INFORMATION TO USERS

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

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

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

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

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

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

NOTE TO USERS

This reproduction is the best copy available

UMI

University Course Registration and Management System – A Distributed Application Using Microsoft Distributed Component Object Model

Qiang Zhang

A Major Report
IN
The Department
OF
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 1999

© Qiang Zhang, 1999

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-43671-3

Canada

Abstract

University Course Registration and Management System – A Distributed Application Using Microsoft Distributed Component Object Model

Qiang Zhang

A University Course Registration and Management System (UCRMS) has been designed and implemented using Microsoft Visual C++ and Microsoft Distributed Component Object Model (DCOM). The UCRMS is a real time, three-tier (presentation-tier, application-tier, and data-tier), distributed application capable of running on number of PCs according to the configuration. It provides a convenient graphic user interface for both students and university administrators. It allows students to make a self-registration for their selected courses. It also allows university administrators to manage the student and course data, such as adding a student, adding a course, deleting a student, deleting a course. To support data efficiency, the data-tier has full control over database access. To support network efficiency, the application-tier provides data storage for frequently used data. To support user efficiency, the presentation-tier provides students with online course descriptions, allowing a easy course selection. To support online data modification, a callback mechanism is implemented for the application-tier server that broadcasts update information to all its clients as needed. An in-depth descriptions of DCOM, a basic technology used in the implementation of the UCRMS, is also provided with this report.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to my supervisor, Professor L. Tao, for his continuous support throughout my studies at Concordia University. Without his extraordinary guidance, this work would not have been done.

My thanks are also due to all the professors and staffs in the department of computer science, especially Halina Monkiewicz, for their excellent supports during the entire studies of my Master's degree. I would like to extend special thanks to my friend, Dr. Jun Song, for his help in getting the right knowledge for my project.

I am deeply indebted to my patient wife for her unwavering support and understanding while I was doing my project and writing my major report, I owe my lovely daughter many park plays. My deep love goes to my wife and my daughter. It is to them that I dedicate this report.

Finally, to my dearest parents, I express my cordial thanks for their continuous encouragement while I was pursuing my studies.

Table of Contents

List of Figures

1 Introduction

This report depicts designing and implementing a University Course Registration and Management System (UCRMS). UCRMS is designed to facilitate the student course registration process. It provides a convenient way for course administrators to manage the courses and students data stored in the database. It also provides a feasible method for implementing a reliable, resource saving, distributed application in Windows environment, which is the main goal and key contribution of this report.

1.1 The reason for the UCRMS

In many universities, course registration is a time consuming process for both students and course administrators. To make course registration, a student needs to select courses first and then submit a registration form to the course administrators who finally enter the student registration data into the database. It is understandable that the workloads for administrators are enormous since they need to enter the registration data for all the students. The student, on the other hand, may need to repeat the registration process if the space for the selected course is already full. This further increases the workload for course administrators. To reduce the workload imposed on both students and course administrators, we need a course registration and management system.

The UCRMS mentioned above is designed to simplify the course registration process. By using the UCRMS, students are capable of making self-registration for their selected courses. To support self-registration, UCRMS allows course administrators to manage courses and students data in the database, such as adding and deleting a course, adding and deleting a student, so that the student can make their course registration. This

self-registration system provides the students with direct access to the course registration, greatly reduces the workload imposed on the course administrators. To speed up the course registration process, the UCRMS also provides students with online course descriptions, allowing students to make their course selection effectively. Once the courses are registered into the system, these courses are guaranteed for the student.

1.2 The needs for component software

The UCRMS is designed to run in a personal computer (PC) on Windows environment. The primary reason for using a PC and Windows operating system is their popularity and accessibility to the students with different computer background. The main constraint for using a PC in the UCRMS is its limited system resource, especially its very limited physical memory. This may cause problems in using PC to run the UCRMS since student and course data in the university could well exceed the memory capacity of a PC. More than 120 MBs memory might be needed for solely storing the student data in a university with 50000 students. Therefore memory management becomes very critical in the UCRMS.

To show the important roles played by the memory resource in the UCRMS, we examine two models, respectively called Model-1 and Model-2. Model-1 implements the UCRMS to run on a single computer. The data to be processed can be retrieved from remote database via network connection. These data could be either stored in the local memory for the next usage after retrieved from the database, or they are not stored but retrieved from the remote database each time a data is needed. In the first case, the number of network transmissions can be significantly reduced. But since all the data is

loaded into a single PC, the physical memory of a PC may not be able to hold all these data. As a result, the UCRMS may not work correctly in such a heavy loaded condition. At the very least, the performance of the UCRMS might be greatly downgraded. In the second case, the system resource is saved by not loading the data into the local memory, but at the cost of possible system response-time delay caused by heavy data transmission on the network and the frequent access to the remote database.

Model-2 intends to remove the drawbacks caused by the insufficient system resource and network delay. To reduce the data transmission delay on the network and thus reduce the number of access to the remote database, the data needs to be loaded into local memory. To reduce the PC system resource usage so that the UCRMS could be executed correctly at any time, the UCRMS itself needs to be cut into several pieces with different pieces to run on different computers. In such a distributed environment, the data can be loaded and distributed into local memory of different computers by different pieces of the UCRMS. These different pieces work together to complete their common task, that is to help students register their courses and to help administrators manage student and course data. Each of these software pieces is the component of the UCRMS, which could be implemented by using the so-called component software technology. Until now two parallel technologies are available for the component software. One of the technologies — the Distributed Component Object Model (DCOM) [1, 2] is developed by the Microsoft Cooperation to support communication among objects on different computers. The other one — the Common Object Request Broker Architecture (CORBA) [3, 4] is developed by the Object Management Group (OMG) to support the invocations of operations on objects located anywhere on a network.

Both DCOM and CORBA support the following common features in the development of server components:

- Multiple programming languages. All the main programming language can be used for the component implementation.
- Location transparency. Client can invoke server object without the details of server location.
- Strong security. Different levels of security checks are performed for server object invocation and access.
- Multithreaded server. Concurrent multiple-client accesses to the server object are supported.

In addition to the above common features, several key differences between DCOM and CORBA can be summarized as below:

- High-quality development tools are available for building DCOM components. On the contrary, no development tools are provided for building CORBA components.
- DCOM has large selection of commercially available ActiveX components for use. CORBA does not have such support, causing longer development cycle on average.
- CORBA supports broad operating system. DCOM only support Windows NT 4.0, Windows 95/98, Sun Solaris 2.5, and Digital UNIX.
- CORBA supports implementation inheritance, one interface can inherit another's components. DCOM only supports interface inheritance, the derived interface does not inherit the components available under the original implementation.

In the case of UCRMS where PC and Windows system are used, it is more natural to use DCOM rather than CORBA since more developing supports are available for DCOM in Windows environment.

1.3 The Distributed Application Model for the UCRMS

A typical application that interacts with a user, such as UCRMS, usually consists of three elements: presentation, application logic, and data. Presentation focuses on interacting with the user. Application logic performs calculations and determines the flow of the application execution. Data elements manage information that must persist across sessions or be shared between users.

Two-tier, or standard client/server applications, as described in Model-1 in section 1.2, group presentation and application logic components on the client computer and access a shared data source using a network connection. The advantage of such a configuration is that the data is centralized. This centralization benefits an organization by sharing data, providing consistency in accessed data, and reducing duplication and maintenance. But there are also numbers of limitations for the two-tier applications, such as poor scalability, poor maintainability, poor reusability, and poor network performance.

In three-tier architectures, presentation, application logic, and data elements are conceptually separated. Presentation components manage user interaction and request application services by calling middle-tier (application-tier) components. Application components perform business logic and make requests to databases. Application design becomes more flexible because clients can call server-based components as needed to complete a request, and components can call other components to improve code reuse.

Three-tier applications implemented using multiple servers across a network are referred to as distributed applications.

Three-tier architectures are often called server-centric, because they uniquely enable application components to run on middle-tier servers, independent of both the presentation interface and database implementation. The independence of application logic from presentation and data offers many benefits:

- Multi-language support. Application components can be developed using general programming languages.
- Centralized components. Components can be centralized for easy development, maintenance, and deployment.
- Load balancing. Application components can be spread across multiple servers, allowing for better scalability.
- Efficient data access. The problems for database connection are minimized since the database now sees only the application component, and not all of its clients. Also database connections and drivers are not required on the client.
- Improved security. Middle-tier application components can be secured centrally using a common infrastructure. Access can be granted or denied on a component-by-component basis, simplifying administration.
- Simplified access to external resources. Access to external resources, such as mainframe applications and other databases, is simplified; a gateway server becomes another component that is used by the application.

The structure of three-tier application is shown in Figure 1. The adjacent tiers are connected through the network.

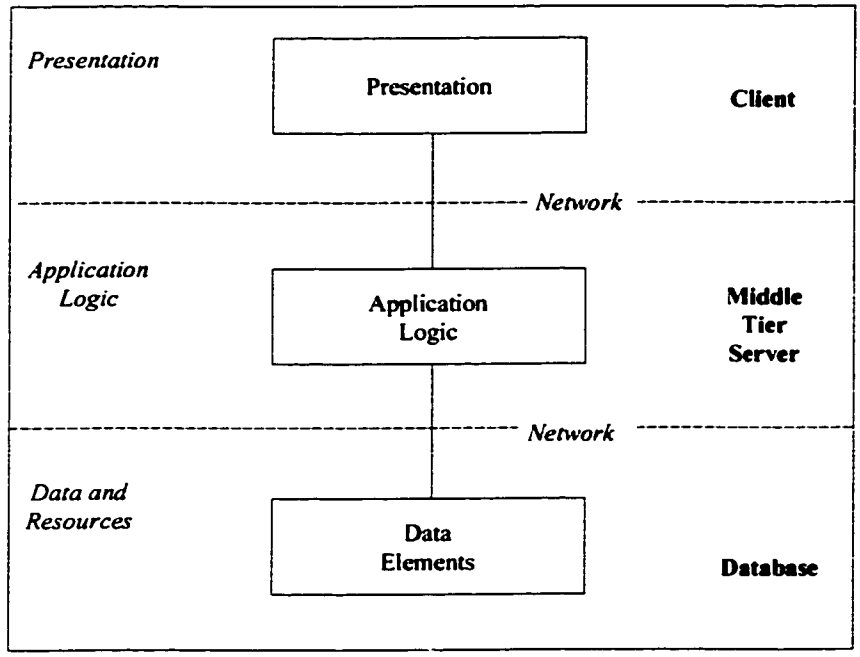


Figure 1: Structure for Three-tier Application

The UCRMS fits well into three-tier distributed application. The students and course administrators interact with the system through the graphic user interface provided by the presentation tier. The application services are provided to the presentation-tier component by the middle-tier component. In the UCRMS, the middle tier should also provide the essential data storage to reduce the network traffic. The data-tier component manages the access to the database, as for example adding a course into database. All these software components should be configured and distributed into several computers so that the collection of system resources is sufficient to guarantee the correct execution of the UCRMS.

The reason for the UCRMS to use distributed three-tier application is not only the resource sharing, but also the benefits gained from distributing the components into different computers. There are enhanced efficiency, scalability, and reliability of the

system by partitioning and distributing a complex application into presentation, application logic, and data sections.

- Scalable: As the number of users or workload increases, the application does not degrade significantly.
- Reliable: Reliable applications do not stop users from doing their jobs due to a hardware or software failure. Reliable applications also have a high level of confidence from their users of the correctness of their operation and availability.
- Efficient: Efficient applications do their work quickly and are effective at helping the user reaching their goal.

2 DCOM Functionality

In this chapter, we present an in-depth description of the Microsoft Distributed Component Object Model (DCOM). This component technology will be used in the implementation of application-tier and data-tier of the UCRMS.

DCOM is a specification and set of services that allow software developer to create modular, object-oriented, customizable, upgradable, and distributed applications using a number of language [5]. It supports communication among objects on different computers, whether on a local area network (LAN), a wide area network (WAN), or even the Internet. In DCOM architecture, applications are built from packaged binary components with well-defined interfaces [1, 5]. It allows flexible update of existing applications, provides a higher-degree of application customization, encourages large-scale software reuse, and provides a natural migration path to distributed applications. The Component Object Model (COM) [6] is an approach to achieving component

software architecture. COM specifies a way for creating components and for building applications from components. Specifically, it provides a binary standard that components and their clients must follow to ensure dynamic interoperability. DCOM is the distributed extension of COM. It is an application-level protocol for object-oriented remote procedure call. With DCOM technology, an application can be configured and distributed at any locations. Because DCOM is an extension of COM, one can take advantage of the existing COM-based components and knowledge to quickly build a new distributed application.

DCOM is currently an active research field, especially in the area of reliability and security [7, 8]. The new version of DCOM, to be packaged with Windows NT 5.0, will provide improved supports in reliability, security, and easy developing. In the following we present the key aspects of DCOM for the current version, packaged with Windows NT 4.0.

2.1 Object Activation

One of the important features of DCOM is a mechanism for establishing connections to components and creating new instances of components either locally or remotely. In the COM/DCOM world, object classes are named with globally unique identifiers (GUIDs), which are called Class IDs. These Class IDs are nothing more than fairly large integers (128 bits) that provide a collision free, decentralized namespace for object classes. The COM libraries provide *CoCreateInstanceEx*, *CoGetInstanceFromFile* for remote object creation, which in turn calls *QueryInterface* implemented in the server component.

In order to be able to create a remote object, the COM libraries need to know the network name of the server. Once the server name and the Class Identifier (CLSID) are known, the service control manager (SCM) on the client machine connects to the SCM on the server machine and requests creation of this object. As shown in Figure 2, the DCOM protocol is layered on top of the OSF DCE RPC specification [9]. Next to the DCE RPC is the *Security Provider* offering security checks for the method calls between client and object. Proxy and stub are responsible for marshalling and unmarshalling for any method calls, their details will be given in the following section.

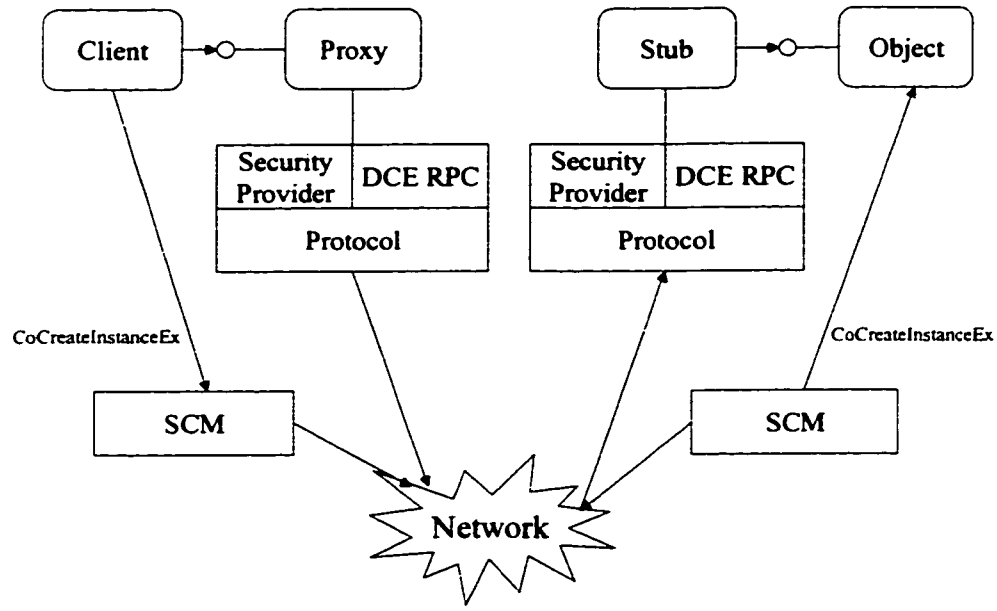


Figure 2: DCOM Architecture

Two fundamental mechanisms allow clients to indicate the remote server name when an object is created:

- As a fixed configuration in the system registry.
- As an explicit parameter to *CoCreateInstanceEx*, *CoGetInstanceFromFile*.

The first mechanism is extremely useful for maintaining location transparency. By making the remote server name part of the server configuration information registered on the client machine, clients do not have to worry about maintaining or obtaining the server location. If the server name changes, the registry is changed and the application continues to work without further action.

Some applications require explicit run-time control over the server to be connected. For this kind of application, COM allows the remote server name to be explicitly specified as a parameter to *CoCreateInstanceEx*, *CoGetInstanceFromFile*. The developer of the client code is in complete control of the server name being used by COM for remote activation.

2.2 Marshaling and Unmarshaling

When a client wants to call an object in another address space, the parameters to the method call must be passed from the client's process to the object's process. The client places the parameters on the stack. For remote invocations, the caller and the object don't share the same stack. Some COM libraries need to read all parameters from the stack and write them to a memory buffer so they can be transmitted over a network. The process of reading parameters from the stack into a memory buffer is called "marshaling." The counterpart to marshaling is the process of reading the flattened parameter data and recreating a stack that looks exactly like the original stack set up by the caller. This process is called unmarshaling. Once the stack is recreated, the object can be called. As the call returns, any return values and output parameters need to be marshaled from the object's stack, sent back to the client, and unmarshaled into the client's stack.

COM provides sophisticated mechanisms for marshaling and unmarshaling method parameters that build on the remote procedure call (RPC) infrastructure defined as part of the distributed computing environment (DCE) standard. In order for COM to be able to marshal and unmarshal parameters correctly, it needs to know the exact method signature, including all data types. This information is provided using an interface definition language (IDL), which is also built on top of the DCE RPC [9] standard IDL. IDL files are compiled using a special IDL compiler — MIDL compiler, which is part of the Win32 SDK. The IDL compiler generates C source files that contain the code for performing the marshaling and unmarshaling for the interface described in the IDL file. The client-side code is called the "proxy," while the code running on the object side is called the "stub." The MIDL generated proxies and stubs are COM objects that are loaded by the COM libraries when needed. By looking up the Interface Id (IID) from the system registry, COM can find the proxy/stub combination for a particular interface.

2.3 Object Connection Control

An object's lifetime is controlled by a mechanism called reference counting, which uses the *AddRef* and *Release* methods of interface *IUnknown*. Every COM/DCOM interface must inherit from interface *IUnknown*; every COM/DCOM component must implement *AddRef* and *Release*. *AddRef* and *Release* are called quite often, and sending every call to a remote object would introduce a serious network performance penalty. Hence, DCOM optimizes *AddRef* and *Release* calls for remote objects. To do so, remote reference counting is conducted per interface rather than per connection, allowing for greater network efficiency.

Remote reference counting would be entirely adequate if clients never terminated abnormally, but in fact they do. To make system robust in the face of clients terminating abnormally when they hold remote references, a *Pinging* mechanism is used for detecting any client abnormal termination. At every elapse of predefined ping period time, the server object sends a ping signal to the connected client. If the ping period elapses without receiving a ping on that server object, all the remote references to interfaces associated with that server object are considered "expired" and can be garbage collected.

2.4 Connection Points

Many real-world distributed applications require bidirectional communication between two objects. An object may need to notify a client of a certain event or objects might want to push data back as it becomes available.

Connection points standardize passing an interface pointer to an object. With connection points, the object that implements the interface for registering the "callback" (*IConnectionPoint*) is required to be a separate object from the actual object. Multiple clients register themselves with the same connection point, and the connection point sends notifications to all the clients. However, since the server must synchronously call each client in turn, dealing with fragile network links, slow clients, and network time-outs require additional design. One approach is to use multiple threads to notify several clients simultaneously. If a client does not respond within a reasonable time, the object spawns an additional thread to notify the next client.

2.5 DCOM Threading Models

To support multi-clients accessing the server objects concurrently, multi-threading models are required for the DCOM architecture.

2.5.1 Single-Threaded Apartments (STA)

In a single-threaded apartment, each object lives in a thread. Each thread must initialize COM using either **CoInitialize** or **CoInitializeEx.**

The basic concurrency unit in an STA is the individual thread that initializes COM. If two objects, for example A and B, live in the same apartment, and A is processing a call, no other client can make a call, to either A or B, until A completes its service. As a result, instance data that is exclusive to an object need not be protected, because only one thread can ever enter this instance of the object. But this may delay the server response and thus downgrade the system performance.

2.5.2 Multithreaded Apartment (MTA)

A multithreaded apartment is an easier model compared to STA. Incoming RPC calls directly use the thread assigned by the RPC run time. The object does not live in any specific thread. Clients from any thread can directly call any object inside the MTA. However, MTA requires extreme caution on the shared the data. Multiple threads can call an object method at the same time. Therefore the object must provide synchronized access to any instance using synchronization primitives such as critical section and semaphores.

Although complex to implement, MTA objects offer the possibility of higher performance and better scalability than STA objects since the generic synchronization

that COM performs on STA is relatively expensive [1, 2]. A good design technique is to isolate the critical areas of an application into separate objects and move the critical objects into MTAs to achieve high overall performance and scalability. Before using MTA, a thread must initialize COM by calling **CoInitializeEx**.

2.6 Security Issues

One of the most difficult issues in the design of distributed application is that of security. DCOM provides an extensible and customizable security framework for the developers.

2.6.1 Access security

The most obvious security requirement on distributed applications is the need to protect objects against unauthorized access. Only authorized users are supposed to be able to connect to an object. Current implementations of DCOM provide declarative access control on a per-process level. Existing components can be securely integrated into a distributed application by simply configuring their security policy as appropriate.

2.6.2 Launch security

Another related requirement on a distributed infrastructure is to maintain control of object creation. Since all COM objects on a machine are potentially accessible via DCOM, it is critical to prevent unauthorized users from creating instances of these objects. For this purpose, the COM libraries perform special security validations on object activation. If a new instance of an object is to be created, COM validates that the

caller has sufficient privileges to perform this operation. The privilege information is configured in the registry, external to the object.

2.6.3 Authentication

The above mechanisms for access and launch permission checks require some mechanism for determining the security identity of the client. This client authentication is performed by one of the security providers, which returns unique session tokens that are used for ongoing authentication once the initial connection has been established. The initial authentication often requires multiple round trips between caller and object.

DCOM uses the access token to speed up security checks on calls. To avoid the additional overhead of passing the access token on each and every call, DCOM by default only requires authentication when the initial connection between two machines is established. It then caches the access token on the server side and uses it automatically whenever it detects a call from the same client. For many applications this level of authentication is a good compromise between performance and security. However, some applications may require additional authentication on every call, as for example passing in credit card information or other sensitive information, the object might require calls to be individually authenticated.

2.6.4 Impersonation levels

A more subtle implication of security in distributed applications is the issue of protecting callers from malicious objects. Since DCOM allows objects to impersonate callers, objects can actually perform operations they do not have sufficient privileges to perform alone. To prevent malicious objects from using the caller's credentials, the caller

can indicate what it wants to allow objects to do with the security token it obtains. The following options are currently defined:

- Anonymous: the object is not allowed to obtain the identity of the caller. This is the safest setting for the client but the least powerful for the object.
- Identify: the object can detect the security identity of the caller, but can not impersonate the caller. This call is still safe for the client since the object will not be able to perform operations using the security credentials of the caller.
- Impersonate: the object can impersonate and perform local operations, but it can not call other objects on behalf of the caller. This mode is potentially unsecure for the caller, since it allows the object to use the client's security credential to perform arbitrary operations.

These options are defined as part of the Windows NT security infrastructure. Again, DCOM allows these settings to be both programmatically controlled and externally configured.

3 System Requirements

In the area of software engineering, the development of software is based on the software requirements. These requirements must be fully satisfied by the system implementation. After the implementation is completed, the system must be validated against the requirements. Any invalid implementation must be corrected and validated again. In this chapter, the requirements for the UCRMS are presented.

3.1 Hardware requirements

A personal computer with color screen is required for displaying the visual interface, mouse and keyboard are required for user input, and Internet connection facility is required for connecting to the remote servers and for transferring the data between the different software components. At least one power PC is needed for install the server component. More PCs might be needed for running different server components in a distributed environment.

3.2 Software requirement

UCRMS is a distributed, real-time application designed to be used in Windows environment. Since Windows 95/98 has a weak security feature, Windows NT is recommended. To guarantee the real-time and distributed feature of UCRMS, the implementation is performed using Visual C++, MFC, and distributed component object models (DCOM). Before using UCRMS, MFC library and DCOM should be properly installed. Moreover, an ActiveX Flex gridline control is integrated into the system to give a clear presentation of student and course data. Therefore the gridline control must be also installed and registered into the Windows operation system.

3.3 Graphic User Interface requirements

UCRMS consists of five main windows as its user interface: the login window, the student registration window, the course add and drop window, the course and student management window, and the super user window. The specifications for these windows are not intended to cover all the possible user requirements.

3.3.1 login window

Upon the execution of UCRMS program, a login window is displayed to the user.

- User is prompted to enter user ID and password, and select login type into the corresponding fields, as shown in Figure 20.
- The system starts processing user login information when user clicks on "Enter" button with a mouse.
- An invalid login information brings up a message box informing the user that system failed to login him onto the system. Upon the acknowledgement of the system message, the system clears the password fields.
- A successful login brings up either student registration window, or course add and drop window, or course and student management window, or administrator management window, depending on the identification of the user being logged onto the system.
- Clicking on "Quit" button will terminate the UCRMS application.

3.3.2 Student Course Registration Window

Upon a successful login as a student (login type: student), a course registration window appears (see Figure 21).

- On upper portion of the window locates a pre-select course list that lists the total courses available for selection by the student. On lower portion of the window locates a pre-register course list that lists the courses to be registered by the student.
- Student can select one course at a time by clicking on the course presented in either of the course lists. The selected course becomes highlighted.

- Student can put the selected course into pre-registered list by clicking "Select" button. Once a course is selected from the pre-select list and "Select" button is clicked, it appears in the pre-register list. At the same time, it is removed from pre-select list to avoid duplicate selection on the same course.
- The course in the pre-register list can be removed by selecting it first and then clicking the "Remove" button. Once a course is removed from pre-register list, it re-appears in the pre-select list for possible re-selection.
- Student can register the courses in the pre-register list by clicking "Register" button. A dialog box appears that displays the courses to be registered and asks the student to confirm the registration. Student can confirm the registration by clicking "OK" button, and consequently the courses are registered into database. Student can cancel the registration by clicking "Cancel" button and system goes back to the course registration window.
- System goes back to login window when "Quit" button is clicked.

3.3.3 Course Add and Drop Window

If a student has already made course registration when he logs onto the system, a Course Add and Drop window appears (see Figure 24).

- Student can perform course add operation by clicking on the "Add" button. A message box will appear asking the confirmation from the student.
- Student can perform course drop operation by clicking on the "Drop" button. A message box will appear asking the confirmation.
- The system goes to login window when "Quit" button is clicked.

3.3.4 Course and Student Administration Window

Upon a successful login as an administrator (login type: administrator), a student and course management window appears (see Figure 25)

- On the upper portion of the window locates a course list where the courses can be added or removed from the database.
- Administrator can select one course at a time by clicking on the course displayed in the course list. The selected course becomes highlighted.
- The selected course can be removed from the list by clicking "Remove" button.
- The administrator can add a new course by clicking "Add" button. A dialog box appears prompting the administrator to enter the course information.
- Adding a course can be committed by clicking "OK" button. The added course appears in the course list immediately after the commit. Adding a course can be cancelled by clicking "Cancel" button.
- On the lower portion of the window locates a student list where the student can be added or removed from the database.
- Administrator can select one student at a time by clicking on the student displayed in the student list. The selected student becomes highlighted.
- The selected student can be removed from the list by clicking "Remove" button.
- The administrator can add a new student by clicking "Add" button. A dialog box appears prompting the administrator to enter the student information.
- Adding a student can be committed by clicking "OK" button. The added student appears in the student list immediately after the commit. Adding a student can be cancelled by clicking "Cancel" button.

- The system goes back to login window when "Quit" button is clicked.

3.3.5 Administrators Management Window

Upon a successful login as a super user (login type: administrator), an administrator management window appears (see Figure 27)

- An administrator list is displayed on the window. The administrator in the list can be selected when clicked on it. When selected, it becomes highlighted.
- A new administrator can be added by clicking on "Add" button. An administrator-add dialog box appears, prompting the super user to enter the required administrator information. Adding an administrator can be committed by clicking on "OK" button. The added administrator appears in the administrator list immediately after the commit. Adding an administrator can be cancelled by clicking "Cancel" button.
- Clicking on "Remove" button removes the selected administrator from the list.
- System goes back to login window when "Quit" button is clicked.

3.4 Middle-tier/Server Requirement

To support three-tier distributed application, the UCRMS consists of several server components configured to run on different computers. These components form the application-tier of the UCRMS. They should have the following features to fulfil the performance requirements.

- The data services are obtained by calling data-tier component only.
- The data managed by each middle-tier server is to be stored into its corresponding local memory after retrieved from the data-tier for the first time. The whole system

data should be distributed into these different server computers to avoid all the data being loaded into a single computer.

- The number of middle-tier servers should be configurable based on the amount of data and the capacity of server computers.
- To ensure the data consistency, the servers responsible for the dynamically changeable data should be able to call back to the presentation-tier to dynamically update the data being loaded into different user's computers but changed by one of the users. If callback mechanism is not implemented, the other mechanism should be in place to guarantee the run-time update.

3.5 Data-tier requirement

In some of the universities, the databases may be built in the UNIX environment, and in other universities they may be built in Windows NT environment. Therefore to support the database location transparency in the application-tier, a database server is needed in the data-tier. It's the data-tier server that directly accesses either local database or remote database whether it is on UNIX or on Windows NT. Such a configuration allows the middle-tier servers completely independent of database operations.

4 System Design

System design is an essential step in developing a reliable, secure, and user-friendly application. A full analysis of users and the system environment is critical for a successful system design.

4.1 User Description

- The UCRMS is to be used by the students in universities to select and register their courses at the beginning of each semester. It is also to be used by the administrators to manage the courses and student data stored in the database on the basis of frequent usage.
- Low computer literacy is required for students to use the system; low to moderate computer literacy is required for administrators to use the system.
- Heavy usage is expected for students during registration time that starts at the beginning of each semester.

4.2 Design Consideration

The UCRMS intends to be designed as a three-tier distributed application. The whole system design includes the design of presentation-tier — the Graphic User Interface, the design of application-tier — the application services, and the design of data-tier — the database access.

4.2.1 Presentation-tier — Graphic User Interface

Based on the user description, following points are considered in the design of graphic user interface:

- UCRMS is a distributed multi-user application; data are transferred between the server site and client (user) site through the Network. The workload on the communication channel and the data server is expected very heavy at the beginning of each semester. In order to improve the performance, the connection and data

transfers between client and server is performed only once for all selected courses rather than once per selected course.

- The UCRMS is to be used mainly by students in universities to register their courses. It is also to be used by the administrators to manage students and courses data stored in the database. The common student registration procedure is considered as a metaphor for the UCRMS.
- Different kind of students may have different level of knowledge in using computer. Thus for students the UCRMS is designed to be error-protected. No student input can cause system error or incorrect database update. To do so, we need to reduce keyboard input as far as possible since keyboard input is a potential source of error. Here the only keyboard input required for students are login process and changing password process, which are fully controlled by the system. All the other interactions between students and the UCRMS are mouse-oriented, which prevents the system data from any incorrect input.
- The UCRMS requires moderate keyboard input for administrators. To minimize the possible input error, a dialog box is popped up for confirmation before each database update is finalized, allowing the administrators to modify or abort the incorrect update.
- UCRMS is designed to teach the user about the system incrementally. An informative message box is popped up for each invalid or incorrect user action, which informs the user of possible cause and how to correct it.
- UCRMS is designed to be consistent. As described above in the User Interface Requirement, two course lists, pre-select course list and pre-register course list, are

displayed in the Course Registration Window. Before registration, courses to be selected are all displayed in pre-select list. The courses that are selected into pre-register list will disappear from the pre-select list. The courses that are removed from pre-register list will re-appear in the pre-select list. This ensures the consistency that the course to be selected is always the course not selected yet.

4.2.2 Middle-tier — Application Services

Based on the middle-tier/servers requirements, the following points are considered as a guideline for designing the middle-tier servers. As mentioned in the introduction of this report, the middle-tier components/servers are going to be implemented using DCOM, therefore DCOM is considered as parts of our consideration in design issues.

- The number of servers should be configurable according to the maximum workload each server can handle.
- Multiple users should be allowed to connect to the servers and be provided with application services. To support maximum performance of the servers, the concurrent accesses to the servers should be allowed; to avoid the possible errors caused by the race conditions, each critical operations should be protected by either the critical sections or semaphores.
- A user is allowed to modify the data shared by the other users through the middle-tier servers. To keep the data consistency, a mechanism should be provided to make the servers capable of calling back to all the connected user processes to update the data changed by that particular user. To support update, these servers should keep an update list containing all the actively connected user objects.

- The middle-tier servers should be able to detect the abnormal termination of client processes in order to delete that client from servers' update list as soon as possible. Otherwise, a memory leak might occur when the servers try to update the data by calling the faulty client object. The "ping" mechanism provided by the DCOM is able to detect the abnormal termination of the client processes. But in the case of the UCRMS, that detection of abnormal termination may arrive too late to avoid the system failure since active update may happen at any time after user process terminates abnormally. One of the solutions could be to apply "try" and "catch" pair to all the client update operations.
- The middle-tier servers should control the number of requests to the data-tier for reducing the workload imposed on the database. Thus the frequently requested data should be kept in the memory of server computers once retrieved from the database. To keep the data consistency, the servers should update all the data copies stored in the server computers as well as the copies stored in the client objects. Critical section should be used in these active data update.
- Consider the following situation: the server is trying to update the client while that client calls Release() of the server and then quits by destroying itself. To avoid calling client object that already destroyed, we should use critical section to protect these concurrent operations.
- To make best use of the data loaded from data server through the network, and thus reduce the data transmissions over the network, some servers should be implemented as persistent DCOM server so that the loaded data is always there to serve the user requests.

4.2.3 Data-tier — Database Access

From the requirements for data-tier, the following points need to be considered in the design of data-tier component.

- To reduce workload for database, the number of database access must be minimized.
- A set of services is to be presented to the middle-tier servers. All the database queries and operations should be operated through public interface of the server, and be encapsulated into data server to avoid invalid access to the database by the outside world.

4.3 Task Analysis

There are four distinguish tasks involved in UCRMS: registration task, course add and drop task, student and course management task, and administrator management task. Each of these tasks corresponds to one specific type of user action, that is: a student performs registration; a student adds or drops courses; an administrator manages student and course data; and a super user manages administrator data.

The detailed descriptions of these tasks and their hierarchical task analysis diagram (HTA) are respectively given as below:

- **Registration Task**: The task starts from login session. After successful login, the student can perform course registration task that includes view course description, select courses, and register for the selected courses. The task ends when registration is done or user explicitly quits. The HTA diagram is shown in Figure 3.

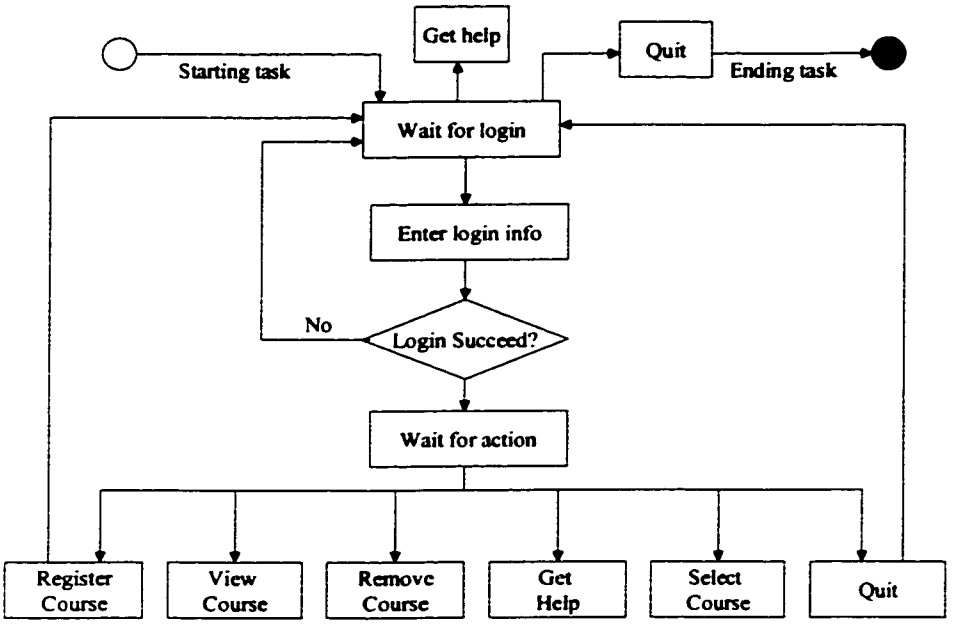


Figure 3: Registration task analysis diagram

- **Course Add and Drop task:** the task starts from login session. After successful login, the student is presented with a course add and drop window if registered course(s) already exist in the database for that particular student. Then student can perform course add and drop task that includes adding and deleting a course to and from his registered course list. The task ends when user explicitly quits from the course add and drop window. The HTA diagram is shown in Figure 4.
- **Student and Course Management Task**: The task starts from login session. After a successful login, the administrator can perform students and courses management task that includes adding and removing course from existing course list, adding and removing student from existing student list. The task ends when administrator quits from the management window. The HTA diagram is shown in Figure 5.

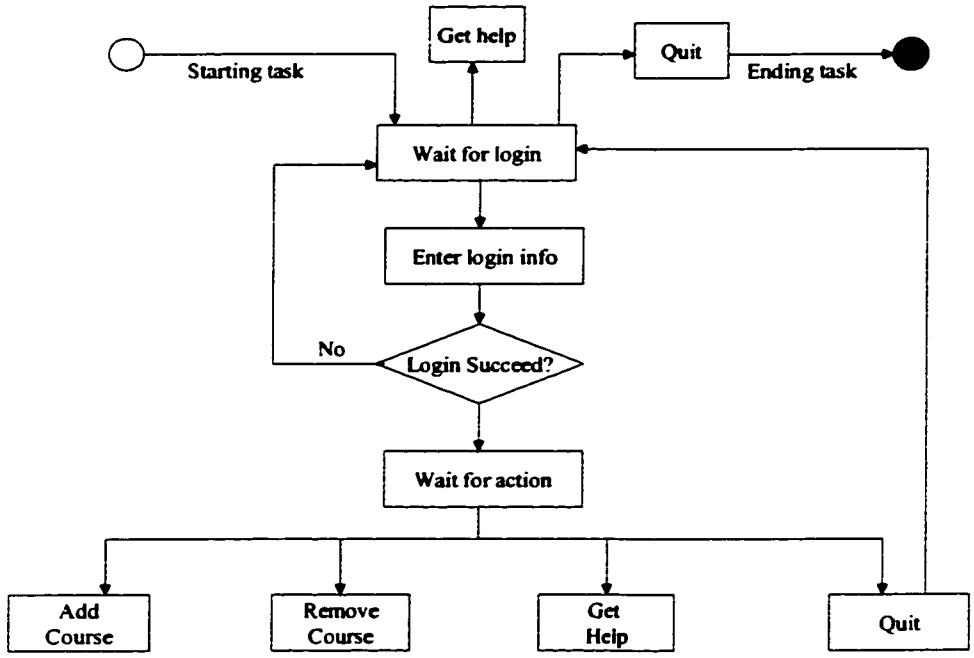


Figure 4: Course Add and Drop task analysis diagram

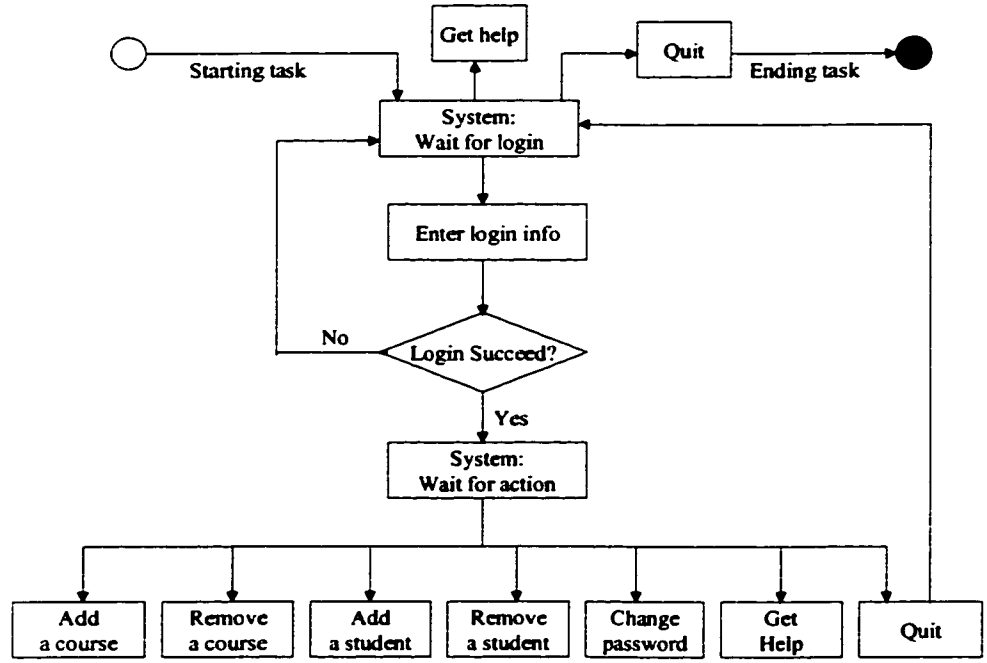


Figure 5: Student and Course Management task analysis diagram

- **Administrator Management Task**: The task starts from login session. After successful login, the super-user can perform the administrator management task that includes adding and removing an administrator from the existing administrator list, modifying administrator's record. The task ends when the super user explicitly quits from the management window. The HTA diagram is shown in Figure 6.

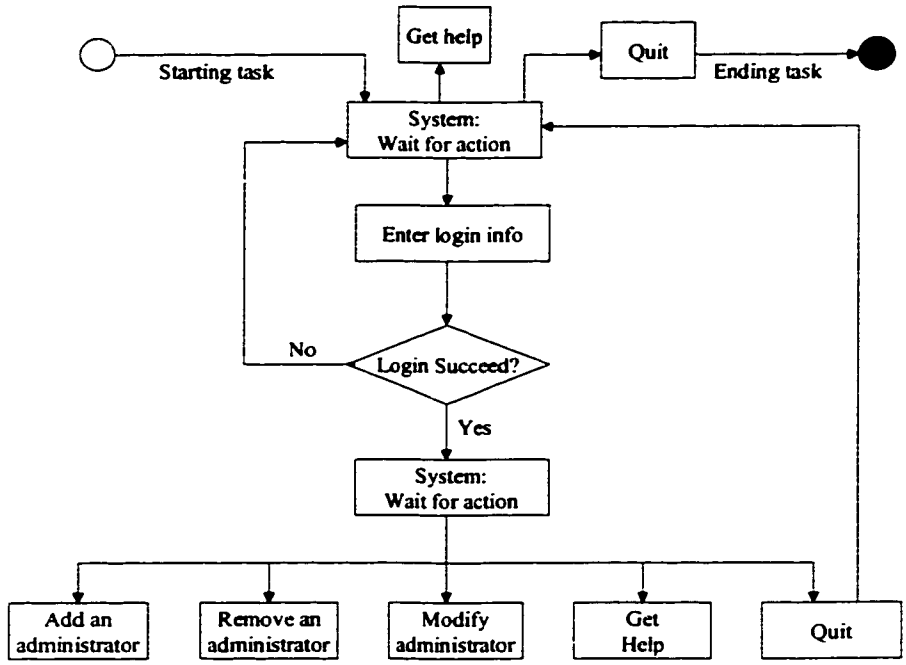


Figure 6: Administrator Management task analysis diagram

4.4 System Architecture

The UCRMS architecture design is shown is Figure 7. The whole system consists of six basic modules, the *GUI* module, the *GuiBase* module, the *DataControler* module, the *SecurityMgr* module, the *SpecMgr* module, and finally the *DataAccess* module.

User interacts with the system via *GUI* module. The *GUI* module provides the users with convenient graphic user interface. The user can either get the infomation from the system or enter the information into the system through the graphic user interface.

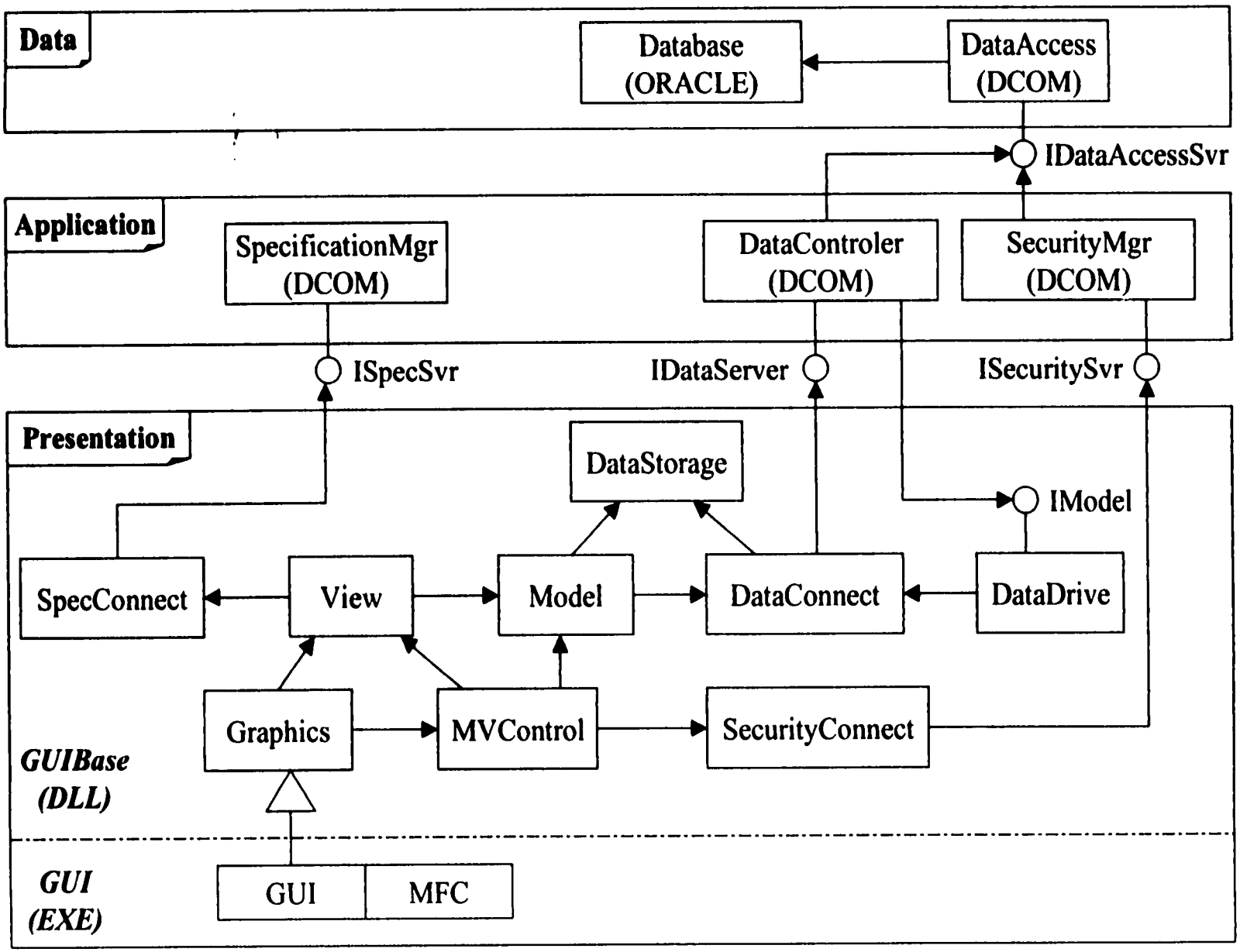


Figure 7: System Architecture for the UCRMS

A *GuiBase* module is built on top of *GUI* module to support the processing of information requested by the users through the graphic user interface provided by the *GUI* module. The *GuiBase* module interacts with the outside world via *Graphics* class that exports the necessary functions for use by *GUI* module. The *View* class is responsible for getting the required information for displaying the graphic user interface, while the *Model* class is responsible for managing and processing the user data. The *MVControl* class is responsible for creating and deleting the Model and View objects as needed. At initialization stage, *GuiBase* module creates the proper Model and View objects. It then establishes the connections between *GuiBase* and the middle-tier servers — *SpecificationMgr*, *DataControler*, and *SecruityMgr* via *SpecConnect*, *DataConnect*, and *SecurityConnect* classes. At any given time there may be several user processes connected with middle-tier servers. To support active update of the data presented in the *GUI* module, *GuiBase* module also provides a public interface, *IModel*, to be implemented in the *DataDrive* class by using DCOM technology. Once an active update is required, the related servers in the middle-tier will call the update functions in the *GuiBase* through its public interface *IModel*. Then the *GuiBase* module transfers the update information to the *GUI* module where users get the update information.

As mentioned above, three middle-tier servers, the *SpecificationMgr*, the *DataControler*, the *SecurityMgr*, are to be implemented using DCOM technology. One of the great benefits gained by using DCOM is its capability of dynamic invocation and termination of the servers. The server is launched when there is an active request to the server. The server automatically shut down when released by the user process. Thus the system resource is consumed only when there is an active connection to the server. Since

a PC has a very limited system resource, this mechanism will greatly improve PC performance.

The basic functionality of these middle-tier servers is to provide necessary application services to the *GuiBase* module so that the course registration and management tasks could be completed. These application services are exposed by the public interfaces — *IDataServer*, *ISecuritySvr*, and *ISpecSvr*. In addition to the application services, middle-tier servers should also keep a copy of the essential data frequently requested by the user. After the data is retrieved from the database for the first time, the essential data is loaded into the memory of the server computer. When the second user requests the same data as the first user does, the related middle-tier server simply get the data from its data storage and immediately returns it to the user via *GuiBase* module. Therefore database access is avoided for any subsequent data requests, and the workload for database is thus greatly reduced. The data services are provided by the *DataAccess* server through the public interface *IDataAccessSvr*. *DataAccess* server is also implemented by DCOM technology. It directly accesses the database through its inner class that encapsulates all the database operations and queries. The only way for the outside world to access the database is using the methods provided in *IDataAccessSvr*.

4.5 System Design — Object Model

The diagrams displayed in Figures 8, 9, 10, 11, 12, 13 present the modules and their classes in details.

- The class diagram for *GUI* module is presented in Figure 8. When system is started, *CCourseRegDlg* graphic object will be created. Upon the run-time situation, it creates

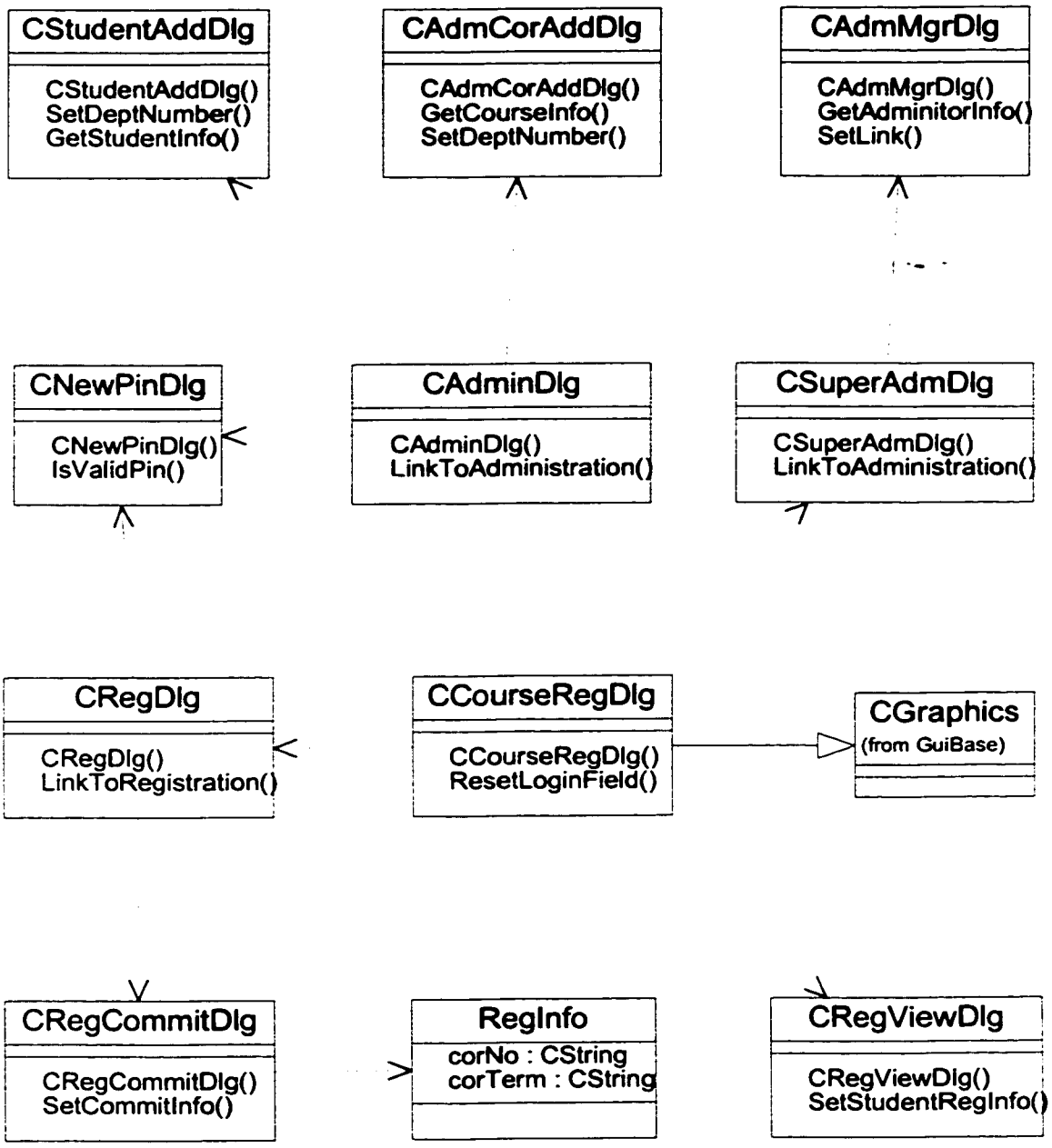


Figure 8: Class Diagram for GUI Module

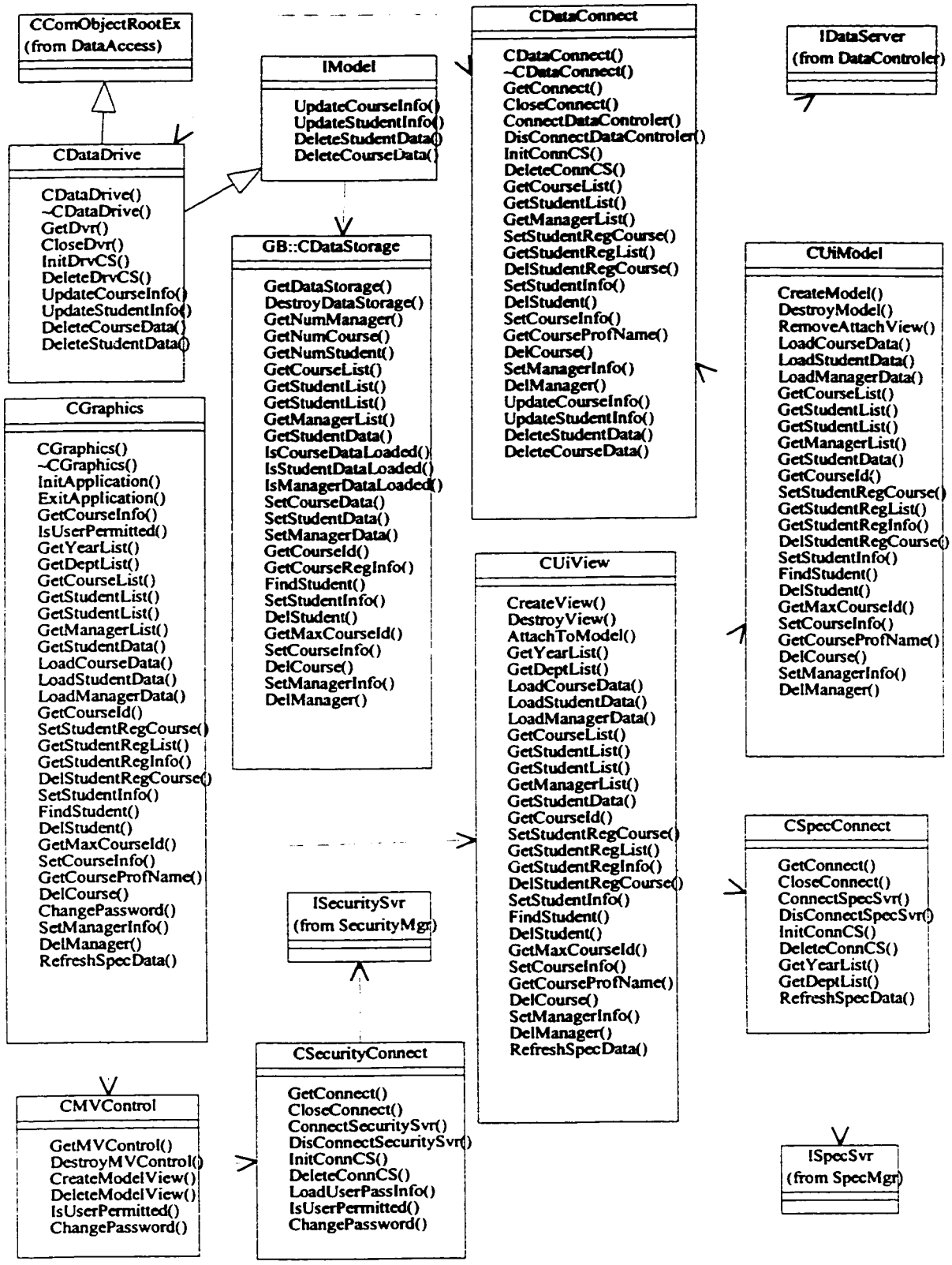


Figure 9: Class Diagram for GuiBase Module

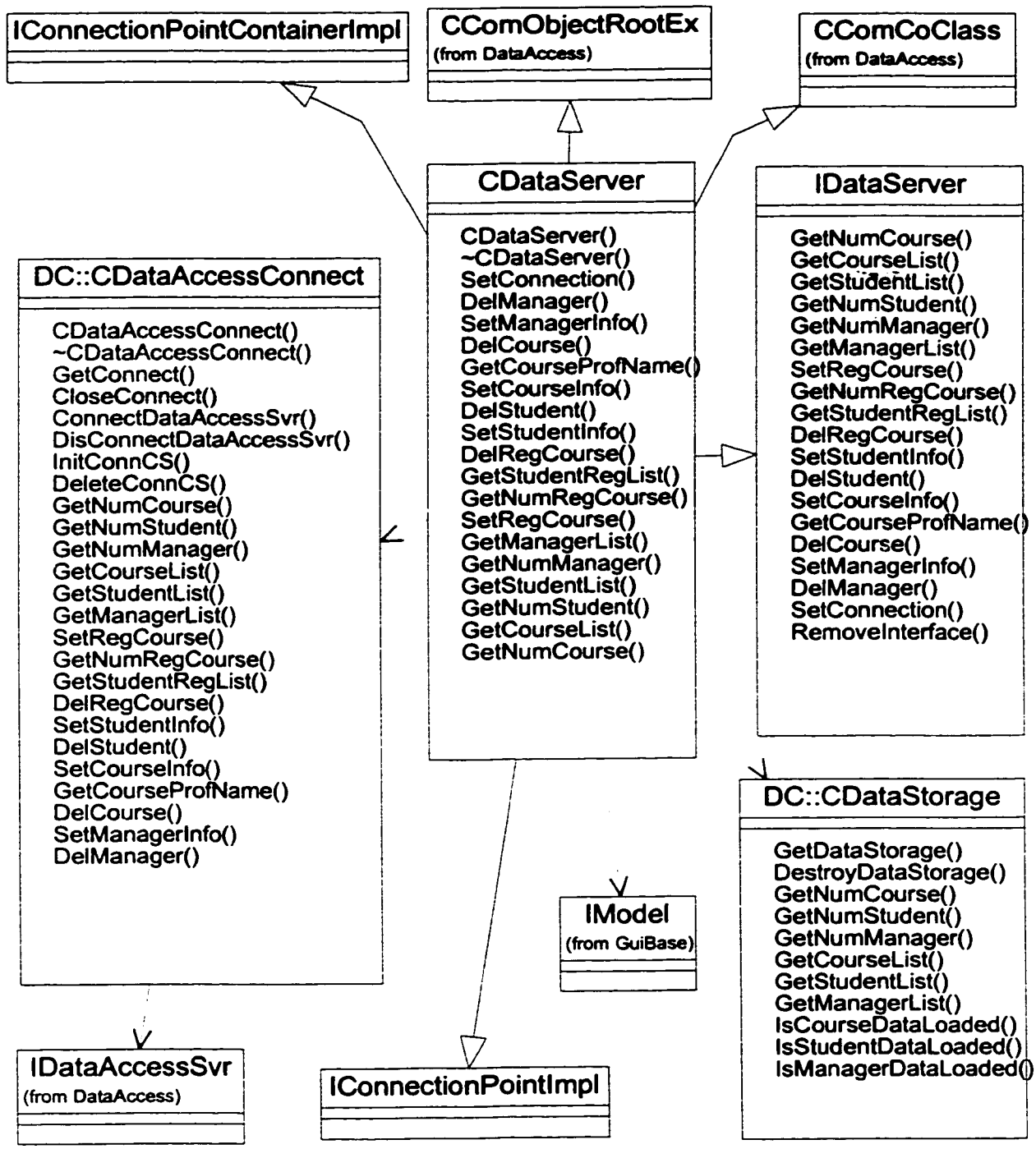


Figure 10: Class Diagram for DataControler Module

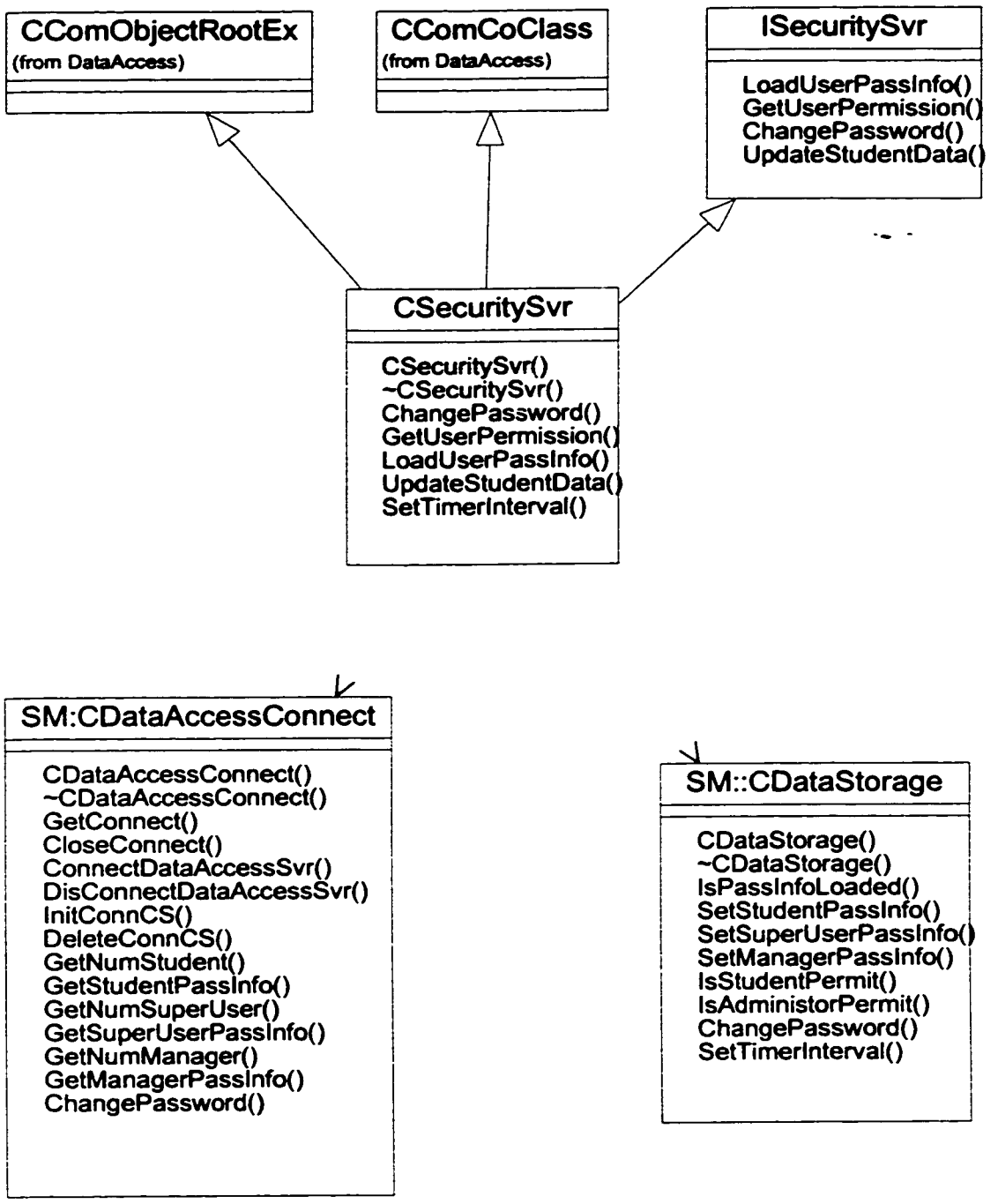


Figure 11: Class Diagram for SecurityMgr Module

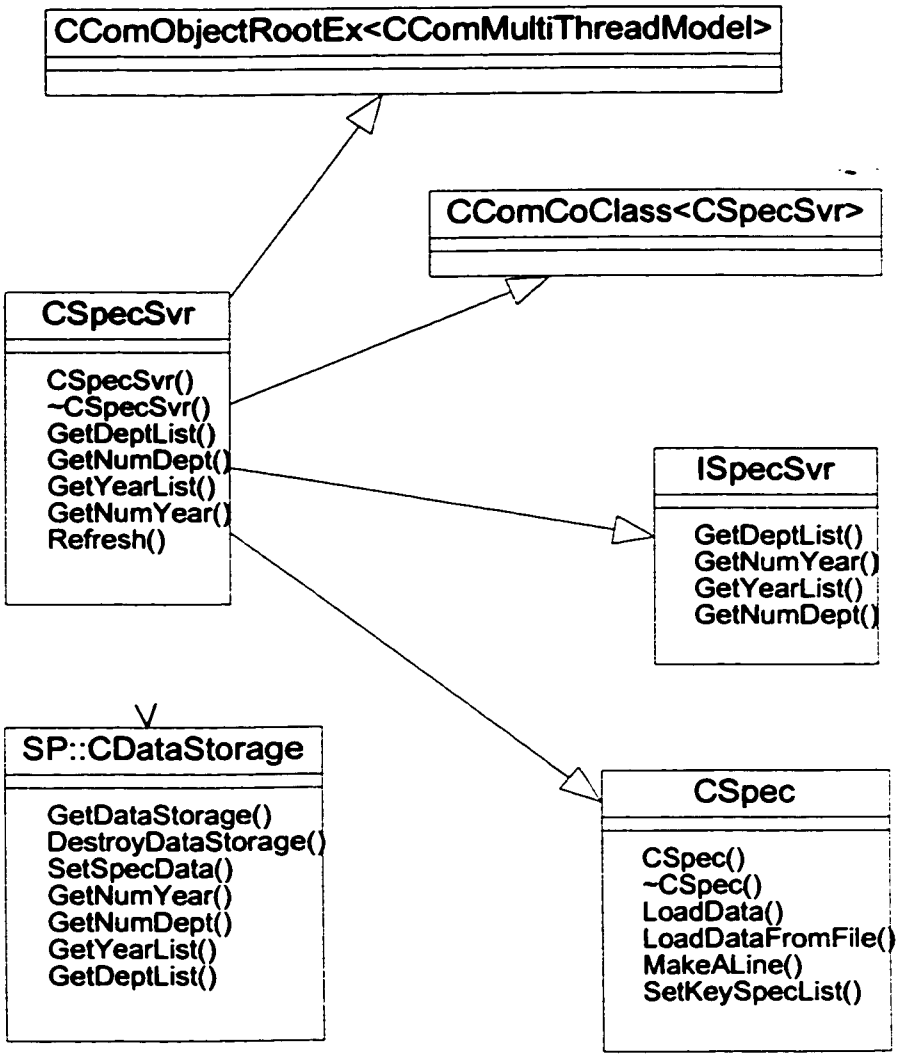


Figure 12: Class Diagram for SpecMgr Module

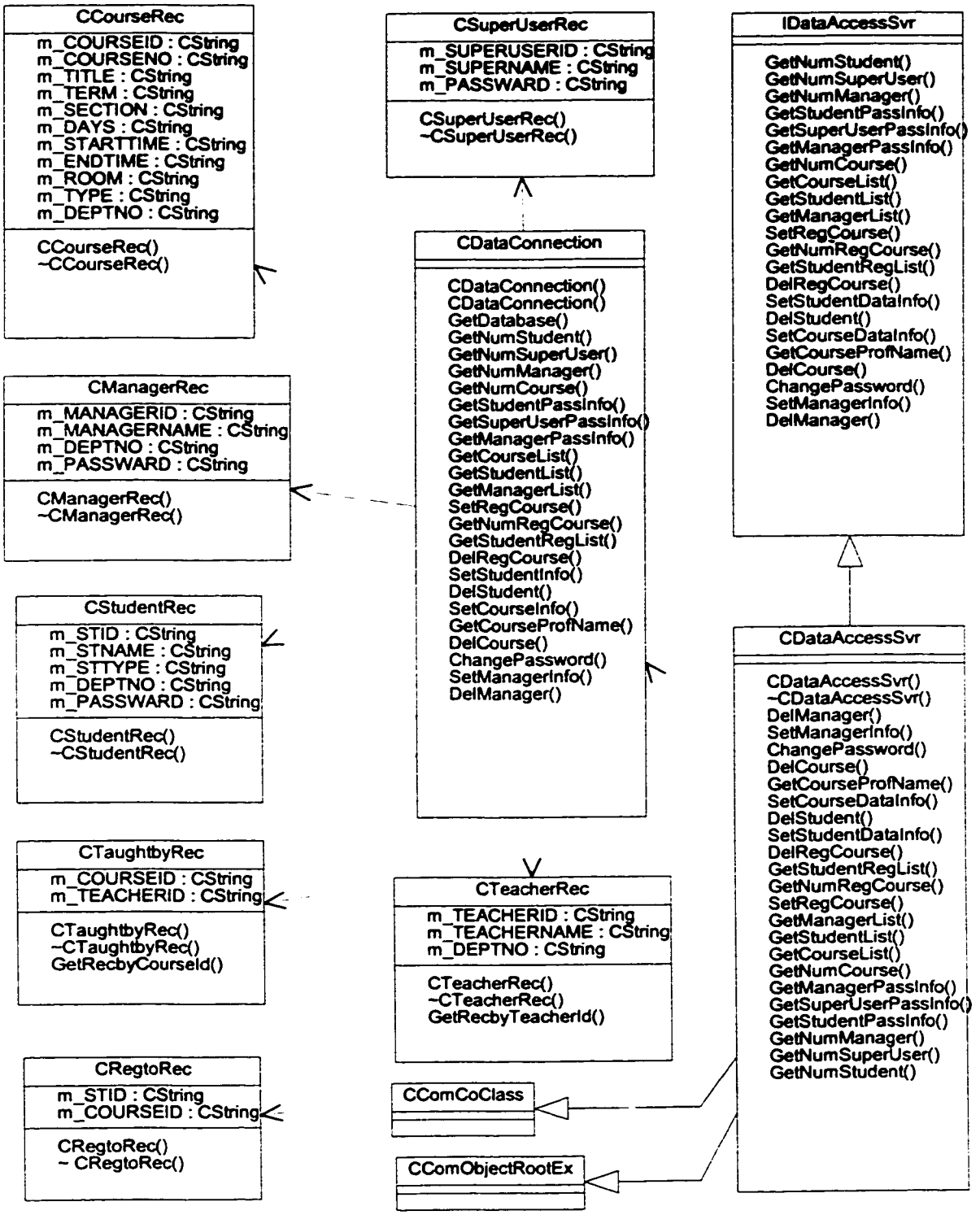


Figure 13: Class Diagram for DataAccess Module

the other graphic objects. It also initiates the other parts of the system by calling into *GuiBase* Module through the exported class *CGraphics*.

- The class diagram for *GuiBase* module is presented in Figure 9. Inside GuiBase module, only *CGraphics* class is exported to the outside world and thus can be accessed by the *GUI* module. At the initial stage, all the related objects are properly created. When a function call is made from *GUI* module to a function defined in the exported class *CGraphics* of *GuiBase* module, the *CGraphics* calls into other objects of *GuiBase* to get the services requested by the *GUI* module. These other objects may further request the middle-tier services by calling the public functions defined in *IDataServer*, *ISecuritySvr*, and *ISpecSvr*. These public interfaces are respectively implemented in *DataControler* module, *SecurityMgr* module, and *SpecMgr* module.
- The class diagram for *DataControler* module is presented in Figure 10. When a data request is received from *GuiBase* module, it will first search its data storage in *CdataStorage* class. If necessary it will call the public functions defined in *IDataAccessSvr*, which is implemented in *DataAccess* module. It then returns the requested data to *GuiBase* module. In some case, the *DataControler* module needs to call back to the *GuiBase* module through the public functions defined in *IModel* interface.
- The class diagram for *SecurityMgr* module is presented in Figure 11. When a password information is requested from *GuiBase* module, it will first search its data storage in *CDataStorage*. If data not found, it will call the public functions defined in *IDataAccessSvr* to get the password information and return them to *GuiBase* module.

- The class diagram for *SpecMgr* module is presented in Figure 12. When a request for configuration and help information is received, it will search its data storage in *CDataStorage*. If necessary it will load the requested data from files and return the data to *GuiBase* module.
- The class diagram for *DataAccess* module is presented in Figure 13. When data request is received from either *DataControler* or *SecurityMgr*, it will get the data records from the database and return them to the caller.

4.6 System Design — Dynamic Model

We present sequence diagrams for the major use case scenarios. These are respectively displayed in Figures 14, 15, 16, 17, 18.

- The use case sequence diagram for loading an application is presented in Figure 14. The use case starts from a call to *InitApplication()* which in turn creates *Model* and *View* objects. Then it establishes the connections with middle-tier servers by calling *CoCreateInstanceEx()*.
- The use case sequence diagram for exiting an application is presented in Figure 15. The use case starts from a call to *Exit()* which in turn deletes *Model* and *View* objects. It then disconnects middle-tier servers from *GuiBase* by calling *Release()*.
- The use case sequence diagram for user login is presented in Figure 16. The use case starts from user entering password information. Step by step, it goes into *SecurityMgr* to check the user permission by calling *IsUserPermitted()*, and create the user window if permission is obtained.

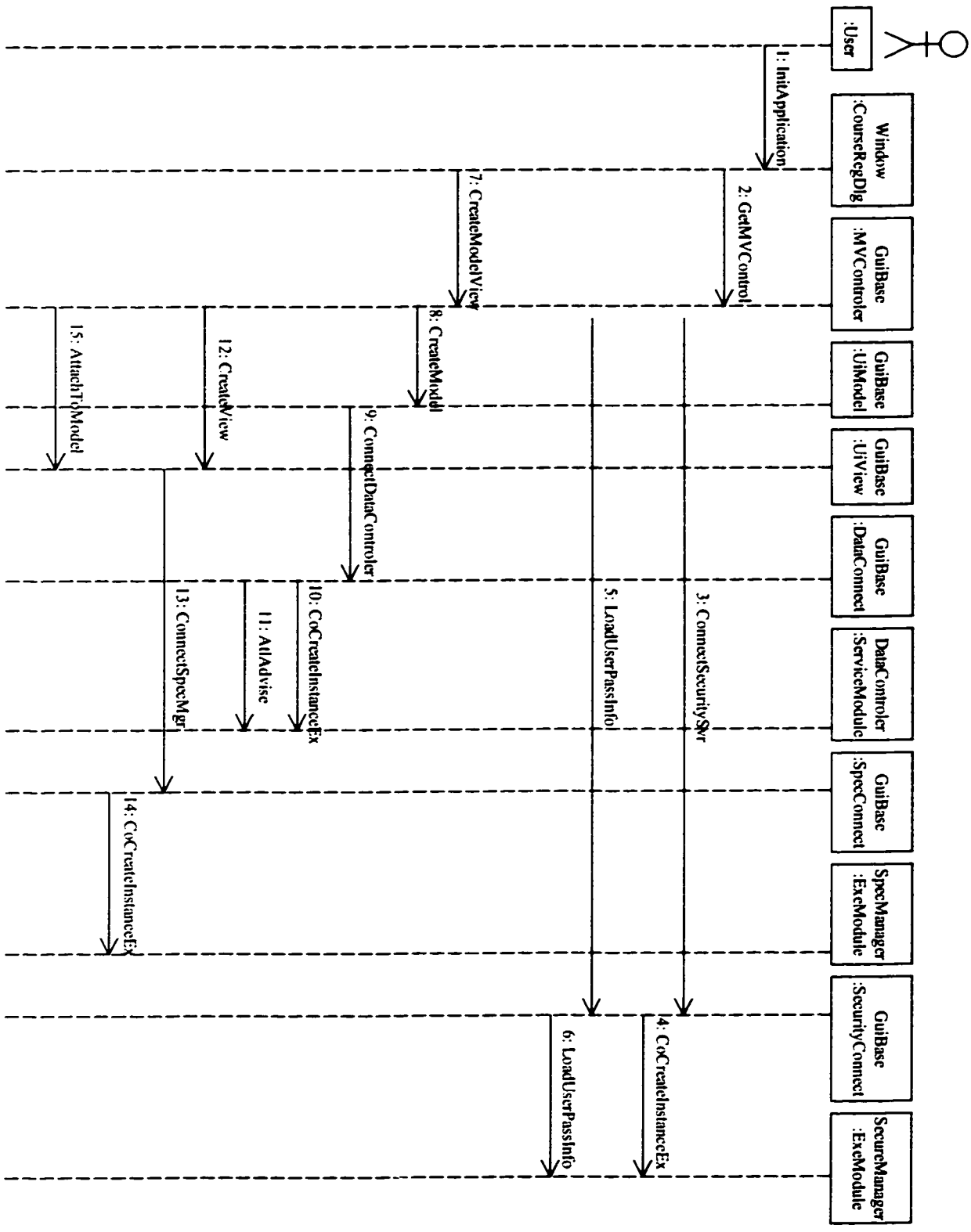


Figure 14: Use Case - Sequence Diagram: Load Application

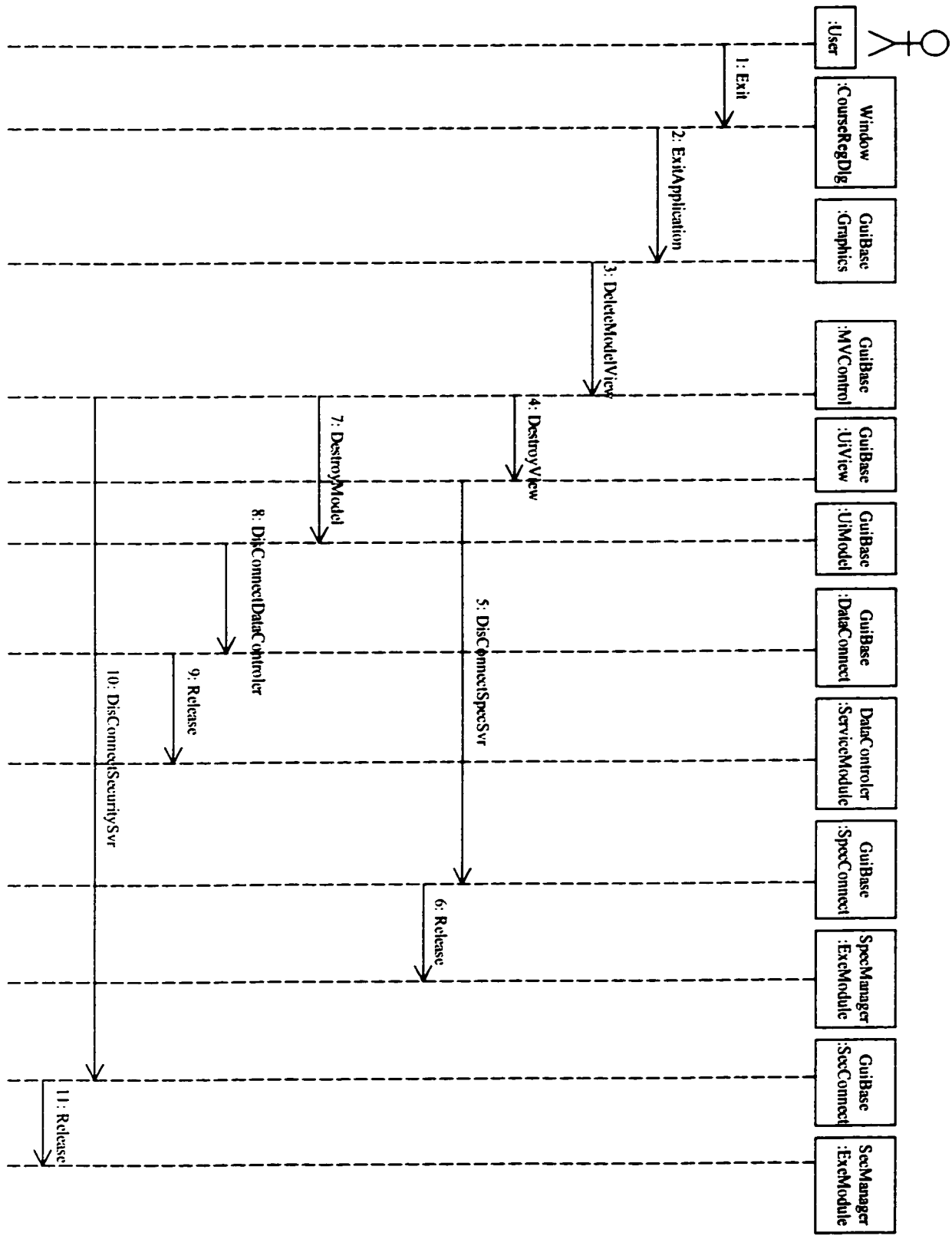


Figure 15: Use Case - Sequence Diagram: Exit Application

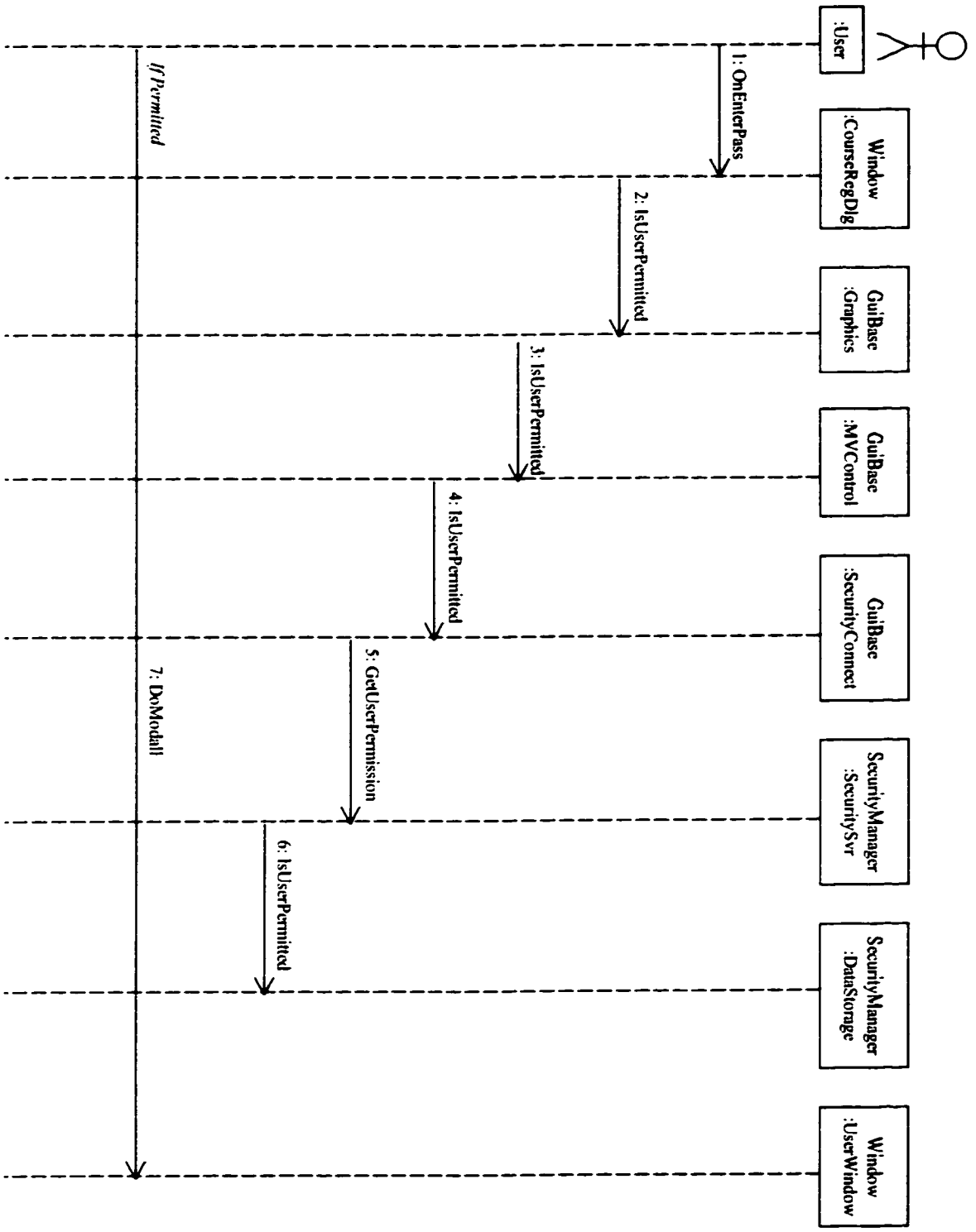


Figure 16: Use Case - Sequence Diagram: User Login

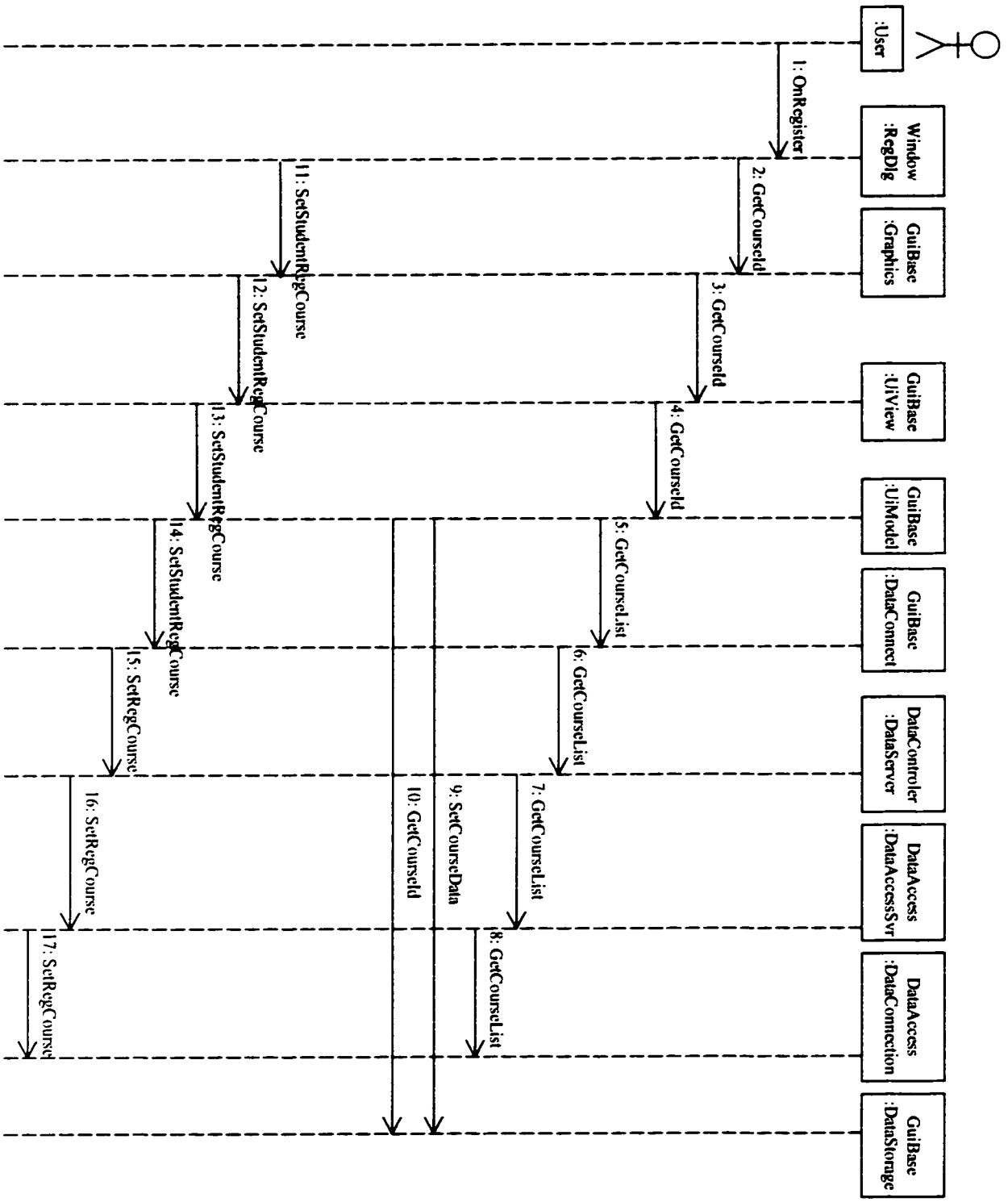


Figure 17: Use Case - Sequence Diagram: Student Register Courses

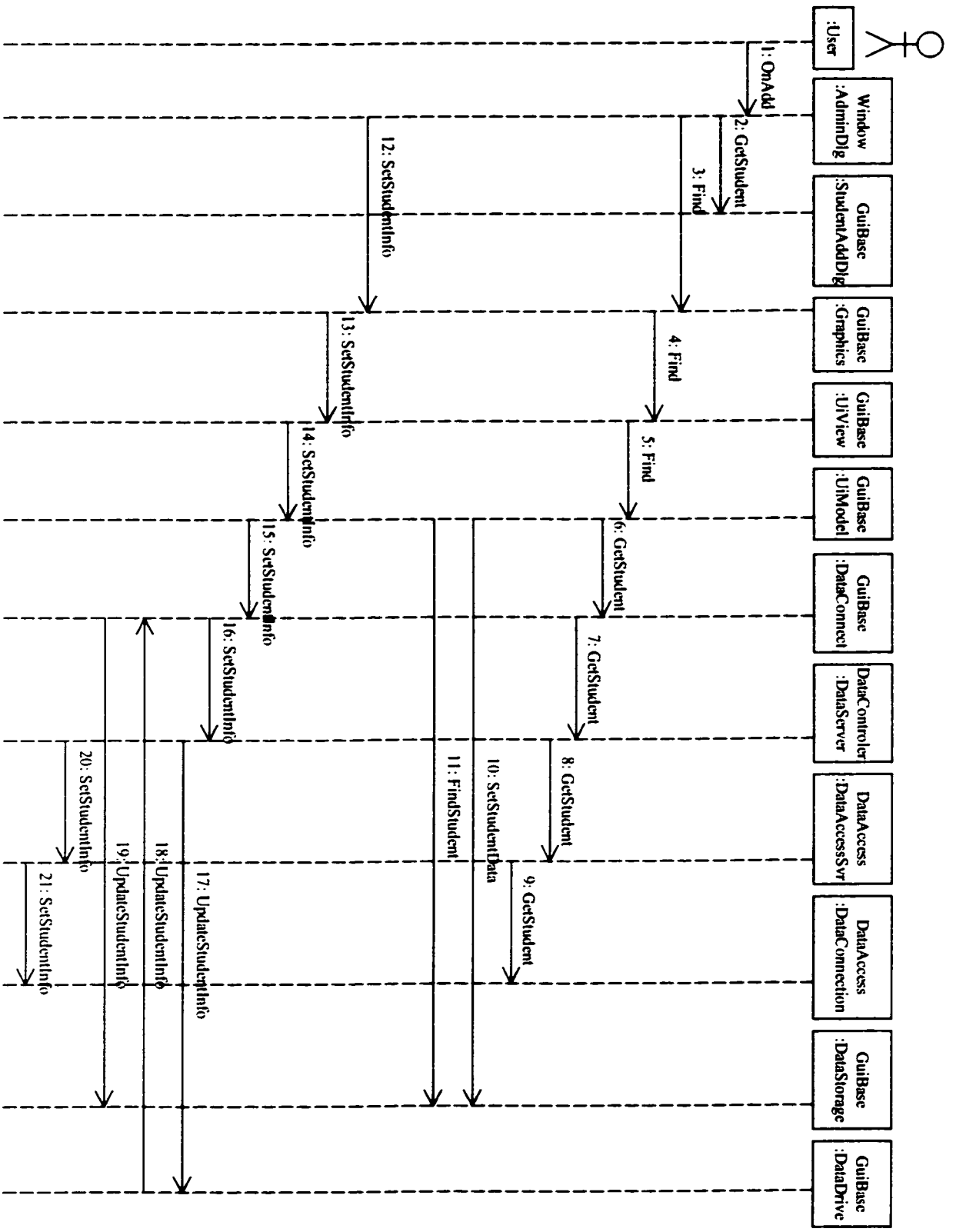


Figure 18: Use Case - Sequence Diagram: Add a Student

- The use case sequence diagram for student registering courses is presented in Figure 17. The use case starts from user entering register command. The student registration data is transferred to the data-tier server through middle-tier *DataControler* server. Finally the data is set into database by the data-tier server.
- The use case sequence diagram for adding a student by the administrator is presented in Figure 18. The use case starts from user adding a new student. The student data is retrieved from input. These data are checked with the existing data by calling *FindStudent()* to avoid adding redundant data into database. The data is then sent to the data-tier server through middle-tier *DataControler* server and finally set into database. To maintain consistency, an update of the student data is made from *DataControler* server to all the connected clients by calling *UpdateStudentInfo()*.

4.7 Data Flow

Figure 19 describes how the data is retrieved and stored between the three tiers in the distributed system. When the application is started, the login information is loaded from the data server via middle-tier *SecurityMgr* server and the login window is presented to the user. After the user types in the login information, the system compares the entered information with loaded login information. A successful match will bring the user to the next window depending on the user type. If user is a student, the system will load the student and related course information from the data server via middle-tier server *DataControler* and presented to the student with registration window in which the loaded student and related course information are displayed. Upon student's request, the registration data will be stored into database via middle-tier *DataControler* server and

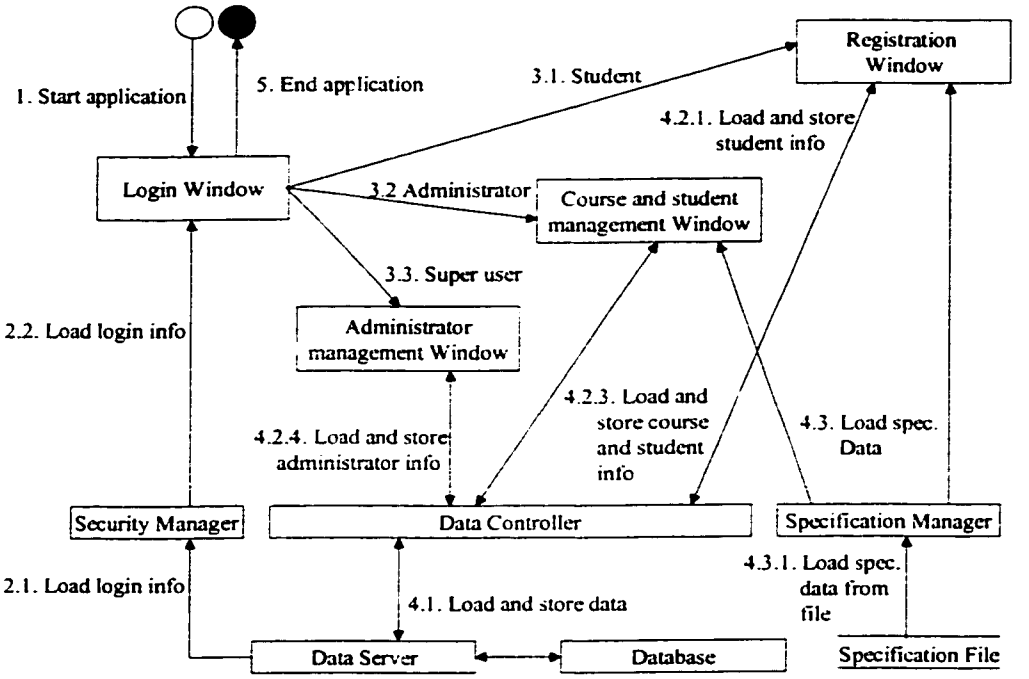


Figure 19: Data flow and sequence diagram

data-tier *DataAccess* server. If user is an administrator, the system will load the total students and total courses information from database and presents to the administrator with student and course management window in which the loaded information is displayed. Any information relating to the adding and removing of student and course data will be stored into database through *DataControler* and *DataAccess* servers. If user is a super user, the system will load administrator information from the database and presents to the super user with administrator management window in which the loaded administrator information is displayed. Any information relating to the adding and removing of administrator data will be stored into database.

5 System Implementation

The system implementation directly determines the reliability and usability of the UCRMS. An efficient implementation of graphic user interface is the key to the effective use of the system, especially for commercial software since the graphic user interface provides the only way for the interaction between the system and its users. To achieve a reasonable system performance, the server components must be implemented carefully. An analysis of the system features is also needed for a complete implementation of the system.

5.1 System Characteristics

As mentioned earlier, UCRMS is designed to be a real-time, multi-user supported, distributed system. Thus it is advantageous to use C++, Visual C++, distributed component object models (DCOM) to implement the whole system. In some cases, multithread technology should be used to improve the system response time. To maintain the system performance as number of the user increases, the UCRAMS must support system scalability and load balancing.

5.2 Implementation Details

In this section, we respectively explain and examine the implementation of *GUI* module, *GuiBase* module, *DataControler* module, *SecurityMgr* module, *SpecMgr* module, and *DataAccess* module in details. Corresponding to six modules, Six different projects are constructed by using Microsoft Visual C++.

5.2.1 GUI module — the Graphic User Interface

Based on the user descriptions and task analysis, we choose dialog-based interactive style for the interactions between the user and the system. The advantages of this choice are their simplicity. These on-screen controls provide contextual information for the user, allowing them to make a related set of choices, choose from a set of options that may change depending on the context, and acknowledge a piece of information before proceeding. To support a clear presentation of the courses and the students data, an Active X component called Microsoft Gridline Control is used in the construction of the graphic user interface. The Active X component can be either directly loaded into dialog panel at the design time, or it can be created at run time according to the situation. For the UCRMS, since the number of columns needed to present the data in gridline control is known beforehand, therefore simply loading the Gridline control at design time is all we need. To reduce the data transmission over the network, only login information is loaded into the system from the database at the starting of the UCRMS.

One of the functionality implemented in *GUI* module is to help *SecurityMgr* module and *SpecMgr* module to achieve online data-update (details will be given when discussing the implementations of these modules). For this purpose, we implement a timer mechanism as shown below:

```
void CCourseRegDlg::OnTimer(UINT nIDEvent)
{
        RefreshSpec();          //Send refresh signal for update Specification Manager
        RefreshSecurity();      //Send refresh signal for update Security Manager
        CDialog::OnTimer(nIDEvent);
}
```

After a predefined time interval, the client will launch a refresh signal. Upon receiving the refresh signal, the *SecurityMgr* and *SpecMgr* will update their data storage. The

implementation of *GUI* module is completed according to the class definitions given in Figure 8.

5.2.2 GuiBase Module

The *GuiBase* module is the essential part of the user process. It is built as a Dynamic Link Library, and will be loaded by the *GUI* process once the UCRMS application is launched by the user. It serves as the base module for the *GUI* module. It provides all the necessary operations for the *GUI* module to complete the user tasks. It is responsible for establishing the connections with the middle-tier servers. To get the best system performance *GuiBase* also keeps the essential data of the UCRMS in its local memory.

One of the classes, *CCourseRegDlg*, defined in *GUI* module is inherited from class *CGraphics* in *GuiBase* module. Thus every protected or public methods in *CGraphics* class are parts of members in *CCourseRegDlg* class. At the initial stage of the UCRMS, *CCourseRegDlg* is created by the system, which in turn calls *InitApplication()* defined in *CGraphics*. The primary task of *InitApplication()* is to create the model and view through Model-View Controller, it then establishes the connections with three middle-tier servers — the *DataControler*, the *SecurityMgr*, and the *SpecMgr* by calling *CoCreateInstanceEx()* to create the remote server object. *CoCreateInstanceEx()* will return status information in HRESULT type. This status information tells us whether or not the remote server object is created successfully. We can use the following code to test its status:

```
HRESULT hr = CoCreateInstanceEx(...);
if (SUCCEEDED(hr)) {
        // continue
}
```

```
else {
        // error handling and terminate program
}
```

Although in most of the cases, *CoCreateInstanceEx()* and some other functions return zero for signaling success, it is not safe to simply test if it is zero since there are some other cases in which successful operations return nonzero. In all cases, we should use macro *SUCCEEDED* or *FAILED* to test the operation status.

For ordinary middle-tier servers, such as *SecurityMgr* and *SpecMgr*, only a set of application services are provided and no call-back is necessary since the data managed by these servers are not shared by the other users, or they are shared but can be easily updated by other mechanism. The only exception in the UCRMS is the *DataControler* server that manages the dynamically changeable data and needs a callback mechanism to update the data stored in the user computers for maintaining the data consistency. To support middle-tier *DataControler* server to call back to the user module — the *GuiBase* module, we need to implement a *DataDrive* class in *GuiBase* module. *DataDrive* class must inherited from *CComObjectRootEx* to become a DCOM component that receives the callbacks from *DataControler* server.

Currently there is no direct support from Microsoft "ATL COM AppWizare" for setting up the callback interface. Therefore some deep knowledge about the DCOM and its working mechanism is required for a successful implementation of callback interface. Here we try to explain in details about the implementation process.

We begin the implementation of the callback mechanism with *DataControler* module, which will be discussed next. When a project for a DCOM server is built for the first time, "ATL COM AppWizard" will generate an IDL (interface definition language) file (see Appendix A). All the public operations supported by the server should be

defined in the IDL file. The interface and its supported public operations can be added into server project through studio tools provided by the Microsoft Visual Studio. All these public operations are generated by the AppWizard as pure virtual functions and must be implemented by its *CoClass* [2]. To make call back available, *DataControler* module should know the public interface supported by *GuiBase* module. The most convenient way to do that is to add the interface definition into the IDL file of *DataControler* module, and implement these pure virtual functions on the side of user module, i.e. in the class *DataDrive* of *GuiBase* module. Thus the *GuiBase* module can receive the calls made from the *DataControler* module. But the interface definition for the callback should be added manually without the help of Visual Studio tools. If Visual Studio tools are used for adding public interface in *DataControler* module, the tools will regard the newly added interface as one of *DataControler* interfaces rather than *CuiBase* supported interface. As a result, implementation code is generated on the server side rather than on the client side. The consequence is the failure for establishing a mechanism that allows *GuiBase* module to receive the call from *DataControler* module. The following codes are added into IDL file of *DataControler* manually:

```
[
        object,
        uuid(63B3BFA0-7BE2-11d2-819B-000000000000),

        helpstring("IModel Interface"),
        pointer_default(unique)
]
interface IModel : IDispatch
{
        [helpstring("method UpdateCourseInfo")]
        HRESULT UpdateCourseInfo([in] CourseInfo* courseInfo);
        [helpstring("method UpdateStudentInfo")]
        HRESULT UpdateStudentInfo([in] StudentInfo* studentInfo);
        [helpstring("method DeleteStudentData")]
        HRESULT DeleteStudentData([in] ULONG studentId);
        [helpstring("method DeleteCourseData")]
        HRESULT DeleteCourseData([in] ULONG courseId);
```

```
        [helpstring("method SetCourseSpaceFull")]
        HRESULT SetCourseSpaceFull ([in] ULONG courseId);
};
```

In the language of COM/DCOM, the class *DataDrive* that implements the pure virtual functions defined in the IDL of *DataControler* is called *CoClass* of that defined interface. To become a DCOM component, the class *DataDrive* must inherit from *CComObjectRootEx<CComMultiThreadModel>* that are responsible for handling the implementation of the *IUnknown* methods and threads handling methods. The class *DataDrive* must also inherit from *IDispatchImpl<IModel, &IID_IModel, &LIBID_DATACONTROLERLib>* that involves interface *IModel* defined in IDL of *DataControler*. The purpose of this inheritance is to make the class *DataDrive* become an automation server that facilitates the callback. This inheritance also allows the *DataDrive* to implement the pure virtual functions defined in *IModel* of *DataControler* server.

The next thing to do is to add a COM map in the head file of the class *DataDrive*:

```
BEGIN_COM_MAP(CDataControlerDriver)
        COM_INTERFACE_ENTRY(IUiModel)
        COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

This macro-wrapped structure keeps track of the interfaces exposed by *DataDrive* class. Each interface exposed by *QueryInterface()* must have an entry in the COM map. The primary functionality of COM map is to tell the interface where the implementation code is located. Therefore when a client calls the methods in the interface, it will be able to map to its implementation by the COM map.

To implement the pure virtual functions defined in the IDL of *DataControler* server, we must use a standard macro to redefine them in the head file of *DataDrive* class:

```
STDMETHOD(UpdateCourseInfo)(/*[in]*/ CourseInfo* courseInfo);
STDMETHOD(UpdateStudentInfo)(/*[in]*/ StudentInfo* studentInfo);
STDMETHOD(DeleteCourseData)(/*[in]*/ ULONG courseId);
STDMETHOD(DeleteStudentData)(/*[in]*/ ULONG studentId);
STDMETHOD(SetCourseSpaceFull)(/*[in]*/ ULONG courseId);
```

We must do the implementation in the implementation file of DataDrive class:

```
STDMETHODIMP CDataDrive::UpdateCourseInfo(CourseInfo* courseInfo)
{
        CDataConnect::GetConnect()->UpdateCourseInfo(courseInfo);

        return S_OK;
}

STDMETHODIMP CDataDrive::UpdateStudentInfo(StudentInfo* studentInfo)
{
        CDataConnect::GetConnect()->UpdateStudentInfo(studentInfo);

        return S_OK;
}

STDMETHODIMP CDataDrive::DeleteStudentData(ULONG studentId)
{
        CDataConnect::GetConnect()->DeleteStudentData(studentId);

        return S_OK;
}

STDMETHODIMP CDataDrive::DeleteCourseData(ULONG courseId)
{
        CDataConnect::GetConnect()->DeleteCourseData(courseId);

        return S_OK;
}

STDMETHODIMP CDataDrive::SetCourseSpaceFull(ULONG courseId)
{
        CDataConnect::GetConnect()->SetCourseSpaceFull(courseId);

        return S_OK;
}
```

Here STDMETHOD and STDMETHODIMP are standard macros from Microsoft ATL that allows the software developer to implement the user defined COM/DCOM interface.

One tricky situation might arrive that involves *DataDrive* calling *Release()* of *DataControler* and *DataControler* calling *Update()* of *DataDrive* in *GuiBase*. Suppose *DataControler* is trying to call *UpdateStudent()* to one of the client, and that client

decides to quit. So the client calls *Release()* of *DataControler*, and then destroys itself. That will leave *DataControler* calling an object that already destroyed. To tackle this problem, *DataControler* must block the *Release()* of a particular client when making *UpdateStudent()* for that client. This problem will be discussed further when discussing the implementation details of *DataControler* module.

As we will see in the implementation of *SpecMgr* server and SecurityMgr server, no callback mechanisms are provided for these servers. To do the online update for specification data and security data, a timer mechanism is implemented in the GUI module, which finally calls *RefreshSpec()* in the class *SpecConnect* and *RefreshSecurity()* in the class *SecurityConnect* of the *GuiBase* module:

```
void CSpecConnect::RefreshSpec()
{
        HRESULT hr = 0;
        if (m_pSpecSvr != NULL) {
                hr = m_pSpecSvr->Refresh();
                if (FAILED(hr)) {
                        TCHAR errBuf[128];
                        sprintf(errBuf, "RefreshSpec failed with error code %x", hr);
                        AfxMessageBox(errBuf);
                        return;
                }
        }
}

void CSecurityConnect::RefreshSecurity()
{
        HRESULT hr = 0;
        if (m_pSecuritySvr != NULL) {
                hr = m_pSecuritySvr->Refresh();
                if (FAILED(hr)) {
                        TCHAR errBuf[128];
                        sprintf(errBuf, "RefreshSecurity failed with error code %x", hr);
                        AfxMessageBox(errBuf);
                        return;
                }
        }
}
```

The implementation of *GuiBase* module is completed according to the detailed class diagram presented in Figure 9.

5.2.3 DataControler Module

DataControler server is one of the middle-tier servers and is the essential component of the UCRMS. It provides most of the application services required to complete the user tasks. All the dynamically changeable data is managed by the *DataControler* server and the amount of these data well exceeds 2/3 of the total data managed by middle-tier servers. In addition, *DataControler* server provides more application services than the other middle-tier server does. As a result, *DataControler* might need to run in a more powerful PC in which all the system sources are reserved for that server.

When implementing DCOM server, one must decide which type of server is most appropriate. One type of server, the ordinary DCOM server, always unloads itself when no more client connecting to it. The other type of server is the so-called NT service, a persistent server that exists even when there is no more clients connecting to it. The advantage of ordinary DCOM server is its automatic release of system resource. This characteristic is crucial for an ordinary PC where system resource is limited. The benefit of the NT service server is evident in reducing the number of network data transmission. Assume only one client is currently connected with a DCOM server of NT service type, and assume that the required data is already loaded into DCOM server. After finishing the user task, the client calls *Release()* of the server and quits. Although there is no more client connection to the server, the server does not unload itself and the data managed by the server still exists. When a client makes a new connection to the server, the loaded data can be reused. Consequently new network data transmissions are avoided by using persistent DCOM server. If ordinary DCOM server is used, all the data kept in the

DCOM server will be destroyed when server unload itself in case of no more client connection. At that moment when a client makes a new connection, the server must reload the data through the network transmissions.

By taking into account the amount of data managed by the *DataControler* server, we implement *DataControler* server as NT services to avoid the overloading to the network. To allow the *DataControler* server to call back to the client, the client must send the interface pointer of itself to the connected server by calling *RegisterInterfaceToList()* on the server side. After finishing the user tasks, the client must delete the interface pointer registered on the server side to avoid server updating the destroyed object. The *DataControler* server implements *RemoveInterface()* to serve this purpose.

Another implementation detail deserve to be discussed is the possible memory leak caused by concurrent operations of server updating client and client releasing server, just as mentioned in the previous section. To solve this problem, the client calls *RemoveInterface()* of *DataControler* server before calling *UnAdvise()* of *DataControler* server at the time of quit. We use critical section to lock the operations of server updating client and client releasing server. Specifically, if the server is updating the client, the operation of client releasing server is blocked; on the other hand, if the client is releasing the server, the operations of server getting pointer to the client object and updating client are blocked. When client finishes the call of releasing the server, the server should already have removed the client from its client list, and the subsequent update will only update the client still existing in the client list. Thus the possible updating of destroyed

client object is avoided. To protect the client from releasing server while active update is in effect, we use critical section as shown in the following codes:

```
STDMETHODIMP CDataControler::RemoveInterface(IModel *pIModel)
{
        POSITION pos = m_interfaceList.GetHeadPosition();
        while (pos != NULL) {
                POSITION oldPos = pos;
                IModel* pIListModel = m_interfaceList.GetNext(pos);
                If (pIListModel == pIModel) {
                        EnterCriticalSection(&m_clientISect);
                        m_interfaceList.RemoveAt(oldPos);
                        LeaveCriticalSection(&m_clientISect);
                        return S_OK;
                }
        }
        return S_FAILED;
}
```

To protect the server from updating client while active release is in effect, the same critical section is used in *DataControler* server for updating the client data:

```
EnterCriticalSection(&m_clientISect);
POSITION pos = m_interfaceList.GetHeadPosition();
HRESULT hr = 0;
while (pos != NULL) {
        IModel* pIModel = m_interfaceList.GetNext(pos);
        hr = pIModel->UpdateCourseInfo(pCourseInfo);
        if (FAILED(hr)) {
                char errBuf[128];
                memset(errBuf, '0', 128);
                sprintf(errBuf, "UpdateCourseInfo failed with error code %x", hr);
                AfxMessageBox(errBuf);
                LeaveCriticalSection(&m_clientISect);
                return hr;
        }
}
LeaveCriticalSection(&m_clientISect);
```

To detect the possible abnormal termination of client process, we apply "*try*" and "*catch*" pair to all the client update operations in order to catch the potential system error. The implementation of *DataControler* module is performed based on the class definitions presented in Figure 10.

5.2.4 SecurityMgr module

SecurityMgr is a middle-tier server responsible for checking user login, setting user info, and changing user password when requested. Upon receiving the client request for password information, *SecurityMgr* will look at its local memory whether or not the password information is already loaded from the data server. In case the password information is not loaded, it will loads the password information from data server, sets the password information into its local memory, and then return password information to the client. *SecurityMgr* only manages a small amount of data — the password information, so it is implemented as ordinary DCOM server. When no more active client is connected to *SecurityMgr*, it will unload itself to release the system resource. Although the next client request will cause a new data transmission, the workload imposed to the network will not be too heavy since the amount of data transmitted over the network is not too big. Less than one Megabytes data are needed to transfer over the network.

SecurityMgr is an ordinary DCOM server. It supports automatic release of system resource, but it does not support the client callback. The reason is that SecurityMgr server does not have knowledge about when to make active update. All the data modification is managed and set into the database by the DataControler server, so only the DataControler server knows when to make active update. Without knowing when to make active update, how does SecurityMgr handle the update of the password information modified by the administrators? The easiest way to do, without wasting too much system resource, is to implement a *Refresh()* function that is called by the connected clients at their timer event handler. Inside *Refresh()* function we can implement an operation to empty the password information that is kept in the local memory of *SecurityMgr* server. Thus the next client

request for the password information will cause *SecurityMgr* to reload the password information from data server since the loaded password information stored in the *SecurityMgr* is already emptied by the *Refresh()* operation. As a result, the updated password information will be loaded into local memory of *SecurityMgr*. This timer mechanism is implemented on the client side in *GUI* module as mentioned in the implementation of *GUI* module. The predefined time interval for the timer event is obtained from the *SpecMgr* server that reads the data from the configuration file. The predefined time interval should be determined according to the system requirements. If a new student account needs to be activated six hours after an administrator set up the account for that student. Then the timer should be set with the predefined time interval equal to six hours. The timer event actually calls *Refresh()* of *SecurityMgr* as shown below:

```
void CDataStorage::Refresh(UINT nIDEvent)
{
        POSITION pos = m_studentPassList.GetHeadPosition();
        while (pos != NULL) {
                delete m_studentPassList.GetNext(pos);
        }
        m_studentPassList.RemoveAll();
        pos = m_superUserPassList.GetHeadPosition();
        while (pos != NULL) {
                delete m_superUserPassList.GetNext(pos);
        }
        m_superUserPassList.RemoveAll();
        pos = m_managerPassList.GetHeadPosition();
        while (pos != NULL) {
                delete m_managerPassList.GetNext(pos);
        }
        m_managerPassList.RemoveAll();

        m_timerId = SetTimer(m_key, m_timerInterval, NULL);
}
```

The complete implementation of the *SecurityMgr* module is conducted according to the class definitions given in Figure 11.

5.2.5 SpecMgr module

SpecMgr is a middle-tier DCOM server responsible for the configuration and specification of the system. It also provides the capability of online help for the UCRMS, such as course descriptions. It is independent of data-tier data server and thus independent of database, giving it the flexibility to be configured to run at any computers without the client side configuration for the data server. Thus it can be run either on a single computer without the presence of any other UCRMS components, or on a computer with the presence of other UCRMS components. *SpecMgr* is also an ordinary DCOM server, capable of unloading itself when not used by any client. The data needs to be reloaded from files when new request arrives. The client side configuration and server side configuration will be discussed in the next chapter.

In principle, *SpecMgr* works with the file system. It will load the "configuration" and "help" data into its local memory from files. These data are loaded only once for a lifecycle of *SpecMgr*. To allow the university administrators to edit the files easily, only a simple rule is set to govern the format of the file contents. Part of the "configuration" file "Data.cfg" is shown below (see Appendix C):

!!!!! Data.cfg

```
YEAR : 1998-1999;
YEAR : 1999-2000;
DEPT : 0, (Chemical Engineering);
DEPT : 1, (Civil Engineering);
DEPT : 2, (Computer Science);
DEPT : 3, (Electronic Engineering);
DEPT : 4, (Mechanical Engineering);
TM_STEP : 21600;
```

Part of the "help" file "Help.cfg" is shown below (see Appendix C):

```
COR_NUM : (ELEC629);
COR_DES : COMP628, (COMP 628 - Computer Systems Design
|Prerequisite: COMP 546.
|
```

```
|Migration from Von Neumann to parallel processing
|architectures: fine grained and coarse grained
|concurrency, multi-threaded computers, massively
|parallel computers, fundamental problems in hardware
|architecture, and memory consistency. Embedding of
|algorithms on shared-memory and message-passing
|architectures. Parallel programming supports: a
|parallel program model, embedding of parallel
|programs on multiprocessors and distributed systems.
|Key concepts in distributed systems.);
```

To guarantee that the data stored in the files is loaded only once for the life cycle of *SpecMgr* server, we first test whether or not data is loaded. We load the data from files only if the data is not loaded. By doing so, we can reduce the number of access to the file system, and thus improve the system performance. In addition, the test for data loading and the subsequent loading of data from files must be protected by the critical section to avoid unnecessary file loading caused by two clients sending the requests at the same time and both pass the tests.

For simplicity, *SpecMgr* itself has no support for run-time updating its data loaded from files. Usually the configuration and specification file is the essential part of the UCRMS, so these files must be ready before UCRMS can be used. Therefore it's natural that no online support is given for the modification of these files. But usually the course administrators should be allowed to modify the "help" files dedicated for the course descriptions. The working mechanism of *SpecMgr* makes these modifications impossible to be loaded into its memory by *SpecMgr* itself to reflect these new changes. The reason is simple: *SpecMgr* server works with files and file modification is unable to send signals to the *SpecMgr* to inform the data update. To support online update of "help" files, we use timer mechanism to perform the runtime update. This timer mechanism is invoked and controlled by the client process. At each elapse of timer event, client calls *Refresh()* of *SpecMgr*, which then reloads the "help" information from the file:

```
STDMETHODIMP CSpecSvr::Refresh()
{
        LoadData();
        return S_OK;
}
```

SpecMgr server works with files, and these files may be located either on a local drive or on a network drive. Any changes concerning the file location should not require the re-compilation of one or more components of the UCRMS. For this purpose, we use the environment variable setting and *GetEnvironmentVariable()* function to ensure the file location transparency, the codes are shown below:

```
LPTSTR fileName = NULL;
DWORD len = GetEnvironmentVariable(_T("SPEC_CFG"), NULL, 0);
if (len > 0) {
        fileName = new TCHAR[len + 1];
        GetEnvironmentVariable(_T("SPEC_CFG"), fileName, len + 1);
}
```

At the time of configuration, we can set the system environment variables for these files. The complete implementation of *SpecMgr* module is conducted according to the class definitions presented in Figure 12.

5.2.6 DataAccess Module

DataAccess is a data-tier server responsible to offer the data services for the middle-tier servers. To guarantee the safe operation of database access from outside of the data server, the database operations and queries should be encapsulated in the data server. To do so, we implement *DataAccess* server to expose a set of data services and functions. Database is accessed by the outside world through the public interface and is operated only inside these exposed data services. Thus if these services and functions are carefully coded, the safe database operations are then guaranteed.

As discussed in the previous sections, middle-tier servers are implemented to minimize the data base access by keeping the data into their local memory. For this reason, *DataAccess* does not maintain a memory to keep the data retrieved from database. Once database service is requested, it opens database and makes corresponding database operation. It immediately closes the database after finishing the database services to reduce the possibility of database errors.

Here we want to further mention the implementations of *DataControler* server and *SecurityMgr* server since they are closely related with *DataAccess* server. As described above, the number of database access is very limited, we can then implement the *DataControler* server and *SecurityMgr* server to set up the connection to *DataAccess* server just before calling its data services, and release the connection immediately after getting the data services. Thus the number of active database connections is very limited at any given time. The implementation is performed based on the class definitions presented in Figure 13.

5.2.7 Project Settings

To complete their common tasks, dependency relations must present between these modules. In order to successfully compile and build these modules, we need to make appropriate settings for these different projects.

- *DataAccess* is built as a data-tier server component, only providing the data services to the other modules. So it does not depend on any other modules. As a result, no specific settings are needed to build this module.

- *SpecMgr* module is built as a middle-tier server component. But since it only works with files locating either on the local drive or on the network drive, it does not depend on the other modules. No specific settings are necessary.
- *DataControler* module is built as a middle-tier server component, an NT service. It not only provides the application services to the client, but also requires the data services from the data-tier server when needed. Thus we need to do some settings for *DataControler* module. First of all, we must include a head file named "dataaccess.h" that tells the compiler the complete definition of data-tier services available for the *DataControler* module. To allow the compiler to find the specified include file, we need to add a corresponding path to the project path-include list, which can be done by using sub-menu "options..." of main menu "Tools". Moreover we must add a file named "DataAccess_i.c" into the project, which allows the compiler knows all the interface definitions. This file can be added by using a sub-menu "Add to project" from main menu "Project".
- *SecurityMgr* module is built as a middle-tier server component. It has the similar functionality as *DataControler* component. Therefore same settings need to be performed for *SecurityMgr* module.
- *GuiBase* module is built as a presentation-tier DLL component, responsible for initializing the application, establishing the connections to the middle-tier servers, requiring the needed application services from middle-tier servers, and providing the corresponding data to the graphic user interface. Therefore *GuiBase* module depends on *DataControler* module, *SecurityMgr* module, and *SpecMgr* module. To successfully compile the *GuiBase* project, it must include the head files

"DataControler.h", "SecurityMgr.h", and "SpecMgr.h", respectively from *DataControler* project, *SecurityMgr* project, and *SpecMgr* project, and their corresponding paths must be added into the path-include list of *GuiBase* project. To allow the compiler to know the interface definitions of all middle-tier servers used in the *GuiBase* module, the files "DataControler_i.c", "SecurityMgr_i.c", and "SpecMgr_i.c" must added into the *GuiBase* project.

- *GUI* module is built as a presentation-tier EXE component, responsible for starting the UCRMS application and providing the graphic user interface to the users. It will load "GuiBase.dll" into its workspace and requires application services from the middle-tier servers through the functions implemented in the *GuiBase* module. Thus *GUI* module depends on *GuiBase* module. All the functions accessible to the *GUI* module are exported from the class *CGraphics* of *GuiBase* module. Thus the head file "Graphics.h" needs to be included in the *GUI* project. The corresponding path must also added into the path-include list of the *GUI* project. To successfully compile the project, the compiler also needs to know the external functions provided in the *GuiBase*. For this purpose we need to add "GuiBase.lib" into lib-include list of the *GUI* project, which can be done by adding "GuiBase.lib" into the edit field marked "object/library modules" in the "link" page of project settings.

6 Installation and Execution

After the implementation of the UCRMS is completed, we need to distribute and configure the different components of the UCRMS into their designated computers so that they can work together in a distributed environment.

6.1 The Distribution of GuiBase.dll (dynamic linked library)

At the starting of the UCRMS, the "GuiBase.dll" generated from *GuiBase* module will be loaded into the working space of "CourseReg.exe" application, which is generated from *GUI* module. To allow the "CourseReg.exe" to find the "GuiBase.dll", "GuiBase.dll" must be set in one of the following locations:

- Windows NT system32 directory.
- The directories specified in the path setting of Windows NT system environment.
- The same directory where starting file "CourseReg.exe" locates.

The easiest way is to set the output files of the *GUI* project and the *GuiBase* project to the same common directory. This can be done by entering a common path name in the edit field marked "output files" in the "General" page of their project settings.

6.2 The System Environment Variable Settings

As described above, *SpecMgr* server works with files, and these files may be located anywhere, either on a local drive or on a network drive. To ensure file location transparency, we need to set the environment variables for these files on the computers where *SpecMgr* server is to be launched by the user request. We can set these environment variables by double clicking the system icon on the Control Panel and choosing Environment page. To be appropriate, these environment variables should be set as user variables since the UCRMS is a user application. Click on the user variables field and then type in the required variable and its value as below:

Variable → DATA_CFG
Value → D:\ZQReport\SpecMgr\CFG\Data.cfg

Variable → HELP_CFG
Value → D:\ZQReport\SpecMgr\CFG\Help.cfg

Here the "Value" field must be filled with the full path of the file to be loaded by the SpecMgr server at run time.

6.3 Server side configuration

Each DCOM server must be registered into the system registry of the computers where they are launched upon client's request. Usually the server project is built by "ATL COM AppWizard" to support automatic registration of the server interface and their classes at the compile time. Therefore, if the server is built and is going to run on the same machine, the registration step is not needed for that server. For *DataControler* server, we should repeat the server registration process. This is because that only the ordinary DCOM server can be registered by the automatic registration. On the other hand, if the server is developed and built on one computer and is to be executed on another computer, we also need to do the server registration process on the computer dedicated for that server. If a server is implemented as NT service, it can be either registered as ordinary DCOM server or registered as NT service. Otherwise it can be registered only as ordinary DCOM server. To register *DataControler* as NT service we type the following command at command line:

DataControler /Service

To register *SecurityMgr* as ordinary DCOM server, we use the following command:

SecurityMgr /RegServer

The registration for all the other ordinary DCOM servers uses the same process as that of *SecurityMgr* server.

For *DataControler* server, we can remove the server registration from the system registry by using the following command at command line:

DataControler /UnregServer

All the other servers follow the same unregistration process.

When server registration is done, we have to set the server access permission and launch permission, which can be performed by using "DCOMCNFG.EXE" provided in the Windows NT package. When "DCOMCNFG.EXE" is executed, a DCOM server configuration dialog box appears. First of all, one should make sure that the check box "Enable Distributed COM on this computer" on the "Default Properties" page of that dialog box is checked. The other settings on this page should be kept unchanged. After that, go to the "applications" page of that dialog box, which displays all the DCOM objects and interfaces being registered in the system. To make a configuration for a particular server, one should find and select the server to be configured, then click "properties" button. Another dialog box will appear. To allow particular clients capable of launching and accessing the DCOM server remotely, one should set the related options from Security page and Identity page. From "Security" page, one can give the access permission to particular users from remote computers by clicking the corresponding "Edit..." button and then add these users onto the *AccessPermission* list. Similarly one can give launch permission to particular users from remote computers by clicking the corresponding "Edit..." button and then add these users onto the *LaunchPermission* list.

From "Identity" page, one should select "This user" from three of the option radio buttons. By clicking "Browse..." button. One can select a user from user list. The selected user should have an account on that particular server computer and should be able to launch the server. One should also type in the password needed for that particular user to log onto that computer. After the above process is completed, the server is ready to receive the client requests and provides the related services to the client.

6.4 Client side configuration

Client side configuration includes the registration for the Proxy and the registration for the remote server.

6.4.1 Registration for the Proxy

Proxy is used by the client in the process that performs *Marshalling* and *Unmashalling* for the calls between the client and the server. Before registering the server Proxy, one should generate Proxy DLL using nmake.exe program provided by the Microsoft visual studio. When server project is established, a Proxy make file is generated by the AppWizard. The name of the make file is set by the AppWizard to be the project name postfixed with "ps.mk". In the following we use *DataControler* server as an example to explain how to generate and register the Proxy DLL.

Among the project files of *DataControler* server, one can find a Proxy make file named as "DataControlerps.mk". This file should be copied to the computer on the client side to generate a Proxy DLL and then make proper registration. One can enter the following command at the command line to generate the Proxy DLL file:

nmake DataControlerps.mk

"nmake.exe" is one of the Microsoft program maintenance utilities that builds projects based on commands contained in a description file. The output of "nmake.exe" for Proxy make file is the Proxy DLL file named as "DataControlerps.dll". This Proxy DLL can be registered using regsvr32.exe program, provided with Windows NT package. At the command line, type the following command:

regsvr32 DataControlerps.dll

The result of the registration, either successful or failed, will be informed to the user. If one wants to remove the Proxy registration after it successfully registered, one can use the following command:

regsvr32 /u DataControlerps.dll

Once the Proxy DLL is successfully done, we can go to the next step. The registrations for all the other server proxies can follow the same procedure as described above.

6.4.2 Registration for Remote Servers

The servers should be registered on both server computer and client computer. It is natural to think that server should be registered on the server computer since system should be able to find the server when a client request arrives. For more or less the same reason, the same server should also be registered on the client computer. When a client needs the services from the server, it will ask the service control manager (SCM) to create the server object for him. The SCM will look at the system registry to find the location of the server. If the server locates remotely, the local SCM will ask the remote SCM to create the specified server object. The client then gets the services from that server by calling the functions supported by the server.

There are two methods to register the server on the client computer. For the first one, just follow the same server registration procedure described in the server configuration. The first method may be not convenient since one must copy the server component and all related registration files, ".RGS files", into the client computer.

The second method is more convenient. It requires only the export of the server registration information. After the server is registered in the server computer, one can search the registry database to find the corresponding server class and its interface, which can be done by using "regedit.exe" program provided by Windows NT package. The execution of "regedit.exe" program provides the user with menu-supported presentation of registry database. By using the export options from the menu, one can get the registration information for the server class and its interface, and further save them as registration files (see Appendix B). These files can be copied into somewhere in the client computer. Simply double click on these registration files from Windows NT explorer, the server will be registered into the system registry of client computer.

Both registration methods described above need further settings. After registration of server on the client computer, one should use "DCOMCNFG.EXE" program to reset the location of the server that allows the client system manager to launch and access the server on the dedicated location. Execution of "DCOMCNFG.EXE" program will bring up a dialog box to the user. Clicking on the "properties" button on that dialog box will bring up another dialog box, from which one can set the location of the server by using the "Location" page. Simply deselect "Run application on this computer" and select "Run application on the following computer", and further select the dedicated computer into

the corresponding edit field by using "Browse..." button. Then the setting for server location is done.

The registration procedure described here seems complex. As described in the chapter of DCOM Functionality, there exists another method for running the distributed application without the needs to perform the server registration on the client computer. This method requires the client codes to pass the detailed server location when calling *CoCreateInstanceEx()* to create server object. The advantage of this latter method is its simplicity and control over the server. Its disadvantage is evident. Whenever the server changes its location, the client codes need to be recompiled.

The latter solution, although simple, is not acceptable for the distributed application even the existing server never change their locations. The reason is that sometimes two or more servers of the same type are needed to release the server workload caused by increased user requests. In this case, one should build several versions of client component that pass different server locations to the *CoCreateInstanceEx()*, thus causing management difficulties.

Using server registration on client computer can guarantee the server location transparency. One needs not to know the server location when implementing client codes. The disadvantage is that a particular client must always get services from a particular server, which somewhat discourages the automatic server load-balancing.

6.5 Settings for Database

The data managed and accessed by the UCRMS is stored in the Oracle database. To allow the data-tier server to operate the data stored in the Oracle database, we use

ODBC as connection interfaces from data-tier server to Oracle database. For this purpose, we need to set a database connection using Microsoft ODBC driver for Oracle. The settings for database also include the establishment of the database.

6.5.1 Setting Database User Account

A new database user account can be set up using "SQL Plus" or "Database Navigator" provided in the Oracle database package. If using "SQL plus", one must use system or sys account to have the privilege for setting up a new user account. The easiest way to set up a new user account is to use "Database Navigator". Simply launch the "Database Navigator", and select "database" presented on the left pane of Navigator, a list of database items is displayed. Select "User" from the list and right-button mouse click on the selected item, a pop-up menu appears. One can then add a user account by entering the user's name and password into "New User Properties" page that is brought up by select "New" from the pop-up menu.

6.5.2 Setting Database Connection

The database connection for the ODBC driver can be established by using the "ODBC" from Control Panel. Clicking on "ODBC" icon brings up "ODBC Data Source Administrator" dialog box, from which one can add a new User DSN. When "Add" button is clicked, a list of ODBC drivers is presented to the user. Select "Microsoft ODBC for Oracle" and click "Finish" button, one is then prompted to enter the Data Source Name and its Description. Enter "Oracle Connection". After this information is entered, the database connection is then established. One can also add a new file DSN by using the similar procedure. From the file DSN page, click "Add" button and select

"Microsoft ODBC for Oracle" from ODBC driver list, then click "next". One is prompted to enter the name of the file data source. Enter the name as "Oracle Connection". After the name is given, click on "next", and then "Finish". one is prompted to enter the user name, password, and the server name. Upon the completion of entering user and database server information, one can click "OK" button to finish the setting. The Oracle database is then ready to be accessed by the UCRMS data server.

6.5.3 Creating Database Tables

Database tables can be created from either "Database Navigator", "SQL Plus", or database project created by Microsoft Visual Studio. The most convenient way is to run "SQL Plus" program and use "create" command at the command line of "SQL Plus". One can create a file postfixed with .sql and write all the table-create commands into this file. For our purpose, the file named "student_course.sql" is created (see Appendix D). The format of the commands written in this file are shown below:

```
create table Student
stid        number(7, not null,
stname      varchar2(20),
sttype      number(2),
deptno      number(4),
password    char(4)
);
```

By running the following command at the command line of "SQL Plus", the database tables can be properly created:

start student_course.sql

6.5.4 Inserting Values into Database Tables

Database values can be easily entered into database by using "insert" command. Simply create a file postfixed with .sql and write the required "insert" command into this

file, then run these commands from "SQL Plus" command line. The file named "insertvalues.sql" (see Appendix D) is created for this purpose. The command written in this file follows the format shown below:

```
insert into Teacher values(6,'lixin Tuo',1);
```

Running the following command at the command line of "SQL Plus" will execute all the commands written in the file:

start insertvalues.sql

As a result, the values written in the files are properly inserted into the database tables.

6.6 Using the UCRMS

According to the configuration guidelines described above, we have conducted following settings for the UCRMS to be used in a distributed environment. We used three Pentium II personal computers to execute the UCRMS. Windows NT work station is installed for all these three PCs.

On the first PC, we installed Oracle database, with database tables properly created. We then inserted sufficient data values into these tables. We also established a new ODBC connection that is called "Oracle Connection" and is to be used by *DataAccess* server. We installed *DataAccess* server and properly register it as ordinary DCOM server; their registration information is exported into registration files: "DataAccessClass.reg" and "DataAccessInterface.reg" (see Appendix B). On the second PC, we installed *DataControler* server, *SecurityMgr* server, and *SpecMgr* server. *DataControler* server is registered as NT service, and the other two servers are registered as ordinary DCOM server. With the help of *DataAccess* registration files, we registered

DataAccess server on the second PC and reset it to run on the first PC. *DataAccess* proxy DLL is also registered by using "nmake.exe" and "regsvr32.exe". Moreover we made environment variable settings for SpecMgr server according to the location of the "configuration" and "help" files. The registration information for all these three servers on the second PC are exported into registration files respectively named as: "DataControlerClass.reg", "DataControlerInterface.reg", "SecurityMgrClass.reg", "SecurityMgrInterface.reg", "SpecMgrClass.reg" and SpecMgrInterface.reg" (see Appendix B). On the third PC, we installed the starting program for the graphic user interface. This includes "CourseReg.exe" and "GuiBase.dll". Actually the "GuiBase.dll" is to be loaded by "CourseReg.exe" to its own working space and is the base of the graphic user interface. Finally, also on the third PC we made a server registration for all the middle-tier servers and reset them to run on the second PC. After the above settings, the UCRMS is ready to be used from the client computer, the third PC. Here we describe how the user tasks can be completed with the help of graphic user interface.

6.6.1 Login Window

Login window is shown in Figure 20. To login to the system, user needs to enter a valid user ID and select a corresponding login type from a list of types. There are three buttons displayed on the login window. Clicking on "Enter" button will cause the system to process the user's login information. If login is successful, the user is then presented with either course registration window, or course add and drop window, or student and course management window, or administrator management window, depending on user's login information. For an invalid login, a message box appears informing users about the login failure. Then system flushes the login fields and waits for a new login session.

Clicking on "Help" button will bring up a text box to tip the user about correct login procedure. Clicking on "Quit" button will cause UCRMS to terminate.

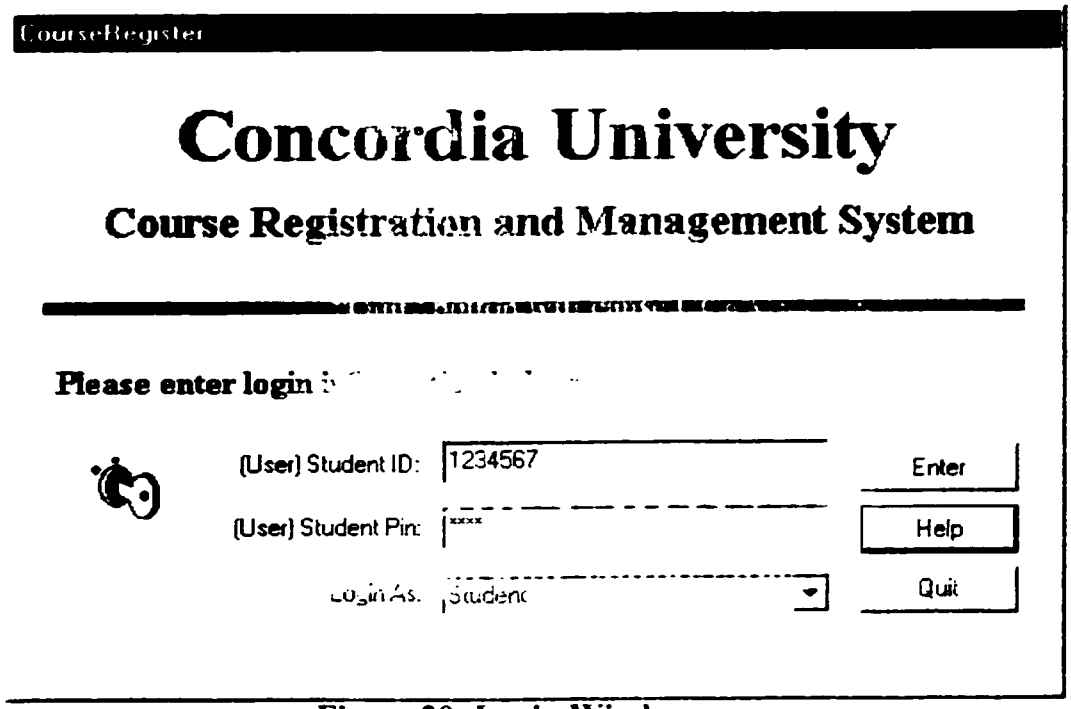
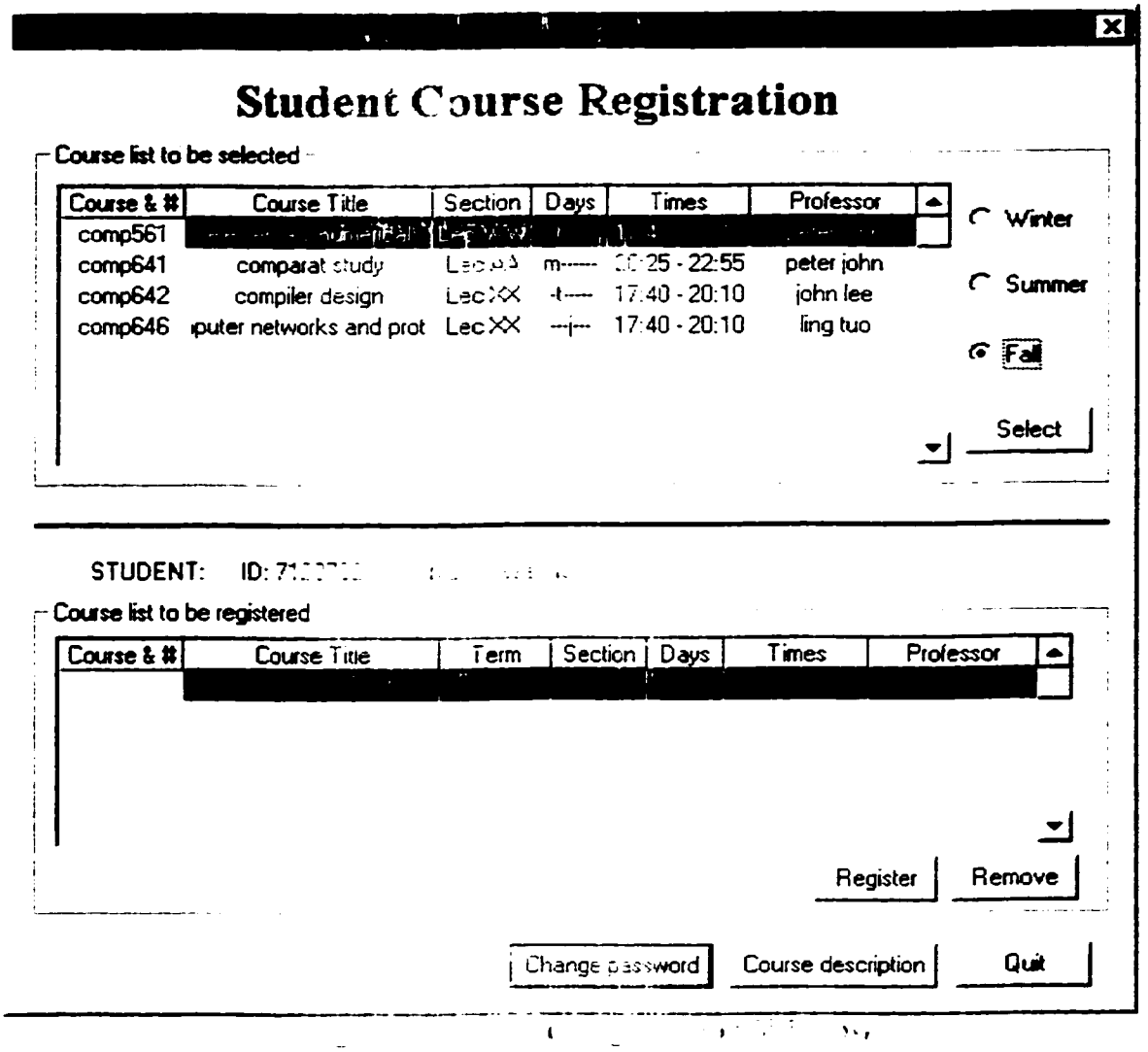


Figure 20: Login Window

6.6.2 Course Registration Window

For a successful STUDENT login, the system presents the course registration window to the student. As shown in Figure 21, there are one pre-select course list and one pre-register course list, clearly displayed in window's ActiveX gridline controls. The student can select the courses from pre-select course list into pre-register course list. Clicking on course displayed on the pre-select list will select the course. The selected course will be put into pre-register course list when "Select" button is clicked. The same course can appear only in one course list.



The courses that are selected into pre-register list are going to be stored into database if the student choose to register these courses. Clicking on "Register" button will bring up a confirmation dialog box (as shown in Figure 22) to the student, showing the courses to be registered. Student needs to confirm the registration before the data can be set into the database. Student can change his password by clicking "Change password" button, a dialog box appears allowing the student to type in the new password. Student can view a particular course description by clicking on "Course description" button. A

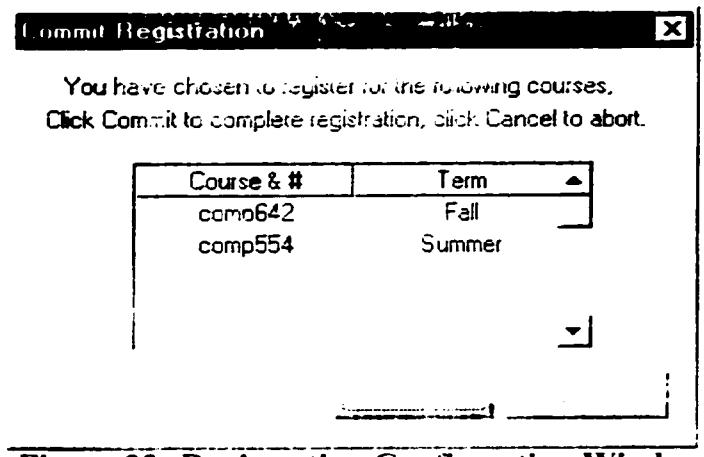
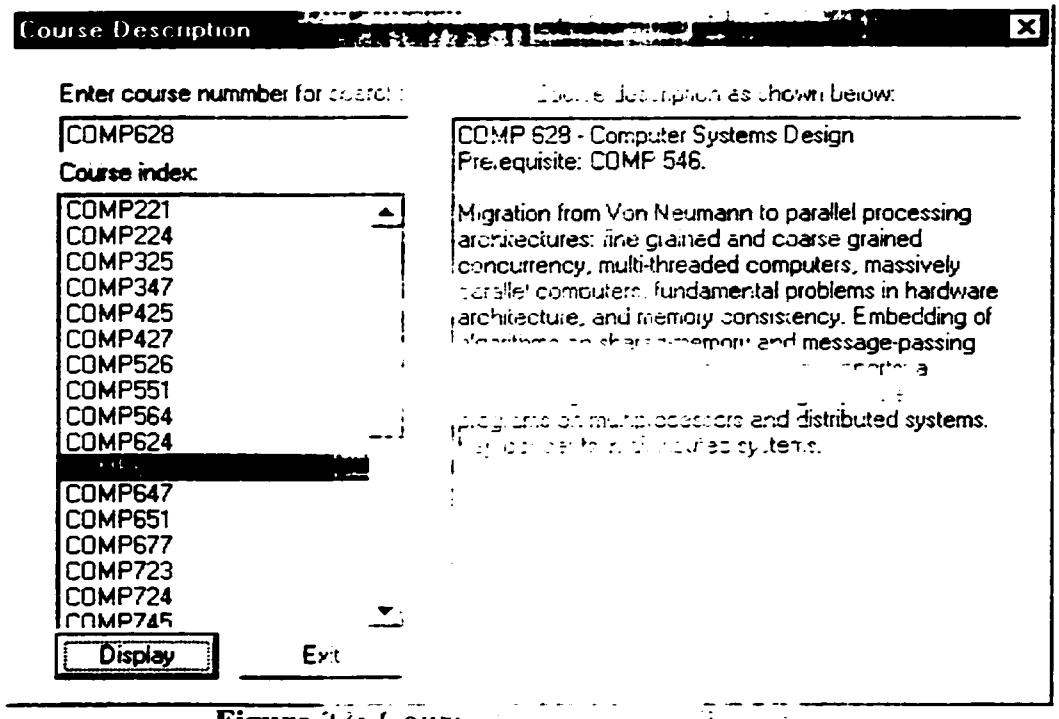


Figure 22: Registration Confirmation Window

course description window is presented to the student. As shown in Figure 23, the course description window supports full index search of a particular course.



Figure

Selecting on a course displayed on the left pane of the window and clicking on "Display" button, a detailed course description is presented to the student on the right pane of the

window. Student can quit from registration window by clicking "Quit" button, and the system goes back to login window waiting for new login session.

6.6.3 Course Add and Drop Window

If the UCRMS detects a valid login for a student who already has courses registered, the course add and drop window will be presented to the student, as shown in Figure 24.

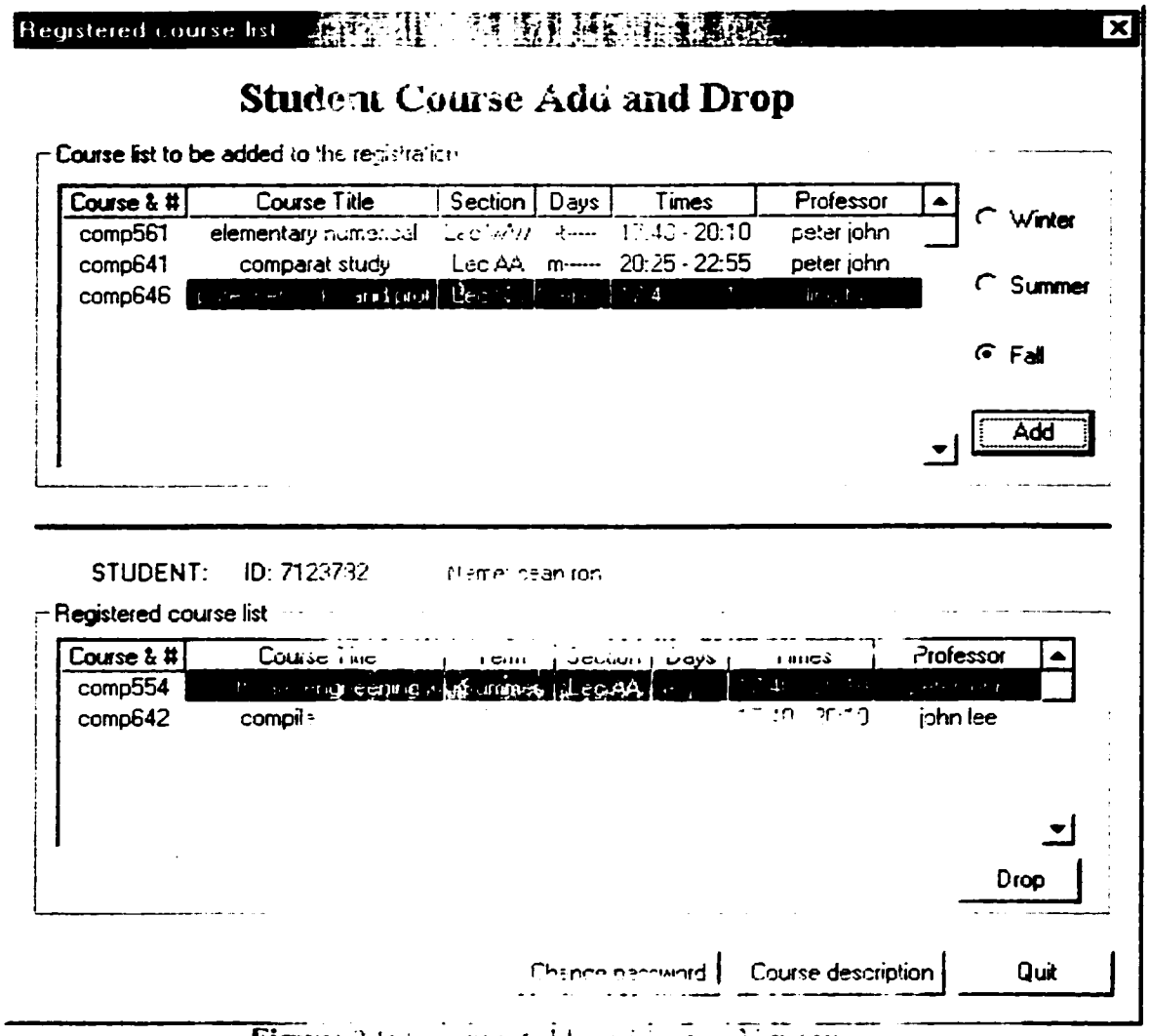


Figure 24: Course Add and Drop Window

On the upper portion of the window displays the course select list; on the lower portion displays the registered course list. When a course is selected from the course select list and "Add" button is clicked, a message box appears asking for the confirmation for registering the selected course into the database. The user can either commit or abort the action. When a course is selected from the registered course list and "Drop" button is clicked, a message box appears asking for the confirmation for removing the selected course from the courses registered in the database. The user can either commit or abort the action.

6.6.4 Student and Course Management Window

For a successful ADMINISTRATOR login, the system will present a **student and course management window** to the administrator, as shown in Figure 25.
Two course lists are presented on the window: one is the course list and the other is the student list. The courses displayed on the course list represents the total courses available for student registration. Administrator can add a new course into the course list by clicking "Add" button. The administrator is prompted to enter the new course information into the edit fields of dialog box, as shown in Figure 26.

Upon the commit of course-adding procedure (by clicking "OK" button), the new added course is stored into the remote database. An administrator can remove a selected course from the course list by clicking "Remove" button. The removing of a course requires the confirmation from the super user. Once confirmed, the selected course is removed from the database.

The students displayed on the student list represents the total student permitted to use the UCRMS. The administrator can also add a new student into student list or remove

a student from student list with the same rules governing course add and remove procedures. Administrator can quit from student and course management window by clicking "Quit" button and system goes back to login window waiting for new login session.

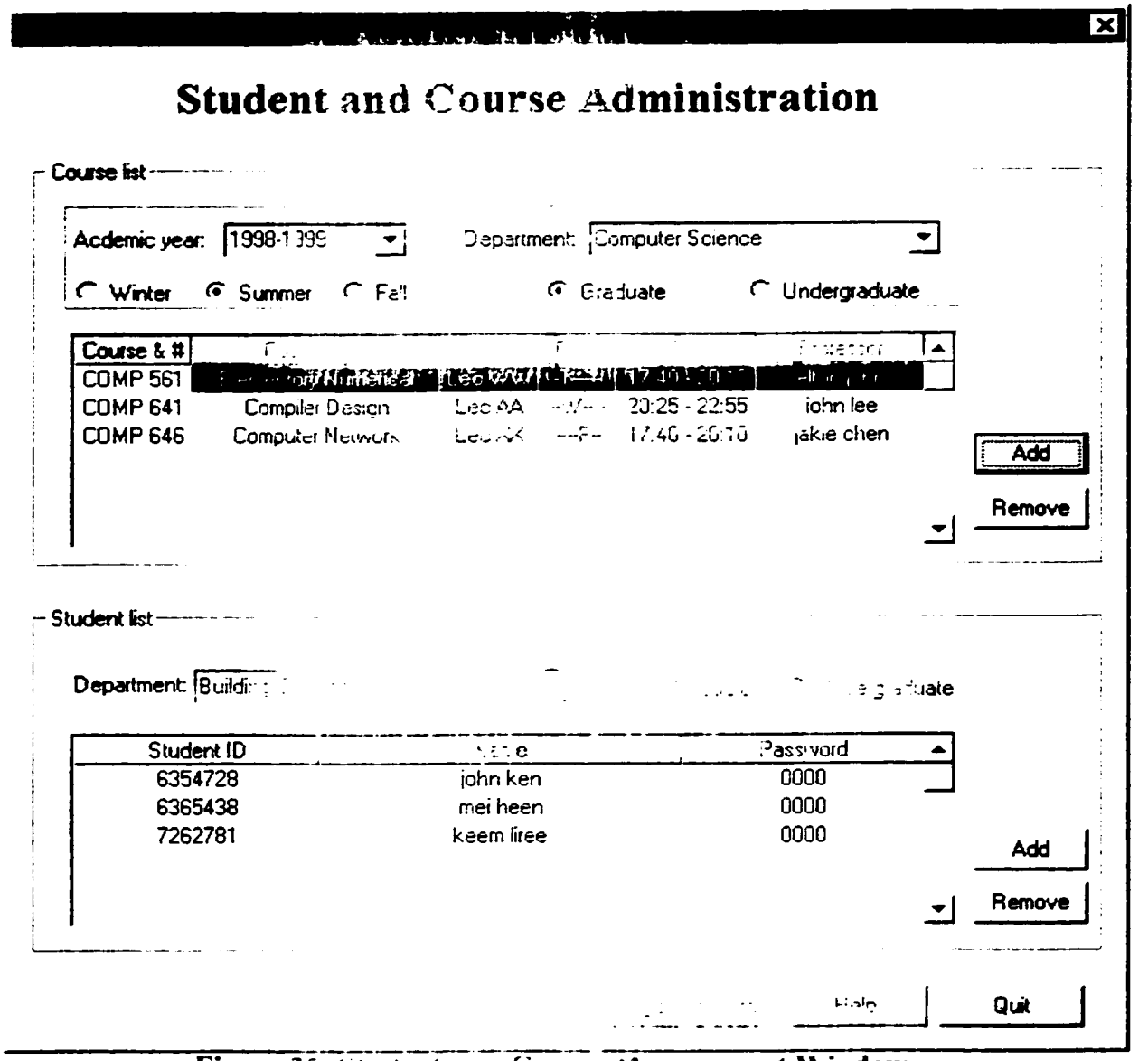


Figure 25: Student and Course Management Window

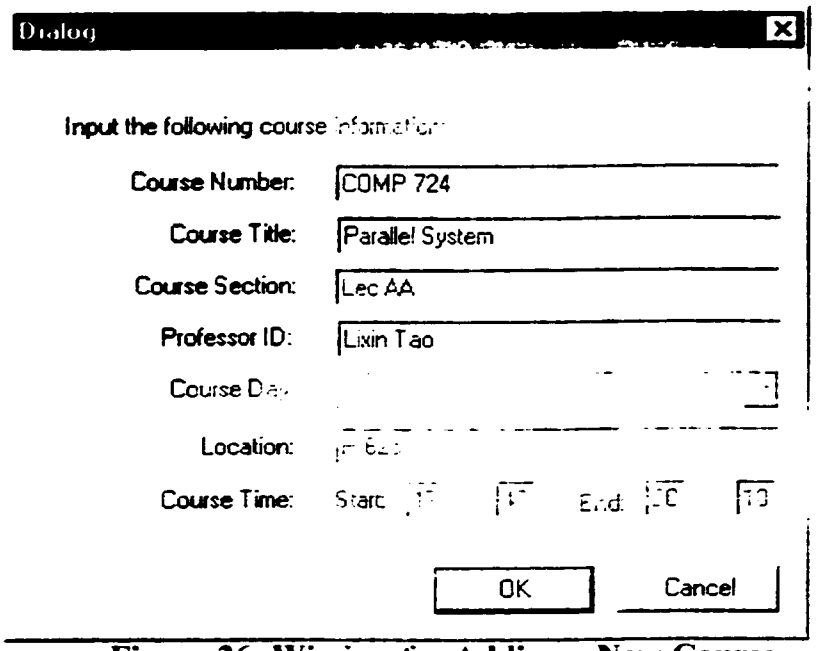


Figure 26: Window for Adding a New Course

6.6.5 Administrator Management Window

For a successful SUPER USER login, the system presents an administrator management window to the super user, as shown in Figure 27.

An administrator list is displayed on the window. A new administrator can be added into the list by clicking the "Add" button. A dialog box appears that prompts the super user to enter the required information about the administrator to be added (see Figure 28). An administrator can be selected. The selected administrator can be removed by clicking "Remove" button. A confirmation message box appears asking for the confirmation from the super user. If confirmed, the remote database will be updated. Otherwise no action will be taken and the system goes back to the administrator management window. Super user can quit from administrator management window by clicking "Quit" button and system goes back to login window waiting for new login session.

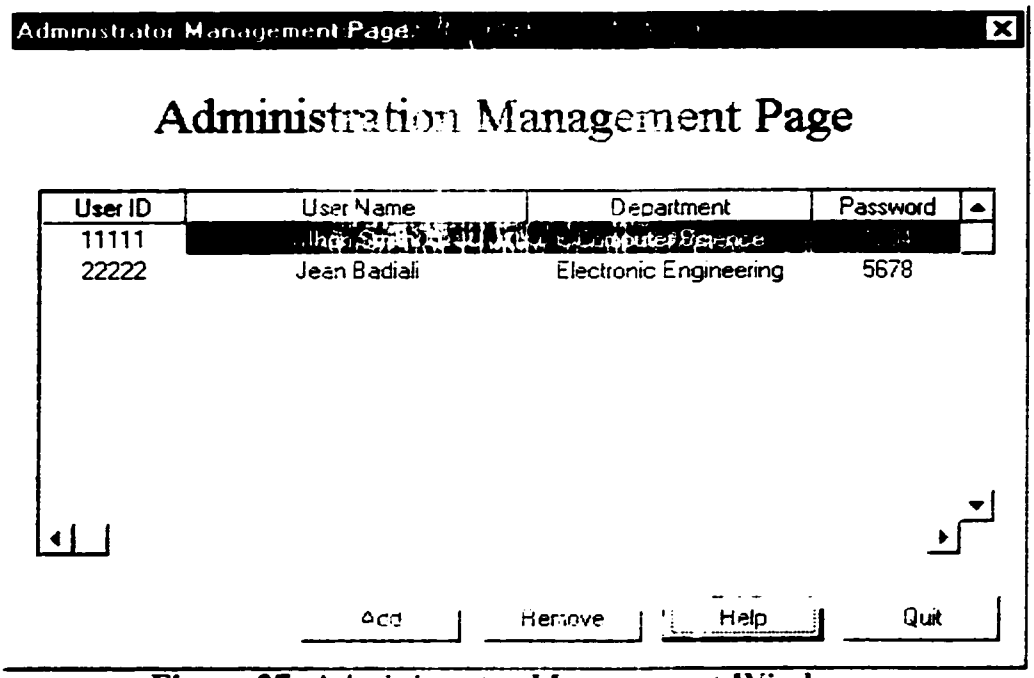


Figure 27: Administrator Management Window

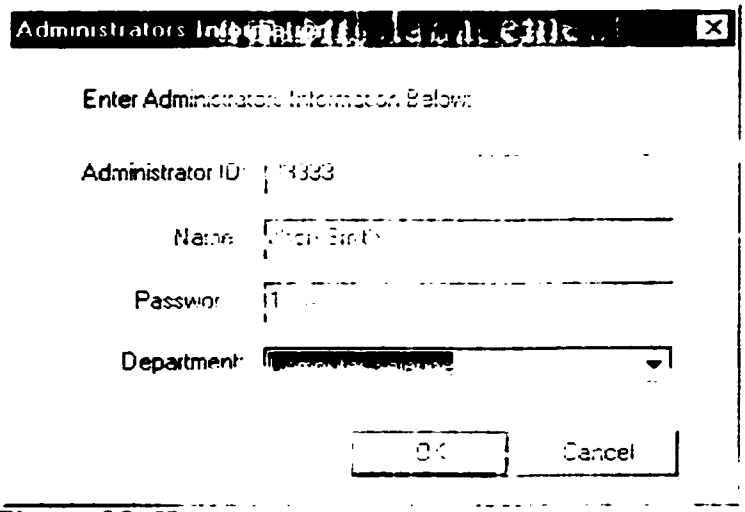


Figure 28: Window for Adding a new Administrator

7 System-Testing and Bug-Fixing

A usability testing has been conducted for UCRMS. The purpose of the testing is to find the following points:

- Is there anything that confuses the user?
- Is there any task that can not be completed by the system?
- Is there any bug in the system?

Several problems have been found from the above system testing. Some of the problems and their corrections are summarized below:

- It's not clear for the first time user that courses in the pre-select list should be first selected into pre-register list in order for these selected courses to be finally registered into system. We have added short help-tips on top of each course lists to correct this problem.
- When users quit the working window, the login information is not removed from the login window, leaving the possibility of unauthorized access to the users' account. This problem has been corrected by clearing the login password field immediately after the user quits the working window.
- From student and course management window, the courses with two lectures per week cannot be added into system since the choices are given only for one lecture per week. This problem has been fixed by adding the choices for setting two lectures per week.

8 Conclusions

A University Course Registration and Management System (UCRMS) is developed on Windows NT by using Microsoft Visual Studio. UCRMS is a three-tier, distributed application capable of running on number of different computers according to the configuration. It is designed and implemented to be a user-centered course registration system that is easy to use and easy to learn for students and administrators. The primary goal has been achieved, which can be described below:

- UCRMS is easy to learn. The reason that users are capable of learning to use the system easily is that the system is designed to be simple for all the users. Another reason is that users can learn the registration procedure incrementally, each invalid operation will pop up a message to tell what the users should do.
- UCRMS is error-protected. This is true especially for students to register their courses. Students are allowed to do every possible operation without harm to the system since the only source of error is at the time of login and changing password in which system performs full control.
- UCRMS is secure. The security for server component is set on the component basis with different security levels including launch permission, access permission, and Windows NT permission.
- UCRMS is efficient. Students can self-register their courses without the help of course administrators. The on-line course descriptions can help students make their course selections easily and quickly.

- UCRMS is modifiable. Most parts of the user interface can be easily modified to suit the different user requirements, such as text box and radio box. The extended services can be easily added into the public interface of the server components.
- UCRMS is scalable. As the number of user increases, the system can be reconfigured to use more middle-tier server components to maintain the application performance.
- UCRMS is efficient. The implementation is done using object-oriented C++ language capable of generating efficient application program.
- UCRMS is reliable. Two or more middle-tier servers can be used to improve the system reliability. The failure of one server will not stop the whole system from functioning.
- UCRMS supports independent service update. If more services and functionality are added to a server component in the UCRMS, the other components need not necessarily be recompiled.
- UCRMS supports online data update. The dynamic changeable data can be updated without shutting down the server.
- UCRMS supports efficient data access. Problems of database connection limitation are minimized since only data-tier server is connected with the database.
- UCRMS supports component reuse. The UCRMS, for example, can be re-built into Web-based application with the modification of only *GUI* module. All the other components can be reused.

From the experience of design and implementation of UCRMS, we are convinced that the three-tier distributed application is more efficient, more reliable, and more scalable than the conventional two-tier application. The Microsoft Visual Studio is able to offer the

developers with excellent developing environment for implementing three-tier distributed applications.

Because of the time constraint, some of the features in the UCRMS need to be enhanced and improved in the future version of the UCRMS:

- Database operations is not as reliable as it should be. If something goes wrong in the middle of database operation, some faulty database state may appear. The solution to this problem might be to add a new layer between data-tier server and the database. This new layer can be implemented using Microsoft Transaction Server (MTS) capable of providing a transaction guarantee that either the transaction is committed or nothing is done.
- *DataAccess* server is the only data-tier server that manages active data transaction. In order to use MTS actively in database operations, it's more efficient to implement *DataAccess* server as a DLL component rather than EXE component. Then MTS is able to manage *DataAccess* component directly and provides it with surrogate and transaction services at the same time.
- The future version of UCRMS should be able to provide more effective online help and course descriptions. These supports are crucial for the students to select their courses and finally make course registration quickly.
- A new functionality that allows the course administrator to customize the number of courses permitted for each student should be added in the future version of UCRMS.
- Currently UCRMS only supports single course selection. In the future version, the system should be able to support multiple course selections to speed up the course registration process.

- As the Internet usage becomes more and more popular, the Web-based UCRMS should be developed in the future. As mentioned above, only GUI module needs to be rebuilt. For this purpose, we need to build a Web-based graphic user interface by using Microsoft Active Server Page (ASP). Moreover a COM component must be implemented to serve as a bridge between the ASP and the *GuiBase* module. The Web-based *GUI* and the new built COM component together form the entire *GUI* module. Under such a configuration, client request is sent to *GuiBase* component through COM component, and GuiBase component further calls the other middle-tier servers, returning any information requested by the client. Needless to say, COM/DCOM plays a significant role in the development of distributed application.

References

[1] D. Rogerson, Inside COM, Microsoft Press, 1996.

[2] R. Grimes, Professional DCOM Programming, Wrox Press, 1997.

[3] J.Siegel, CORBA Fundamentals and Programming, Jhon Wiley & Sons, 1996

[4] M. Henning and S. Vinoski, Advanced CORBA Programming with C++, Addison-Wesley Pub Co., 1999

[5] DCOM Technical Overview, http://www.microsoft.com/windows/common/pdcwp.htm

[6] N. Brown and C. Kindel, Distributed Component Object Model Protocol, http://www.miccrosoft..com/oledcom/draft-brown-dcom-v1-spec-01.txt

[7] Y.M. Wang, Y. Huang, and W.K. Fuchs Progressive Retry for Software Error Recovery in Distributed Systems, IEEE Fault-Tolerance Computing Symp., pp. 138, 1993

[8] K.P. Birman, Building Secure and Reliable Network Applications, Manning Publications CO., 1996

[9] OSF DCE RPC Specification, http://www.ost.org.mail/dce/free_dce.htm

Appendix A — IDL Files

```
// DataControler.idl : IDL source for DataControler.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (DataControler.tlb) and marshalling code.


import "oaidl.idl";
import "ocidl.idl";

typedef struct _CourseInfo {
        ULONG courseId;
        BSTR courseNo;
        BSTR courseTitle;
        ULONG courseTerm;
        BSTR courseSec;
        BSTR courseDays;
        BSTR startTime;
        BSTR endTime;
        BSTR courseRoom;
        ULONG courseType
        ULONG deptNum;
        BSTR profName;
} CourseInfo;

typedef struct _StudentInfo {
        ULONG studentId;
        BSTR studentName;
        ULONG studentType;
        ULONG deptNum;
        BSTR studentPass;
} StudentInfo;

typedef struct _ManagerInfo {
        ULONG managerId;
        BSTR managerName;
        ULONG deptNum;
        BSTR managerPass;
```

```
} ManagerInfo;

typedef struct _StudentRegInfo {
        ULONG studentId;
        ULONG courseId;
} StudentRegInfo;

        [
                object,
                uuid(63B3BFA0-7BE2-11d2-819B-000000000000),

                helpstring("IModel Interface"),
                pointer_default(unique)
        ]
        interface IModel : IDispatch
        {
                [helpstring("method UpdateCourseInfo")]
                HRESULT UpdateCourseInfo([in] CourseInfo* courseInfo);
                [helpstring("method UpdateStudentInfo")]
                HRESULT UpdateStudentInfo([in] StudentInfo* studentInfo);
                [helpstring("method DeleteStudentData")]
                HRESULT DeleteStudentData([in] ULONG studentId);
                [helpstring("method DeleteCourseData")]
                HRESULT DeleteCourseData([in] ULONG courseId);
        };

        [
                object,
                uuid(082EC2FF-7BDE-11D2-819B-000000000000),

                helpstring("IDataServer Interface"),
                pointer_default(unique)
        ]
        interface IDataServer : IUnknown
        {

                [helpstring("method GetNumCourse")]
                HRESULT GetNumCourse([out] ULONG* pNumCourse);
                [helpstring("method GetCourseList")]
                HRESULT GetCourseList([in] ULONG numCourse, [out, size_is(numCourse)]
CourseInfo* pvecCourseInfo);
                [helpstring("method GetNumStudent")]
                HRESULT GetNumStudent([out] ULONG* pNumStudent);
                [helpstring("method GetStudentList")]
                HRESULT GetStudentList([in] ULONG numStudent, [out, size_is(numStudent)]
StudentInfo* pvecStudentInfo);
                [helpstring("method GetNumManager")]
                HRESULT GetNumManager([out] ULONG* pNumManager);
                [helpstring("method GetManagerList")]
                HRESULT GetManagerList([in] ULONG numManager, [out, size_is(numManager)]
ManagerInfo* pvecManagerInfo);
                [helpstring("method SetRegCourse")]
                HRESULT SetRegCourse([in] ULONG numRegCourse, [in, size_is(numRegCourse)]
StudentRegInfo* pvecRegCourse);
                [helpstring("method GetNumRegCourse")]
                HRESULT GetNumRegCourse([in] ULONG stuId, [out] ULONG* numRegCourse);
```

```
        [helpstring("method GetStudentRegList")]
        HRESULT GetStudentRegList([in] ULONG stuId, [in] ULONG numRegCourse, [out,
size_is(numRegCourse)] StudentRegInfo* pvecRegInfo);
        [helpstring("method DelRegCourse")]
        HRESULT DelRegCourse([in] ULONG numRegCourse, [in, size_is(numRegCourse)]
StudentRegInfo* pvecRegInfo);
        [helpstring("method SetStudentInfo")]
        HRESULT SetStudentInfo([in] StudentInfo* pStudentInfo);
        [helpstring("method DelStudent")]
        HRESULT DelStudent([in] ULONG stuId);
        [helpstring("method SetCourseInfo")]
        HRESULT SetCourseInfo([in] CourseInfo* pCourseInfo, ULONG teacherId);
        [helpstring("method GetCourseProfName")]
        HRESULT GetCourseProfName([in] ULONG profId, [out] BSTR* profName);
        [helpstring("method DelCourse")]
        HRESULT DelCourse([in] ULONG corId);
        [helpstring("method SetManagerInfo")]
        HRESULT SetManagerInfo([in] ManagerInfo* pManagerInfo);
        [helpstring("method DelManager")]
        HRESULT DelManager([in] ULONG mgrId);
        [helpstring("method SetConnection")]
        HRESULT SetConnection([in] IModel* pIModel);
        [helpstring("method ReleaseInterface")]
        HRESULT ReleaseInterface([in] IModel* pIModel);
    };


[
    uuid(082EC2F3-7BDE-11D2-819B-000000000000),
    version(1.0),
    helpstring("DataServer 1.0 Type Library")
]
library DATACONTROLLERLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(082EC301-7BDE-11D2-819B-000000000000),
        helpstring("_IDataServerEvents Interface")
    ]
    dispinterface _IDataServerEvents
    {
        properties:
        methods:
    };

    [
        uuid(082EC300-7BDE-11D2-819B-000000000000),
        helpstring("DataServer Class")
    ]
    coclass DataServer
    {
        [default] interface IDataServer;
        [default, source] dispinterface _IDataServerEvents;
    };
```

```
};


// SecurityMgr.idl : IDL source for SecurityMgr.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (SecurityMgr.tlb) and marshalling code.


import "oaidl.idl";
import "ocidl.idl";

        [
                object,
                uuid(8D2F46[illegible]-D77B-11D2-81B5-000000000000),
                helpstring("ISecuritySvr Interface"),
                pointer_default(unique)
        ]
        interface ISecuritySvr : IUnknown
        {
                [id(1), helpstring("method UpdateStudentData")]
                HRESULT UpdateStudentData();
                [id(2), helpstring("method LoadUserPassInfo")]
                HRESULT LoadUserPassInfo();
                [helpstring("method GetUserPermission")]
                HRESULT GetUserPermission([in] BSTR bId, [in] BSTR bPass, [in] ULONG logType,
[out] ULONG* pUserType);
                [helpstring("method ChangePassword")] HRESULT ChangePassword([in] ULONG
userType, [in] ULONG userId, [in] BSTR newPass);
                [helpstring("method Refresh")] HRESULT Refresh();
        };

[
        uuid(8D2F4613-D77B-11D2-81B5-000000000000),
        version(1.0),
        helpstring("SecurityMgr 1.0 Type Library")
]
library SECURITYMGRLib
{
        importlib("stdole32.tlb"),
        importlib("stdole2.tlb"),

        [
                uuid(ED7E915C-D762-11D2-81B5-000000000000),
                helpstring("SecuritySvr Class")
        ]
        coclass SecuritySvr
        {
                [default] interface ISecuritySvr;
        };
};


// SpecMgr.idl : IDL source for SpecMgr.dll
//
```

```
// This file will be processed by the MIDL tool to
// produce the type library (SpecMgr.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";

typedef struct _YearInfo {
        BSTR year;
} YearInfo;

typedef struct _DeptInfo {
        ULONG deptNum;
        BSTR deptName;
} DeptInfo;

typedef struct _CorNumInfo {
        BSTR corNum;
} CorNumInfo;

        [
                object,
                uuid(AF56ECF3-D77B-11D2-81B5-000000000000),
                helpstring("ISpecSvr interface"),
                pointer_default(unique)
        ]
        interface ISpecSvr : IUnknown
        {
                [helpstring("method GetDeptList")]
                HRESULT GetDeptList([in] ULONG numDept, [out, size_is(numDept)] DeptInfo*
pvecDeptInfo);
                [id(1), helpstring("method GetNumYear")]
                HRESULT GetNumYear([out] ULONG* numYear);
                [id(2), helpstring("method GetYearList")]
                HRESULT GetYearList([in] ULONG numYear, [out, size_is(numYear)] YearInfo*
pvecYearInfo);
                [id(3), helpstring("method GetNumDept")]
                HRESULT GetNumDept([out] ULONG* pNumDept);
                [helpstring("method Refresh")] HRESULT Refresh();
                [helpstring("method GetTimerStep")]
                HRESULT GetTimerStep([out] ULONG* timerStep);
                [helpstring("method GetNumDeptNum")]
                HRESULT GetNumCorNum([out] ULONG* numCorNum);
                [helpstring("method GetCorNumList")]
                HRESULT GetCorNumList([in] ULONG numCorNum, [out, size_is(numCorNum)]
CorNumInfo* pvecCorNumInfo);
                [helpstring("method GetDescription")] HRESULT GetDescription([in] BSTR corNum,
[out] BSTR* description);
        };

[
        uuid(AF56ECF3-D77B-11D2-81B5-000000000000),
        version(1.0),
        helpstring("SpecMgr 1.0 Type Library")
]
library SPECMGRLib
```

```
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                uuid(95FD24DE-D76F-11D2-81B5-000000000000),
                helpstring("SpecSvr Class")
        ]
        coclass SpecSvr
        {
                [default] interface ISpecSvr;
        };
};


// DataAccess.idl : IDL source for DataAccess.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (DataAccess.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";

typedef struct _EntryPassInfo {
        BSTR userId;
        BSTR userPass;
} EntryPassInfo;

typedef struct _CourseDataInfo {
        ULONG courseId,
        BSTR courseNo;
        BSTR courseTitle;
        ULONG courseTerm;
        BSTR courseSec;
        BSTR courseDays;
        BSTR startTime;
        BSTR endTime;
        BSTR courseRoom;
        ULONG courseType;
        ULONG deptNum;
        BSTR profName;
} CourseDataInfo;

typedef struct _StudentDataInfo {
        ULONG studentId;
        BSTR studentName;
        ULONG studentType;
        ULONG deptNum;
        BSTR studentPass;
} StudentDataInfo;

typedef struct _ManagerDataInfo {
        ULONG managerId;
        BSTR managerName;
        ULONG deptNum;
```

```
        BSTR managerPass;
} ManagerDataInfo;

typedef struct _StudentRegDataInfo {
        ULONG studentId;
        ULONG courseId;
} StudentRegDataInfo;


        [
                object,
                uuid(947EA1D0-C094-11D2-81CF-C00000000000),
                helpstring("IDataAccessSvr Interface"),
                pointer_default(unique)
        ]
        interface IDataAccessSvr : IUnknown
        {
                [id(1), helpstring("method UpdateStudentData")]
                HRESULT UpdateStudentData();
                [id(2), helpstring("method GetNumStudent")]
                HRESULT GetNumStudent([out] ULONG* pNumStudent);
                [id(3), helpstring("method GetNumSuperUser")]
                HRESULT GetNumSuperUser([out] ULONG* pNumSuperUser);
                [id(4), helpstring("method GetNumManager")]
                HRESULT GetNumManager([out] ULONG* pNumManager);
                [id(5), helpstring("method GetStudentPassInfo")]
                HRESULT GetStudentPassInfo([in] ULONG numStudent, [out, size_is(numStudent)]
EntryPassInfo* pvecPassInfo);
                [id(6), helpstring("method GetSuperUserPassInfo")]
                HRESULT GetSuperUserPassInfo([in] ULONG numSuperUser, [out,
size_is(numSuperUser)] EntryPassInfo* pvecPassInfo);
                [id(7), helpstring("method GetManagerPassInfo")]
                HRESULT GetManagerPassInfo([in] ULONG numManager, [out, size_is(numManager)]
EntryPassInfo* pvecPassInfo);
                [helpstring("method GetNumCourse")]
                HRESULT GetNumCourse([out] ULONG* pNumCourse);
                [helpstring("method GetCourseList")]
                HRESULT GetCourseList([in] ULONG numCourse, [out, size_is(numCourse)]
CourseDataInfo* pvecCourseInfo);
                [helpstring("method GetStudentList")]
                HRESULT GetStudentList([in] ULONG numStudent, [out, size_is(numStudent)]
StudentDataInfo* pvecStudentInfo);
                [helpstring("method GetManagerList")]
                HRESULT GetManagerList([in] ULONG numManager, [out, size_is(numManager)]
ManagerDataInfo* pvecManagerInfo);
                [helpstring("method SetRegCourse")]
                HRESULT SetRegCourse([in] ULONG numRegCourse, [in, size_is(numRegCourse)]
StudentRegDataInfo* pvecRegDataInfo);
                [helpstring("method GetNumRegCourse")]
                HRESULT GetNumRegCourse([in] ULONG stuId, [out] ULONG* numRegCourse);
                [helpstring("method GetStudentRegList")]
                HRESULT GetStudentRegList([in] ULONG stuId, [in] ULONG numRegCourse, [out,
size_is(numRegCourse)] StudentRegDataInfo* pvecRegDataInfo);
                [helpstring("method DelRegCourse")]
                HRESULT DelRegCourse([in] ULONG numRegCourse, [in, size_is(numRegCourse)]
StudentRegDataInfo* pvecRegDataInfo);
```

```
        [helpstring("method SetStudentDataInfo")]
        HRESULT SetStudentDataInfo([in] StudentDataInfo* pStudentDataInfo);
        [helpstring("method DelStudent")]
        HRESULT DelStudent([in] ULONG stuId);
        [helpstring("method SetCourseDataInfo")]
        HRESULT SetCourseDataInfo([in] CourseDataInfo* pCourseDataInfo, [in] ULONG
teacherId);
        [helpstring("method GetCourseProfName")]
        HRESULT GetCourseProfName([in] ULONG profId, [out] BSTR* profName);
        [helpstring("method DelCourse")]
        HRESULT DelCourse([in] ULONG corId);
        [helpstring("method ChangePassword")]
        HRESULT ChangePassword([in] ULONG userType, [in] ULONG userId, [in] BSTR
newPass);
        [helpstring("method SetManagerInfo")]
        HRESULT SetManagerInfo([in] ManagerDataInfo* pManagerDataInfo);
        [helpstring("method DelManager")]
        HRESULT DelManager([in] ULONG mgrId);
    };

[
    uuid(947EA1C4-C094-11D2-81CF-000000000000),
    version(1.0),
    helpstring("DataAccessSvr 1.0 Type Library")
]
library DATAACCESSLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(947EA1D4-C094-11D2-81CF-000000000000),
        helpstring("DataAccessSvr Class")
    ]
    coclass DataAccessSvr
    {
        [default] interface IDataAccessSvr;
    };
};
```

Appendix B — Server Registration Files

<******* DataAccessClass.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{947EA1D1-C094-11D2-81CF-000000000000}]
@="DataAccessSvr Class"
"AppID"="{947EA1C5-C094-11D2-81CF-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{947EA1D1-C094-11D2-81CF-000000000000}\LocalServer32]
@="D:\\ZQReport\\DATAAC~1\\Debug\\DATAAC~1.EXE"

[HKEY_CLASSES_ROOT\CLSID\{947EA1D1-C094-11D2-81CF-000000000000}\ProgID]
@="DataAccess.DataAccessSvr.1"

[HKEY_CLASSES_ROOT\CLSID\{947EA1D1-C094-11D2-81CF-000000000000}\Programmable]

[HKEY_CLASSES_ROOT\CLSID\{947EA1D1-C094-11D2-81CF-000000000000}\TypeLib]
@="{947EA1C4-C094-11D2-81CF-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{947EA1D1-C094-11D2-81CF-
000000000000}\VersionIndependentProgID]
@="DataAccess.DataAccessSvr"

<******* End of File *******>

<******* DataAccessInterface.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\Interface\{947EA1D0-C094-11D2-81CF-000000000000}]
@="IDataAccessSvr"

[HKEY_CLASSES_ROOT\Interface\{947EA1D0-C094-11D2-81CF-000000000000}\NumMethods]
@="26"

[HKEY_CLASSES_ROOT\Interface\{947EA1D0-C094-11D2-81CF-000000000000}\ProxyStubClsid]
@="{00020424-0000-0000-C000-000000000046}"

[HKEY_CLASSES_ROOT\Interface\{947EA1D0-C094-11D2-81CF-000000000000}\ProxyStubClsid32]
@="{947EA1D0-C094-11D2-81CF-000000000000}"

[HKEY_CLASSES_ROOT\Interface\{947EA1D0-C094-11D2-81CF-000000000000}\TypeLib]
@="{947EA1C4-C094-11D2-81CF-000000000000}"
"Version"="1.0"

<******* End of File *******>

<******* DataControlerClass.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{082EC300-7BDE-11D2-819B-000000000000}]
@="DataServer Class"
"AppID"="{082EC2F4-7BDE-11D2-819B-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{082EC300-7BDE-11D2-819B-000000000000}\LocalServer32]
@="D:\\ZQReport\\DATACO~1\\Debug\DATACO~1.EXE"

[HKEY_CLASSES_ROOT\CLSID\{082EC300-7BDE-11D2-819B-000000000000}\ProgID]
@="DataControler.DataServer.1"

[HKEY_CLASSES_ROOT\CLSID\{082EC300-7BDE-11D2-819B-000000000000}\TypeLib]
@="{082EC2F3-7BDE-11D2-819B-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{082EC300-7BDE-11D2-819B-
000000000000}\VersionIndependentProgID]
@="DataControler.DataServer"

<******* End of File *******>

<******* DataControlerInterface.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\Interface\{082EC2FF-7BDE-11D2-819B-000000000000}]
@="IDataServer"

[HKEY_CLASSES_ROOT\Interface\{082EC2FF-7BDE-11D2-819B-000000000000}\NumMethods]
@="21"

[HKEY_CLASSES_ROOT\Interface\{082EC2FF-7BDE-11D2-819B-000000000000}\ProxyStubClsid32]
@="{63B3BFA0-7BE2-11D2-819B-000000000000}"

<******* End of File *******>

<******* SecurityMgrClass.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{ED7E915C-D762-11D2-81B5-000000000000}]
@="SecuritySvr Class"
"AppID"="{8D2F4614-D772-11D2-81B5-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{ED7E915C-D762-11D2-81B5-000000000000}\LocalServer32]

@="D:\\ZQReport\\SECURI~1\\Debug\\SECURI~1.EXE"

[HKEY_CLASSES_ROOT\CLSID\{ED7E915C-D762-11D2-81B5-000000000000}\ProgID]
@="SecurityMgr.SecuritySvr.1"

[HKEY_CLASSES_ROOT\CLSID\{ED7E915C-D762-11D2-81B5-000000000000}\Programmable]

[HKEY_CLASSES_ROOT\CLSID\{ED7E915C-D762-11D2-81B5-000000000000}\TypeLib]
@="{8D2F4613-D77B-11D2-81B5-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{ED7E915C-D762-11D2-81B5-
000000000000}\VersionIndependentProgID]
@="SecurityMgr.SecuritySvr"

<******* End of File *******>

<******* SecurityMgrInterface.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\Interface\{8D2F461F-D77B-11D2-81B5-000000000000}]
@="ISecuritySvr"

[HKEY_CLASSES_ROOT\Interface\{8D2F461F-D77B-11D2-81B5-000000000000}\NumMethods]
@="7"

[HKEY_CLASSES_ROOT\Interface\{8D2F461F-D77B-11D2-81B5-000000000000}\ProxyStubClsid]
@="{00020424-0000-0000-C000-000000000046}"

[HKEY_CLASSES_ROOT\Interface\{8D2F461F-D77B-11D2-81B5-000000000000}\ProxyStubClsid32]
@="{8D2F461F-D77B-11D2-81B5-000000000000}"

[HKEY_CLASSES_ROOT\Interface\{8D2F461F-D77B-11D2-81B5-000000000000}\TypeLib]
@="{8D2F4613-D77B-11D2-81B5-000000000000}"
"Version"="1.0"

<******* End of File *******>

<******* SpecMgrClass.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\CLSID\{95FD24DE-D76F-11D2-81B5-000000000000}]
@="SpecSvr Class"
"AppID"="{AF56ECF4-D77B-11D2-81B5-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{95FD24DE-D76F-11D2-81B5-000000000000}\LocalServer32]
@="D:\\ZQReport\\SpecMgr\\Debug\\SpecMgr.exe"

[HKEY_CLASSES_ROOT\CLSID\{95FD24DE-D76F-11D2-81B5-000000000000}\ProgID]
@="SpecMgr.SpecSvr.1"

[HKEY_CLASSES_ROOT\CLSID\{95FD24DE-D76F-11D2-81B5-000000000000}\Programmable]

[HKEY_CLASSES_ROOT\CLSID\{95FD24DE-D76F-11D2-81B5-000000000000}\TypeLib]
@="{AF56ECF3-D77B-11D2-81B5-000000000000}"

[HKEY_CLASSES_ROOT\CLSID\{95FD24DE-D76F-11D2-81B5-000000000000}\VersionIndependentProgID]
@="SpecMgr.SpecSvr"

<******* End of File *******>

<******* SpecMgrInterface.reg *******>

REGEDIT4

[HKEY_CLASSES_ROOT\Interface\{95FD24DD-D76F-11D2-81B5-000000000000}]
@="ISpecSvr"

[HKEY_CLASSES_ROOT\Interface\{95FD24DD-D76F-11D2-81B5-000000000000}\NumMethods]
@="7"

[HKEY_CLASSES_ROOT\Interface\{95FD24DD-D76F-11D2-81B5-000000000000}\ProxyStubClsid]
@="{00020424-0000-0000-C000-000000000046}"

[HKEY_CLASSES_ROOT\Interface\{95FD24DD-D76F-11D2-81B5-000000000000}\ProxyStubClsid32]
@="{95FD24DD-D76F-11D2-81B5-000000000000}"

[HKEY_CLASSES_ROOT\Interface\{95FD24DD-D76F-11D2-81B5-000000000000}\TypeLib]
@="{95FD24D1-D76F-11D2-81B5-000000000000}"
"Version"="1.0"

<******* End of File *******>

Appendix C — Configuration and Help Files

!!! Data.cfg

```
YEAR : 1998-1999;
YEAR : 1999-2000;

DEPT : 0, (Chemical Engineering);
DEPT : 1, (Civil Engineering);
DEPT : 2, (Computer Science);
DEPT : 3, (Electronic Engineering);
DEPT : 4, (Mechanical Engineering );
DEPT : 5, (Engineering);
DEPT : 6, (Building Engineering);
DEPT : 7, (Physics);
DEPT : 8, (Mathematics);
DEPT : 9, (Chemistry );

TM_STEP : 600000;
```

```
!!! Help.cfg

COR_NUM : (COMP221);
COR_NUM : (COMP224);
COR_NUM : (COMP325);
COR_NUM : (COMP347);
COR_NUM : (COMP425);
COR_NUM : (COMP427);
COR_NUM : (COMP526);
COR_NUM : (COMP551);
COR_NUM : (COMP564);
COR_NUM : (COMP624);
COR_NUM : (COMP628);
COR_NUM : (COMP647);
COR_NUM : (COMP651);
COR_NUM : (COMP677);
COR_NUM : (COMP723);
COR_NUM : (COMP724);
COR_NUM : (COMP745);
COR_NUM : (ELEC351);
```

```
COR_NUM : (ELEC564);
COR_NUM : (ELEC624);
COR_NUM : (ELEC628);
COR_NUM : (ELEC647);
COR_NUM : (ELEC651);
COR_NUM : (ELEC677);
COR_NUM : (ELEC723);
COR_NUM : (ELEC724);
COR_NUM : (ELEC745);
COR_NUM : (ELEC771);

COR_DES : COMP221, (COMP 221
|Computer calculation and evaluation.
|Focus on the training of calculation skill.);
COR_DES : COMP224, (COMP 224
|Computer programming and methodology.
|Focus on the training of programming skill.);
COR_DES : COMP325, (COMP 325
|Basic mathematics and evaluation method.
|Focus on the training of calculation skill.);
COR_DES : COMP347, (COMP 347
|Computer operating system architecture.
|Focus on the training of design CPU skill.);
COR_DES : COMP425, (COMP 425
|Computer Web design and programming.
|Focus on the training of calculation skill.);
COR_DES : COMP427, (COMP 427
|Object oriented design and methodology.
|Focus on the training of calculation skill.);
COR_DES : COMP526, (COMP 526
|Computer architecture and design.
|Focus on the training of designing CPU processor.);
COR_DES : COMP551, (COMP 551
|Data structure and algorithm.
|Focus on the training of programming skill.);
COR_DES : COMP564, (COMP 564
|Computer graphics and their design and programming.
|Focus on the training of calculation skill.);
COR_DES : COMP624, (COMP 624
|High performance processor design principle.
|Focus on the training of calculation skill.);
COR_DES : COMP628, (COMP 628 - Computer Systems Design
|Prerequisite: COMP 546.
|
|Migration from Von Neumann to parallel processing
|architectures: fine grained and coarse grained
|concurrency, multi-threaded computers, massively
|parallel computers, fundamental problems in hardware
|architecture, and memory consistency. Embedding of
|algorithms on shared-memory and message-passing
|architectures. Parallel programming supports: a
|parallel program model, embedding of parallel
|programs on multiprocessors and distributed systems.
|Key concepts in distributed systems.);
COR_DES : COMP647, (COMP 647
|Object oriented design, future trends of software.
|Focus on the training of design skill.);
```

```
COR_DES : COMP651, (COMP 651
|Object oriented database design and programming.
|Focus on the training of database design skill.);
COR_DES : COMP677, (COMP 677
|Computer multi-medias, including Web and Game.
|Focus on the broad knowledge in the computer field.);
COR_DES : COMP723, (COMP 723
|Distributed computer calculating, including
|distributed protocol design and implementation.);
COR_DES : COMP724, (COMP 724
|Parallel computer system, including the design
|of high performance computer system.);
COR_DES : COMP745, (COMP 745
|Advanced computer topics in the field of computer.
|science. topics is selected according to the focus.);
COR_DES : ELEC551, (ELEC 551
|Electroics circuits and design.
|Focus on the training of basic knowledge.);
COR_DES : ELEC564, (ELEC 564
|Semi-conductor theory and their applications.
|Focus on the training of knowledge skill.);
COR_DES : ELEC624, (ELEC 624
|Hard ware design for the computer machine interface.
|Focus on the training of design skill.);
COR_DES : ELEC628, (ELEC 628
|A large scale electronics circuit design.
|Focus on the training of broad knowledge.);
COR_DES : ELEC647, (ELEC 647
|Computer interface with electronics sensor.
|Focus on the training of knowledge.);
COR_DES : ELEC651, (ELEC 651
|Multi-topics on the field of electronics engineering.
|Focus on the training of broad knowledge.);
COR_DES : ELEC677, (ELEC 677
|Software engineering for electronics engineer.
|Focus on the training of computer skill.);
COR_DES : ELEC723, (ELEC 723
|New frontier in the electronics engineering.
|Advanced topics in recent development. ;
COR_DES : ELEC724, (ELEC 724
|Computer communication and design.
|Focus on the training of knowledge base.);
COR_DES : ELEC745, (ELEC 745
|Computer communication hardware and application.
|Focus on the training of simulation skill.);
COR_DES : ELEC771, (ELEC 771
|Protocol design and validation.
|Focus on the design of communication protocol.);
```

Appendix D — Database Files

<******* File student_course.SQL *******>

```
create table Student(
stid        number(7) not null,
stname      varchar2(20),
sttype      number(2),
deptno      number(4),
password    char(4)
);

create table Course(
courseid    number(4),
courseno    varchar2(8) not null,
title       varchar2(30),
term        number(2) not null,
section     varchar2(6),
days        varchar2(7),
starttime   varchar2(5),
endtime     varchar2(5),
room        varchar2(6),
type        number(2),
deptno      number(4)
);

create table Regist_to(
stid        number(7) not null,
courseid    number(4) not null
);

create table Taught_by(
courseid    number(4) not null,
teacherid   number(4) not null
);

create table Dept(
deptno      number(4) not null,
deptname    varchar2(20) not null
);

create table Teacher(
```

```
teacherid   number(7) not null,
teachername varchar2(20),
deptno      number(4)
);

create table CourseManager(
managerid   number(7) not null,
managername varchar2(20),
deptno      number(4),
password    char(4) not null
);

create table SuperUser(
superuserid number(7) not null,
supername   varchar2(20),
password    char(4) not null
);
```

<***** File insertvalues.SQL *******>**

```
insert into Student values(6354728,'john ken',0,0,'0000');
insert into Student values(7262782,'kim lee',0, 2,'0000');
insert into Student values(6365438,'mei heen',0,0,'0000');
insert into Student values(7262781,'keem liree',0,0,'0000');
insert into Student values(1234728,'jarn kenmy',0,2,'0000');
insert into Student values(4352782,'johnaic lewe',0,3,'0000');
insert into Student values(6354123,'love yrken',0,2,'0000');
insert into Student values(7123782,'sean ron',0, 2,'0000');

insert into Course values(1,'comp554','software engneening',1,'Lec
AA','---j---','17:40','20:10','h634',0,2);
insert into Course values(2,'comp561','elementary numerical',2,'Lec
WW','-t-----','17:40','20:10','h614',0,2);
insert into Course values(3,'comp628','computer system design',1,'Lec
AA','---j---','20:25','22:55','h627',0,2);
insert into Course values(4,'comp635','topics/scientific
computation',1,'Lec AA','---j---','17:40','20:10','h517',0,2);
insert into Course values(5,'comp641','comparat study',2,'Lec AA','m---
---','20:25','22:55','h310',0,2);
insert into Course values(6,'comp642','compiler design',1,'Lec U','m-w-
---','13:15','14:30','h634',0,2);
insert into Course values(7,'comp642','compiler design',2,'Lec XX','-t-
----','17:40','20:10','h637',0,2);
insert into Course values(8,'comp646','computer networks and
protocol',1,'Lec AA','-t-----','17:40','20:10','h222',0,2);
insert into Course values(9,'comp646','computer networks and
protocol',2,'Lec XX','---j---','17:40','20:10','h611',0,2);

insert into Regist_to values( 7123782,8);
insert into Regist_to values( 7123782,1);
insert into Regist_to values( 7123782,3);
insert into Regist_to values( 7262782,8);
insert into Regist_to values( 7262782,4);
insert into Regist_to values( 6354728,8);
insert into Regist_to values( 6354728,6);
insert into Regist_to values( 4352782,5);
```

```
insert into Taught_by values(1,1);
insert into Taught_by values(2,1);
insert into Taught_by values(3,2);
insert into Taught_by values(4,3);
insert into Taught_by values(5,1);
insert into Taught_by values(6,4);
insert into Taught_by values(7,4);
insert into Taught_by values(8,5);
insert into Taught_by values(9,6);

insert into Dept values(2,'computer sci.');
insert into Dept values(1, civil');
insert into Dept values 3,'elec. e.');
insert into Dept values(4, mechanical');
insert into Dept values(5,'chemical');

insert into Teacher values 1,'peter john',0);
insert into Teacher values(2,'elton john',0);
insert into Teacher values 3,'joan lie',2);
insert into Teacher values(4,'john lee',2);
insert into Teacher values(5,'jakie chen',0);
insert into Teacher values 6,'ling tuo',1);

insert into CourseManager values(111,'man he',0,'0000');
insert into CourseManager values(111,'man he',1,'0000');
insert into CourseManager values(222,'qiang zhang',2,'0000');
insert into CourseManager values(222,'qiang zhang',3,'0000');
insert into CourseManager values(333,'helena',4,'0000');

insert into SuperUser values(11111,'Qiang Zhang','1217');
insert into SuperUser values(22222,'Man He','6612');
```