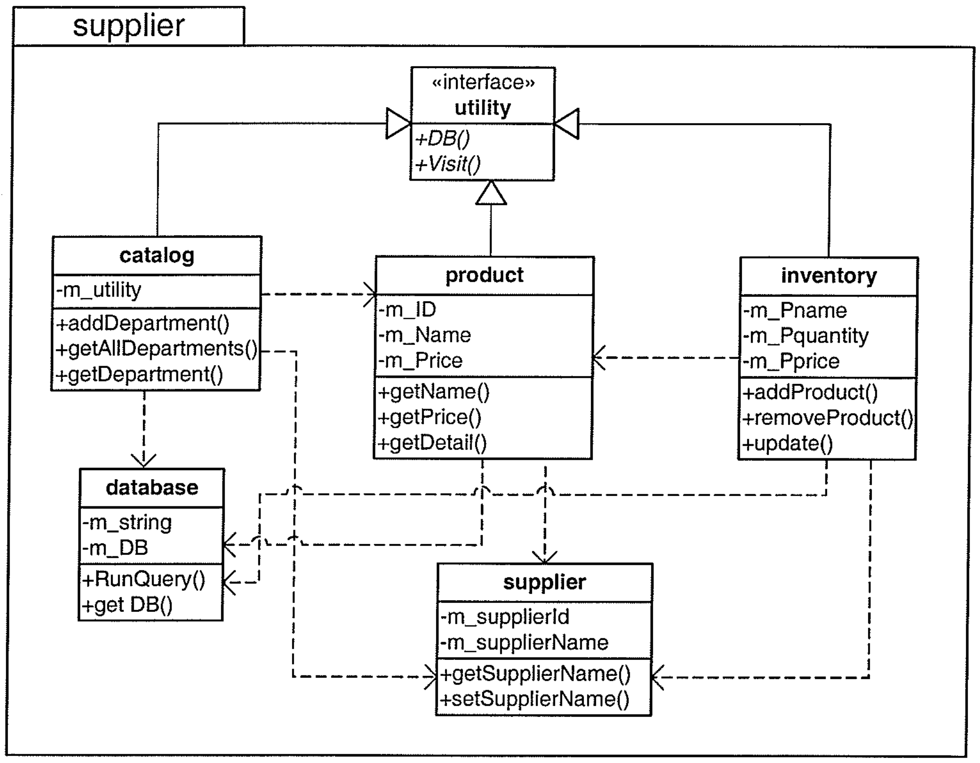


Figure 3.23 Component Diagram for Supplier

2. Class Diagram for Customer Component

The customer component includes two main classes: customers and customer. The customer class is used for holding a single customer's detailed information. The customers class is used for holding all operations that related to customer management, such as creating a new customer, updating an existing customer's information, etc.

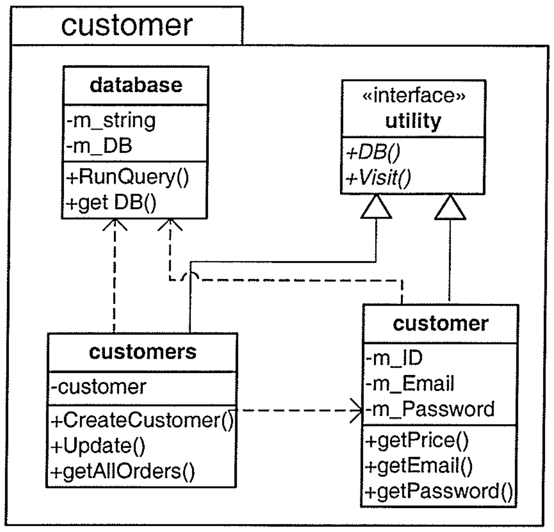


Figure 3.24 Component Diagram for Customer

3. Class Diagram for Shopping Cart Component

The shopping Cart component contains a cart class, which is responsible for adding products to shopping carts, increasing the quantity of a product, emptying shopping carts, etc.

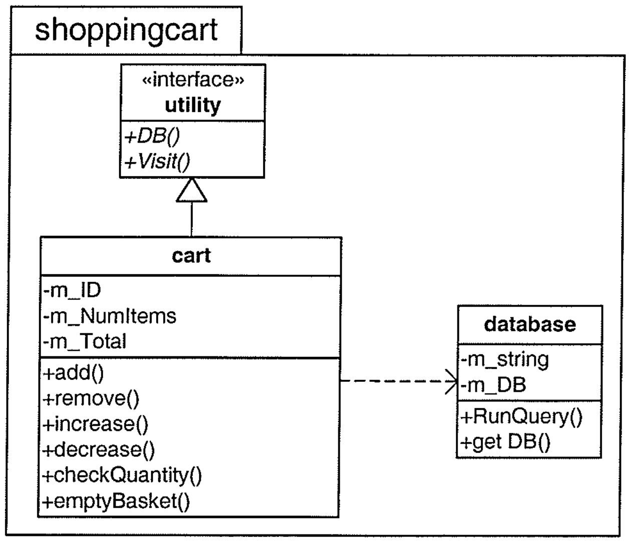


Figure 3.25 Component Diagram for Shopping Cart

4. Class Diagram for Payment Component

The payment component contains a payment class, which is used for managing customers' payment methods, including adding or removing a payment method, and modifying an existing payment method, etc.

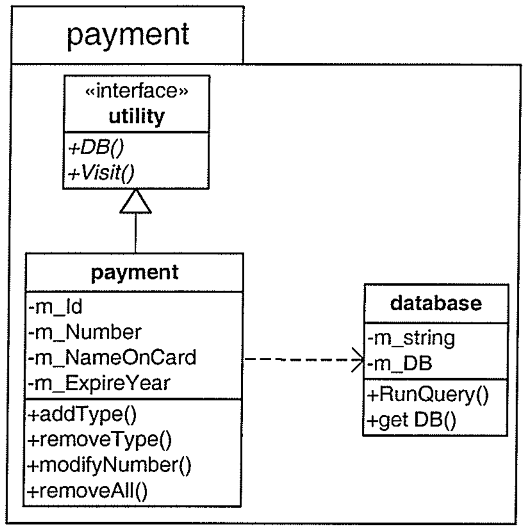


Figure 3.26 Component Diagram for Payment

4. **Class Diagram for Third Party Component**

The third party component includes a payment authentication class and a bank account class, which are responsible for authenticating customers' credit card information and transmitting money from customer accounts to supplier accounts.

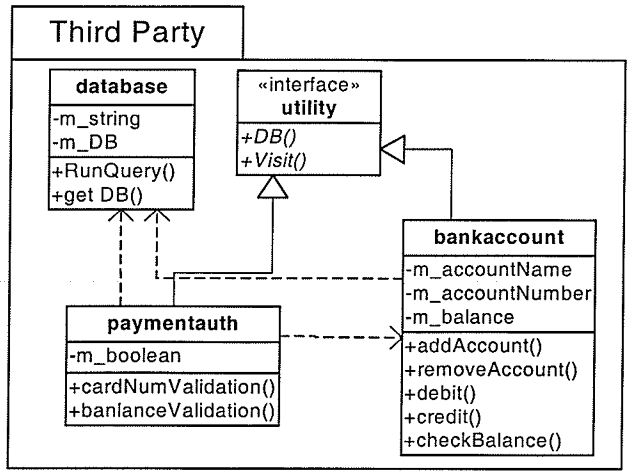


Figure 3.27 Component Diagram for Third Party

6. Class Diagram for Shipment Component

The shipment component contains a shipment class, which is used for managing the shipment methods that are provided by an e-commerce system. The component provides services like adding, modifying, and removing shipment methods.

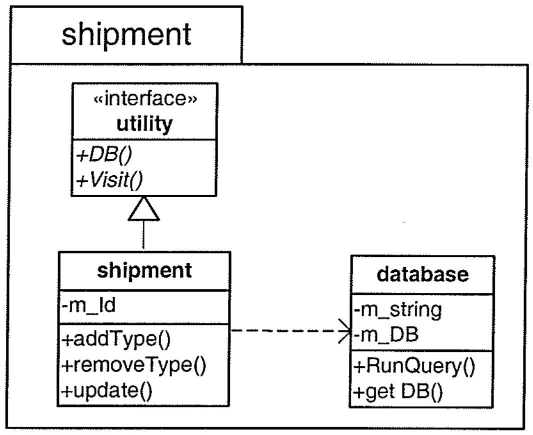


Figure 3.28 Component Diagram for Shipment

7. Class Diagram for Transaction Component

The transaction component contains classes that are related to e-commerce transactions. These classes include order, invoice, and order processing pipeline. This component is responsible for creating orders, creating invoices, and calculating order totals. Additionally, the component is also used for notifying customers, and notifying suppliers.

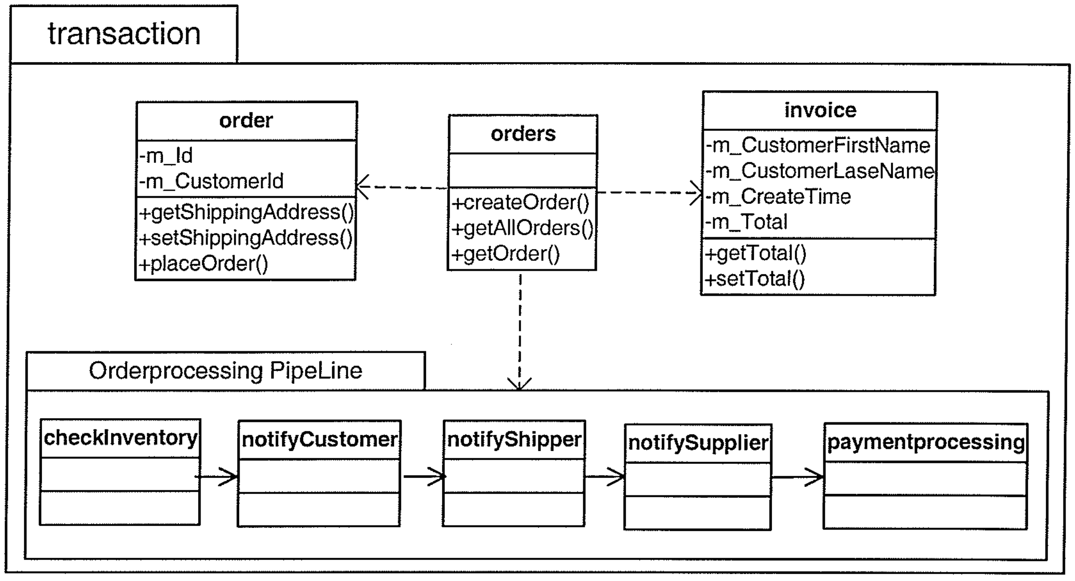


Figure 3.29 Component Diagram for Transaction

3.6.4 Detailed Design

The detailed design of each class is documented using pseudo code. The following information is used to describe the detailed design for each class:

Index: a reference uniquely assigned for identifying this class.

Name: a unique name that describes the corresponding class.

Purpose: briefly describes the objective of the class.

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

A method of a class is described in the follow style:

Index: a reference uniquely assigned for identifying this method.

Name: a unique name that describes the corresponding method.

Purpose: briefly describes the objective of the method.

Input parameters: a list of input parameters required by the method.

Output parameters: a list of output parameters that the method is expected to return.

Pseudo code: describes the operation of this method including how the functions utilize the data and how the program reacts to events and state changes.

Remarks: explains details concerning the method, which are not captured by any of the above.

Below is an example description for the customer class and its methods.

Customers Class:

Index:	Class_01
Name:	Customers
Purpose:	providing methods that are used for managing customer information, such as creating a new customer or updating an existing customer's account.
Reference classes:	customer
Remarks:	No

Methods of customers class:

Index:	Method_01
Name:	createCustomer
Purpose:	register a new customer
Visibility:	Public
Input parameters:	firstName, lastName, email, password, address, phone,City, Country, Zip
Output parameters:	customerId
Pseudo code:	

```
recordset = SELECT * from customer where EMAIL = email
if   end of recordset then
        add new customer;
        return customerId;
else
        return 0;
end if
```

Remarks:	firstName, lastName, password, address, phone are optional parameters

Index:	Method_02
Name:	updateCustomer

Purpose:	update an existing customer's personal information
Visibility:	Public
Input parameters:	Id, FirstName, LastName, EMail, Password, Address, Phone, City, Country, Zip
Output parameters:	Boolean (true or false)
Pseudo code:	

```
recordset = Select * from customer where ID = Id
if not end of recordset
        update record;
        return true;
else
        return false
end if
```

Remarks:	No

Index:	Method_03
Name:	GetOneCustomer
Purpose:	retrieve one customer's record value from the customer table, which is specified by customer ID
Visibility:	Public
Input parameters:	Id
Output parameters:	recordset
Pseudo code:	

```
recordset = Select * from customer where customerID = Id
if not end of table
        return recordset;
else return null;
```

Remarks:	No

Index:	Method_04
Name:	CheckLogon
Purpose:	attempts to logon a user based on an e-mail and password
Visibility:	Private
Input parameters:	email, password
Output parameters:	boolean
Pseudo code:	

```
recordset = select * from customer where EMAIL=email and
PASSWORD = password
if not end of table
        return true;
else
        return false;
```

Remarks: No

Fifteen classes are identified in this research. Other classes and their methods descriptions are provided in appendix A.

Chapter 4

Implementation

This chapter discusses issues related to the implementation of the middleware framework, such as implementation strategies, implementation environment, and quality assurance. An example application derived from the proposed middleware framework is also provided.

4.1 Implementation Strategies

In the implementation phase, the design documents that have been created must be translated into code. All the functionality specified in the design must be present and fully functional. To ensure functionality, I used a combined top-down and bottom-up approach.

In object-oriented development, a top-down implementation means that the classes where executions start are implemented first, while supporting classes and reference classes are implemented later. In this middleware framework project, all the proposed components work through a façade class, namely *visit.* Therefore, I started by building this class and presented all its referenced classes as dummy classes (i.e., a class that contains its necessary methods with incomplete method bodies). By doing so, whenever a component had been implemented (e.g., order, customer, and transaction), I could test it through the *visit* class. In order to test the correctness of a component, I also had to build the GUI first (i.e., web pages implemented using ASP), which provides a graphical user interface for collecting customer input.

The bottom-up implementation means that classes at the bottom of the class inheritance hierarchy are built first. In other words, classes that only allocate or reference primitive or predefined classes are coded first. For example, in this project, an interface named *utility* and a base class named *database* were coded and tested first since they only reference to primitive classes. Classes like *order, customer, product,* were built later because they used the *utility* and *database* class. Therefore, a combined method was used in the implementation of this middleware framework.

4.2 Quality Assurance

To ensure the framework's quality in term of functionality, reliability, availability, and performance, I used extensive testing methods, including unit testing, integration testing, and system testing, to improve the framework's quality.

Unit testing is testing the individual components that comprise the system. After a component had been built by following bottom-up implementation strategy, I proceeded to test functionality by generating a set of test cases that represent all possible situations. The purpose of functional testing is to ensure that the observed and expected behaviors are the same in all situations. However, unit testing of a component can only test part of the correctness of the component. It was uncertain whether the component would function correctly when interacting to other components. Therefore, I used integration testing to further test components' dynamic behaviors.

Integration testing is testing a group of components' behavior when they are working together. The goal of integration testing is to ensure that a group of components behaviors are the same as specified in the requirements. Once I was satisfied with each individual component of the framework, I integrated these components into a working system to perform integration testing. During the integration testing, I used an incremental approach. I added one component at a time to the working system instead of adding them together. When the first component added to the working system tested and demonstrated correct behavior, I added the second component. If a problem arose, the most likely source for the problem was the most recently added component. In most instances, bugs can be found relatively easily.

Finally, I performed the system testing, which meant that the derived middleware components and web pages that called these middleware components were deployed on a server computer. Clients browsed the web pages via the Internet. A set of test cases were performed to verify the functions provided by middleware components. In some cases, the system must be fault tolerant, which means when an error occurs the overall system must be able to recover. In system testing, middleware components' fault tolerances were tested. In addition, the performance of the proposed middleware was also tested, using metrics such as the response time of a web page and the number of transactions performed per second.

4.3 Implementation Environment

The proposed middleware framework is implemented using Microsoft COM+ technology. The reasons for using COM+ are that it achieves binary encapsulation and binary compatibility and it supports a wide range of implementation languages.

In addition, it is faster and more robust than either EJB or CORBA technologies [10]. The developmental environment I used is Microsoft Visual Studio and the programming language is Visual Basic.

To verify the design and the implementation of the framework, an example e-commerce application is created from the proposed middleware framework. ASP technology is used for implementing the web server layer (i.e., web pages). The ASP web pages are deployed in Microsoft IIS web server container. Microsoft's SQL server 2000 is used as the backend database and Microsoft Windows XP is used as the operating system.

4.4 Run-Time Behavior of an Example Application

A customer can logon to the sample e-commerce application by typing the URL of the system (i.e., http://142.161.74.219/shoppingcenter/asp/frame.asp). The system then displays its entry page, as shown in Figure 4-1. The left side of the entry page shows a search engine, the contents of the current shopping cart, and the department list of the system. The search engine provides a quick method for searching for a particular product. The search key words could include name, price, or supplier of a product. The shopping cart content area shows the total number of items and the total price of all items in the current shopping cart. The department list displays all the department of this e-commerce system. Users can browse any of the department by clicking the corresponding hyper link.

The main frame of the entry page shows all the featured products for each department. A customer can click the hyperlink of the department name or click on the image of the featured product to explore detailed information regarding a particular department. A customer can also choose to immediately login the system, instead of doing so at check out. An existing customer can also track his/her shopping records and check his/her contact information by clicking the respective hyperlink.

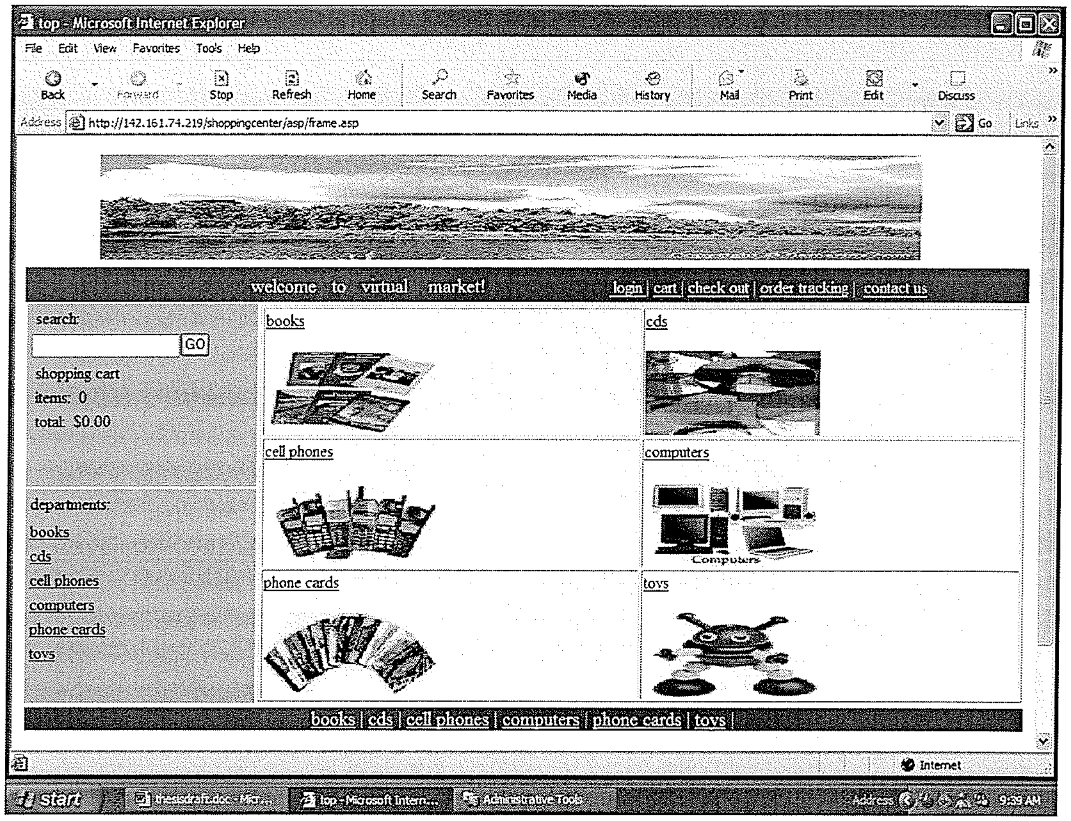
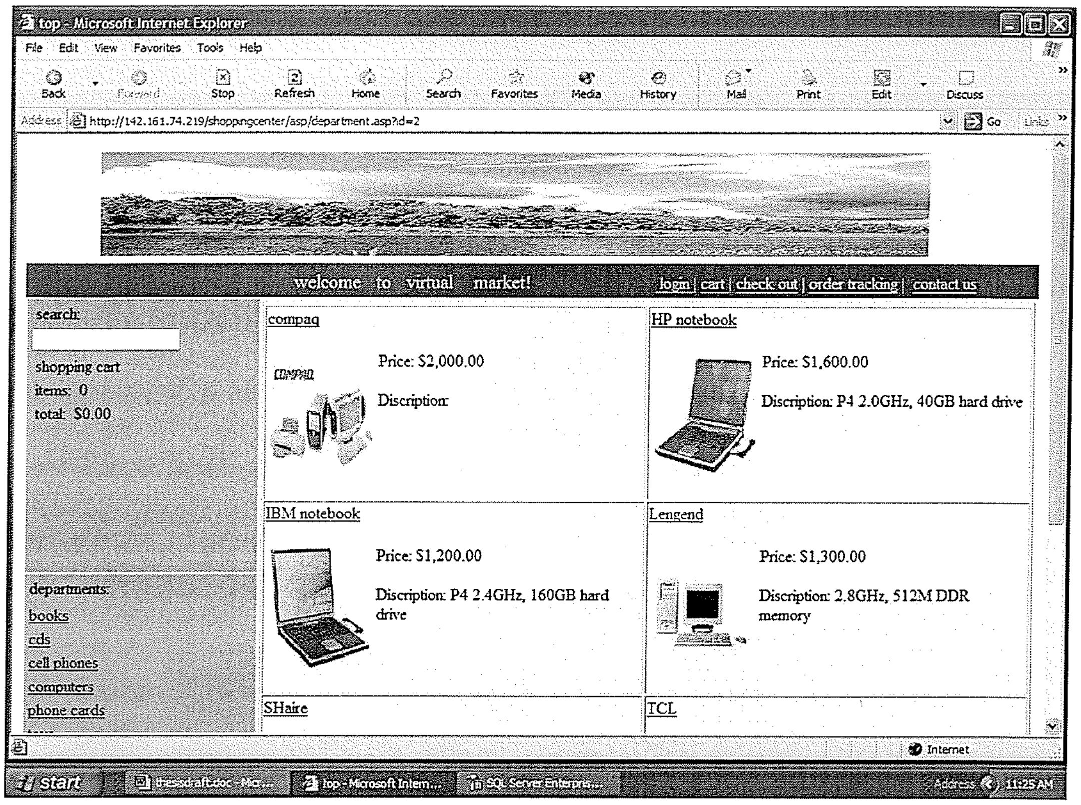

top - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History Mail Print Edit Discuss
Address http://142.161.74.219/shoppingcenter/asp/frame.asp
welcome to virtual market!
login | cart | check out | order tracking | contact us
search:
GO
shopping cart
items: 0
total: $0.00
departments:
books
cds
cell phones
computers
phone cards
toys
books
cds
cell phones
computers
phone cards
toys
books | cds | cell phones | computers | phone cards | toys |
Internet
start

Figure 4-1 Entry page of an E-Commerce System


top - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History Mail Print Edit Discuss
Address http://142.161.74.219/shoppingcenter/asp/department.asp?id=2
Go Links
Internet
start
thesisdraft.doc - Micr...
top - Microsoft Intern...
SQL Server Enterpris...
11:25 AM

welcome to virtual market!

login | cart | check out | order tracking | contact us

search:

shopping cart
items: 0
total: $0.00

departments:
books
cds
cell phones
computers
phone cards

compaq
Price: $2,000.00
Discription:

HP notebook
Price: $1,600.00
Discription: P4 2.0GHz, 40GB hard drive

IBM notebook
Price: $1,200.00
Discription: P4 2.4GHz, 160GB hard drive

Lengend
Price: $1,300.00
Discription: 2.8GHz, 512M DDR memory

SHaire

TCL

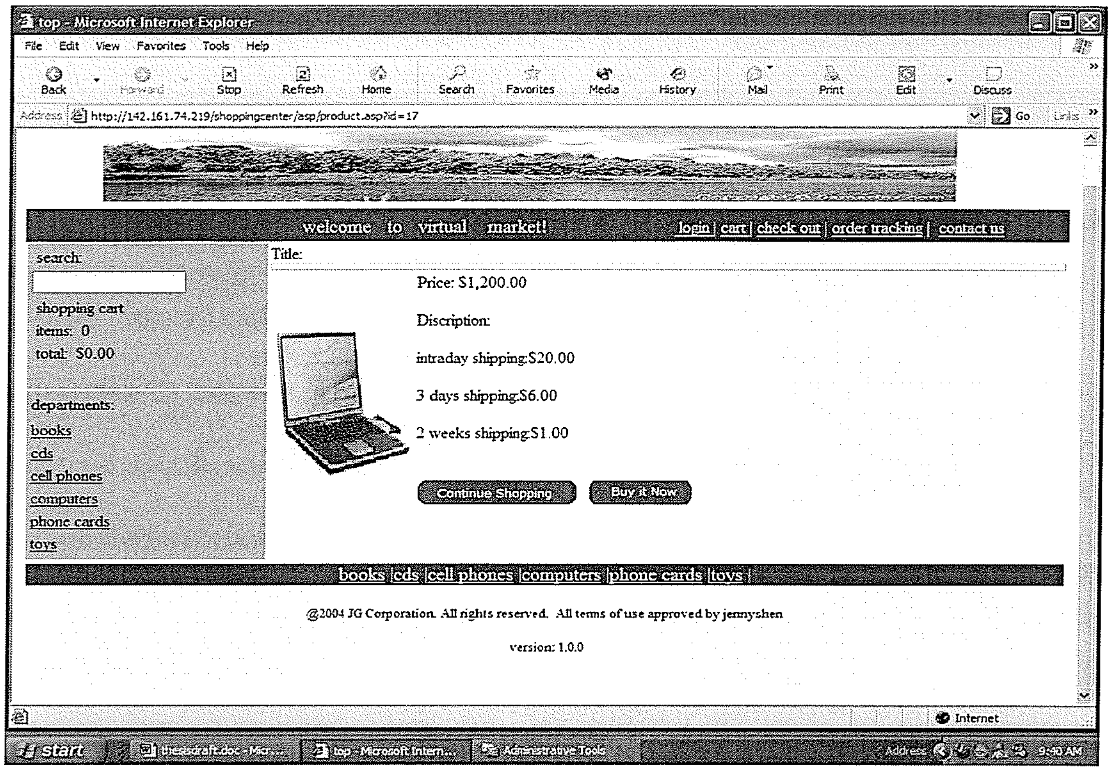
Figure 4- 2 Department page of an E-commerce System

If a customer is interested in one department, he/she can click the name of the department or the image of the department to browse all the corresponding products. Figure 4-2 presents the product information of a computer department. If a customer is interested in one particular product, he/she can click the name of that product or the image to see the detailed information, as shown in Figure 4-3.

Figure 4-3 shows one particular product's image, name, price, description, and shipping methods. From this page, a customer can click the "buy it now" button to drop this product into his/her shopping cart, or click the "continue shopping" button to browse other products.

Figure 4-4 shows the contents of a customer's shopping cart including all product names, product prices, quantity, and their value. From this page, a customer can increase or decrease the quantity of a product, or delete items from the shopping cart by clicking the respective hyper links. In addition, a customer can click on the "continue shopping" button to keep shopping or click on the "proceed to check out" button to check out.

In the check out process, a customer first needs to login the e-commerce system. From Figure 4-5 we can see that if a customer is new to the system, he/she is required to input an email address and check the "I have never shopped at your store before" radio button. If a customer is a returned customer, he/she is required to input an email address and password. For a new customer, the e-commerce system will proceed to ask for the user's name, address, country, telephone, and password, as shown in Figure 4 – 6. When all of the required information is entered correctly, the system will ask for credit card information. As show in Figure 4-7, a customer can choose a credit card type (e.g. Visa Card, Master Card) and input the name on the card, the card number, and expiration data. When all the personal information is entered correctly, the credit card information will be sent to a third party for validation. Once the credit card has been verified, a valid order will be created. Otherwise, the customer will be asked to input corrected credit card information.


top - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History Mail Print Edit Discuss
Address http://142.161.74.219/shoppingcenter/asp/product.asp?id=17
Go Links
Continue Shopping
Buy it Now
Internet
start
thesisdraft.doc - Mic...
top - Microsoft Intern...
Administrative Tools
9:40 AM

welcome to virtual market!   login | cart | check out | order tracking | contact us

search:

shopping cart
items: 0
total: $0.00

departments:
books
cds
cell phones
computers
phone cards
toys

Title:

Price: $1,200.00

Discription:

intraday shipping:$20.00

3 days shipping:$6.00

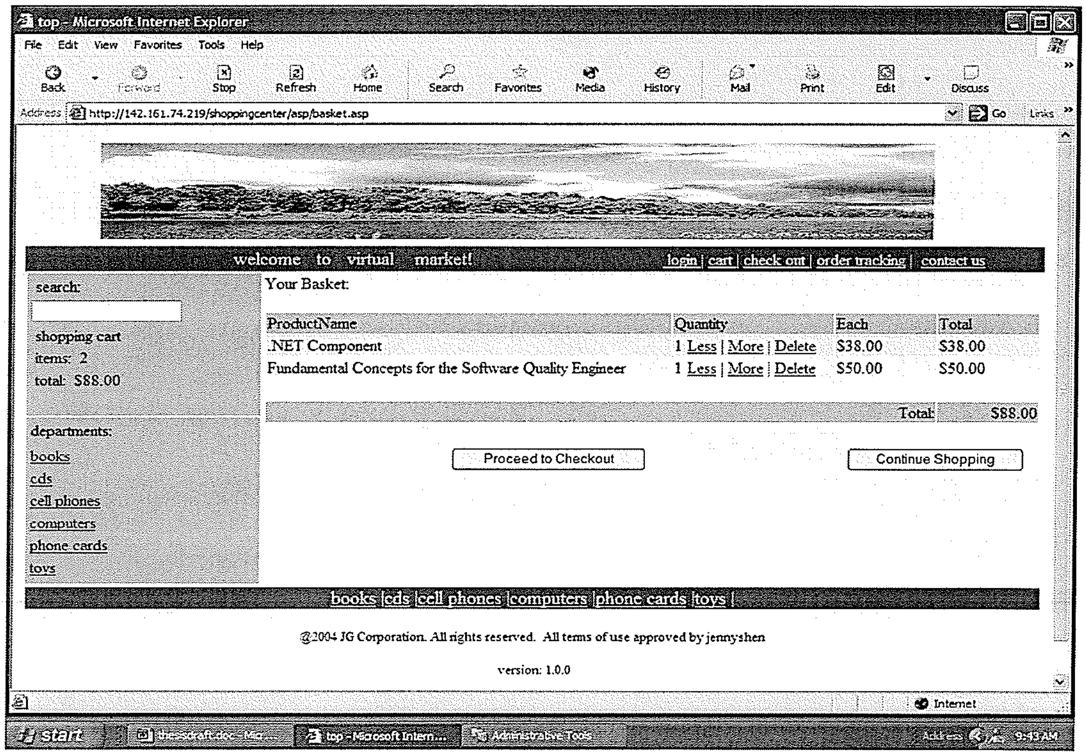
2 weeks shipping:$1.00

books |cds |cell phones |computers |phone cards |toys |

@2004 JG Corporation. All rights reserved. All terms of use approved by jennyshen

version: 1.0.0

Figure 4-3 Product Page of an E-Commerce System


top - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History Mail Print Edit Discuss
Address http://142.161.74.219/shoppingcenter/asp/basket.asp
Go Links
Proceed to Checkout
Continue Shopping
Internet
start
thesisdraft.doc - Mic...
top - Microsoft Intern...
Administrative Tools
9:43 AM

welcome to virtual market!   login | cart | check out | order tracking | contact us

search:

shopping cart
items: 2
total: $88.00

departments:
books
cds
cell phones
computers
phone cards
toys

Your Basket:

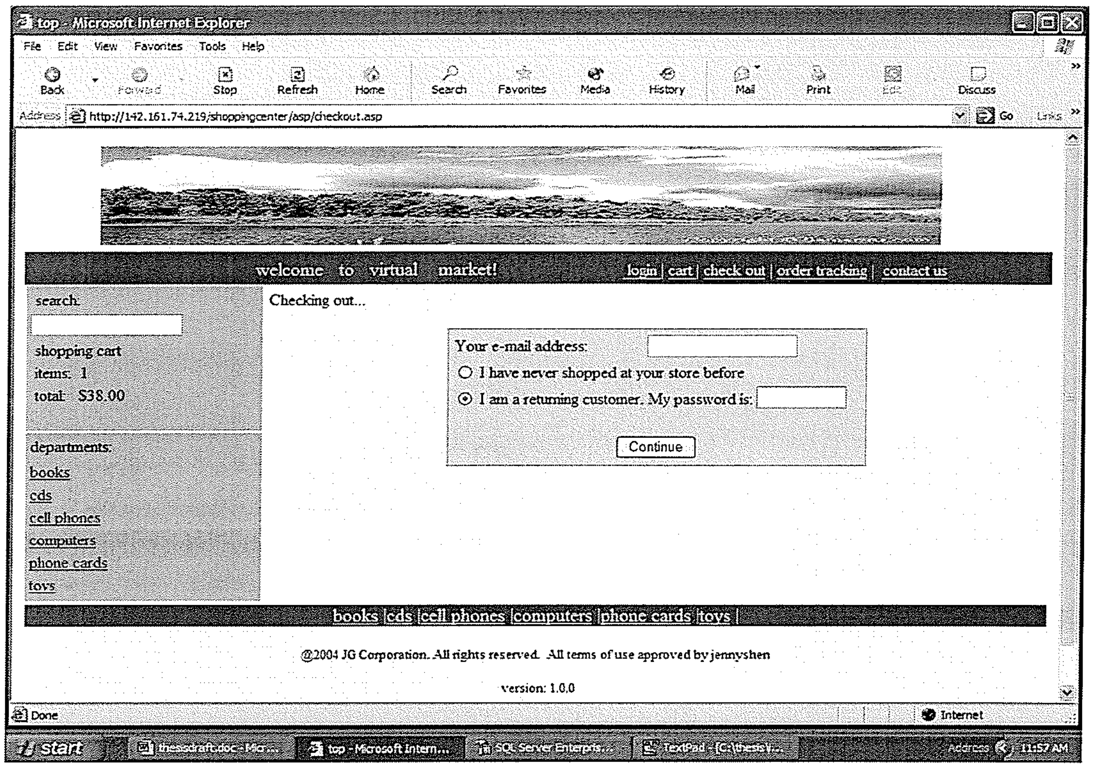
ProductName	Quantity	Each	Total
.NET Component	1 Less | More | Delete	$38.00	$38.00
Fundamental Concepts for the Software Quality Engineer	1 Less | More | Delete	$50.00	$50.00
		Total:	$88.00

books |cds |cell phones |computers |phone cards |toys |

@2004 JG Corporation. All rights reserved. All terms of use approved by jennyshen

version: 1.0.0

Figure 4-4 Shopping Cart Page of an E-Commerce System



http://142.161.74.219/shoppingcenter/asp/checkout.asp

welcome to virtual market! login | cart | check out | order tracking | contact us

search:

shopping cart
items: 1
total: $38.00

departments:
books
cds
cell phones
computers
phone cards
toys

Checking out...

Your e-mail address:
○ I have never shopped at your store before
⊙ I am a returning customer. My password is:

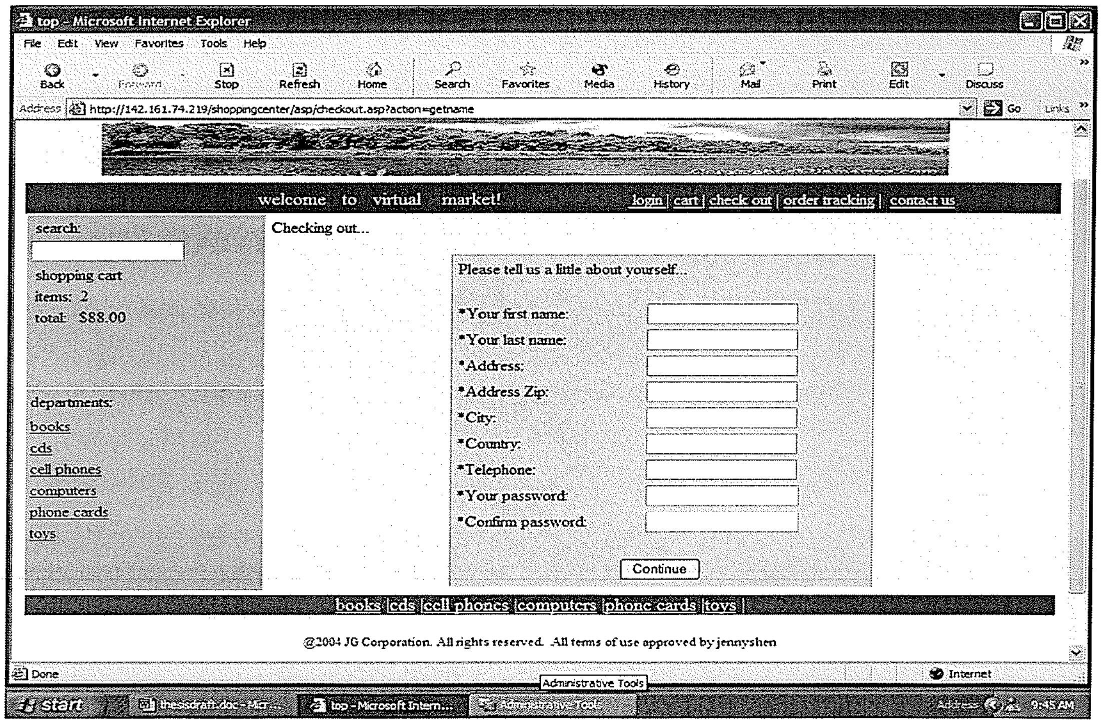
Continue

books |cds |cell phones |computers |phone cards |toys |

@2004 JG Corporation. All rights reserved. All terms of use approved by jennyshen

version: 1.0.0

Figure 4-5 Email Address Collection Page of an E-Commerce System



http://142.161.74.219/shoppingcenter/asp/checkout.asp?action=getname

welcome to virtual market! login | cart | check out | order tracking | contact us

search:

shopping cart
items: 2
total: $88.00

departments:
books
cds
cell phones
computers
phone cards
toys

Checking out...

Please tell us a little about yourself...

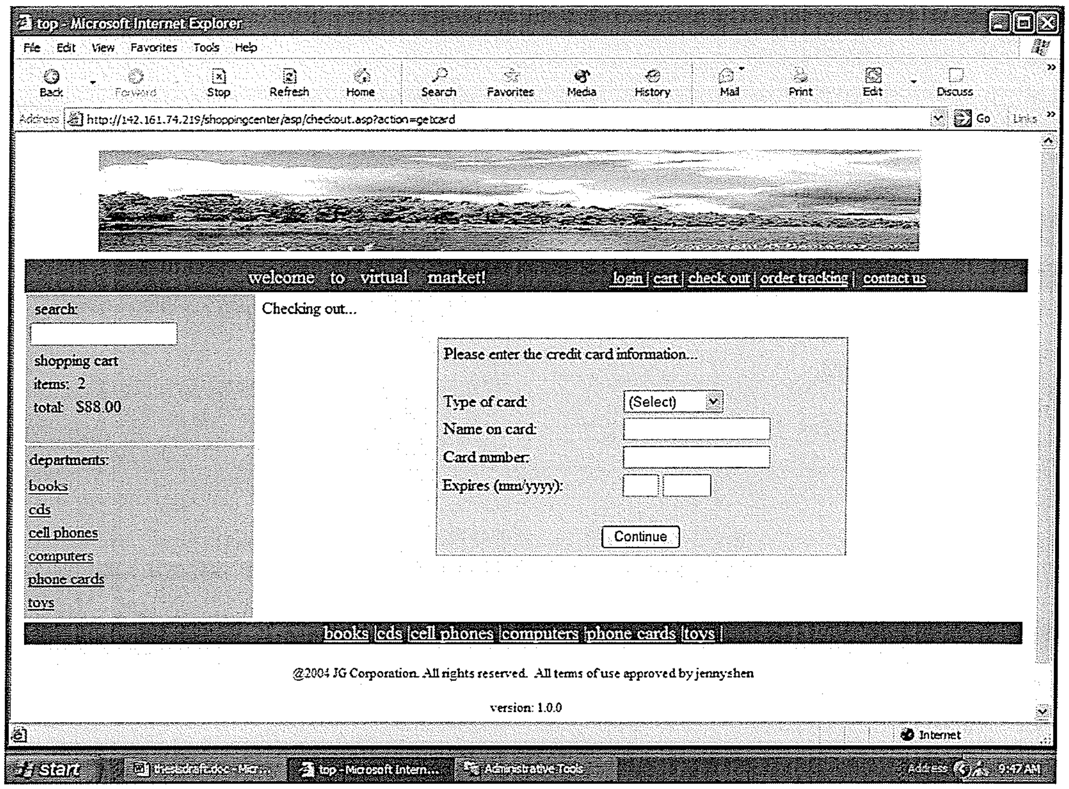
*Your first name:
*Your last name:
*Address:
*Address Zip:
*City:
*Country:
*Telephone:
*Your password:
*Confirm password:

Continue

books |cds |cell phones |computers |phone cards |toys |

@2004 JG Corporation. All rights reserved. All terms of use approved by jennyshen

Figure 4-6 Personal Information Collection Page of an E-Commerce System


top - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History Mail Print Edit Discuss
welcome to virtual market!
login | cart | check out | order tracking | contact us
search:
departments:
books
cds
cell phones
computers
phone cards
toys
books | cds | cell phones | computers | phone cards | toys |
Continue
Internet

Address http://142.161.74.219/shoppingcenter/asp/checkout.asp?action=getcard

shopping cart
items: 2
total: $88.00

Checking out...

Please enter the credit card information...

Type of card:	(Select)
Name on card:	
Card number:	
Expires (mm/yyyy):	

@2004 JG Corporation. All rights reserved. All terms of use approved by jennyshen

version: 1.0.0

Figure 4-7 Credit Card Information Collection Page of an E-Commerce System

4.5 Evaluation

The quality of the proposed middleware framework can be evaluated from both qualitative and quantitative aspects. In the qualitative analysis, I examined the framework's functionality, reusability, technology dependency, and security. In the quantitative analysis, I measured the example application's average response time, transaction success rate, and the system's capacity.

4.5.1 Qualitative Evaluation

- **Functionality**

The proposed middleware framework supports the common functions required by online commerce applications. The functions include customer account creation, a shopping cart, search engine, various payment and shipment methods, product browsing, checkout, and order tracking, etc.

The framework supports a product browsing function. Any user of an e-commerce web site is able to navigate the web site just like walking in a real shop. Whenever users intend to purchase any products from the web site, they provide their personal information to become registered customers (creating an account) or provide their ID and password to be recognized by the system (login).

Furthermore, the framework supports a shopping cart function, which can be used by end users add products in their shopping cart, and to view the contents of a cart during the purchasing process. In addition, this framework provides a search engine function, which reduces product search times. Since numerous methods are provided in the product class, a search engine can be easily built to accept keywords such as product name, price, and supplier.

Another basic function supported by the framework is credit card payment. Customers provide their credit card information to the e-commerce system. Then, the e-commerce system will encrypt the information and send it through the Secure Socket Layer (SSL) to third parties, where the credit card information is verified. If the credit card information passes verification, an order will be created. Otherwise, the order will be rejected. After the order is created, it will go into a processing procedure, which includes checking the stock, making a payment, and notifying customers, suppliers, and shippers.

The framework also supports different shipment methods, which can be chosen by customers, and helps to ensure user satisfaction. Registered customers can also track their shopping records. However, some of the functions that are not common are omitted from this framework.

- **Reusability**

The main benefits of the proposed framework are that both its design and implementation are reusable. During the design stage, I used a systematic method to identify the components. When I made the decision on each component's granularity, I considered the component's cohesion and coupling so that reusability and maintainability could be confirmed.

- **Technology Dependency**

This framework describes the design of middleware components that enable B2C e-commerce transactions. The design is presented using UML diagrams and pseudo code, which is independent from any implementation technology. Therefore, the proposed middleware framework may be implemented using different middleware platforms, namely CORBA, COM+, and EJB. Particularly when COM+ technology is used, a wide range of programming languages (e.g., Visual Basic, C++, and C#) can be used to implement the framework. The implementation results (i.e., a set of components) could be deployed in either a Unix or Windows environment depending on the chosen implementation middleware platform.

- **Security**

Security is a key concern in any e-commerce system. Protection includes customers' personal, purchasing, and credit card information. This framework provides multiple access control mechanisms to reduce both customer and merchant risks. For example, customers who wish to purchase products from an e-commerce system need to provide unique email address and password to register. Only registered customers can place orders and check their shopping history. Only authorized administrators can modify catalog, product, and inventory information, and only authorized persons can view customer information. However, information transport security and operating system security are not main concerns of this work.

4.5.2 Quantitative Evaluation

To evaluate the proposed middleware framework's performance and usability, I deployed the example application on the server machine in the E-Commerce Laboratory (i.e., Dr. Ehikioya's Advanced E-Commerce Systems Development Laboratory), measured the example applications' average web page response time, the page loading success rate, and compared the results with an existing two-tier architecture e-commerce application (i.e. a course project developed by a graduate student) installed on the same server machine. In addition, I measured my example application's transaction accuracy rate, and the deployed system's capacity.

The server machine has a Pentium III 1GHz CPU, 256M RAM, Windows 2000 Advanced Server operating system, and Microsoft SQL server 2000 database. I used the computers in the Cargill Lab and the Heterogeneous Computing Laboratory as client machines to logon to both of the two tier and the three tier e-commerce applications through the local area network.

I used a Java program named JMETER [32] to simulate multiple simultaneous users to logon to the two e-commerce systems. JMETER is open-source software used to test the performance of server systems such as web, FTP, and databases servers.

- **Average Response Time**

A web page's response time includes the time for the server to handle request, the time for network communication, and the time for processing on the client machine (e.g., formatting the response). In this research, I captured sample web page response times and calculated the average response times of both the two-tier architecture e-commerce system and the three tier architecture e-commerce system as shown in table 4-1. From Figure 4-9, we can see that with an increase in the number of users, both the two-tier and the three-tier e-commerce system's average response time increases and the three-tier system's average response time increases more sharply than the two-tier system. When the number of users reaches 40 for the two-tier system and 20 for the three-tier system, the average response time will drop. However, at these points in time, both systems' success page loading rate drops too, as depicted in both Table 4-2 and Figure 4-10

Number of Users	Two Tier	Three Tier
1	12	378
2	51	726
4	146	1355
6	168	1914
8	260	3091
10	363	5163
20	751	7127
40	1036	4008
80	446	2374
160	295	1130

Table 4-1 Average Response Time for the Two Tier and Three Tier E-Commerce Application (in ms)

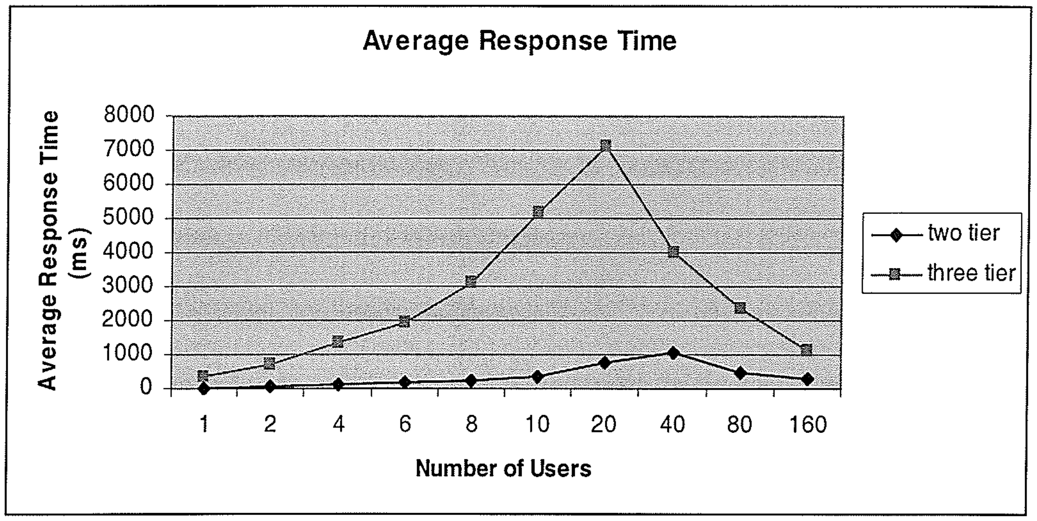


Figure 4-8. Average Response Time for the Two Tier and Three Tier E-Commerce Application

Number of Users	Two Tier	Three Tier
1	100%	100%
2	100%	100%
4	100%	100%
6	100%	100%
8	100%	100%
10	100%	100%
20	100%	100%
40	100%	38%
80	46.20%	12.25%
160	43.70%	3.10%

Table 4-2 Success Page Loading Rate for the Two Tier and Three Tier E-Commerce Application

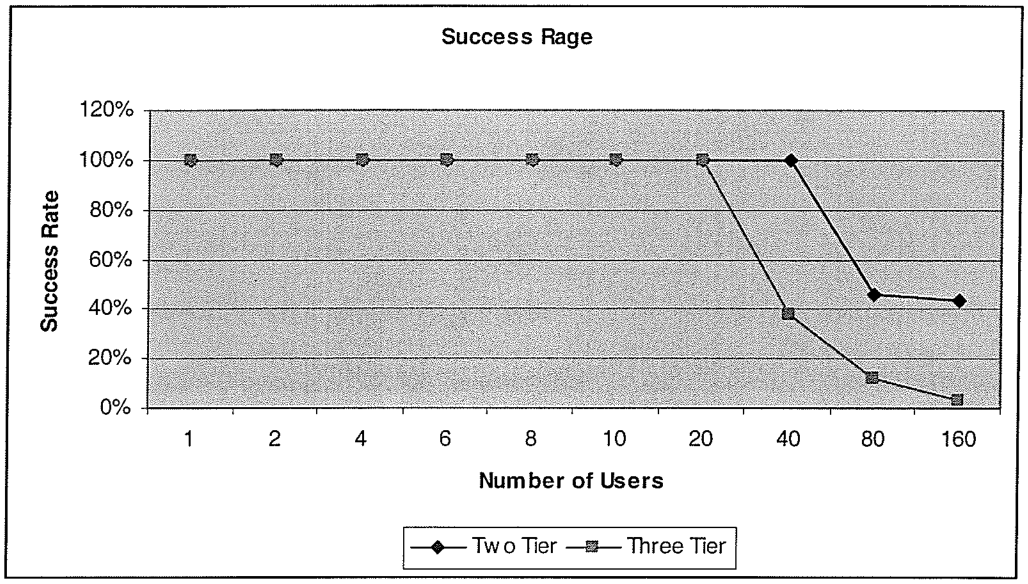


Figure 4-9 Success Page Loading Rate

- **Capacity**

To determine the e-commerce system's capacity, I measured the maximum number of users the system supported. JMETER was used to simulate multiple users and each user performed a login operation. I increased the number of users until the server machine crashed. The maximum number recorded was 200, which means one server machine can support 200 users simultaneously.

- **Integrity**

To test the sample application's integrity, I used three groups of participants' help to do the experiment. The participants were friends and students in Computer Science Department at the University of Manitoba. The first group had one participant, the second group had five participants and the third group had ten participants. People in each group were asked to logon to the three tier e-commerce system simultaneously and perform various purchasing transactions. The percentage of successful transactions and the number of correctly accessed web pages was 100%. The system availability was 98%.

- **Impact of Component Size**

In this study, I was also interested in the impact of component size on the example application's performance. Therefore, I regrouped the components into one singular component, and rebuilt the example application based on the unified component. Then I compared the average response time and success page loading rate with the example application based on ten components.

Table 4-3 lists the average response time and success page loading rate of example application using one component. And Table 4-4 lists the similar data for example application using ten components. Both these two example applications use three-tier architecture and deployed on the same server machine.

Number of Users	Average Response Time(ms)	Success Rate	Component Number	Architecture
1	431	100%	1	three tier
2	892	100%	1	three tier
4	1060	100%	1	three tier
6	1652	100%	1	three tier
8	3387	100%	1	three tier
10	4033	100%	1	three tier
20	4607	50%	1	three tier
40	2516	25%	1	three tier
80	1458	11.25%	1	three tier
160	848	6.50%	1	three tier

Table 4-3 Example Application Based on One Component

Number of users	Average Response Time(ms)	Success Rate	Component Number	Architecture
1	378	100%	10	three tier
2	726	100%	10	three tier
4	1355	100%	10	three tier
6	1914	100%	10	three tier
8	3091	100%	10	three tier
10	5163	100%	10	three tier
20	7127	100%	10	three tier
40	4008	38%	10	three tier
80	2374	12.25%	10	three tier
160	1130	3.10%	10	three tier

Table 4-4 Example Application Based on Ten Components

Figure 4-11 shows that the example application that uses one component follows almost the same diagram shape as the example application that uses ten components. However, the single component has slightly better average response time than the application using ten smaller components. Nevertheless, the framework with ten components is easier to understand, to reuse, and to extend than the singular component framework because the users can pick any of the components, digest and use them in their applications.

Similarly, both example applications' successful page loading rate drops when the number of users reaches 20 for the application based on one component and 40 for the application based on ten components, as shown in Figure 4-12.

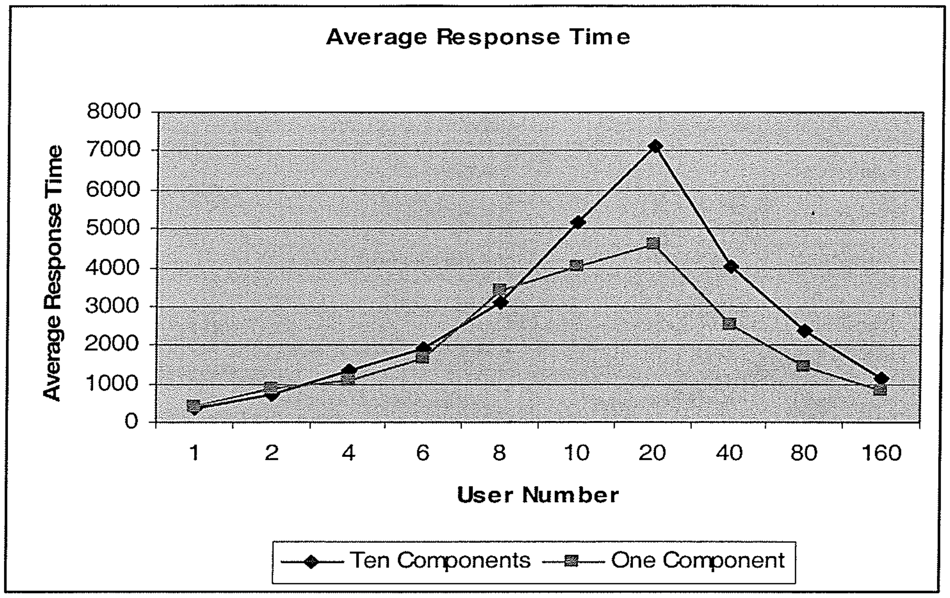


Figure 4-10 Average Response Time for Example Application Based on One Component and Ten Components

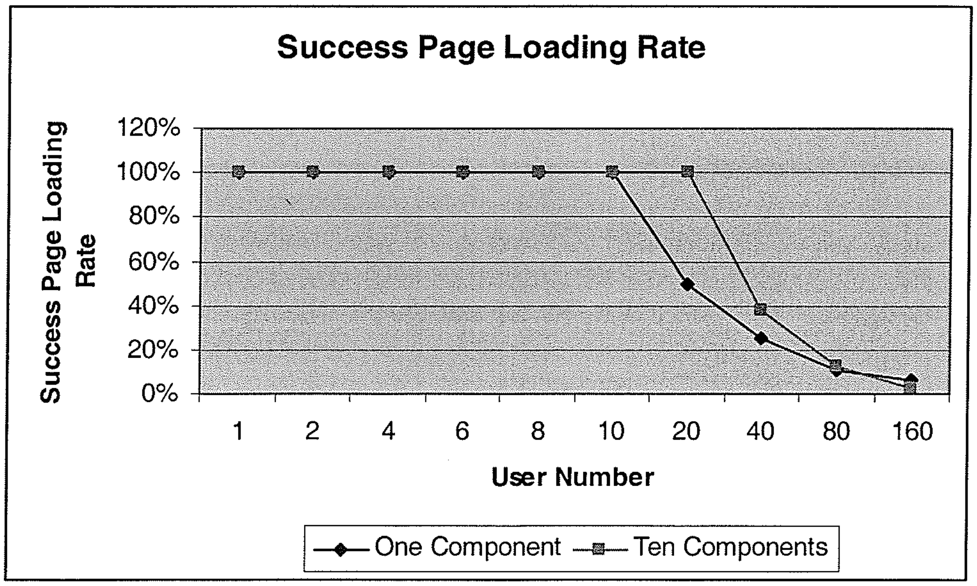


Figure 4-11 Success Page Loading Rate for Example Application Based on One Component and Ten Components

Chapter 5

Conclusion

5.1 Summary of Contributions

In this research, I provided an object-based middleware framework, which identified all the common services necessary for e-commerce transactions, such as customer information, shopping cart and order management. This framework is represented by a set of components and provides its services through these components' interfaces. Using the proposed middleware framework to build online e-commerce applications can dramatically improve developer's productivity, as well as enhance the quality and reliability of e-commerce applications.

This research involves the design of common services that are required by similar online store applications and the implementation of a subset of the design to validate the design. The design of the framework is independent of any underlying platform and programming language, therefore it is much easier to reuse. Using the common services provided by the framework to build online e-commerce application is cheaper and faster than building from scratch.

The UML design methodology was used widely in the requirement analysis and design. Specifically, UML use case diagrams were used to capture the system's functional requirements, UML collaboration diagrams were used to describe the use case scenario, and UML state chart diagrams were used to show the dynamic behavior of objects. Pseudo code was used to document the detailed design (i.e., methods provided by each object).

To confirm the correctness and the performance of the proposed middleware framework, design issues including component granularity, concurrency, communication, and consistency were discussed in chapter 3 and were examined at both the design and implementation stage. Therefore, the performance, reusability, and maintainability of the framework could be predicted.

A systematic method was used to identify the components from UML use case, collaboration and class diagrams. Component reusability, maintainability, and performance could be predicted since the components are derived through conforming the high cohesion and low coupling principle.

The proposed middleware framework was implemented using Microsoft COM+ technology, because it is faster and more stable than other middleware platforms and it also supports a wide range of implementation languages.

To verify the quality of the framework, I have also developed an example application using the proposed middleware framework. Comparisons were made with a non-middleware based e-commerce system. The middleware based e-commerce system was found to be more flexible, scalable, and reliable.

5.2 Future Work

Components provided in this middleware framework are mainly focused on supporting B2C e-commerce transactions. However, this work could be further extended to support complete system administration, like customer, order, and inventory administration. In addition, the framework could also be extended to support order tracking and data mining so that the derived e-commerce system could help merchants to make correct decisions and receive additional benefits.

Security is a very important issue in the B2C e-commerce domain. Both customers and merchants face many risks, like private information loss and credit card fraud. To achieve the highest degree of security, many new technologies have been used throughout e-commerce the applications' multiple tier such as Client (Web Browser), Web Server, and Application Server. To develop a security framework and incorporate it with the proposed middleware framework presents a beneficial research area.

Currently, I have evaluated the proposed middleware framework by testing an example application and comparing it with a non-middleware based e-commerce application. In the future, I hope to extend this work by comparing it with other B2C e-commerce middleware frameworks. A comparison with other B2C commerce middleware frameworks will identify the advantages and disadvantages of each framework and will contribute to the standardization of a middle framework for the B2C commerce domain.

Today's e-commerce environment consists of many components and faces numerous challenges. A middleware framework must be flexible to accommodate these challenges. Some new components may need to be added to the framework. These new components must be easy to create and must interoperate with the existing ones. Therefore, improving the framework's interoperability is an important research area.

I believe that the proposed middleware framework allows for the fast development of new robust B2C e-commerce applications in a simple way.

References:

[1] Markus Aleksy, Axel Korthaus, and Martin Schader. A CORBA-based architecture for electronic auction applications. In *Proceeding of the First ACIS Annual International Conference on Computer and Information Science*, pages 186–194, Orlando, Florida, 2001.

[2] Luis Anido, Manuel Caeiro, Juan M. Santos, and Judith Rodriguez. Design of a component-based software architecture for web-based learning system: EJB vs COBRA. In *Proceedings of International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications,* pages 277–282, Brazil, 2002.

[3] Martin Bichler, Carrie Beam, and Arie Segev. O_er: A broker-centered object framework for electronic requisitioning. *Trends in Distributed Systems for Electronic Commerce,* pages 154–165, 1998.

[4] Grady Booch, James Rumbaugh, Ivar Jacobson, and Jim Rumbaugh. *The Unified Modeling Language: User Guide*. Addison-Wesley, 1999.

[5] MiSook Choi, HyonHee Koh, YongIk Yoon, and JaiNyun Park. Component identification method based on use case. In *Proceeding of the First ACIS Annual International Conference on Computer and Information Science*, pages 203–210, Orlando, Florida, 2001.

[6] Wolfgang Emmerich. *Engineering Distributed Objects*. John Wiley and Sons, Ltd., 2000.

[7] Richard E. Fairley. Software Engineering Concepts. McGraw-Hill, Inc., 1985.

[8] Nathalie Gaertner and Bernard Thirion. Working with business patterns frameworks: A case study for fuzzy logic control. In *Proceedings of the ECOOP'99 Workshop for PhD Students in OO Systems (PhDOOS '99)*, pages 128–135.

[9] Jong-Pyo Jang, Sang-Jun Lee, and Byung-Ki Kim. Component specification activities using Z. In *Proceeding of the First ACIS Annual International Conference on Computer and Information Science,* pages 256–262, Orlando,

Florida, 2001.

[10] Markku Karppinen. *Enterprise Java Beans.* O'Reilly Associate Inc., 2000.

[11] Dong Kwan Kim, Hyo Taeg Jung, and Chae Kyu Kim. Techniques for systematically generating framework diagram based on UML. In *Proceedings of Asia-Pacific Software Engineering Conference,* pages 203–210, Taipei, Taiwan, December 1998.

[12] Gil-Jo Kim, In-Geol Chun, Ja-Kying Koo, Jin-Ho Jang, and Roger Y. Lee. A framework for software component quality improvement. In *Proceeding of the First ACIS Annual International Conference on Computer and Information Science*, pages 195–202, Orlando, Florida, 2001.

[13] Zeynep Kiziltan, Torsten Jonsson, and Brahim Hnich. On the Definition of Concepts in Component Based Software Development. *Technical Report*, Department of Information Science, Uppsala University, 2000.

[14] Chris Loosley and Frank Douglas. *High-Performance Client/Server: A Guild to Building and Managing Robust Distributed System.* John Wiley and Sons, 1997.

[15] Microsoft Corporation. *Microsoft COM Specification.* http://www.microsoft.com/ com/resources/specs.asp.

[16] Vojislav B. Misic. Cohesion is structural, coherence is functional: Different views, different measures. In *Seventh International Software Metrics Symposium,* pages 135–144, London, England, 2001.

[17] Object Management Group. *Common Object Request Broker: Architecture and Specification, Revision 2.6, OMG.* 2000. http://www.omg.org/technology/docaments/ formal/corba_iiop.htm.

[18] PageWise Inc. *Object Paradigm vs. Procedural Paradigm.* 2002. http://lala. essortment.com/objectparadigm-ruqg.htm.

[19] Rob Pooley and Perdita Stevens. *Using UML Software Engineering with Objects and Components.* Addison-Wesley, 1999.

[20] Dusan Progovac. Understanding core requirements: Intrusion module. In *Proceedings of the ISCA 14th International Conference, Computers and their Applications*, pages 74–77, Cancun, Mexico, 1999.

[21] Pedro S. Ripper, Marcus Felipe Fontoura, Ayrton Maia Neto, and Carlos Jose

P.de Lucena. *V-market: A framework for agent ecommerce systems.* World Wide Web, 3:43–52, 2000.

[22] Dave Wood Robert Eckstein, Marc Loy. Java Swing. O'REILLY, 1998.

[23] W.P. Stevens, J.Myers G, and L.L.Constantine. Structured design. *IBM Systems Journal,* 13(2):115–139, 1974.

[24] Sun Microsystems. *Enterprise Java Beans Technology.* http://java.sun.com/products/ ejb/.

[25] Wheelwright Wigley and Andy Wigley. *Microsoft .NET Compact Framework.* Microsoft Press, 2003.

[26] Marty Hall. *Core Servlets and JavaServer Pages.* Prentice Hall PTR, 2000.

[27] Alex Homer. *Professional Asp Techniques for Webmasters.* Wrox Press Inc. 1998.

[28] John Zukowski. *Java AWT Reference.* O'REILLY. 1997.

[29] Young Jong Yang, Song Yong Kim, Gui Ja Choi, Eun Sook Cho, Chul Jin Kim, and Soo Dong Kim. A UML-based object-oriented framework development methodology. In Proceedings of Asia-Pacific Software Engineering Conference, Pages 211-218, Taipei, Taiwan, December 1998.

[30] Alan W. Brown. *Large-scale, Component-Based Development.* Prentice Hall Press. 2000.

[31] Evelyn Stiller and Cathie Leblanc. *Project-based software engineering: An Object-Oriented Approach.* Addison Wesley. 2001.

[32] The Apache Jakarta Project JMeter. http://jakarta.apache.org/jmeter.

[33] Antonia Stefani and Michalis Xenos. *Greek vs. International E-commerce Systems: an Evaluation Based on User-centered Characteristics*

[34] Kenneth C. Laudon and Carol Guercio Traver. *E-Commerce.* Addison Wesley, 2002.

[35] Perdita Stevens and Rob Pooley, *Using UML Software Engineering with objects and components*, Addison-Wesley,1999.

Appendix A

Customer Component:

Index: Class_01
Class Name: Customers
Purpose: Providing methods that are used for managing customer's information, such as creating a new customer and update a customer's information.
Reference Classes: IUtility
Remarks: No

Methods of customers class:

Index: Method_01
Name: createCustomer
Purpose: To register a new customer.
Visibility: Public
Input parameters: firstName, lastName, email, password, address, phone, city, country, zip
Output parameters: customerId
Pseudo code:

```
recordset = SELECT * from customer where EMAIL = email
if   end of recordset then
        add new customer;
        return customerId;
else
        return 0;
end if
```

Remarks: firstName, lastName, password, address, phone, city, country, and zip are optional parameters

Index: Method_02
Name: updateCustomer
Purpose: Update an existing customer's personal information.
Visibility: Public
Input parameters: id, firstName, lastName, email, password, address, phone, city, country, zip
Output parameters: Boolean (true or false)
Pseudo code:

```
recordset = Select * from customer where ID = id
if not end of recordset
        update record;
        return true;
else
        return false
end if
```

Remarks: firstName, lastName, email, password, address, phone, city, country, zip are optional parameters

Index:	Method_03
Name:	getOneCustomer
Purpose:	Retrieve one customer's record value from customer table, which is specified by customer id.
Visibility:	Public
Input parameters:	id
Output parameters:	recordset

Pseudo code:

```
recordset = Select * from customer where customerID = id
if not end of table
        return recordset;
else return null;
```

Remarks: No

Index:	Method_04
Name:	CheckLogon
Purpose:	attempts to logon a user based on an e-mail and password.
Visibility:	Private
Input parameters:	email, password
Output parameters:	Boolean

Pseudo code:

```
recordset = select * from customer where EMAIL=email and PASSWORD = password
if not end of table
        return true;
else
        return false;
```

Remarks:

Index:	Class_02
Class Name:	Customer
Purpose:	Representing one customer record of customer table and including accessor and mutator method of each field.
Reference Classes:	IUtility
Remarks:	No

Methods of customer class:

Index:	Method_01
Name:	checkLoad
Purpose:	Check if a customer's record has been loaded from the database.
Visibility:	Private
Input parameters:	customerId
Output parameters:	Boolean

Pseudo code:

```
if loaded = false
        recordset = select * from customer where CUSID = customerId
        if not end of recordset
                firstName = recordset ("FNAME");
                lastName = recordset ("LNAME");
                email = recordset ("EMAIL");
                password = recordset ("PASSWORD");
```

```
                    address =   recordset ("ADDRESS");
                    phone =   recordset ("PHONE");
                    city = recordset ("CITY");
                    country =   recordset ("COUNTRY");
                    zip =   recordset ("ZIP");
                    loaded = true;
          else
                    loaded = false;
```

Remarks:	None

Index:	Method_02
Name:	getFirstName
Purpose:	Access a customer's first name.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String firstName
Pseudo code:	

```
if (checkLoad() == true)
          return firstName;
else
          return null;
```

Remarks:	None

Index:	Method_03
Name:	getLastName
Purpose:	Access a customer's last name.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String lastName
Pseudo code:	

```
if (checkLoad() == true)
          return lastName;
else
          return null;
```

Remarks:	None

Index:	Method_04
Name:	getEmail
Purpose:	Access a customer's email.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String email
Pseudo code:	

```
if (checkLoad() == true)
          return email;
else
          return null;
```

Remarks:	None

Index:	Method_05
Name:	getPassword
Purpose:	Access a customer's password.
Visibility:	Public

Input parameters:	customerId
Output parameters:	String password
Pseudo code:	

```
if (checkLoad() == true)
        return password;
else
        return null;
```

Remarks:	None

Index:	Method_06
Name:	getAddress
Purpose:	Access a customer's address.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String address
Pseudo code:	

```
if (checkLoad() == true)
        return address;
else
        return null;
```

Remarks:	None

Index:	Method_07
Name:	getPhone
Purpose:	Access a customer's phone number.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String phone
Pseudo code:	

```
if (checkLoad() == true)
        return phone;
else
        return null;
```

Remarks:	None

Index:	Method_08
Name:	getCity
Purpose:	Access a customer's city.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String city
Pseudo code:	

```
if (checkLoad() == true)
        return city;
else
        return null;
```

Remarks:	None

Index:	Method_09
Name:	getCountry
Purpose:	Access a customer's country.
Visibility:	Public
Input parameters:	customerId

Output parameters:	String country
Pseudo code:	

```
if (checkLoad() == true)
        return country;
else
        return null;
```

Remarks:	None

Index:	Method_10
Name:	getZip
Purpose:	Access a customer's zip.
Visibility:	Public
Input parameters:	customerId
Output parameters:	String zip
Pseudo code:	

```
if (checkLoad() == true)
        return zip;
else
        return null;
```

Remarks:	None

ShoppingCart Component:

Index:	Class_03
Class Name:	ShoppingCart
Purpose:	Hold an order instance temporally and hold methods that could operate on a shopping cart
Reference Classes:	IUtility
Remarks:	No

Methods of ShoppingCart class:

Index:	Method_1
Name:	contains
Purpose:	checks to see if an item exists in the shopping cart, if it does exist, return quantity of this product, otherwise, return zero.
Visibility:	Public
Input parameters:	productId, cartId
Output parameters:	long
Pseudo code:	

```
recordset = SELECT * from CART WHERE CARTID = cartId and PRODUCTID = productId
if not eof recordset
        return recordset ("QUANTITY");
else
        return 0;
```

Remarks:	None

Index:	Method_2
Name:	add
Purpose:	add an item to the shopping cart
Visibility:	Public

Input parameters:	productId, cartId
Output parameters:	None
Pseudo code:	

```
recordset = SELECT * FROM CART WHERE PRODUCTID = productId
if not eof recordset
    QUANTITY++;
else
    recordset.addnew
    recordset ("CARTID") = cartId;
    recordset ("PRODUCTID") = productId;
    recordset ("QUANTITY") = 1;
```

Remarks:	None

Index:	Method_3
Name:	remove
Purpose:	removes an item from the shopping cart
Visibility:	Public
Input parameters:	productId, cartId
Output parameters:	None
Pseudo code:	

```
DELETE * from CART WHERE CARTID = cartId and PRODUCTID = productId
```

Remarks:	None

Index:	Method_4
Name:	changeQuantity
Purpose:	sets the quantity of an item in the shopping cart
Visibility:	Public
Input parameters:	productId, cartId, quantity
Output parameters:	None
Pseudo code:	

```
if contains(productId, cartId) = 0
    add (productId, cartId);
else
    UPDATE CART SET QUANTITY = quantity WHERE CARTID = cartId and PRODUCTID = productId
```

Remarks:	None

Index:	Method_5
Name:	increment
Purpose:	increase an item's quantity in the shopping cart
Visibility:	Public
Input parameters:	productId, cartId
Output parameters:	None
Pseudo code:	

```
if contains(productId, cartId) = 0
    add (productId, cartId);
else
    UPDATE CART SET QUANTITY = QUANTITY+1 WHERE CARTID = cartId and PRODUCTID = productId
```

Remarks:	None

Index:	Method_6
Name:	decrement
Purpose:	decrease an item's quantity in the shopping cart
Visibility:	Public
Input parameters:	productId, cartId
Output parameters:	None
Pseudo code:	

```
if contains(productId, cartId) = 0
        add (productId, cartId);
else
        UPDATE CART SET QUANTITY = QUANTITY-1 WHERE
CARTID = cartId and PRODUCTID = productId
```

Remarks:	None

Index:	Method_7
Name:	empty
Purpose:	remove all the items in the shopping cart
Visibility:	Public
Input parameters:	cartId
Output parameters:	None
Pseudo code:	

```
DELETE FROM CART WHERE CARTID = cartId
```

Remarks:	None

Index:	Method_8
Name:	getAllItems
Purpose:	return all the items in the shopping cart
Visibility:	Public
Input parameters:	cartId
Output parameters:	None
Pseudo code:	

```
SELECT * FROM CART WHERE CARTID = cartId
```

Remarks:	None

Index:	Method_9
Name:	getNumberItems
Purpose:	return the total number of items in the shopping cart
Visibility:	Public
Input parameters:	cartId
Output parameters:	long
Pseudo code:	

```
SELECT sum(QUANTITY) FROM CART WHERE CARTID = cartId
```

Remarks:	None

Payment Component:

Index: Class_04
Class Name: Payment
Purpose: Hold methods that are used for operate on payment information, such as add a new payment type, get all the payment types of a particular customer, etc.
Reference Classes: IUtility
Remarks: No

Methods of Payment class:

Index: Method_1
Name: checkLoad
Purpose: checks if a particular payment type has been loaded from the database
Visibility: Private
Input parameters: customerId, paymentId
Output parameters: boolean
Pseudo code:

```
if (loaded == false)
        recordset = SELECT * FROM PAYMENT WHERE CUSTOMERID = customerId AND PAYMENTID = paymentId
        if not eof recordset
                type = recordset ("TYPE");
                name = recordset ("NAME");
                number = recordset ("NUMBER");
                expYear = recordset ("YEAR");
                expMonth = recordset ("MONTH");
                loaded = true;
        endif
endif
```

Remarks: None

Index: Method_2
Name: getCardType
Purpose: return the card type of one particular customer's one particular payment method
Visibility: Public
Input parameters: customerId, paymentId
Output parameters: String
Pseudo code:

```
call checkLoad()
return type;
```

Remarks: None

Index: Method_3
Name: getNameOnCard
Purpose: return the name on card of one particular customer's one particular payment method
Visibility: Public
Input parameters: customerId, paymentId
Output parameters: String
Pseudo code:

```
call checkLoad()
return name;
```

Remarks: None

Index: Method_4
Name: getNumberOnCard
Purpose: return the number on card of one particular customer's one particular payment method
Visibility: Public
Input parameters: customerId, paymentId
Output parameters: String
Pseudo code:

```
call checkLoad()
return number;
```

Remarks: None

Index: Method_5
Name: getExpireYear
Purpose: return the expire year of card of one particular customer's one particular payment method
Visibility: Public
Input parameters: customerId, paymentId
Output parameters: long
Pseudo code:

```
call checkLoad()
return year;
```

Remarks: None

Index: Method_6
Name: getExpireMonth
Purpose: return the expire month of card of one particular customer's one particular payment method
Visibility: Public
Input parameters: customerId, paymentId
Output parameters: long
Pseudo code:

```
call checkLoad()
return month;
```

Remarks: None

Index: Method_7
Name: addType
Purpose: add a new type of payment method to a customer
Visibility: Public
Input parameters: customerId, cardType, number, name, year, month
Output parameters: none
Pseudo code:

```
recordset = SELECT * FROM PAYMENT WHERE NUMBER = number and CUSTOMERID = customerId
if not eof recordset
        query = SELECT * FROM PAYMENT
        query.add new
        query("NAME") = name;
        query ("NUMBER") = number;
        query ("TYPE") = type;
        query ("EXPIREYEAR") = year;
```

```
            query ("EXPIREMONTH") = month;
            query.update;
        endif
```

Remarks:	none
Index:	Method_8
Name:	removeType
Purpose:	remove a type of payment method for a customer
Visibility:	Public
Input parameters:	customerId, paymentId
Output parameters:	none
Pseudo code:	
	DELETE * FORM PAYMENT WHERE PAYMENTID = paymentId
Remarks:	none
Index:	Method_9
Name:	removeAll
Purpose:	remove all type of payment method form the database
Visibility:	Public
Input parameters:	none
Output parameters:	none
Pseudo code:	
	DELETE * FORM PAYMENT
Remarks:	none
Index:	Method_10
Name:	removeCustomer
Purpose:	remove all type of payment method of a customer
Visibility:	Public
Input parameters:	customerId
Output parameters:	none
Pseudo code:	
	DELETE * FORM PAYMENT WHERE CUSTOMERID = customerId
Remarks:	none
Index:	Method_11
Name:	getPayments
Purpose:	get all payment methods for a single customer
Visibility:	Public
Input parameters:	customerId
Output parameters:	none
Pseudo code:	
	SELECT * FROM PAYMENT WHERE CUSTOMERID = customerId
Remarks:	none
Index:	Method_12
Name:	getPayment
Purpose:	get a payment method for a single customer
Visibility:	Public
Input parameters:	customerId, paymentId
Output parameters:	none
Pseudo code:	
	SELECT * FROM PAYMENT WHERE CUSTOMERID = customerId

AND PAYMENTID = paymentId

Remarks:	none

Index:	Method_13
Name:	checkCredit
Purpose:	check the expire date of a credit card
Visibility:	Public
Input parameters:	customerId, paymentId
Output parameters:	boolean

Pseudo code:

```
recordset = SELECT * FROM PAYMENT WHERE CUSTOMERID = customerId AND PAYMENTID = paymentId
flag = false;
if not eof recordset
    if recordset ("YEAR") > year (now)
        flag = true;
    else if  recordset ("YEAR") == year (now) && recordset ("MONYH") >month (now)
        flag = ture;
    else
        flag = false;
endif
```

Remarks:	none

Shipment Component:

Index:	Class_05
Class Name:	Shipment
Purpose:	Hold methods that are used for operate on shipment information, such as add a new shipment type, get all the shipment types, etc.
Reference Classes:	IUtility
Remarks:	No

Methods of Shipment class:

Index:	Method_1
Name:	addType
Purpose:	add a type a shipment
Visibility:	Public
Input parameters:	type, charge
Output parameters:	None

Pseudo code:

```
recordset = SELECT * FROM SHIPMENT
recordset.addnew
recordset ("TYPE") = type;
recordset ("CHARGE") = charge;
recordset.update;
```

Remarks:	none

Index:	Method_2
Name:	removeType
Purpose:	remove a type a shipment
Visibility:	Public

Input parameters:	shipmentId
Output parameters:	None
Pseudo code:	recordset = DELETE FROM SHIPMENT WHERE SHOPMENTID = shipmentId;
Remarks:	none

Index:	Method_3
Name:	getAllType
Purpose:	get all shipping type from table shipment
Visibility:	Public
Input parameters:	None
Output parameters:	None
Pseudo code:	recordset = SELECT * FROM SHIPMENT
Remarks:	none

Index:	Method_4
Name:	getOneType
Purpose:	get one shipping type from table shipment
Visibility:	Public
Input parameters:	shipmentId
Output parameters:	None
Pseudo code:	recordset = SELECT * FROM SHIPMENT WHERE SHIPMENTID = shipmentId;
Remarks:	none

Index:	Method_5
Name:	update
Purpose:	update the value of an exsiting shipping type
Visibility:	Public
Input parameters:	shipmentId, type, charge
Output parameters:	None
Pseudo code:	recordset = UPDATE SHIPMENT SET TYPE = type, CHARGE = charge WHERE SHIPMENTID = shipmentId;
Remarks:	none

Supplier Component:

Index:	Class_06
Class Name:	Catalog
Purpose:	hold methods that are used for managing catalog
Reference Classes:	IUtility
Remarks:	No

Methods of Catalog class:

Index:	Method_1
Name:	addDepartment
Purpose:	creates a new department and returns the new departmentId
Visibility:	Public

Input parameters:	departmentName, parentId
Output parameters:	departmentId
Pseudo code:	
	recordset = SELECT * FROM DEPARMENT
	recordset.addnew
	recordset ("DEPARTMENTNAME") = departmentName;
	if parentId <>0 recordset ("PARENTID") = parentId;
	return recordset ("DEPARTMENTID");
Remarks:	none

Index:	Method_2
Name:	getTopDepartment
Purpose:	return the top level department
Visibility:	Public
Input parameters:	
Output parameters:	recordset
Pseudo code:	
	recordset = SELECT * FROM DEPARMENT WHERE PARENTID = NULL;
Remarks:	none

Index:	Method_3
Name:	getAllDepartments
Purpose:	return the top level department
Visibility:	Public
Input parameters:	
Output parameters:	recordset
Pseudo code:	
	recordset = SELECT * FROM DEPARMENT;
Remarks:	none

Index:	Method_4
Name:	getDepartment
Purpose:	return a single department from the department table
Visibility:	Public
Input parameters:	departmentId
Output parameters:	recordset
Pseudo code:	
	recordset = SELECT * FROM DEPARMENT WHERE DEPARTMENTID = departmentId;
Remarks:	none

Index:	Method_5
Name:	getProductsInDepartment
Purpose:	get all products in one department
Visibility:	Public
Input parameters:	departmentId
Output parameters:	recordset
Pseudo code:	
	recordset = SELECT * FROM PRODUCT WHERE DEPARTMENTID = departmentId;
Remarks:	none

Index:	Method_6
Name:	getProduct
Purpose:	get one particular product in a department
Visibility:	Public
Input parameters:	departmentId, productId
Output parameters:	recordset

Pseudo code:

```
recordset = SELECT * FROM PRODUCT WHERE DEPARTMENTID = departmentId AND PRODUCTID = productId;
```

Remarks: none

Index:	Method_7
Name:	addProduct
Purpose:	add a new product and return the new productId
Visibility:	Public
Input parameters:	name, description, price, cost, departmentId, deparmentName, supplierId, ImageURL
Output parameters:	productId

Pseudo code:

```
recordset = SELECT * FROM PRODUCT
recordset.addnew
if name <>"" recordset ("NAME") = name;
if description <>"" recordset ("DESCRIPTION") = description;
if price <>0 recordset ("PRICE") = price;
if cost <>0 recordset ("COST") = cost;
if departmentId <>0 recordset ("DEPARTMENTID") = departmentId;
if departmentName <>"" recordset ("DEPARTMENTNAME") = departmentName;
if supplierId <>0 recordset ("SUPPLIERID") = supplierId;
if imageURL <>"" recordset ("IMAGEURL") = imageURL;
recordset.update
```

Remarks: none

Index:	Method_8
Name:	removeDepartment
Purpose:	remove a department specified by departmentId
Visibility:	Public
Input parameters:	departmentId
Output parameters:	none

Pseudo code:

```
DELETE * FROM DEPARTMET WHERE DEPARTMENTID = departmentId;
```

Remarks: none

Index:	Class_07
Class Name:	Product
Purpose:	entity object that holds data for a particular product
Reference Classes:	IUtility
Remarks:	No

Methods of Product class:

Index:	Method_1
Name:	checkLoad
Purpose:	check if the property values of a product have been loaded from the database
Visibility:	Private
Input parameters:	productId
Output parameters:	none
Pseudo code:	

```
recordset = SELECT * FROM PRODUCT WHERE PRODUCTID = productId;
if not eof recordset
        name = recordset ("NAME");
        description = recordset ("DESCRIPTION");
        price = recordset ("PRICE");
        cost = recordset ("COST");
        departmentId = recordset ("DEPARTMENTID");
        supplierId = recordset ("SUPPLIERID");
        imageURL = recordset ("IMAGEURL");
end if
```

Remarks:	none

Index:	Method_2
Name:	getName
Purpose:	return the product name specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	

```
call checkLoad();
return name;
```

Remarks:	none

Index:	Method_3
Name:	getPrice
Purpose:	return the product price specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	

```
call checkLoad();
return price;
```

Remarks:	none

Index:	Method_4
Name:	getCost
Purpose:	return the cost of product specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	

```
call checkLoad();
return cost;
```

Remarks:	none

Index:	Method_5
Name:	getDescription
Purpose:	return the product description specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	
	call checkLoad();
	return description;
Remarks:	none

Index:	Method_6
Name:	getSupplierId
Purpose:	return the supplierId of a product specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	
	call checkLoad();
	return supplierId;
Remarks:	none

Index:	Method_7
Name:	getImageURL
Purpose:	return the image URL of a product specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	
	call checkLoad();
	return imageURL;
Remarks:	none

Index:	Method_8
Name:	getDepartmentId
Purpose:	return the departmentId of a product specified by productId
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	
	call checkLoad();
	return departmentId;
Remarks:	none

Index:	Class_08
Class Name:	Inventory
Purpose:	hold methods that are used for operating the inventory
Reference Classes:	IUtility
Remarks:	No

Methods of Product class:

Index:	Method_1
Name:	addProduct
Purpose:	add a new product to the inventory
Visibility:	Public
Input parameters:	productId, amount, name, model, color, size, price, supplierId
Output parameters:	none
Pseudo code:	

```
if amount<>0
        recordset = SELECT * FROM INVENTORY
        recordset.addnew
        recordset ("PROID") = productId;
        recordset ("NAME") = name;
        recordset ("QUANTITY") = quantity;
        recordset ("MODEL") = model;
        recordset ("COLOR") = color;
        recordset ("SIZE") = size;
        recordset ("PRICE") = price;
        recordset ("SUPPLIERID") = supplierId;
        recordset.update;
end if
```

Remarks:	none

Index:	Method_2
Name:	removeProduct
Purpose:	remove a product from the inventory
Visibility:	Public
Input parameters:	productId
Output parameters:	none
Pseudo code:	DELETE FORM INVENTORY WHERE PRODUCTID = productId;
Remarks:	none

Index:	Method_3
Name:	changeQuantity
Purpose:	change a product's quantity in the inventory
Visibility:	Public
Input parameters:	productId, quantity
Output parameters:	none
Pseudo code:	UPDATE INVENTORY SET QUANTITY = quantity WHERE PRODUCTID = productId;
Remarks:	none

Index:	Method_4
Name:	checkAmount
Purpose:	return true if a product's quantity is more than the given number, otherwise return false
Visibility:	Public
Input parameters:	productId, number
Output parameters:	boolean
Pseudo code:	recordset = SELECT QUANTITY FROM INVENTORY WHERE PRODUCTID = productId;

```
if recordset ("QUANTITY") < number
        return false;
else
```

```
            return true;
```

Remarks:	none

Index:	Method_5
Name:	update
Purpose:	update a product's attributes in the inventory
Visibility:	Public
Input parameters:	productId, quantity, name, model, color, size, price, supplierId
Output parameters:	none
Pseudo code:	

```
recordset = SELECT * FROM INVENTORY WHERE PRODUCTID = productId;

if not eof recordset
        if quantity <>0 then recordset ("QUANTITY") = quantity;
        if name <>"" then recordset ("NAME") = name;
        if model <>"" then recordset ("MODEL") = model;
        if color <>"" then recordset ("COLOR") = color;
        if size <>0 then recordset ("SIZE") = size;
        if price <>0 then recordset ("PRICE") = price;
        if supplierId <>0 then recordset ("SUPPLIERID") = supplierId;
        recordset.update;
```

Remarks:	quantity, name, model, color, size, price, supplierId are optional

Index:	Class_09
Class Name:	Supplier
Purpose:	Entity object that holds data of a supplier
Reference Classes:	IUtility
Remarks:	No

Methods of Supplier class:

Index:	Method_1
Name:	checkLoad
Purpose:	check if the property values of a supplier have been loaded from the database
Visibility:	Private
Input parameters:	supplierId
Output parameters:	none
Pseudo code:	

```
recordset = SELECT * FROM SUPPLIER WHERE SUPPLIERID = supplierId;

if not eof recordset
        name = recordset ("NAME");
        address = recordset ("ADDRESS");
        city = recordset ("CITY");
        country = recordset ("COUNTRY");
        zip = recordset ("ZIP");
        phone = recordset ("PHONE");
        email = recordset ("EMAIL");
end if
```

Remarks:	none

Index:	Method_2
Name:	getName

Purpose:	return the supplier's name specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none

Pseudo code:	call checkLoad();
	return name;

Remarks:	none

Index:	Method_3
Name:	getAddress
Purpose:	return the supplier's address specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none

Pseudo code:	call checkLoad();
	return address;

Remarks:	none

Index:	Method_4
Name:	getCity
Purpose:	return the supplier's city specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none

Pseudo code:	call checkLoad();
	return city;

Remarks:	none

Index:	Method_5
Name:	getCountry
Purpose:	return the supplier's country specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none

Pseudo code:	call checkLoad();
	return country;

Remarks:	none

Index:	Method_6
Name:	getZip
Purpose:	return the supplier's zip code specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none

Pseudo code:	call checkLoad();
	return zip;

Remarks:	none

Index:	Method_7
Name:	getEmail
Purpose:	return the supplier's email specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none
Pseudo code:	

```
call checkLoad();
return email;
```

Remarks:	none

Index:	Method_8
Name:	getPhone
Purpose:	return the supplier's phone specified by supplierId
Visibility:	Public
Input parameters:	supplierId
Output parameters:	none
Pseudo code:	

```
call checkLoad();
return phone;
```

Remarks:	none

Transaction Component:

Index:	Class_10
Class Name:	Order
Purpose:	Entity object that holds data of an order
Reference Classes:	IUtility
Remarks:	No

Methods of Order class:

Index:	Method_1
Name:	checkLoad
Purpose:	check if the property values of an order have been loaded from the database
Visibility:	Private
Input parameters:	orderId
Output parameters:	none
Pseudo code:	

```
recordset = SELECT * FROM ORDER WHERE ORDERID = orderId;

if not eof recordset
        orderId = recordset ("ORDERID");
        customerId = recordset ("CUSTOMERID");
        paymentId = recordset ("PAYMENTID");
        shipmentId = recordset ("SHIPMENTID");
        total = recordset ("TOTAL");
        created = recordset ("CREATED");
        completed = recordset ("COMPLETED");
        status = recordset ("STATUS");
end if
```

Remarks:	none

Index:	Method_2
Name:	getCustomerId
Purpose:	return the customerId of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	long
Pseudo code:	

```
call checkLoad();
return customerId;
```

Remarks:	none

Index:	Method_3
Name:	getPaymentId
Purpose:	return the paymentId of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	long
Pseudo code:	

```
call checkLoad();
return paymentId;
```

Remarks:	none

Index:	Method_4
Name:	getShipmentId
Purpose:	return the shipmentId of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	long
Pseudo code:	

```
call checkLoad();
return shipmentId;
```

Remarks:	none

Index:	Method_5
Name:	getTotal
Purpose:	return the total of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	double
Pseudo code:	

```
call checkLoad();
return total;
```

Remarks:	none

Index:	Method_6
Name:	getCreated
Purpose:	return the creation time of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	datetime
Pseudo code:	

```
call checkLoad();
```

	return created;
Remarks:	none

Index:	Method_7
Name:	getCompleted
Purpose:	return the complete time of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	datetime
Pseudo code:	

```
call checkLoad();
return completed;
```

Remarks: none

Index:	Method_8
Name:	getStatus
Purpose:	return the status of an order specified by an orderId
Visibility:	Public
Input parameters:	orderId
Output parameters:	long
Pseudo code:	

```
call checkLoad();
return status;
```

Remarks: none

Index: Class_11
Class Name: Orders
Purpose: hold methods that are used for managing orders, such as creating an order, processing an order, etc.
Reference Classes: IUtility
Remarks: No

Methods of Order class:

Index:	Method_1
Name:	createOrder
Purpose:	create an order record in database and return the orderId
Visibility:	Public
Input parameters:	cartId, customerId, paymentId
Output parameters:	orderId
Pseudo code:	

```
recordset = SELECT * FROM ORDER
recordset.addnew;
recordset ("CUSTOMERID") = customerId;
recordset ("PAYMENTID") = paymentId;
recordset ("CREATED") = currentTime;
recordset ("STATUS") = 0;
recordset.update;
return recordset("ORDERID");

query = SELECT * FROM vCartItems WHERE CARTID = cartId;
if not eof query
        line = SELECT * FORM ORDERLINE
        line.addnew
        line ("ORDERID") = recordset ("ORDERID");
        line ("QUANTITY") = query ("QUANTITY");
        line ("PRICE") = query ("PRICE");
        line("TAX") = TAX-RATE;
        line("STATUS") = 0;
        line("LINETOTAL") = query ("LINETOTAL");
        line.update
end if
```

Remarks:	1. currentTime can get using a function, like now(). 2. TAX-RATE is an constant, which may vary from place to place.

Index:	Method_2
Name:	getOrders
Purpose:	return all the orders of a single customer.
Visibility:	Public
Input parameters:	customerId
Output parameters:	recordset
Pseudo code:	recordset = SELECT * FROM ORDER WHERE CUSTOEMRID = customerId
Remarks:	none

Index:	Method_3
Name:	getOrder
Purpose:	return a single order specified by customerId and orderId.
Visibility:	Public
Input parameters:	customerId, orderId
Output parameters:	recordset
Pseudo code:	recordset = SELECT * FROM ORDER WHERE CUSTOEMRID = customerId AND ORDERID = orderId;
Remarks:	none

Index:	Method_4
Name:	getOrderLines
Purpose:	return all the order lines of a single order.
Visibility:	Public
Input parameters:	orderId
Output parameters:	recordset
Pseudo code:	recordset = SELECT * FROM ORDERLINE WHERE ORDERID = orderId
Remarks:	none