



**UNIVERSITY OF
GREENWICH**

University of Greenwich ID Number: 001324892

FPT Student ID Number: GCD220148

Module Code: COMP1551

Module Assessment Title: Application Development

Lecturer Name: Nguyen The Nghia

Submission Date: 19/08/2024

TASK 1 : SYSTEM DESCRIPTION	3
1. Introduction:	3
1.1) Purpose:	3
1.2) Project scope:.....	3
1.3) Overview:	3
2. Introduction:	3
2.1) Overall Description Product:	3
2.2) Operational Environment:	3
3. System Features:	3
3.1) Description:	4
3.2) Functional Requirements:	4
4. User Interface Requirements:.....	4
5. Platform Requirements:	4
6. Quality attributes:.....	4
6.1) Performance:	4
6.2) Security:	4
6.3) Safety:.....	4
TASK 2: UML DIAGRAM	4
1. Class Diagram:	4
2. Use Case Diagram:.....	6
TASK 3: DEVELOP THE DESKTOP INFORMATION SYSTEM:	9
3.1) Class:.....	9
TASK 4: MANUAL TEST:.....	55
DATABASE DESIGN:	56
TASK 5: CONCLUSION:.....	57
APPENDIX:.....	57

TASK 1 : SYSTEM DESCRIPTION

1. Introduction:

This project presents a Desktop Information System for an education center transitioning from paper-based to digital record-keeping. Developed as a C# Console Application, it manages data for Teaching Staff, Administration, and Students, showcasing Object-Oriented Programming principles through a class hierarchy. The system implements functionality for adding, viewing, editing, and deleting records, adhering to software engineering best practices.

1.1) Purpose:

The purpose of this SRS document is to provide a comprehensive overview of the Desktop Information System, detailing high-level requirements and system functionality for Teaching Staff, Administration, and Students. It serves as a primary reference for stakeholders throughout the development lifecycle.

1.2) Project scope:

The project aims to create a Windows-based application that modernizes the existing bookkeeping system. It will feature an intuitive user interface designed with Windows Forms and will employ a secure MySQL database hosted in a Docker container. The application will include modules for user information management, offering functionalities such as adding, viewing, editing, and deleting records. To ensure data protection, the system will implement user authentication and role-based access control. Furthermore, it will utilize a three-layer architecture to enhance maintainability and scalability for future development.

1.3) Overview:

The Desktop Information System replaces a paper-based system with a modern application designed for efficient data management. It emphasizes security, usability, and adherence to Object-Oriented Programming principles..

2. Introduction:

2.1)Overall Description Product:

The product is a Windows-based application designed to centralize user data management through an intuitive interface. It leverages a MySQL database hosted in a Docker container, providing secure access and scalability. The application features user-friendly data entry forms and navigation controls, ensuring a seamless user experience. Additionally, it is built using a three-layer architecture, which separates the presentation, business logic, and data access layers, promoting maintainability and scalability for future enhancements.

2.2) Operational Environment:

The Desktop Information System runs on Windows 10 and above, utilizing a MySQL database in a Docker container for efficient deployment. Network connectivity is essential for secure access to the database. Designed for an educational setting, it provides a stable, user-friendly interface for teaching staff, administrative personnel, and students.

3. System Features:

3.1) Description:

The system allows users to:

- Add new data
- View all existing data
- View data by groups
- Edit and delete records
- Exit

3.2) Functional Requirements:

- **User Authentication:** Secure login for the System Admin with role-based access control.
- **Data Handling:** Efficiently add, view, edit, and delete records with validation and confirmation prompts.

4. User Interface Requirements:

The interface will feature a consistent Windows Forms design, clear navigation, data entry forms with validation, and tables with filtering capabilities.

5. Platform Requirements:

The application will run on Windows 10 and later, requiring 4 GB RAM and a dual-core processor. It will use a MySQL database on port 3307, necessitating Docker installation.

6. Quality attributes:

6.1) Performance:

Fast response times and efficient data retrieval.

6.2) Security:

Admin-only access, encrypted data storage, and secure authentication.

6.3) Safety:

Comprehensive error handling and role-based access controls.

TASK 2: UML DIAGRAM

1. Class Diagram:

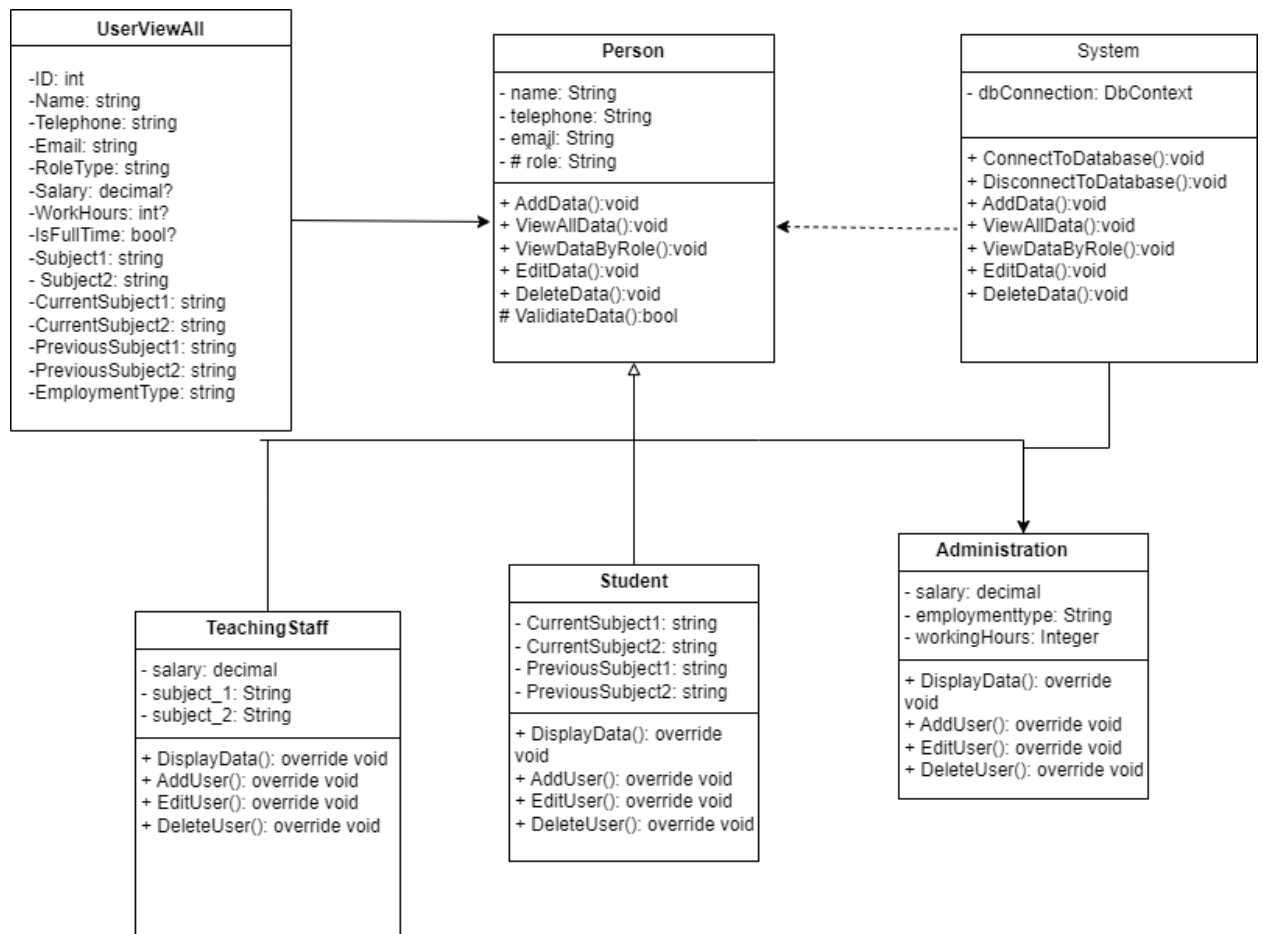


Figure 1: Class Diagram

2. Use Case Diagram:

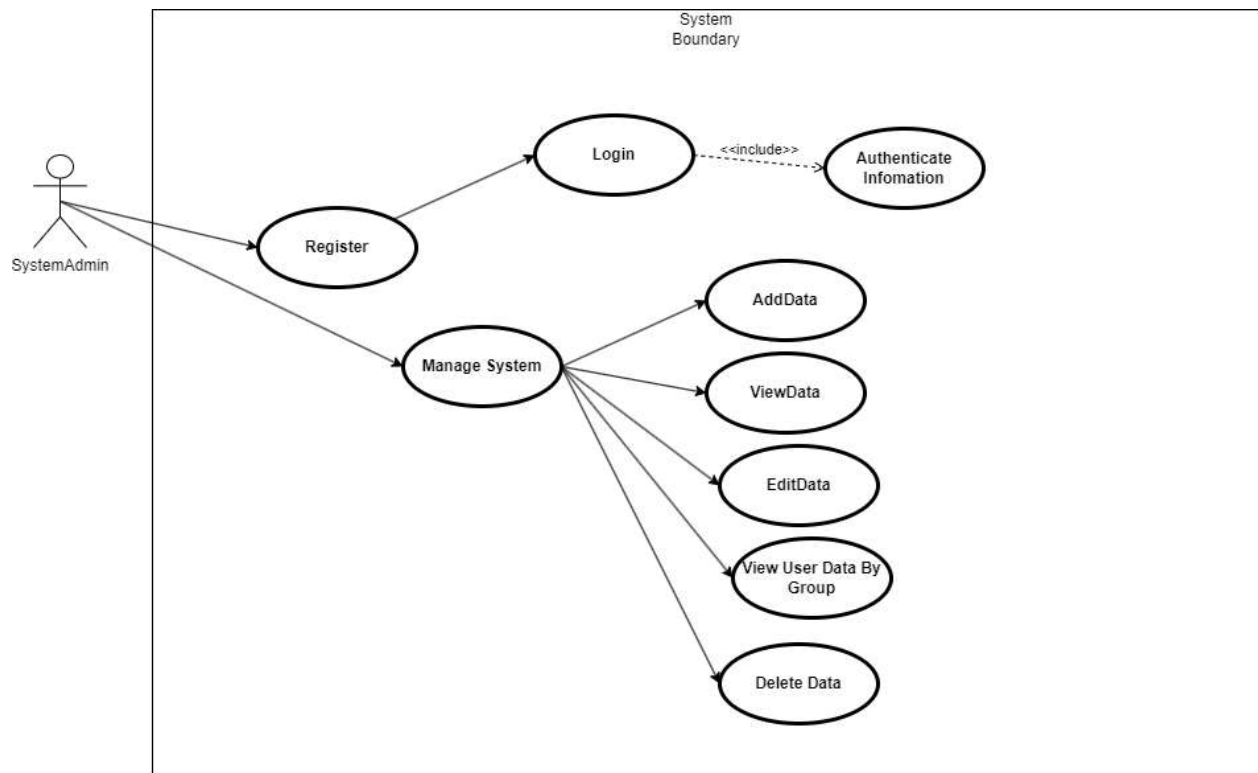


Figure 2: Overall Use Case

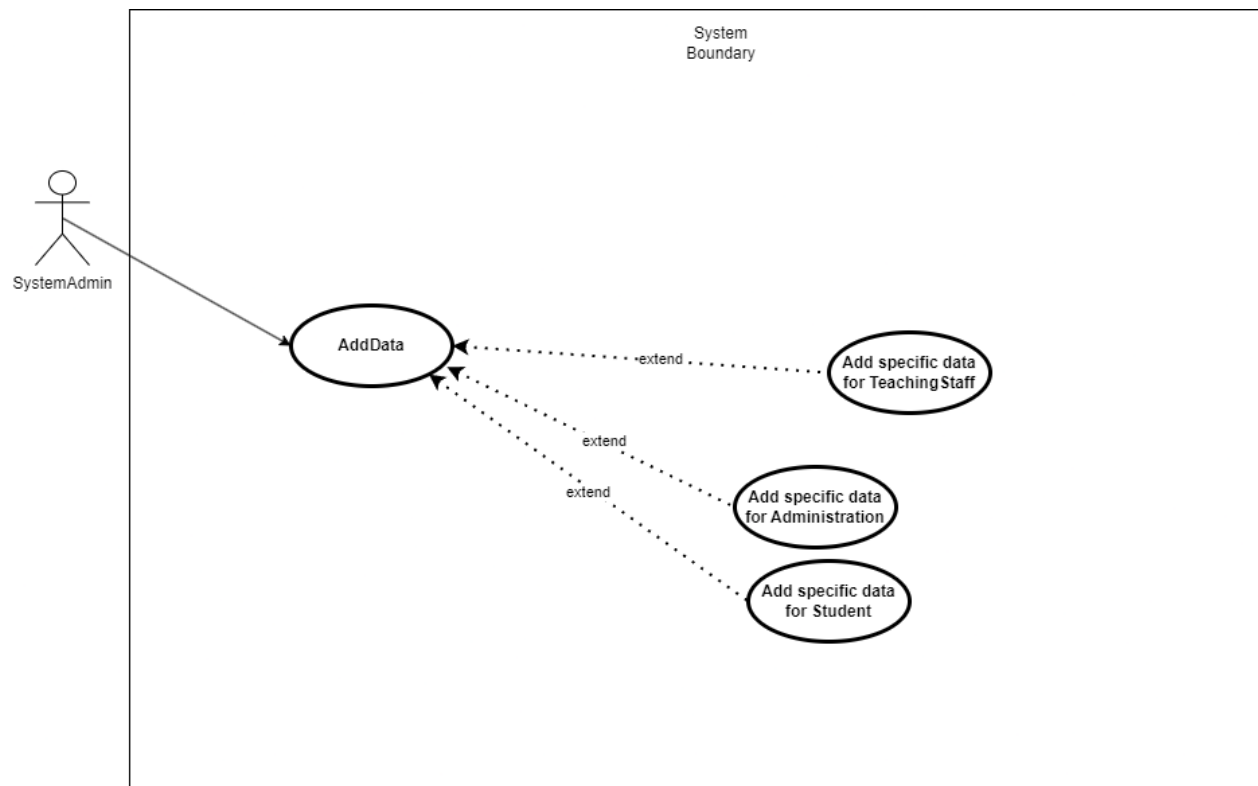


Figure 3: Add Function

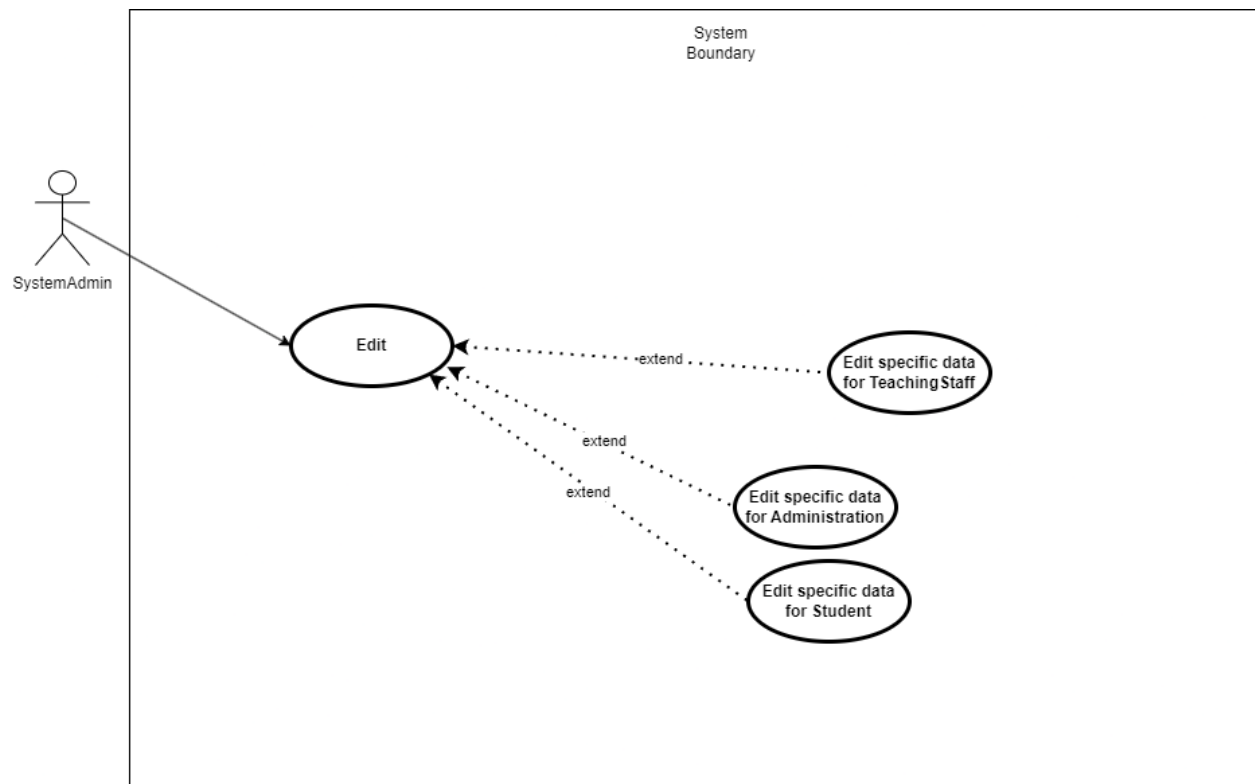


Figure 4: Edit Function

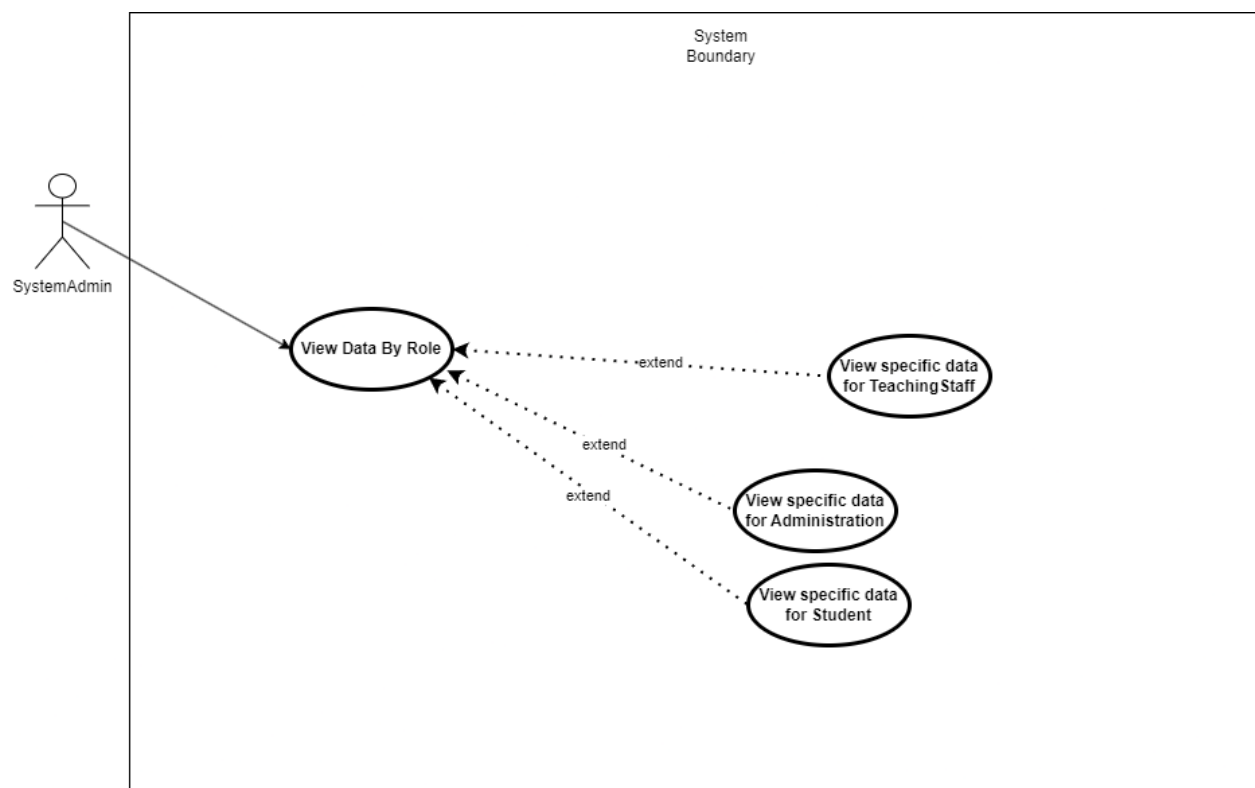


Figure 5: View By Role Function

The SystemAdmin is essential to the management of the Desktop Information System, starting with the registration of new user accounts. They input vital information such as usernames and passwords, ensuring that all new users can access the system securely. Once registered, the SystemAdmin can log in and gain access to a wide range of administrative functionalities.

After logging in, the SystemAdmin can efficiently manage the system by adding, viewing, editing, and deleting data related to students(current subjects and previous subjects), teaching staff (salary, subjects) , and administration(workhours, employment type and salary). They can also view user data by group

Additionally, the SystemAdmin is responsible for maintaining the accuracy of user information. This includes editing existing records to keep them current and deleting obsolete data to uphold the integrity of the database. Through these critical tasks, the SystemAdmin ensures a secure and well-structured environment within the information system.

TASK 3: DEVELOP THE DESKTOP INFORMATION SYSTEM:

3.1) Class:

So after implementing the SRS and the UML diagram, here is the code for the system code:

First of all, we have the code for class IUser, here is the code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ManageSystem.Models
{
    public interface IUser
    {
        string Name { get; }
        string Email { get; }
        string Telephone { get; }
        void AddUser();
        void EditUser();
        void DeleteUser();
    }
}
```

```
public interface IUser
```

- public interface IUser sets up a contract for user-related behaviors. By declaring it as public, It's making it accessible throughout the application, encouraging other classes to implement it and ensuring consistency in how users are managed.

```
public interface IUser
{
    string Name { get; }
    string Email { get; }
    string Telephone { get; }
    void AddUser();
    void EditUser();
    void DeleteUser();
}
```

- **Name:** The string Name { get; } property allows to retrieve the user's name but doesn't allow it to be changed externally. This encapsulation keeps the integrity of the user data intact.
- **Email:** Similarly, string Email { get; } provides read-only access to the user's email, ensuring that email addresses remain secure and consistent.
- **Telephone:** The string Telephone { get; } property follows the same principle, allowing access without permitting modifications from outside.

```
void AddUser();
void EditUser();
void DeleteUser();
```

- **AddUser():** The void AddUser(); method outlines a clear intention. It signals that any implementing class must provide a way to add a user. This method acts as an invitation for classes to define how they manage user creation.
- **EditUser():** With void EditUser();, It's establishing a requirement that any implementing class must detail how user information can be modified. This encourages flexibility and adaptability in user management.
- **DeleteUser():** Finally, void DeleteUser(); indicates that classes must implement a method for user deletion. This method reinforces the idea that user lifecycle management is crucial in the system.

After discussing, here is the full code for IUser with commenting:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ManageSystem.Models
{
    // Define an interface named 'IUser' that represents the contract for user-related operations.
    public interface IUser
    {
        // Property for getting the user's name.
        string Name { get; }
    }
}
```

```

        // Property for getting the user's email address.
        string Email { get; }

        // Property for getting the user's telephone number.
        string Telephone { get; }

        // Method for adding a new user. The implementation will define the details.
        void AddUser();

        // Method for editing an existing user. The implementation will define the details.
        void EditUser();

        // Method for deleting an existing user. The implementation will define the details.
        void DeleteUser();
    }
}

```

Then after the class IUser, is the class User(abstract), which is the parent class:

```
public abstract class User : IUser
```

- The class public abstract class User : IUser defines an abstract class that implements the IUser interface. This means that the User class cannot be instantiated directly; instead, it serves as a blueprint for other user types (e.g., Admin, Student).

```

public int ID { get; set; }
private string _Name = "";
private string _Telephone = "";
private string _Email = "";
private RoleType _RoleType;

```

- **ID:** The public property ID { get; set; } provides access to the unique identifier for each user, allowing both retrieval and assignment of the ID.
- **Private Fields:**
 - _Name, _Telephone, and _Email are private string variables initialized as empty strings. They store the user's name, telephone number, and email address, respectively.
 - _RoleType is a private variable of type RoleType, which defines the role assigned to the user.

```

public User(string name, string telephone, string email, RoleType role)
{
    _Name = name;
    _Telephone = telephone; // Use the property setter for validation
    _Email = email;
    _RoleType = role;
}

```

- **Name:** The public string Name property allows external access to the user's name, enabling modification through its setter.

- **Telephone:** The public virtual string Telephone property provides access to the telephone number, with a virtual setter that allows for potential overriding in derived classes.
- **Email:** The public string Email property allows access to the user's email address, with both getter and setter for external manipulation.
- **Role:** The public RoleType Role property provides access to the user's role, with a setter that allows modification while maintaining encapsulation.

```
public virtual void AddUser()
{
}

public virtual void EditUser()
{
}

public virtual void DeleteUser()
{
}
```

- **AddUser():** The public virtual void AddUser() method is defined but not yet implemented, allowing derived classes to provide specific logic for adding users.
- **EditUser():** Similarly, the public virtual void EditUser() method serves as a placeholder for subclasses to define how user information can be edited.
- **DeleteUser():** The public virtual void DeleteUser() method is also a virtual placeholder, inviting subclasses to implement deletion logic tailored to their needs.

And here is the fully commented code for User class:

```
using ManageSystem.DAL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ManageSystem.Models
{
    // Abstract class 'User' implementing the 'IUser' interface.
    // This class provides a base implementation for user-related operations.
    public abstract class User : IUser
    {
        // Public property for storing the unique identifier for the user.
        public int ID { get; set; }

        // Private fields to store user details.
        private string _Name = "";
        private string _Telephone = "";
        private string _Email = "";
        private RoleType _RoleType;

        // Constructor to initialize a user with basic details.
        public User(string name, string telephone, string email, RoleType role)
        {
            _Name = name;
            _Telephone = telephone; // Use the property setter for validation
            _Email = email;
        }
    }
}
```

```

        _RoleType = role;
    }

    // Public property to get and set the user's name.
    public string Name
    {
        get { return _Name; }
        set { _Name = value; }
    }

    // Virtual property for getting and setting the user's telephone number.
    // Can be overridden in derived classes.
    public virtual string Telephone
    {
        get { return _Telephone; }
        set
        {
            _Telephone = value;
        }
    }

    // Public property to get and set the user's email.
    public string Email
    {
        get { return _Email; }
        set { _Email = value; }
    }

    // Property to get and set the user's role.
    public RoleType Role
    {
        get { return _RoleType; }
        set { _RoleType = value; }
    }

    // Virtual method to add a new user. Can be overridden in derived classes.
    public virtual void AddUser()
    {
        // Implementation to be provided by derived classes.
    }

    // Virtual method to edit an existing user. Can be overridden in derived classes.
    public virtual void EditUser()
    {
        // Implementation to be provided by derived classes.
    }

    // Virtual method to delete a user. Can be overridden in derived classes.
    public virtual void DeleteUser()
    {
        // Implementation to be provided by derived classes.
    }
}

```

After that, we will implement the child class of the User class, is the TeachingStaff class:

```
public class TeachingStaff : User
```

- The line `public class TeachingStaff : User` defines the TeachingStaff class, which inherits from the User abstract class. This inheritance means that TeachingStaff will have all properties and methods from the User class, while also introducing specific attributes and behaviors unique to teaching staff.

```
public decimal Salary { get; private set; }
public string Subject1 { get; private set; }
public string Subject2 { get; private set; }
```

- Salary: The property public decimal Salary { get; private set; } stores the salary of the teaching staff. Having a private setter restricts modifications from outside the class, ensuring the integrity of salary data.
- Subject1 and Subject2: The properties public string Subject1 { get; private set; } and public string Subject2 { get; private set; } hold the subjects taught by the staff member. Like Salary, these properties have private setters to maintain control over their values.

```
public TeachingStaff(string name, string telephone, string email, decimal salary, string subject1,
string subject2)
: base(name, telephone, email, RoleType.TeachingStaff)
{
    Salary = salary;
    Subject1 = subject1;
    Subject2 = subject2;
}
```

- The constructor public TeachingStaff(string name, string telephone, string email, decimal salary, string subject1, string subject2) initializes a new instance of the TeachingStaff class. It calls the base class constructor using : base(name, telephone, email, RoleType.TeachingStaff) to set the inherited properties, while also initializing the Salary, Subject1, and Subject2 properties.

```
public override void AddUser()
{
    var addteachstaff = new LoginRepository();
    addteachstaff.AddTeachingStaff(this);
}
public override void EditUser()
{
    var editteachstaff = new LoginRepository();
    editteachstaff.UpdateTeachingStaff(this);
}
public override void DeleteUser()
{
    var deleteteachstaff = new LoginRepository();
    deleteteachstaff.DeleteTeachingStaff(this.ID);
}
```

- AddUser(): The public override void AddUser() method implements the logic for adding a teaching staff member. It creates an instance of LoginRepository and calls AddTeachingStaff(this), passing the current instance to be added to the database.
- EditUser(): The public override void EditUser() method provides functionality to update the teaching staff member's details. Similar to AddUser(), it utilizes LoginRepository to call UpdateTeachingStaff(this), allowing for modifications in the database.

- `DeleteUser()`: The public override `void DeleteUser()` method is responsible for deleting a teaching staff member. It uses `LoginRepository` to call `DeleteTeachingStaff(this.ID)`, passing the staff member's ID to remove the record from the database.

So in the override method, is the code for implementing data in the database, so before implementing the database, user have to be login account to be a `SystemAdmin` and after that, a test connection to the database will be conducted. In the code for implementing database, First of all, we will implement the connection to the data base, I will explain below:

```
<connectionStrings>
  <add name="MyDatabaseConnection"
    connectionString="Server=localhost;Port=3307;Database=c-sharp;Uid=root;Pwd=helloimkwang2;"
    providerName="MySQL.Data.MySqlClient" />
</connectionStrings>
```

The configuration file includes a section that defines database connection strings utilized by the application, with the `add` element specifying a unique identifier for the connection string (`MyDatabaseConnection`). The `connectionString` details how to connect to the database, including the server address (`localhost`), port number (`3307`), database name (`c-sharp`), username for authentication (`root`), and the associated password (`helloimkwang2`). Additionally, the `providerName` indicates the data provider used for the connection, specifically `MySQL.Data.MySqlClient`, signifying that the application interfaces with a MySQL database.

Then the registration process starts with the user filling out a form to provide their details. Once submitted, the input is validated, and if everything checks out, the data is sent to the backend service for user creation in the database. Finally, the user receives a confirmation of successful registration. Below is the code for this method :

```
public void RegisterUser(string username, string passwordHash)
```

This method registers a new user by accepting a username and a hashed password.

```
using (MySqlConnection conn = new MySqlConnection(_connectionString))
```

A new connection to the MySQL database is created using a connection string stored in `_connectionString`. The `using` statement ensures that the connection is properly disposed of when done.

```
conn.Open();
```

This line opens the database connection to allow for executing queries.

```
string query = "INSERT INTO SystemAdmin(Username, PasswordHash) VALUES (@username, @passwordhash)";
```

The SQL query prepares to insert a new record into the SystemAdmin table with placeholders for parameters.

```
using (var cmd = new MySqlCommand(query, conn))
```

```
cmd.Parameters.AddWithValue("@username", username);  
cmd.Parameters.AddWithValue("@passwordhash", passwordHash);
```

Parameters are added to the command to prevent SQL injection attacks, binding the provided values to the placeholders in the query.

```
cmd.ExecuteNonQuery();
```

This line executes the command, performing the actual insert operation in the database without returning any data.

And here is the full code with commented to fully understand:

```
public void RegisterUser(string username, string passwordHash)  
{  
    // Create a new MySqlConnection using the connection string  
    using (MySqlConnection conn = new MySqlConnection(_connectionString))  
    {  
        // Open the connection to the database  
        conn.Open();  
  
        // SQL query to insert a new user into the SystemAdmin table  
        string query = "INSERT INTO SystemAdmin(Username, PasswordHash) VALUES (@username,  
@passwordhash)";  
  
        // Create a command to execute the SQL query  
        using (var cmd = new MySqlCommand(query, conn))  
        {  
            // Add parameters to prevent SQL injection  
            cmd.Parameters.AddWithValue("@username", username); // Binds the username parameter  
            cmd.Parameters.AddWithValue("@passwordhash", passwordHash); // Binds the password hash  
parameter  
  
            // Execute the command, performing the insert operation  
            cmd.ExecuteNonQuery(); // Executes the insert without returning any results  
        } // cmd is disposed here  
    } // conn is disposed here  
}
```

After implementing the Data Access Layer (DAL), we will now move to the Business Logic Layer (BLL) where we will utilize the DAL methods. In the BLL, the RegisterUser method will be defined to handle user registration, calling the

corresponding method from the DAL to insert the user data into the database. Here is the code for this structure:

```
private readonly LoginRepository roleRepository;
```

Declares a read-only instance of LoginRepository, which will be used to access data-related methods for user registration.

```
public RoleService()  
{ roleRepository = new LoginRepository(); }
```

The constructor initializes the roleRepository instance when a RoleService object is created. This setup allows the BLL to interact with the DAL.

```
public void RegisterUser(string username, string password)  
{ roleRepository.RegisterUser(username, password); }
```

This method acts as a wrapper for the RegisterUser method in the DAL, passing the username and password to it. It facilitates user registration by delegating the database operation to the repository.

And here is the full code for it being commented:

```
// RoleService class manages user-related operations  
public class RoleService  
{  
    // Read-only instance of LoginRepository for data access  
    private readonly LoginRepository roleRepository;  
  
    // Constructor to initialize the LoginRepository  
    public RoleService()  
    {  
        roleRepository = new LoginRepository(); // Instantiate the repository  
    }  
  
    // Method to register a new user  
    public void RegisterUser(string username, string password)  
    {  
        // Call the RegisterUser method from the repository to insert user data into the database  
        roleRepository.RegisterUser(username, password);  
    }  
}
```

And then the registerbutton_Click_1 method manages the user registration process in the UI. It captures input from the registration form, performs validation checks, and interacts with the Business Logic Layer (BLL) to register the user. By utilizing the RoleService to access the Data Access Layer (DAL), it ensures that user data is securely stored in the database while providing feedback to the user based on the registration outcome. And here is the code for it with explanation by using comment :

```

private void registerbutton_Click_1(object sender, EventArgs e)
{
    // Capture user inputs from the form
    string username = txtusername.Text; // Get the username
    string password = txtpassword.Text; // Get the password
    string retypePassword = txtretypebox.Text; // Get the re-entered password

    // Validate that none of the fields are empty
    if (string.IsNullOrEmpty(username) ||
        string.IsNullOrEmpty(password) ||
        string.IsNullOrEmpty(retypePassword))
    {
        // Show an error message if any field is empty
        MessageBox.Show("Please fill in all fields.", "Error Message", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
        return; // Exit the method
    }

    // Validate that the passwords match
    if (password != retypePassword)
    {
        // Show an error message if passwords do not match
        MessageBox.Show("Passwords do not match. Please retype your password.", "Error Message",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return; // Exit the method
    }

    // Attempt to register the user
    try
    {
        roleService.RegisterUser(username, password); // Call the BLL method to register the user
        // Show success message if registration is successful
        MessageBox.Show("Registration successful! Please log in.", "Success", MessageBoxButtons.OK,
        MessageBoxIcon.Information);

        // Optionally, open the login form
        Loginform lform = new Loginform();
        lform.Show(); // Show the login form
    }
    catch (Exception ex)
    {
        // Handle any exceptions that occur during registration
        MessageBox.Show($"Registration failed: {ex.Message}", "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }
}

```

After we completing the user’s registration first, then we will move to the “Login” method here, which is named as “Validate User”:

```

public bool ValidateUser(string username, string password)
{
    // SQL query to select a user based on username and password hash
    string selectData = "SELECT * FROM SystemAdmin WHERE username = @username AND passwordhash = @passwordhash";

    // Create a new MySqlConnection using the connection string
    using (MySqlConnection conn = new MySqlConnection(_connectionString))
    {
        // Open the connection to the database
        conn.Open();
    }
}

```

```

        // Create a command to execute the SQL query
        using (MySqlCommand cmd = new MySqlCommand(selectData, conn))
        {
            // Add parameters to prevent SQL injection
            cmd.Parameters.AddWithValue("@username", username);
            cmd.Parameters.AddWithValue("@passwordhash", password);

            // Execute the query and read the results
            using (MySqlDataReader reader = cmd.ExecuteReader())
            {
                // Return true if a matching record is found
                return reader.Read();
            }
        }
    }
}

```

Then the ValidateUser method is part of the Business Logic Layer (BLL) that handles user authentication. It serves as an intermediary between the presentation layer (UI) and the Data Access Layer (DAL). This method takes a username and password as parameters and calls the ValidateUser method from the roleRepository, which interacts with the database to verify the user's credentials.

```

public bool ValidateUser(string username, string password)
{
    // Call the ValidateUser method in the roleRepository to authenticate the user
    // It returns true if the username and password match a record in the database
    return roleRepository.ValidateUser(username, password);
}

```

And after that, in the UI, The Loginbtn_Click method handles the user login process in a Windows Forms application. It first checks if the username and password fields are empty, displaying an error message if they are. If both fields are filled, it validates the credentials using the roleService. If the credentials are correct, it shows a success message, opens the main application form, and hides the login form. If the login fails, an error message is displayed, informing the user of invalid credentials. This method ensures a smooth and informative login experience.

```

private void Loginbtn_Click(object sender, EventArgs e)
{
    // Check if username or password fields are empty
    if (string.IsNullOrEmpty(username.Text) || string.IsNullOrEmpty(password.Text))
    {
        // Show an error message prompting the user to fill in the fields
        MessageBox.Show("Please fill in all blank fields", "Error Message", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }
    else
    {
        // Validate the user's credentials using the roleService
        bool isValidUser = roleService.ValidateUser(username.Text, password.Text);

        // Check if the user is valid
        if (isValidUser)
        {
            // Show a success message if login is successful

```

```

        MessageBox.Show("Login Successful", "Success", MessageBoxButtons.OK,
        MessageBoxIcon.Information);

        // Open the main application form
        MainForm mform = new MainForm();
        mform.Show();
        this.Hide(); // Hide the login form
    }
    else
    {
        // Show an error message if login fails
        MessageBox.Show("Invalid username or password", "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }
}
}

```

After implantiing successfully the method for login, we will move to the first function of the application is the “Add New Data” for user, the first class we want to add data is the “Teaching Staff” class:

```

public void AddTeachingStaff(TeachingStaff teachingStaff)
{
    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        conn.Open(); // Open the database connection

        // Insert into Users table first to get UserId
        try
        {
            // SQL query to insert a new user into the Users table
            string query = @"
            INSERT INTO Users (Name, Telephone, Email, RoleType)
            VALUES (@Name, @Telephone, @Email, 'TeachingStaff');";

            int userId; // Variable to hold the new UserId

            // Execute the insert command for the Users table
            using (var cmd = new MySqlCommand(query, conn))
            {
                // Add parameters to the command
                cmd.Parameters.AddWithValue("@Name", teachingStaff.Name);
                cmd.Parameters.AddWithValue("@Telephone", teachingStaff.Telephone);
                cmd.Parameters.AddWithValue("@Email", teachingStaff.Email);

                cmd.ExecuteNonQuery(); // Execute the insert command

                // SQL query to get the last inserted UserId
                string getQuery = "SELECT LAST_INSERT_ID();";
                using (var idCmd = new MySqlCommand(getQuery, conn))
                {
                    // Execute the command and convert the result to an integer
                    userId = Convert.ToInt32(idCmd.ExecuteScalar());
                }
            }

            // SQL query to insert the teaching staff details into the TeachingStaff table
            string staffQuery = @"
            INSERT INTO TeachingStaff (UserId, Salary, Subject1, Subject2)
            VALUES (@UserId, @Salary, @Subject1, @Subject2);";

```

```

        // Execute the insert command for the TeachingStaff table
        using (var staffCmd = new MySqlCommand(staffQuery, conn))
        {
            // Add parameters to the command
            staffCmd.Parameters.AddWithValue("@UserId", userId);
            staffCmd.Parameters.AddWithValue("@Salary", teachingStaff.Salary);
            staffCmd.Parameters.AddWithValue("@Subject1", teachingStaff.Subject1);
            staffCmd.Parameters.AddWithValue("@Subject2", teachingStaff.Subject2);

            staffCmd.ExecuteNonQuery(); // Execute the insert command
        }
    }
    catch (Exception ex)
    {
        // Show an error message if an exception occurs
        MessageBox.Show($"Error adding Teaching Staff: {ex.Message}");
    }
} // The connection is automatically closed and disposed of here
}

```

The AddTeachingStaff method is responsible for inserting a new teaching staff member into a database using MySQL. It begins by establishing a connection to the database and opens it.

First, it inserts the staff member's details into the Users table, setting their role as "TeachingStaff." After executing the insert command, it retrieves the UserId of the newly created user using LAST_INSERT_ID().

Next, it inserts the corresponding details into the TeachingStaff table, linking the newly created UserId with their salary and subjects taught. If any errors occur during this process, an error message is displayed to the user. The use of using statements ensures that resources are disposed of properly, maintaining good database management practices.

Continue to view the user data, we will use this code to see relevant data for the class:

```

public List<TeachingStaff> GetTeachingStaffList()
{
    // Initialize a list to hold the teaching staff objects
    var teachingStaffList = new List<TeachingStaff>();

    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        // SQL query to select relevant details from TeachingStaff and Users tables
        string query = @"
        SELECT U.Name, U.Telephone, U.Email, T.UserId, T.Salary, T.Subject1, T.Subject2
        FROM TeachingStaff T
        JOIN Users U ON T.UserId = U.Id";

        // Create a command object to execute the query
        var cmd = new MySqlCommand(query, conn);
        conn.Open(); // Open the database connection

        try
        {
            // Execute the query and use a DataReader to read the results
            using (var reader = cmd.ExecuteReader())
            {
                // Iterate through each record in the result set
            }
        }
    }
}

```

```

        while (reader.Read())
        {
            // Create a new TeachingStaff object and populate its properties
            var teachingStaff = new TeachingStaff(
                reader["Name"].ToString(),
                reader["Telephone"].ToString(),
                reader["Email"].ToString(),
                Convert.ToDecimal(reader["Salary"]),
                reader["Subject1"].ToString(),
                reader["Subject2"].ToString()
            )
            {
                // Set the ID property to the UserId from the database
                ID = Convert.ToInt32(reader["UserId"])
            };

            // Add the newly created TeachingStaff object to the list
            teachingStaffList.Add(teachingStaff);
        }
    }
}
catch (Exception ex)
{
    // Log any errors that occur during the retrieval process
    Console.WriteLine($"Error retrieving teaching staff: {ex.Message}");
}

// Return the populated list of teaching staff
return teachingStaffList;
} // The connection is automatically closed and disposed of here
}

```

The GetTeachingStaffList method retrieves a list of teaching staff members from a MySQL database. It initializes an empty list to store the resulting TeachingStaff objects.

Using a MySqlConnection, it constructs an SQL query that selects relevant details from both the TeachingStaff and Users tables, joining them based on the UserId. After opening the database connection, it executes the query and processes the results with a MySqlDataReader.

For each record read, it creates a new TeachingStaff object, populating its properties with data from the database, including the staff member's name, contact information, salary, and subjects taught. The UserId is assigned to the ID property of the TeachingStaff object. Each object is then added to the list.

If any exceptions occur during the retrieval process, an error message is logged to the console. Finally, the method returns the populated list of teaching staff members, providing a structured way to access the relevant teaching staff data.

So as we have the List for the TeachingStaff class, next is the way how we can see it as a DataTable, we will implement this method in the BLL:

```

public DataTable ConvertToDataTable(List<TeachingStaff> list)
{
    // Create a new DataTable to hold the teaching staff data
    var dataTable = new DataTable();
}

```

```

// Define columns for the DataTable
dataTable.Columns.Add("ID");
dataTable.Columns.Add("Name");
dataTable.Columns.Add("Telephone");
dataTable.Columns.Add("Email");
dataTable.Columns.Add("Salary");
dataTable.Columns.Add("Subject1");
dataTable.Columns.Add("Subject2");

// Iterate over the list of TeachingStaff objects
foreach (var teachStaff in list)
{
    // Add a new row to the DataTable for each TeachingStaff object
    dataTable.Rows.Add(
        teachStaff.ID,
        teachStaff.Name,
        teachStaff.Telephone,
        teachStaff.Email,
        teachStaff.Salary,
        teachStaff.Subject1,
        teachStaff.Subject2
    );
}

// Return the populated DataTable
return dataTable;
}

// Fetch the data from DAL, then convert it to DataTable
public DataTable GetTeachingStaffDataTable()
{
    // Retrieve the list of teaching staff from the repository
    var teachingstafflist = roleRepository.GetTeachingStaffList();

    // Convert the list of teaching staff to a DataTable and return it
    return ConvertToDataTable(teachingstafflist);
}

```

Next is the way how we use this TeachingStaff DataTable in the UI:

```

public AddAdministration()
{
    // Initialize the form components
    InitializeComponent();

    // Create an instance of the RoleService to manage role-related operations
    roleservice = new RoleService();

    // Subscribe to the Load event of the form
    this.Load += AddAdminDataLoad;
}

private void AddAdminDataLoad(object sender, EventArgs e)
{
    // Call the method to load administration data when the form loads
    LoadAdminData();
}

private void LoadAdminData()
{
    // Retrieve the administration data as a DataTable from the RoleService
    var admindatatable = roleservice.GetAdministrationDataTable();
}

```

```

// Bind the DataTable to the DataGridView for display
datagridadmin.DataSource = admindatatable;

// Populate the work type combo box with values from the EmploymentType enum
txtAdminworktype.DataSource = Enum.GetValues(typeof(Administration.EmploymentType));

// Set the ComboBox style to DropDownList to restrict user input to the list
txtAdminworktype.DropDownStyle = ComboBoxStyle.DropDownList;
}

```

So the first step for adding data for “Teaching Staff” is described successfully, the next step is how we implementing the edit user information which contain the “Edit” and “Delete” function. The first one should be mentioned is the “Edit” function which is named as “UpdateTeachingStaff” :

```

public void UpdateTeachingStaff(TeachingStaff teachingStaff)
{
    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        conn.Open(); // Open the database connection

        // Update the Users table
        try
        {
            // SQL query to update user information
            string userQuery = "UPDATE Users SET Name = @Name, Telephone = @Telephone, Email = @Email WHERE ID = @ID";
            using (var userCmd = new MySqlCommand(userQuery, conn))
            {
                // Add parameters to the SQL command
                userCmd.Parameters.AddWithValue("@ID", teachingStaff.ID);
                userCmd.Parameters.AddWithValue("@Name", teachingStaff.Name);
                userCmd.Parameters.AddWithValue("@Telephone", teachingStaff.Telephone);
                userCmd.Parameters.AddWithValue("@Email", teachingStaff.Email);

                // Execute the command to update the user
                userCmd.ExecuteNonQuery();
            }
        }
        catch (Exception ex)
        {
            // Show an error message if the update fails
            MessageBox.Show($"Error Updating User: {ex.Message}");
            return; // Exit the method to avoid proceeding if there's an error
        }

        // Update the TeachingStaff table
        try
        {
            // SQL query to update teaching staff details
            string staffQuery = "UPDATE TeachingStaff SET Salary = @Salary, Subject1 = @Subject1, Subject2 = @Subject2 WHERE UserId = @UserId";
            using (var staffCmd = new MySqlCommand(staffQuery, conn))
            {
                // Add parameters to the SQL command
                staffCmd.Parameters.AddWithValue("@UserId", teachingStaff.ID);
                staffCmd.Parameters.AddWithValue("@Salary", teachingStaff.Salary);
                staffCmd.Parameters.AddWithValue("@Subject1", teachingStaff.Subject1);
                staffCmd.Parameters.AddWithValue("@Subject2", teachingStaff.Subject2);
            }
        }
    }
}

```



```

        // Execute the command to update the teaching staff record
        staffCmd.ExecuteNonQuery();
    }
}
catch (Exception ex)
{
    // Show an error message if the update fails
    MessageBox.Show($"Error Updating TeachingStaff: {ex.Message}");
}
} // The connection is automatically closed and disposed of here
}

```

The UpdateTeachingStaff method updates the information of a teaching staff member in a database by first updating the Users table with the staff member's name, telephone, and email based on their ID. It establishes a connection to the MySQL database and executes an SQL query using a parameterized command to prevent SQL injection. If the user update is successful, it then updates the TeachingStaff table with details such as salary and subjects taught, again using a parameterized query. Error handling is implemented to display messages if any updates fail, and the connection is automatically closed after the operations are complete.

And then we will call it to the BLL to handle between database and UI:

```

public void UpdateTeachingStaff(TeachingStaff teachingStaff)
{
    // Call the repository method to update the teaching staff information in the database
    roleRepository.UpdateTeachingStaff(teachingStaff);
}

```

The UpdateTeachingStaff method in the Business Logic Layer (BLL) calls the UpdateTeachingStaff method of the roleRepository, passing the TeachingStaff object as a parameter.

Next is how to use the function in the UI :

```

public partial class EditTeachingStaff : UserControl
{
    private readonly RoleService roleservice; // Service for handling teaching staff roles

    public EditTeachingStaff()
    {
        roleservice = new RoleService(); // Initialize the RoleService instance
        InitializeComponent(); // Set up the UI components
        this.Load += EditTeachingDataLoad; // Subscribe to the Load event to load data
    }

    private void EditTeachingDataLoad(object sender, EventArgs e)
    {
        // Load teaching staff data when the user control is loaded
        LoadTeachingStaffData();
    }
}

```

```

private void LoadTeachingStaffData()
{
    // Retrieve teaching staff data and bind it to the DataGridView
    var teachstaffdatatable = roleservice.GetTeachingStaffDataTable();
    editteachgrid.DataSource = teachstaffdatatable; // Set the data source for the grid
}

private void editteachgrid_CellClick(object sender, DataGridViewCellEventArgs e)
{
    // Check if a valid row is clicked
    if (e.RowIndex >= 0)
    {
        DataGridViewRow row = editteachgrid.Rows[e.RowIndex]; // Get the clicked row

        // Populate text boxes with the selected row's data
        txtName.Text = row.Cells["Name"].Value.ToString();
        txtTelephone.Text = row.Cells["Telephone"].Value.ToString();
        txtEmail.Text = row.Cells["Email"].Value.ToString();
        txtSalary.Text = row.Cells["Salary"].Value.ToString();
        txtSubject1.Text = row.Cells["Subject1"].Value.ToString();
        txtSubject2.Text = row.Cells["Subject2"].Value.ToString();
    }
}

private void EditTeachingStaff_Load(object sender, EventArgs e)
{
    // Placeholder for any additional setup when the control loads, currently empty
}

private void Updateclick_Click(object sender, EventArgs e)
{
    try
    {
        // Get the selected row from the DataGridView
        var selectedRow = editteachgrid.SelectedRows[0];
        int idColumnIndex = editteachgrid.Columns["ID"].Index; // Get the index of the ID column

        // Retrieve values from the text boxes
        string name = txtName.Text;
        string telephone = txtTelephone.Text;
        string email = txtEmail.Text;
        decimal salary = Convert.ToDecimal(txtSalary.Text); // Convert salary to decimal
        string subject1 = txtSubject1.Text;
        string subject2 = txtSubject2.Text;

        // Create a new TeachingStaff object with the updated data
        var teachingStaff = new TeachingStaff(name, telephone, email, salary, subject1,
subject2)
        {
            ID = Convert.ToInt32(selectedRow.Cells[idColumnIndex].Value) // Set the ID from the
selected row
        };

        // Update the teaching staff data using the RoleService
        roleservice.UpdateTeachingStaff(teachingStaff);

        // Reload the teaching staff data to refresh the grid
        LoadTeachingStaffData();

        // Show a success message to the user
        MessageBox.Show("Teaching Staff updated successfully!", "Success", MessageBoxButtons.OK,
MessageBoxIcon.Information);
    }
    catch (Exception ex)

```

```

        {
            // Show an error message if an exception occurs during the update
            MessageBox.Show($"An error occurred while updating the Teaching Staff: {ex.Message}",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
}

```

The EditTeachingStaff UI is a user control designed for managing teaching staff records, featuring a DataGridView that displays a list of staff members. Users can select a row to populate text input fields for editing details such as name, telephone, email, salary, and subjects taught. The interface includes an Update button that allows users to save changes to the selected staff member's information, while the data grid automatically refreshes to reflect any updates. This streamlined design facilitates efficient record management for teaching staff.

That's the explanation for the "Edit" method, next is the option for deleting the user, we can implement this function as below:

```

public void DeleteTeachingStaff(int userId)
{
    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        conn.Open(); // Open the database connection

        try
        {
            // SQL query to delete a teaching staff record based on UserId
            string deleteTeachingStaffQuery = "DELETE FROM TeachingStaff WHERE UserId = @UserId";
            using (var teachingStaffCmd = new MySqlCommand(deleteTeachingStaffQuery, conn))
            {
                teachingStaffCmd.Parameters.AddWithValue("@UserId", userId); // Add the UserId
parameter
                teachingStaffCmd.ExecuteNonQuery(); // Execute the deletion command
            }

            // SQL query to delete the corresponding user record from the Users table
            string deleteUserQuery = "DELETE FROM Users WHERE Id = @Id";
            using (var userCmd = new MySqlCommand(deleteUserQuery, conn))
            {
                userCmd.Parameters.AddWithValue("@Id", userId); // Add the Id parameter
                userCmd.ExecuteNonQuery(); // Execute the deletion command
            }
        }
        catch (Exception ex)
        {
            // Display an error message if an exception occurs during deletion
            MessageBox.Show($"Error Deleting User or TeachingStaff: {ex.Message}");
        }
    }
}

```

This method, DeleteTeachingStaff, deletes a teaching staff member from the database by first removing their record from the TeachingStaff table based on the provided userId, and then deleting the corresponding entry from the Users table. It uses parameterized SQL

commands within a using statement to ensure proper resource management and to prevent SQL injection. If any errors occur during the deletion process, an error message is displayed to the user.

In the BLL:

```
public void DeleteTeachingStaff(int userId)
{
    // Calls the repository method to delete the teaching staff record based on userId
    roleRepository.DeleteTeachingStaff(userId);
}
```

This method in the Business Logic Layer (BLL) serves as a wrapper that delegates the task of deleting a teaching staff member to the roleRepository. It takes a userId as a parameter and calls the DeleteTeachingStaff method from the repository

The final is the UI:

```
private void Updateclick_Click(object sender, EventArgs e)
{
    try
    {
        // Get the selected row from the DataGridView
        var selectedRow = editteachgrid.SelectedRows[0];
        int idColumnIndex = editteachgrid.Columns["ID"].Index; // Get the index of the ID column

        // Retrieve values from the text boxes
        string name = txtName.Text;
        string telephone = txtTelephone.Text;
        string email = txtEmail.Text;
        decimal salary = Convert.ToDecimal(txtSalary.Text); // Convert salary to decimal
        string subject1 = txtSubject1.Text;
        string subject2 = txtSubject2.Text;

        // Create a new TeachingStaff object with the updated data
        var teachingStaff = new TeachingStaff(name, telephone, email, salary, subject1, subject2)
        {
            ID = Convert.ToInt32(selectedRow.Cells[idColumnIndex].Value) // Set the ID from the
selected row
        };

        // Update the teaching staff data using the RoleService
        roleservice.UpdateTeachingStaff(teachingStaff);

        // Reload the teaching staff data to refresh the grid
        LoadTeachingStaffData();

        // Show a success message to the user
        MessageBox.Show("Teaching Staff updated successfully!", "Success", MessageBoxButtons.OK,
MessageBoxIcon.Information);
    }
    catch (Exception ex)
    {
        // Show an error message if an exception occurs during the update
        MessageBox.Show($"An error occurred while updating the Teaching Staff: {ex.Message}",
"Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

The Updateclick_Click method handles the event when the update button is clicked. It retrieves the selected teaching staff record from the DataGridView, gathers updated information from various text boxes, and creates a new TeachingStaff object with these values. The method then calls the UpdateTeachingStaff method of the RoleService to save the changes to the database. After updating, it refreshes the DataGridView to reflect the changes and displays a success message. If an error occurs during the process, an error message is shown to the user.

So this is all the basic function for the adding user with the roletype as “TeachingStaff”, next I will move to the 2 other class “Administration” class and “Student” class with the same function for adding new data:

First is the “Administration” class:

```
using ManageSystem.DAL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ManageSystem.Models
{
    public class Administration : User // Administration class inherits from User
    {
        public decimal Salary { get; private set; } // Salary property, only settable within the
class
        public int Workhour { get; private set; } // Workhour property, only settable within the
class
        public EmploymentType Employment { get; private set; } // Employment type property

        // Enum to define employment types
        public enum EmploymentType
        {
            FullTime, // Full-time employment
            PartTime // Part-time employment
        }

        // Constructor to initialize Administration object with relevant details
        public Administration(string name, string telephone, string email, decimal salary, int
workhour, EmploymentType employmentType)
            : base(name, telephone, email, RoleType.Administration) // Call to base class
constructor
        {
            Salary = salary; // Set the salary
            Workhour = workhour; // Set the work hours
            Employment = employmentType; // Set the employment type
        }

        // Method to add a new administration user
        public override void AddUser()
        {
            var addadministration = new LoginRepository(); // Create instance of LoginRepository
            addadministration.AddAdministration(this); // Call method to add administration data
        }

        // Method to edit an existing administration user
    }
}
```

```

        public override void EditUser()
        {
            var editadmin = new LoginRepository(); // Create instance of LoginRepository
            editadmin.UpdateAdministration(this); // Call method to update administration data
        }

        // Method to delete an administration user
        public override void DeleteUser()
        {
            var deleteadmin = new LoginRepository(); // Create instance of LoginRepository
            deleteadmin.DeleteAdministration(this.ID); // Call method to delete administration data
        }
    }
}

```

The Administration class represents an administrative user within the ManageSystem application, inheriting from the base User class. It encapsulates properties such as Salary, Workhour, and an EmploymentType enum (which includes FullTime and PartTime) to define the user's role and attributes. The class provides methods for common user management operations—adding, editing, and deleting administration records—by utilizing a LoginRepository for database interactions. This design promotes a clear separation of concerns and enhances maintainability by encapsulating user-specific logic within the Administration class.

So as the “Teaching Staff” class, the first thing I want to explain is the “Add New Data”:

```

public void AddAdministration(Administration administration)
{
    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        conn.Open(); // Open the database connection

        // Insert into Users table first to get UserId
        try
        {
            // SQL query to insert a new user into the Users table
            string query = @"
INSERT INTO Users (Name, Telephone, Email, RoleType)
VALUES (@Name, @Telephone, @Email, 'Administration');";

            int userId; // Variable to hold the newly generated UserId

            // Execute the query to insert the user
            using (var cmd = new MySqlCommand(query, conn))
            {
                // Add parameters to the command
                cmd.Parameters.AddWithValue("@Name", administration.Name);
                cmd.Parameters.AddWithValue("@Telephone", administration.Telephone);
                cmd.Parameters.AddWithValue("@Email", administration.Email);

                cmd.ExecuteNonQuery(); // Execute the insertion command

                // Get the last inserted UserId
                string getQuery = "SELECT LAST_INSERT_ID();"; // SQL to retrieve the last inserted
                ID
                using (var idCmd = new MySqlCommand(getQuery, conn))
                {

```

```

        userId = Convert.ToInt32(idCmd.ExecuteScalar()); // Execute and convert result
        to an integer
    }
}

// Insert into Administration table
string adminQuery = @"
INSERT INTO Administration (UserId, Salary, EmploymentType, WorkingHours)
VALUES (@UserId, @Salary, @EmploymentType, @WorkingHours);";

// Execute the query to insert the administration details
using (var adminCmd = new MySqlCommand(adminQuery, conn))
{
    // Add parameters for the administration details
    adminCmd.Parameters.AddWithValue("@UserId", userId);
    adminCmd.Parameters.AddWithValue("@Salary", administration.Salary);
    adminCmd.Parameters.AddWithValue("@EmploymentType",
administration.Employment.ToString());
    adminCmd.Parameters.AddWithValue("@WorkingHours", administration.Workhour);

    adminCmd.ExecuteNonQuery(); // Execute the insertion command for administration
}
}
catch (Exception ex)
{
    // Display an error message if an exception occurs
    MessageBox.Show($"Error adding Administration: {ex.Message}");
}
}
}

```

The AddAdministration method is responsible for adding a new administrative user to the database. It first inserts the user's basic information into the Users table, retrieving the newly generated UserId. Next, it uses this UserId to insert the corresponding administrative details, such as Salary, EmploymentType, and WorkingHours, into the Administration table. The method includes error handling to display a message if any issues arise during the insertion process, ensuring that the user is informed of any failures.

In the BLL:

```

public void AddAdministration(IUser administration)
{
    // Calls the AddUser method on the provided IUser instance to add the administration user
    administration.AddUser();
}

```

In the UI for this method:

```

private void Addclick_Click(object sender, EventArgs e)
{
    // Retrieve input values from text boxes
    string name = txtAdminName.Text;
    string telephone = txtAdmintelephone.Text;
    string email = txtAdminEmail.Text;
    string workhours = txtworkinghours.Text;

    // Check for duplicate email addresses
    if(roleservice.CheckDuplicate(txtAdminEmail.Text))
    {

```

```

        MessageBox.Show("An user with this email already exists.", "Input Error",
        MessageBoxButtons.OK, MessageBoxIcon.Warning);
        return; // Exit if duplicate found
    }

    // Validate salary input
    if (!decimal.TryParse(txtAdminalary.Text, out decimal salary))
    {
        MessageBox.Show("Please enter a valid salary amount.", "Input Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
        return; // Exit if salary is invalid
    }

    // Validate working hours input
    if (!int.TryParse(txtworkinghours.Text, out int workHours))
    {
        MessageBox.Show("Please enter a valid number of working hours.", "Input Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return; // Exit if working hours are invalid
    }

    // Validate telephone number length
    if (telephone.Length > 11)
    {
        MessageBox.Show("Telephone number cannot exceed 11 characters.", "Input Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return; // Exit if length exceeds limit
    }

    // Validate telephone number format
    if (!telephone.All(char.IsDigit))
    {
        MessageBox.Show("Telephone number must be a number.", "Input Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
        return; // Exit if format is invalid
    }

    // Parse the selected employment type
    Administration.EmploymentType employmentType;
    if (Enum.TryParse(txtAdminworktype.SelectedItem?.ToString(), out employmentType))
    {
        // Create a new Administration object with validated data
        var administration = new Administration(name, telephone, email, salary, workHours,
        employmentType);
        try
        {
            // Attempt to add the new administration record
            roleservice.AddAdministration(administration);
            MessageBox.Show("Administration added successfully", "Success", MessageBoxButtons.OK,
            MessageBoxIcon.Information);
            LoadAdminData(); // Refresh the data display
        }
        catch (Exception ex)
        {
            // Handle any errors during the addition process
            MessageBox.Show($"Error adding Administration: {ex.Message}", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
    else
    {
        // Handle invalid employment type selection
        MessageBox.Show("Please select a valid employment type.", "Input Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}

```



```
}
```

The `Addclick_Click` method handles the click event for adding a new administrative user. It retrieves user input from text fields and performs several validations, including checking for duplicate emails, ensuring the salary and working hours are valid numbers, and verifying the telephone number's format and length. If all validations pass, it creates a new `Administration` object with the provided data and attempts to add it via the `roleservice`. The method also includes error handling to display appropriate messages for any issues encountered during the process, ensuring a smooth user experience. If successful, it refreshes the displayed data to include the new entry.

So if the data added successfully, we will convert a list which contain data for “Administration” to a `DataTable`. Here is how to implement this method:

```
public List<Administration> GetAdministrations()
{
    // Initialize a list to hold Administration objects
    var administrationList = new List<Administration>();

    // Establish a connection to the database
    using (var conn = new MySqlConnection(_connectionString))
    {
        // SQL query to retrieve administration details along with user information
        string query = @"
        SELECT u.Id AS UserId, u.Name, u.Telephone, u.Email, a.Salary, a.EmploymentType,
a.WorkingHours
        FROM Administration a
        JOIN Users u ON a.UserId = u.Id";

        var cmd = new MySqlCommand(query, conn);
        conn.Open(); // Open the database connection

        try
        {
            // Execute the query and read the results
            using (var reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    // Parse the EmploymentType enum from the database
                    if (Enum.TryParse(reader["EmploymentType"].ToString(), out
Administration.EmploymentType employmentType))
                    {
                        // Create a new Administration object with the retrieved data
                        var administration = new Administration(
                            reader["Name"].ToString(),
                            reader["Telephone"].ToString(),
                            reader["Email"].ToString(),
                            Convert.ToDecimal(reader["Salary"]),
                            Convert.ToInt32(reader["WorkingHours"]),
                            employmentType
                        );
                        ID = Convert.ToInt32(reader["UserId"]) // Assign the UserId to the
Administration object
                    };

                    // Add the new administration object to the list
                    administrationList.Add(administration);
                }
            }
        }
    }
}
```

```

    }
}
}
catch (Exception ex)
{
    // Display an error message if an exception occurs during data retrieval
    MessageBox.Show($"Error retrieving administration: {ex.Message}");
}

// Return the list of administrations
return administrationList;
}
}

```

The first step to hold the data for “Administration” is in the DAL, if we want to convert it to a DataTable, we have to convert it in the BLL. Here is the code for it:

```

public DataTable ConvertAdminDataTable(List<Administration> adminlist)
{
    // Create a new DataTable to hold administration data
    var admindatatable = new DataTable();

    // Define the columns for the DataTable
    admindatatable.Columns.Add("ID");
    admindatatable.Columns.Add("Name");
    admindatatable.Columns.Add("Telephone");
    admindatatable.Columns.Add("Email");
    admindatatable.Columns.Add("Salary");
    admindatatable.Columns.Add("EmploymentType");
    admindatatable.Columns.Add("WorkingHours");

    // Populate the DataTable with data from the admin list
    foreach (var admin in adminlist)
    {
        admindatatable.Rows.Add(
            admin.ID,
            admin.Name,
            admin.Telephone,
            admin.Email,
            admin.Salary,
            admin.Employment.ToString(),
            admin.Workhour
        );
    }

    // Return the populated DataTable
    return admindatatable;
}

public DataTable GetAdministrationDataTable()
{
    // Retrieve the list of administrations from the repository
    var administrationlist = roleRepository.GetAdministrations();

    // Convert the list to a DataTable and return it
    return ConvertAdminDataTable(administrationlist);
}

```

The ConvertAdminDataTable method takes a list of Administration objects and converts it into a DataTable. It defines columns for the DataTable corresponding to the properties

of Administration, then iterates through the provided list, adding each administration's details as a new row.

The GetAdministrationDataTable method retrieves the list of administrations from the roleRepository and calls ConvertAdminDataTable to convert this list into a DataTable, which it then returns. This approach allows for easy manipulation and display of administrative data in tabular format.

And then, here is how the viewing new data for the “Administration” class work in UI:

```
public AddAdministration()
{
    // Initialize the form components
    InitializeComponent();

    // Create an instance of the RoleService to manage role-related operations
    roleservice = new RoleService();

    // Subscribe to the Load event of the form
    this.Load += AddAdminDataLoad;
}

private void AddAdminDataLoad(object sender, EventArgs e)
{
    // Call the method to load administration data when the form loads
    LoadAdminData();
}

private void LoadAdminData()
{
    // Retrieve the administration data as a DataTable from the RoleService
    var admindatatable = roleservice.GetAdministrationDataTable();

    // Bind the DataTable to the DataGridView for display
    datagridadmin.DataSource = admindatatable;

    // Populate the work type combo box with values from the EmploymentType enum
    txtAdminworktype.DataSource = Enum.GetValues(typeof(Administration.EmploymentType));

    // Set the ComboBox style to DropDownList to restrict user input to the list
    txtAdminworktype.DropDownStyle = ComboBoxStyle.DropDownList;
}
```

The provided code is part of a form class that facilitates the addition of administrative users. In the constructor, it initializes the form components, creates an instance of RoleService for role management, and subscribes to the form's Load event to trigger data loading. The AddAdminDataLoad method calls LoadAdminData() upon form loading, which retrieves administration data as a DataTable and binds it to a DataGridView for display. Additionally, it populates a ComboBox with values from the EmploymentType enum and sets its style to DropDownList to restrict user input, ensuring a smooth and user-friendly experience for adding new administrators.

After adding and view data for “Administration” successfully, the next thing to do is the implementation for the “Edit” and “Delete” method:

The first function is the Edit function which named as “Update”:

```

public void UpdateAdministration(Administration admin)
{
    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        conn.Open(); // Open the database connection

        // Update Users table
        try
        {
            // SQL query to update user details in the Users table
            string userQuery = "UPDATE Users SET Name = @Name, Telephone = @Telephone, Email = @Email WHERE ID = @ID";
            using (var userCmd = new MySqlCommand(userQuery, conn))
            {
                // Assign parameter values for the user update query
                userCmd.Parameters.AddWithValue("@ID", admin.ID);
                userCmd.Parameters.AddWithValue("@Name", admin.Name);
                userCmd.Parameters.AddWithValue("@Telephone", admin.Telephone);
                userCmd.Parameters.AddWithValue("@Email", admin.Email);

                // Execute the command to update the user information
                userCmd.ExecuteNonQuery();
            }
        }
        catch (Exception ex)
        {
            // Display an error message if an exception occurs during user update
            MessageBox.Show($"Error Updating User: {ex.Message}");
            return; // Exit the method if user update fails
        }

        // Update Administration table
        try
        {
            // SQL query to update administration details in the Administration table
            string adminQuery = "UPDATE Administration SET Salary = @Salary, WorkingHours = @WorkingHours, EmploymentType = @EmploymentType WHERE UserId = @UserId";
            using (var adminCmd = new MySqlCommand(adminQuery, conn))
            {
                // Assign parameter values for the administration update query
                adminCmd.Parameters.AddWithValue("@UserId", admin.ID);
                adminCmd.Parameters.AddWithValue("@Salary", admin.Salary);
                adminCmd.Parameters.AddWithValue("@WorkingHours", admin.Workhour);
                adminCmd.Parameters.AddWithValue("@EmploymentType", admin.Employment.ToString());

                // Execute the command to update the administration information
                adminCmd.ExecuteNonQuery();
            }
        }
        catch (Exception ex)
        {
            // Display an error message if an exception occurs during administration update
            MessageBox.Show($"Error Updating Administration: {ex.Message}");
        }
    }
}

```

The UpdateAdministration method updates the details of an existing Administration object in the database. It begins by establishing a connection to the MySQL database using the connection string. The method first attempts to update the user information in

the Users table using a SQL UPDATE statement, assigning parameter values for Name, Telephone, and Email. If an error occurs during this process, it displays an error message and exits the method. If the user update is successful, the method proceeds to update the Administration table, modifying the Salary, WorkingHours, and EmploymentType based on the provided Administration object. This operation also uses a parameterized query to avoid SQL injection. Any exceptions during the administration update are caught and displayed as error messages, ensuring robust error handling throughout the process.

And how it work in the BLL:

```
public void UpdateAdministration(Administration administration)
{
    // Call the repository method to update the administration information in the database
    roleRepository.UpdateAdministration(administration);
}
```

Next it's the way how the "Update" function work in the UI:

```
private void UpdateClick_Click(object sender, EventArgs e)
{
    try
    {
        // Validate and parse the EmploymentType from ComboBox
        if (txtAdminworktype.SelectedItem != null &&
            Enum.TryParse(txtAdminworktype.SelectedItem.ToString(), out
Administration.EmploymentType employmentType))
        {
            // Get the selected row and other details
            var selectedRow = datagridadmin.SelectedRows[0];
            int idColumnIndex = datagridadmin.Columns["ID"].Index;
            string name = txtAdminName.Text;
            string telephone = txtAdmintelephone.Text;
            string email = txtAdminEmail.Text;
            decimal salary = Convert.ToDecimal(txtAdminsalary.Text);
            int workhour = Convert.ToInt32(txtworkinghours.Text);

            // Create Administration object and set its ID
            var administration = new Administration(name, telephone, email, salary, workhour,
employmentType)
            {
                ID = Convert.ToInt32(selectedRow.Cells[idColumnIndex].Value)
            };

            // Call the RoleService to update the Administration data
            roleservice.UpdateAdministration(administration);
            LoadAdminData();
            MessageBox.Show("Administration updated successfully!", "Success", MessageBoxButtons.OK,
MessageBoxIcon.Information);
        }
        else
        {
            MessageBox.Show("Please select a valid employment type.", "Invalid Selection",
MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }
    }
    catch (Exception ex)
```

```

    {
        MessageBox.Show($"An error occurred while updating the Administration: {ex.Message}",
            "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}

```

So after the “Edit” function which named as “Update”, the next function is the “Delete” function:

```

public void DeleteAdministration(int userId)
{
    // Establish a connection to the database using the connection string
    using (var conn = new MySqlConnection(_connectionString))
    {
        conn.Open(); // Open the database connection

        try
        {
            // SQL query to delete the administration record for the specified user
            string deleteTeachingStaffQuery = "DELETE FROM Administration WHERE UserId = @UserId";
            using (var teachingStaffCmd = new MySqlCommand(deleteTeachingStaffQuery, conn))
            {
                // Assign the user ID parameter for the deletion query
                teachingStaffCmd.Parameters.AddWithValue("@UserId", userId);
                teachingStaffCmd.ExecuteNonQuery(); // Execute the command to delete the
administration record
            }

            // SQL query to delete the user record from the Users table
            string deleteUserQuery = "DELETE FROM Users WHERE Id = @Id";
            using (var userCmd = new MySqlCommand(deleteUserQuery, conn))
            {
                // Assign the user ID parameter for the deletion query
                userCmd.Parameters.AddWithValue("@Id", userId);
                userCmd.ExecuteNonQuery(); // Execute the command to delete the user record
            }
        }
        catch (Exception ex)
        {
            // Display an error message if an exception occurs during deletion
            MessageBox.Show($"Error Deleting User or Administration: {ex.Message}");
        }
    }
}

```

The DeleteAdministration method is responsible for deleting an administrative user from the database based on their userId. It starts by establishing a connection to the MySQL database. Within a try block, it executes two SQL DELETE statements: the first removes the corresponding record from the Administration table using the provided userId, and the second deletes the user from the Users table. Both commands utilize parameterized queries to prevent SQL injection. If an error occurs during either deletion, it catches the exception and displays an error message to the user. This method ensures that both the administration and user records are removed safely and efficiently.

This is how it works in the BLL:

```

public void DeleteAdministration(int userId)
{
    // Calls the repository method to delete the administration record based on userId
    roleRepository.DeleteAdministration(userId);
}

```

The final is that how the “Delete” function work in the UI:

```

private void Deleteadmin_Click(object sender, EventArgs e)
{
    try
    {
        // Check if any row is selected in the DataGridView
        if (datagridadmin.SelectedRows.Count > 0)
        {
            // Get the first selected row
            var selectedRow = datagridadmin.SelectedRows[0];
            // Retrieve the user ID from the selected row
            int userId = Convert.ToInt32(selectedRow.Cells["ID"].Value);

            // Call the DeleteAdministration method to delete the user
            roleservice.DeleteAdministration(userId);

            // Reload the DataGridView to reflect changes
            LoadAdminData();

            // Show a success message to the user
            MessageBox.Show("Administration deleted successfully!", "Success", MessageBoxButtons.OK,
            MessageBoxIcon.Information);
        }
        else
        {
            // Prompt the user to select a record if none is selected
            MessageBox.Show("Please select a record to delete.", "Error", MessageBoxButtons.OK,
            MessageBoxIcon.Warning);
        }
    }
    catch (Exception ex)
    {
        // Display an error message if an exception occurs during deletion
        MessageBox.Show($"An error occurred while deleting the Administration: {ex.Message}",
        "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}

```

After the “Administration” class, the next and the final class is the “Student” class. Here is the detail for it:

```

using ManageSystem.DAL;
using System;

namespace ManageSystem.Models
{
    public class Student : User
    {
        // Properties to store the subjects associated with the student.
        public string Subject1 { get; private set; }
        public string Subject2 { get; private set; }
        public string Presubject1 { get; private set; }
        public string Presubject2 { get; private set; }
    }
}

```

```

        // Constructor to initialize the 'Student' object with specific details.
        public Student(string name, string telephone, string email, string subject1, string
subject2, string presubject1, string presubject2)
            : base(name, telephone, email, RoleType.Student) // Call to the base class constructor.
        {
            Subject1 = subject1; // Assign first subject to the property.
            Subject2 = subject2; // Assign second subject to the property.
            Presubject1 = presubject1; // Assign pre-subject one to the property.
            Presubject2 = presubject2; // Assign pre-subject two to the property.
        }

        // Override of the 'AddUser' method from the base 'User' class.
        public override void AddUser()
        {
            var addstudent = new LoginRepository(); // Create repository instance.
            addstudent.AddStudent(this); // Calls the repository to add this student.
        }

        // Override of the 'EditUser' method from the base 'User' class.
        public override void EditUser()
        {
            var editstudent = new LoginRepository(); // Create repository instance.
            editstudent.UpdateStudent(this); // Calls the repository to update this student.
        }

        // Override of the 'DeleteUser' method from the base 'User' class.
        public override void DeleteUser()
        {
            var deletestudent = new LoginRepository(); // Create repository instance.
            deletestudent.DeleteStudent(this.ID); // Calls the repository to delete this student by
ID.
        }
    }
}

```

The Student class inherits from the abstract User class, representing a student in the management system. It includes properties for two subjects (Subject1 and Subject2) and two pre-subjects (Presubject1 and Presubject2). The constructor initializes these properties while calling the base constructor to set common user attributes such as name, telephone, and email, with the role specified as Student.

The class overrides three methods from the User base class: AddUser, EditUser, and DeleteUser. The AddUser method creates an instance of LoginRepository and calls its method to add the student to the database. Similarly, EditUser updates the student's details, and DeleteUser removes the student from the database using their ID. This structure encapsulates the functionality related to student management, ensuring that all database interactions are handled through the repository pattern for consistency and maintainability.

The same with 2 class above, next we have the method to add information for the user with role as “Student”. Here is how it works:

First, in the DAL:

```

public void AddStudent(Student student)
{

```



```

using (var conn = new MySqlConnection(_connectionString)) // Establish a connection to the
database
{
    conn.Open(); // Open the connection

    int userId;

    // Insert the user into the Users table
    try
    {
        string userQuery = @"
INSERT INTO Users (Name, Telephone, Email, RoleType)
VALUES (@Name, @Telephone, @Email, 'Students');";

        using (var userCmd = new MySqlCommand(userQuery, conn)) // Create command for user
insertion
        {
            userCmd.Parameters.AddWithValue("@Name", student.Name); // Set parameters
            userCmd.Parameters.AddWithValue("@Telephone", student.Telephone);
            userCmd.Parameters.AddWithValue("@Email", student.Email);

            userCmd.ExecuteNonQuery(); // Execute the user insertion

            // Get the last inserted UserId
            using (var idCmd = new MySqlCommand("SELECT LAST_INSERT_ID();", conn)) // Retrieve
the last inserted ID
            {
                userId = Convert.ToInt32(idCmd.ExecuteScalar()); // Store the user ID
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show($"Error adding user: {ex.Message}"); // Show error message
        return; // Exit if user insertion fails
    }

    // Insert into the Students table
    try
    {
        string studentQuery = @"
INSERT INTO Students (UserId, CurrentSubject1, CurrentSubject2, PreviousSubject1, PreviousSubject2)
VALUES (@UserId, @CurrentSubject1, @CurrentSubject2, @PreviousSubject1, @PreviousSubject2);";

        using (var studentCmd = new MySqlCommand(studentQuery, conn)) // Create command for
student insertion
        {
            studentCmd.Parameters.AddWithValue("@UserId", userId); // Set parameters
            studentCmd.Parameters.AddWithValue("@CurrentSubject1", student.Subject1);
            studentCmd.Parameters.AddWithValue("@CurrentSubject2", student.Subject2);
            studentCmd.Parameters.AddWithValue("@PreviousSubject1", student.Presubject1);
            studentCmd.Parameters.AddWithValue("@PreviousSubject2", student.Presubject2);

            studentCmd.ExecuteNonQuery(); // Execute the student insertion
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show($"Error adding student: {ex.Message}"); // Show error message
    }
}

```

The AddStudent method facilitates the addition of a new student to the database by first establishing a connection and executing an SQL query to insert user details into the Users table, setting the role as 'Students'. It retrieves the last inserted user ID to link the student record appropriately. Subsequently, it inserts the student's subject information into the Students table, using parameterized queries to ensure security against SQL injection. The method incorporates error handling with try-catch blocks, displaying error messages via MessageBox.Show if any operation fails, thereby ensuring consistent and reliable data management.

And then, in the BLL operation:

```
public void AddStudent(IUser student)
{
    student.AddUser(); // Calls the AddUser method on the IUser instance to add the student
}
```

After that, here is how the UI interact with this:

```
private void Addclick_Click(object sender, EventArgs e)
{
    // Retrieve input values from text fields
    string name = txtStuname.Text;
    string email = txtStuemail.Text;
    string telephone = txtStutelephone.Text;
    string currentsubject1 = txtcurrentsubject1.Text;
    string currentsubject2 = txtcurrentsubject2.Text;
    string previoussubject1 = txtprevioussubject1.Text;
    string previoussubject2 = txtprevioussubject2.Text;

    // Check for duplicate email
    if (roleservice.CheckDuplicate(email))
    {
        MessageBox.Show("An user with this email already exists.", "Input Error",
        MessageBoxButtons.OK, MessageBoxIcon.Warning);
        return;
    }

    // Validate telephone number length
    if (telephone.Length > 11)
    {
        MessageBox.Show("Telephone number cannot exceed 11 characters.", "Input Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    // Ensure telephone number contains only digits
    if (!telephone.All(char.IsDigit))
    {
        MessageBox.Show("Telephone number must be a number.", "Input Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
        return;
    }

    // Check for duplicate subjects
    if (SubjectDuplicate(currentsubject1, currentsubject2, previoussubject1, previoussubject2))
    {

```

```

        MessageBox.Show("Subjects can't be the same", "Input Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
        return;
    }

    // Create a new Student object
    var student = new Student(name, telephone, email, currentsubject1, currentsubject2,
    previoussubject1, previoussubject2);

    try
    {
        // Add the student to the system
        roleservice.AddStudent(student);
        MessageBox.Show("Student added successfully", "Success", MessageBoxButtons.OK,
        MessageBoxIcon.Information);
        LoadStudent(); // Refresh the student list
    }
    catch (Exception ex)
    {
        MessageBox.Show($"Error adding student: {ex.Message}", "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }
}

```

The Addclick_Click method handles the event when the "Add" button is clicked. It retrieves user input from various text fields, including the student's name, email, telephone, and subjects. The method performs several validations: it checks for duplicate email entries, ensures the telephone number does not exceed 11 characters and contains only digits, and verifies that the subjects are not duplicated. If any validation fails, it displays an appropriate error message to the user. If all checks pass, a new Student object is created, and an attempt is made to add it to the system via the roleservice. Upon successful addition, a success message is displayed, and the student list is refreshed; if an error occurs, an error message is shown. This method effectively ensures data integrity before adding a new student.

The next thing is the way how we can see the data for the Student after adding it. As the method is the same with two class above, I will use comment in my code to explain how my code work within 3 layer for other functions

```

public List<Student> GetStudents()
{
    var studentList = new List<Student>(); // Initialize a list to hold students
    using (var conn = new MySqlConnection(_connectionString)) // Establish a connection to
the database
    {
        string query = @"
SELECT u.Id AS UserId, u.Name, u.Telephone, u.Email,
      s.CurrentSubject1, s.CurrentSubject2,
      s.PreviousSubject1, s.PreviousSubject2
FROM Students s
JOIN Users u ON s.UserId = u.Id"; // SQL query to join Students and Users tables

        var cmd = new MySqlCommand(query, conn);
        conn.Open(); // Open the connection

        try
        {

```

```

        using (var reader = cmd.ExecuteReader()) // Execute the query and read results
        {
            while (reader.Read()) // Loop through each record
            {
                var student = new Student(
                    reader["Name"].ToString(),
                    reader["Telephone"].ToString(),
                    reader["Email"].ToString(),
                    reader["CurrentSubject1"].ToString(),
                    reader["CurrentSubject2"].ToString(),
                    reader["PreviousSubject1"].ToString(),
                    reader["PreviousSubject2"].ToString()
                )
                {
                    ID = Convert.ToInt32(reader["UserId"]) // Set the student ID
                };

                studentList.Add(student); // Add the student to the list
            }
        }
    }
}
catch (Exception ex)
{
    MessageBox.Show($"Error retrieving student: {ex.Message}"); // Handle any errors
}

return studentList; // Return the list of students
}
}

```

In the BLL:

```

public DataTable ConvertStudentDataTable(List<Student> studentlist)
{
    var studenttable = new DataTable(); // Initialize a new DataTable
    // Define columns for the DataTable

    studenttable.Columns.Add("ID");
    studenttable.Columns.Add("Name");
    studenttable.Columns.Add("Telephone");
    studenttable.Columns.Add("Email");
    studenttable.Columns.Add("CurrentSubject1");
    studenttable.Columns.Add("CurrentSubject2");
    studenttable.Columns.Add("PreviousSubject1");
    studenttable.Columns.Add("PreviousSubject2");

    // Populate the DataTable with student data
    foreach (var student in studentlist)
    {
        studenttable.Rows.Add(student.ID, student.Name, student.Telephone, student.Email,
            student.Subject1, student.Subject2, student.Presubject1, student.Presubject2);
    }

    return studenttable; // Return the populated DataTable
}

public DataTable GetStudentDataTable()
{
    var studentlist = roleRepository.GetStudents(); // Retrieve the list of students
    return ConvertStudentDataTable(studentlist); // Convert the list to a DataTable
}

public void UpdateTeachingStaff(TeachingStaff teachingStaff)
{
}

```

```

        // Call the repository method to update the teaching staff information in the database
        roleRepository.UpdateTeachingStaff(teachingStaff);
    }
    public void UpdateAdministration(Administration administration)
    {

        // Call the repository method to update the administration information in the database
        roleRepository.UpdateAdministration(administration);
    }
}

```

After adding and viewing the data, the next thing is the “Edit” and “Delete” function

First is the “Edit” function which named as “Update”:

```

public void UpdateStudent(Student student)
{
    using (var conn = new MySqlConnection(_connectionString)) // Establish a connection to the
    database
    {
        conn.Open(); // Open the connection
        try
        {
            // Update query for the Users table
            string userquery = "UPDATE Users SET Name = @Name, Telephone = @Telephone, Email =
            @Email WHERE ID = @ID";
            using (var userCmd = new MySqlCommand(userquery, conn))
            {
                // Adding parameters to prevent SQL injection
                userCmd.Parameters.AddWithValue("@ID", student.ID);
                userCmd.Parameters.AddWithValue("@Name", student.Name);
                userCmd.Parameters.AddWithValue("@Telephone", student.Telephone);
                userCmd.Parameters.AddWithValue("@Email", student.Email);

                userCmd.ExecuteNonQuery(); // Execute the command
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show($"Error Updating User: {ex.Message}"); // Handle user update errors
            return; // Exit if user update fails
        }

        try
        {
            // Update query for the Students table
            string stuquery = "UPDATE Students SET CurrentSubject1 = @CurrentSubject1,
            CurrentSubject2 = @CurrentSubject2, PreviousSubject1 = @PreviousSubject1, PreviousSubject2 =
            @PreviousSubject2 WHERE UserId = @UserId";
            using (var stucmd = new MySqlCommand(stuquery, conn))
            {
                // Adding parameters for the student update
                stucmd.Parameters.AddWithValue("@UserId", student.ID);
                stucmd.Parameters.AddWithValue("@CurrentSubject1", student.Subject1);
                stucmd.Parameters.AddWithValue("@CurrentSubject2", student.Subject2);
                stucmd.Parameters.AddWithValue("@PreviousSubject1", student.Presubject1);
                stucmd.Parameters.AddWithValue("@PreviousSubject2", student.Presubject2);

                stucmd.ExecuteNonQuery(); // Execute the command
            }
        }
        catch (Exception ex)
        {

```

```

        MessageBox.Show($"Error Updating Student: {ex.Message}"); // Handle student update
errors
    }
}

```

In the BLL:

```

public void UpdateStudent(Student student)
{
    // Call the repository method to update the student information in the database
    roleRepository.UpdateStudent(student);
}

```

In the UI:

```

private void Update_Click(object sender, EventArgs e)
{
    try
    {
        // Get the selected row from the data grid
        var selectedrow = studatagrid.SelectedRows[0];
        int idcolumnindex = studatagrid.Columns["ID"].Index; // Get the index of the ID column

        // Retrieve input values from text fields
        string name = txtStuname.Text;
        string telephone = txtStutelephone.Text;
        string email = txtStuemail.Text;
        string currentsubject1 = txtcurrentsubject1.Text;
        string currentsubject2 = txtcurrentsubject2.Text;
        string previoussubject1 = txtprevioussubject1.Text;
        string previoussubject2 = txtprevioussubject2.Text;

        // Create a new Student object with the updated values
        var Student = new Student(name, telephone, email, currentsubject1, currentsubject2,
previoussubject1, previoussubject2)
        {
            ID = Convert.ToInt32(selectedrow.Cells[idcolumnindex].Value) // Set the ID from the
selected row
        };

        // Update the student information in the database
        roleservice.UpdateStudent(Student);
        LoadStudent(); // Refresh the student list
        MessageBox.Show("Student updated successfully!", "Success", MessageBoxButtons.OK,
MessageBoxIcon.Information); // Success message
    }
    catch (Exception ex)
    {
        // Display an error message if an exception occurs
        MessageBox.Show($"An error occurred while updating the Student: {ex.Message}", "Error",
MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}

```

So this is all for the “Edit” function, next is the “Delete” function:

And the first thing is how we implement this in the DAL:

```
public void DeleteStudent(int userId)
{
    using (var conn = new MySqlConnection(_connectionString)) // Establish a database connection
    {
        conn.Open(); // Open the connection

        try
        {
            // Query to delete the student record
            string deleteTeachingStaffQuery = "DELETE FROM Students WHERE UserId = @UserId";
            using (var teachingStaffCmd = new MySqlCommand(deleteTeachingStaffQuery, conn))
            {
                teachingStaffCmd.Parameters.AddWithValue("@UserId", userId); // Add parameter to
                // prevent SQL injection
                teachingStaffCmd.ExecuteNonQuery(); // Execute the delete command
            }

            // Query to delete the user record
            string deleteUserQuery = "DELETE FROM Users WHERE Id = @Id";
            using (var userCmd = new MySqlCommand(deleteUserQuery, conn))
            {
                userCmd.Parameters.AddWithValue("@Id", userId); // Add parameter to prevent SQL
                // injection
                userCmd.ExecuteNonQuery(); // Execute the delete command
            }
        }
        catch (Exception ex)
        {
            // Display an error message if an exception occurs
            MessageBox.Show($"Error Deleting User or Student: {ex.Message}");
        }
    }
}
```

The next is the BLL:

```
public void DeleteStudent(int userId)
{
    // Calls the repository method to delete the student record based on userId
    roleRepository.DeleteStudent(userId);
}
```

And the way how the UI interact with it:

```
private void Delete_Click(object sender, EventArgs e)
{
    try
    {
        // Check if any rows are selected in the data grid
        if (studatagrid.SelectedRows.Count > 0)
        {
            var selectedRow = studatagrid.SelectedRows[0]; // Get the first selected row
            int userId = Convert.ToInt32(selectedRow.Cells["ID"].Value); // Retrieve the user ID

            // Call the service to delete the student
            roleservice.DeleteStudent(userId);
        }
    }
}
```

```

        LoadStudent(); // Refresh the student list

        // Show success message
        MessageBox.Show("Students deleted successfully!", "Success", MessageBoxButtons.OK,
        MessageBoxIcon.Information);
    }
    else
    {
        // Inform the user to select a record if none is selected
        MessageBox.Show("Please select a record to delete.", "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Warning);
    }
}
catch (Exception ex)
{
    // Display an error message if an exception occurs
    MessageBox.Show($"An error occurred while deleting the Students: {ex.Message}", "Error",
    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
}

```

So this is all the function for three classes: **TeachingStaff**, **Administration**, and **Student**. these classes create a robust framework for efficiently managing educational data.

Also, one thing I forgot to mention about in the adding method for 3 class is the “Check Duplicate” method. In short, this method check for duplicate user in the database whether it exists or not, I will explain a little bit about this:

In the DAL:

```

public bool CheckDuplicateUser(string email)
{
    using (var conn = new MySqlConnection(_connectionString)) // Establish a database connection
    {
        var command = new MySqlCommand(@"
        SELECT COUNT(*) FROM Administration a
        JOIN Users u ON a.UserId = u.Id
        WHERE u.Email = @Email
        UNION ALL
        SELECT COUNT(*) FROM Students s
        JOIN Users u ON s.UserId = u.Id
        WHERE u.Email = @Email
        UNION ALL
        SELECT COUNT(*) FROM TeachingStaff t
        JOIN Users u ON t.UserId = u.Id
        WHERE u.Email = @Email", conn);

        command.Parameters.AddWithValue("@Email", email); // Add parameter to prevent SQL injection
        conn.Open(); // Open the connection

        int totalCount = 0;
        using (var reader = command.ExecuteReader()) // Execute the query
        {
            while (reader.Read())
            {
                totalCount += reader.GetInt32(0); // Sum the counts from all tables
            }
        }
        return totalCount > 0; // Return true if any duplicates are found
    }
}

```


The CheckDuplicateUser method checks for duplicate user entries based on an email address across three tables: Administration, Students, and TeachingStaff. It constructs a SQL query that counts entries in each table where the email matches the provided parameter. The results from these counts are summed up using a UNION ALL query. After executing the command, if the total count is greater than zero, the method returns true, indicating that duplicates exist; otherwise, it returns false. This approach ensures efficient and safe checking for duplicate users in the database.

In the BLL:

```
public bool CheckDuplicate(string email)
{
    // Calls the repository method to check the user record based on email
    return roleRepository.CheckDuplicateUser(email);
}
```

The CheckDuplicate method serves as a simple interface to verify if a user with a given email already exists in the database. It calls the CheckDuplicateUser method from the roleRepository, which handles the actual database query.

And in the UI, example from adding data for the “TeachingStaff” class:

```
if (roleservice.CheckDuplicate(txtEmail.Text))
{
    MessageBox.Show("An user with this email already exists.", "Input Error", MessageBoxButtons.OK,
    MessageBoxIcon.Warning);
    return;
}
```

So this is all the basic method for adding, editing and deleting record for 3 classes. The forwarding method will be mainly about how the system can view all data for user if view user by their unique role, what will be the output.

The first method to discussed about is the “View All” method. As the main function is to view all the record from 3 classes which will be displayed in 1 DataTable. And here is how we implement is:

First we will have a unique class to view all the user called “UserViewAll”:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ManageSystem.Models
{
    public class UserViewModel
    {
        public int ID { get; set; }
    }
}
```

```

public string Name { get; set; }
public string Telephone { get; set; }
public string Email { get; set; }
public string RoleType { get; set; }
public decimal? Salary { get; set; }
public int? WorkHours { get; set; }
public bool? IsFullTime { get; set; }
public string Subject1 { get; set; } // For TeachingStaff
public string Subject2 { get; set; } // For TeachingStaff
public string CurrentSubject1 { get; set; } // For Student
public string CurrentSubject2 { get; set; } // For Student
public string PreviousSubject1 { get; set; } // For Student
public string PreviousSubject2 { get; set; } // For Student
public string EmploymentType { get; set; } // For Administration
}
}

```

The UserViewModel class serves as a data transfer object that encapsulates user-related information for different roles within the management system. It includes properties such as ID, Name, Telephone, Email, and RoleType to identify and describe the user. Additional properties like Salary, WorkHours, and IsFullTime cater to teaching staff, while Subject1 and Subject2 reflect the subjects they teach. For students, properties like CurrentSubject1, PreviousSubject1, and others track their academic subjects. The EmploymentType property is specific to administrative roles, ensuring that the model can flexibly represent different employment types within the system.

And then, here is how we implement the DAL:

```

public void DeleteStudent(int userId)
{
    using (var conn = new MySqlConnection(_connectionString)) // Establish a database connection
    {
        conn.Open(); // Open the connection

        try
        {
            // Query to delete the student record
            string deleteTeachingStaffQuery = "DELETE FROM Students WHERE UserId = @UserId";
            using (var teachingStaffCmd = new MySqlCommand(deleteTeachingStaffQuery, conn))
            {
                teachingStaffCmd.Parameters.AddWithValue("@UserId", userId); // Add parameter to
prevent SQL injection
                teachingStaffCmd.ExecuteNonQuery(); // Execute the delete command
            }

            // Query to delete the user record
            string deleteUserQuery = "DELETE FROM Users WHERE Id = @Id";
            using (var userCmd = new MySqlCommand(deleteUserQuery, conn))
            {
                userCmd.Parameters.AddWithValue("@Id", userId); // Add parameter to prevent SQL
injection
                userCmd.ExecuteNonQuery(); // Execute the delete command
            }
        }
        catch (Exception ex)
        {
            // Display an error message if an exception occurs

```

```

        MessageBox.Show($"Error Deleting User or Student: {ex.Message}");
    }
}

public List<UserViewModel> GetAllData()
{
    // Initialize a list to hold the user data
    var users = new List<UserViewModel>();

    // SQL query to retrieve user data from three different roles
    string query = @"
SELECT 'Administration' AS RoleType, UserId AS id, name, telephone, email, salary, WorkingHours AS
WorkHours,
    NULL AS subject1, NULL AS subject2, NULL AS currentSubject1, NULL AS currentSubject2,
    NULL AS previousSubject1, NULL AS previousSubject2, EmploymentType
FROM Administration
JOIN Users ON Administration.UserId = Users.Id
UNION
SELECT 'TeachingStaff' AS RoleType, UserId AS id, name, telephone, email, salary,
    NULL AS WorkHours, Subject1, Subject2,
    NULL AS currentSubject1, NULL AS currentSubject2,
    NULL AS previousSubject1, NULL AS previousSubject2, NULL AS EmploymentType
FROM TeachingStaff
JOIN Users ON TeachingStaff.UserId = Users.Id
UNION
SELECT 'Students' AS RoleType, UserId AS id, name, telephone, email,
    NULL AS salary, NULL AS WorkHours,
    NULL AS subject1, NULL AS subject2, CurrentSubject1, CurrentSubject2,
    PreviousSubject1, PreviousSubject2, NULL AS EmploymentType
FROM Students
JOIN Users ON Students.UserId = Users.Id
ORDER BY RoleType, name; // Order the results by role type and name
";

    // Establish a connection to the database
    using (MySQLConnection conn = new MySQLConnection(_connectionString))
    {
        // Prepare the SQL command
        using (MySQLCommand cmd = new MySQLCommand(query, conn))
        {
            conn.Open(); // Open the database connection

            // Execute the command and read the results
            using (MySQLDataReader reader = cmd.ExecuteReader())
            {
                // Iterate through the results
                while (reader.Read())
                {
                    // Map each row to a UserViewModel object
                    var userViewModel = new UserViewModel
                    {
                        ID = Convert.ToInt32(reader["id"]), // Convert and assign ID
                        Name = reader["name"].ToString(), // Assign name
                        Telephone = reader["telephone"].ToString(), // Assign telephone
                        Email = reader["email"].ToString(), // Assign email
                        RoleType = reader["RoleType"].ToString(), // Assign role type
                        Salary = reader["Salary"] != DBNull.Value ?
Convert.ToDecimal(reader["Salary"]) : (decimal?)null, // Handle nullable salary
                        WorkHours = reader["WorkHours"] != DBNull.Value ?
Convert.ToInt32(reader["WorkHours"]) : (int?)null, // Handle nullable work hours
                        Subject1 = reader["Subject1"]?.ToString(), // Assign subject1 if not
null
                        Subject2 = reader["Subject2"]?.ToString(), // Assign subject2 if not
null
                    };
                }
            }
        }
    }
}

```

```

        CurrentSubject1 = reader["CurrentSubject1"]?.ToString(), // Assign
current subject 1 if not null
        CurrentSubject2 = reader["CurrentSubject2"]?.ToString(), // Assign
current subject 2 if not null
        PreviousSubject1 = reader["PreviousSubject1"]?.ToString(), // Assign
previous subject 1 if not null
        PreviousSubject2 = reader["PreviousSubject2"]?.ToString(), // Assign
previous subject 2 if not null
        EmploymentType = reader["EmploymentType"]?.ToString() // Assign
employment type if not null
    };

    // Add the UserViewModel to the list
    users.Add(userViewModel);
}
}
}

// Return the list of users
return users;
}

```

The GetAllData method retrieves user information from a MySQL database, combining data from three distinct roles: Administration, Teaching Staff, and Students, into a unified list of UserViewModel objects. It constructs a SQL query using UNION to select relevant fields from each role, ensuring consistent output structure. The method opens a database connection, executes the query, and reads the results into UserViewModel instances, handling potential null values gracefully. Finally, it returns a list of these instances, allowing easy access to user data across different roles

And then, after retrieving data from the database, next in the BLL:

```

public DataTable ConvertUserDataToDataTable(List<UserViewModel> userViewModels)
{
    // Create a new DataTable to hold user data
    var dataTable = new DataTable();

    // Define columns for the DataTable with appropriate data types
    dataTable.Columns.Add("ID", typeof(int));
    dataTable.Columns.Add("Name", typeof(string));
    dataTable.Columns.Add("Telephone", typeof(string));
    dataTable.Columns.Add("Email", typeof(string));
    dataTable.Columns.Add("RoleType", typeof(string));
    dataTable.Columns.Add("Salary", typeof(decimal));
    dataTable.Columns.Add("WorkHours", typeof(int));
    dataTable.Columns.Add("Subject1", typeof(string));
    dataTable.Columns.Add("Subject2", typeof(string));
    dataTable.Columns.Add("CurrentSubject1", typeof(string));
    dataTable.Columns.Add("CurrentSubject2", typeof(string));
    dataTable.Columns.Add("PreviousSubject1", typeof(string));
    dataTable.Columns.Add("PreviousSubject2", typeof(string));
    dataTable.Columns.Add("EmploymentType", typeof(string));

    // Populate the DataTable rows with user data
    foreach (var user in userViewModels)
    {
        dataTable.Rows.Add(
            user.ID,
            user.Name,

```

```

        user.Telephone,
        user.Email,
        user.RoleType,
        // Use DBNull.Value for nullable Salary
        user.Salary.HasValue ? (object)user.Salary.Value : DBNull.Value,
        // Use DBNull.Value for nullable WorkHours
        user.WorkHours.HasValue ? (object)user.WorkHours.Value : DBNull.Value,
        // Use DBNull.Value for nullable subjects
        user.Subject1 ?? (object)DBNull.Value,
        user.Subject2 ?? (object)DBNull.Value,
        user.CurrentSubject1 ?? (object)DBNull.Value,
        user.CurrentSubject2 ?? (object)DBNull.Value,
        user.PreviousSubject1 ?? (object)DBNull.Value,
        user.PreviousSubject2 ?? (object)DBNull.Value,
        user.EmploymentType ?? (object)DBNull.Value
    );
}

// Return the populated DataTable
return dataTable;
}

public DataTable GetAllUserData()
{
    // Retrieve all user data from the repository
    var userdatalist = roleRepository.GetAllData();
    // Convert the user data list to a DataTable and return it
    return ConvertUserDataToDataTable(userdatalist);
}

```

The ConvertUserDataToDataTable method converts a list of UserViewModel objects into a DataTable. It first defines the structure of the DataTable by adding columns with appropriate data types. Nullable properties like Salary and WorkHours are handled using DBNull.Value to ensure that null values are correctly represented in the DataTable. The method then iterates through the list of UserViewModel objects, adding each user's data as a new row in the DataTable. Special care is taken to handle nullable fields to avoid runtime errors.

The GetAllUserData method retrieves all user data from the repository and converts it into a DataTable format using the ConvertUserDataToDataTable method

And in the UI, here is how we interact with it:

```

private void viewall_Click(object sender, EventArgs e)
{
    // Clear existing controls in the PanelContainer
    PanelContainer.Controls.Clear();

    // Create a new instance of the ViewAll control
    ViewAll viewallcontroll = new ViewAll();

    // Set the control to fill the entire panel
    viewallcontroll.Dock = DockStyle.Fill;

    // Add the new control to the PanelContainer
    PanelContainer.Controls.Add(viewallcontroll);
}

```

Here is all the information for the “View All” function, the final function will be discussed is the “View By Role” function. As we have the DataTable for each class before, this function doesn’t require much implementation in the DAL or BLL. Here is the code for the “View By Role” function:

```
private void check_Click(object sender, EventArgs e)
{
    // Check if a role has been selected from the dropdown
    if (selectedrolebox.SelectedItem == null)
    {
        MessageBox.Show("Please select a role.", "Input Error", MessageBoxButtons.OK,
        MessageBoxIcon.Warning);
        return; // Exit if no role is selected
    }

    // Get the selected role as a string
    string selectedrole = selectedrolebox.SelectedItem.ToString();

    // Switch statement to handle different role types
    switch (selectedrole)
    {
        case "Teaching Staff":
            label3.Text = "Teaching Staff Data:"; // Update label for teaching staff
            label3.Visible = true; // Make the label visible
            var teachstaffdatatable = roleservice.GetTeachingStaffDataTable(); // Retrieve data
            roledata1.DataSource = teachstaffdatatable; // Bind data to grid
            break;

        case "Administration":
            label3.Text = "Administration Data:"; // Update label for administration
            label3.Visible = true; // Make the label visible
            var admindatatable = roleservice.GetAdministrationDataTable(); // Retrieve data
            roledata1.DataSource = admindatatable; // Bind data to grid
            break;

        case "Student":
            label3.Text = "Student Data:"; // Update label for students
            label3.Visible = true; // Make the label visible
            var studentdatatable = roleservice.GetStudentDataTable(); // Retrieve data
            roledata1.DataSource = studentdatatable; // Bind data to grid
            break;
    }
}
```

The check_Click method handles the click event of a button, validating user selection from a dropdown list of roles. If no role is selected, it displays a warning message. Upon selecting a role, it uses a switch statement to retrieve and display relevant data—either for "Teaching Staff," "Administration," or "Student"—updating a label and binding the retrieved data to a data grid.

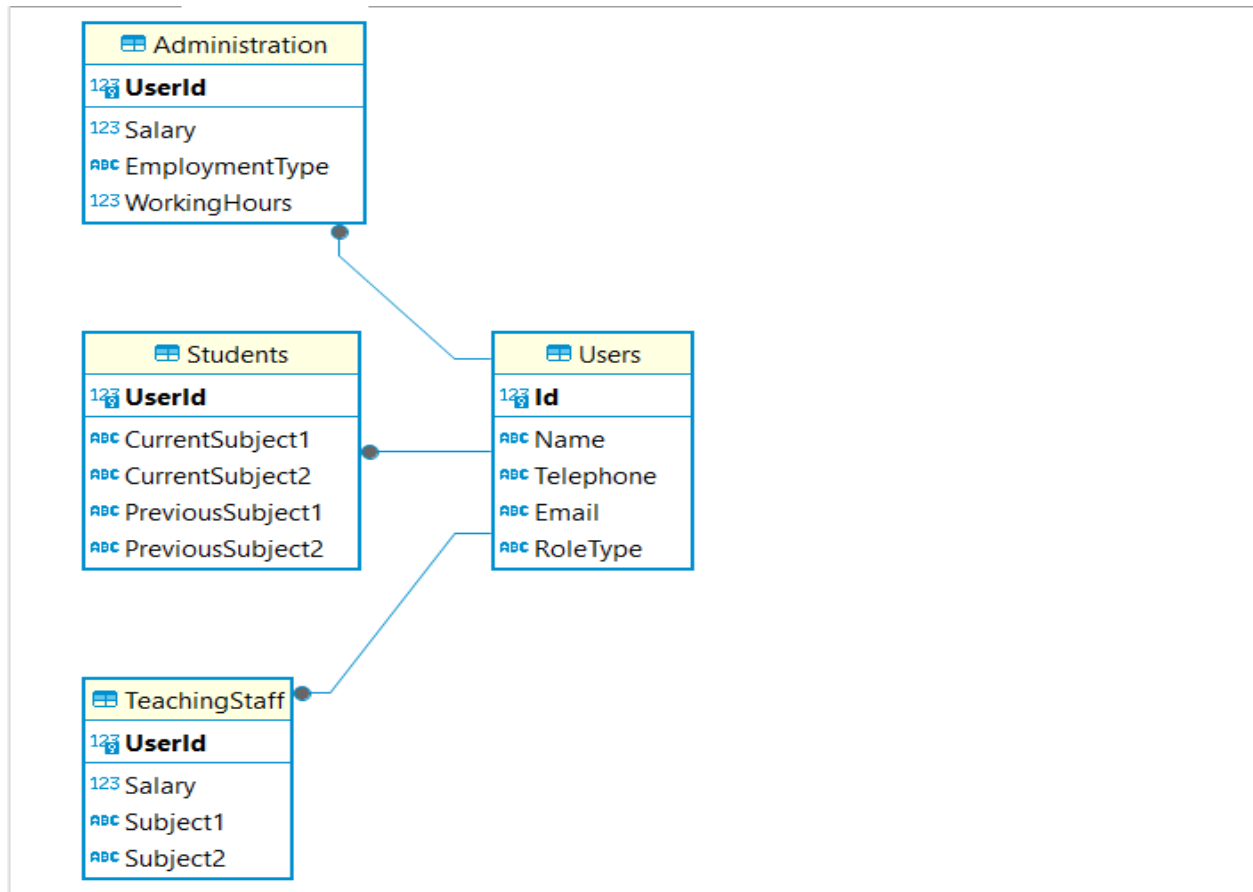
These are all the basic function for the system, all the implemented functions collectively enhance the functionality and user experience of the management system.

TASK 4: MANUAL TEST:

To ensure that each of the essential features are operational, below is a manual test table that includes the test case, expected result, evidence, and result.

Test case	Expected result	Evidence	Result
Run the application	Application run with successfully	Figure6	Pass
Register Admin Account	Registers successfully	Figure7	Pass
Login	Login successfully	Figure8	Pass
Add New Data	Data added successfully	Figure9	Pass
View All Data	All data displayed in 1 table	Figure10	Pass
View Data By Specific Role	Each data for each role displayed successfully	Figure11	Pass
Edit Existing Data	Data for each class is updated successfully	Figure12	Pass
Delete Existing Data	No more data for that class will be displayed	Figure13	Pass
Check Error For Input	Error will be shown when it had blank input	Figure14	Pass
Check Duplicate User	Information for duplicated user will be displayed	Figure15	Pass
Exit The System	Exit the system without error	Figure16	Pass

DATABASE DESIGN:



The database design consists of four main tables:

Users, Administration, Students, and TeachingStaff. Each table serves a specific purpose and is interrelated.

1. Users Table

- Id: A unique identifier for each user (primary key).
- Name: The name of the user.
- Telephone: The contact number of the user.
- Email: The email address of the user.
- RoleType: Indicates the role of the user (e.g., Administration, Teaching Staff, or Student).

This table acts as the core entity, storing common information about all users in the system.

2. Administration Table

- UserId: A foreign key that links to the Id in the Users table, establishing a one-to-one relationship.
- Salary: The salary of the administrative user.
- EmploymentType: Indicates the type of employment (e.g., full-time, part-time).
- WorkingHours: The number of hours the administrative user works.

This table holds specific information relevant to administrative users, extending the general user details.

3. Students Table

- UserId: A foreign key that links to the Id in the Users table, establishing a one-to-one relationship.
- CurrentSubject1: The first current subject the student is enrolled in.
- CurrentSubject2: The second current subject the student is enrolled in.
- PreviousSubject1: The first subject the student previously took.
- PreviousSubject2: The second subject the student previously took.

This table captures the academic details of students, allowing for tracking of their current and past subjects.

4. TeachingStaff Table

- UserId: A foreign key that links to the Id in the Users table, establishing a one-to-one relationship.
- Salary: The salary of the teaching staff member.
- Subject1: The first subject the teacher specializes in.
- Subject2: The second subject the teacher specializes in.

This table contains information specific to teaching staff, including their salary and the subjects they handle.

Relationships:

- Users to Administration: One-to-one relationship, where each administrative user corresponds to one entry in the Administration table.
- Users to Students: One-to-one relationship, where each student corresponds to one entry in the Students table.
- Users to TeachingStaff: One-to-one relationship, where each teaching staff member corresponds to one entry in the TeachingStaff table.

Overall, each table is connected through the **UserId**, ensuring referential integrity and enabling comprehensive queries across user types.

TASK 5: CONCLUSION:

In this coursework project, principles of object-oriented programming and C# are leveraged to create an educational administration system. The application establishes a connection to a database that maintains information for different user roles, such as Administrations, Teaching Staff, and Students. It includes a graphical user interface (GUI) that facilitates user data modifications and registration processes. The database design accommodates subject management, salary tracking, and employment information.... Additionally, concepts like inheritance and encapsulation are effectively applied in the C# programming.

APPENDIX:

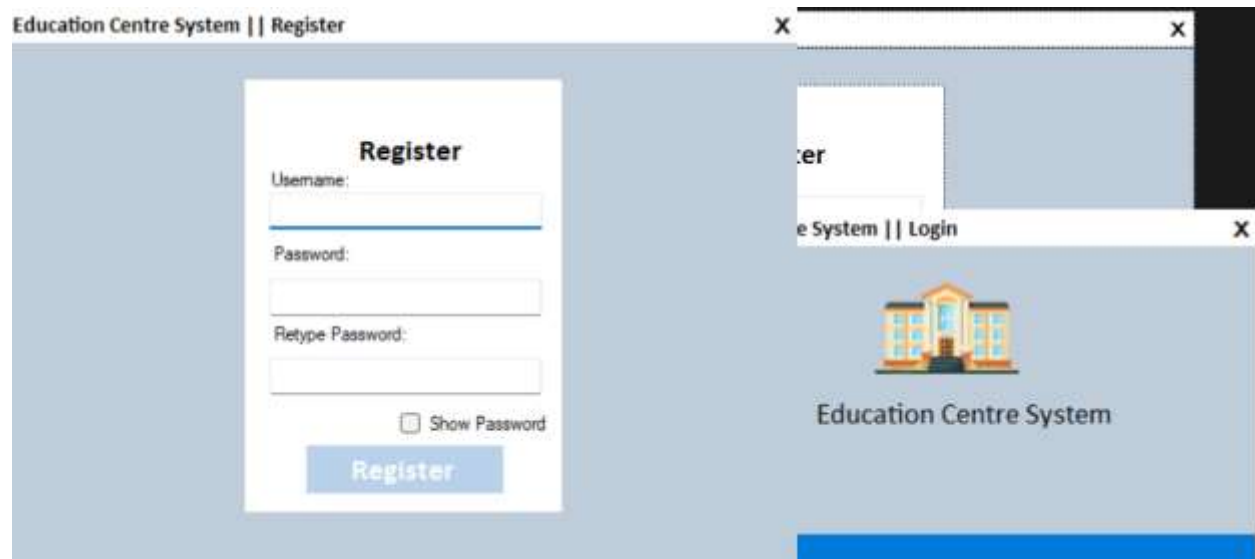


Figure 6: Main Program



Figure 7: Register

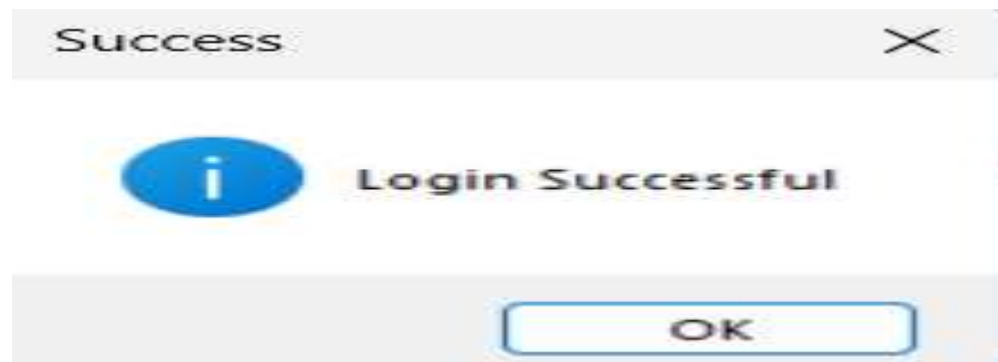


Figure 8: Login

Teaching Staff's Data:

	ID	Name	Telephone	Email	Salary	Su
▶	23	quang1	0342408183	lebao0824@gmai...	30000000.00	Mat
	24	g	03424081835	lebao0824@gmai...	3000000.00	Mat
	28	qaaaa	0342408185	lebao08245@gm...	2000000.00	Mat
	30	Huy Quang	0342408183	lebao08243@gm...	3333333.00	OO
*						

Name: Salary:


Telephone: Subject 1:

Email: Subject 2:

Add

Clear

Success

 Teaching staff added successfully

OK

Figure 9: Add new data

View All User:

ID	Name	Telephone	Email	Role Type	Salary	Work Ho
21	ninh	3434123123	lebao0824@gmail...	Administration	4000000.00	5
25	quang	0342408183	lebao0824@gmail...	Administration	4000000.00	5
29	quang	0342408184	lebao0824@gmail...	Administration	4000000.00	4
26	Huy Quang	0342408183	lebao08224@gmail...	Students		
24	g	03424081835	lebao0824@gmail...	Teaching Staff	3000000.00	
30	Huy Quang	0342408183	lebao08243@gmail...	Teaching Staff	3333333.00	
28	aaaa	0342408185	lebao08245@gmail...	Teaching Staff	2000000.00	
23	quang1	0342408183	lebao0824@gmail...	Teaching Staff	30000000.00	
31	thinh an	0342408183	helloimikang2@...	Teaching Staff	5000000.00	

Figure 10: View All Data

View User By Roles:

Teaching Staff

Teaching Staff Data:

ID	Name	Telephone	Email	Salary	St
23	quang1	0342408183	lebao0824@gmail...	30000000.00	Ma
24	g	03424081835	lebao0824@gmail...	3000000.00	Ma
28	aaaa	0342408185	lebao08245@gmail...	2000000.00	Ma
30	Huy Quang	0342408183	lebao08243@gmail...	3333333.00	OC
31	thinh an	0342408183	helloimikang2@...	5000000.00	Ma
*					

Figure 11: View Data By Role

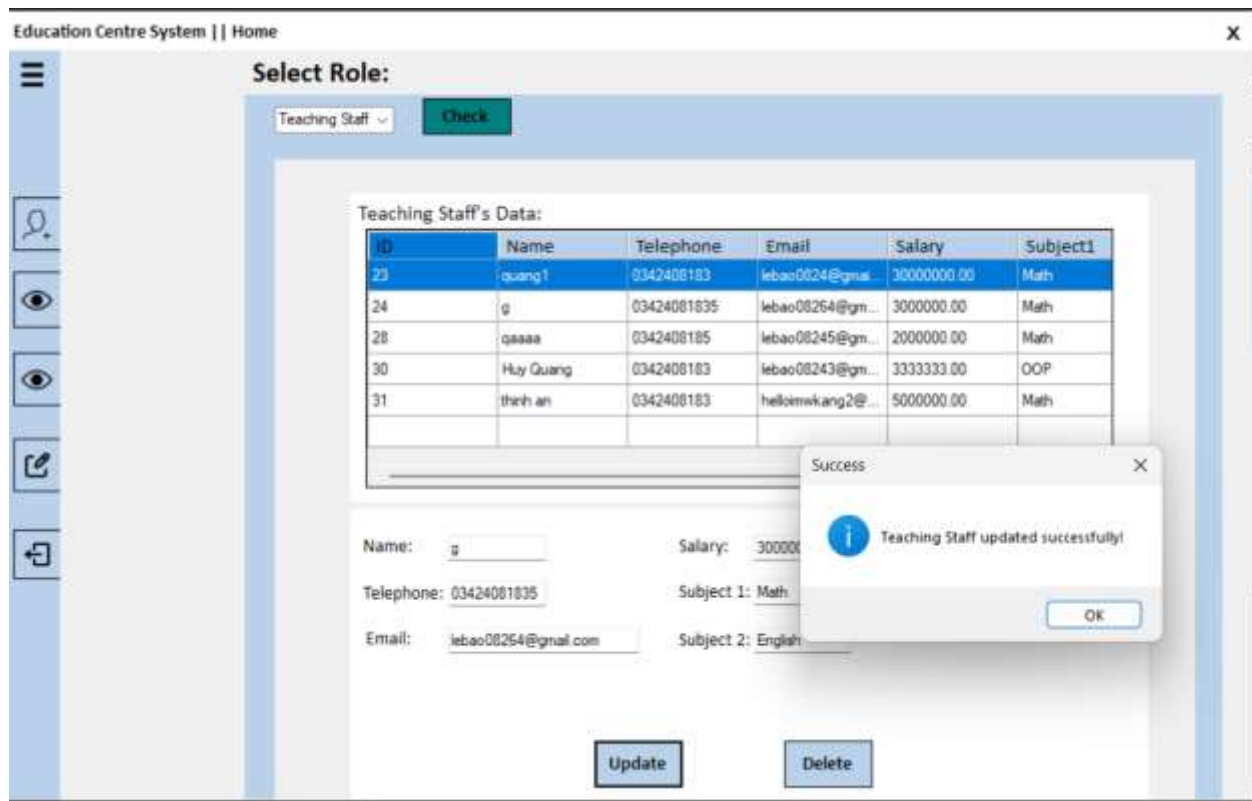


Figure 12: Edit Data For Each Role

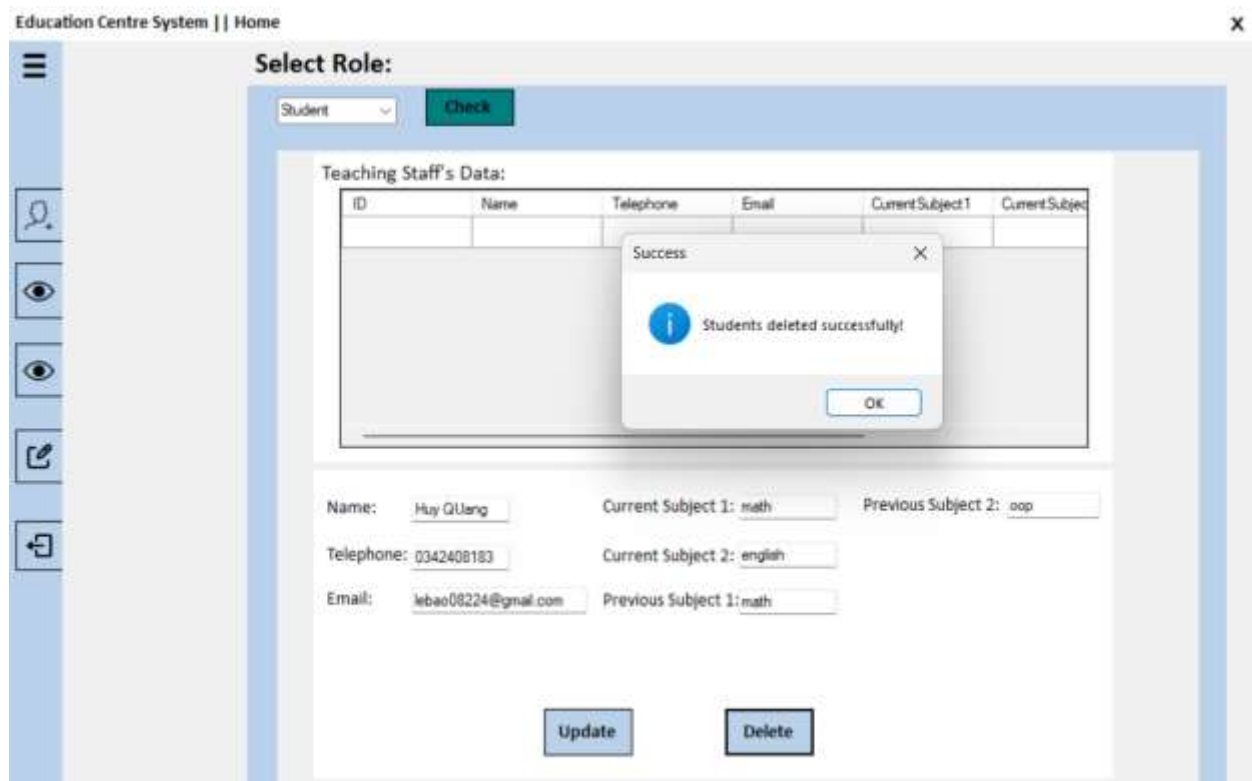


Figure 13: Delete Data For Each Role

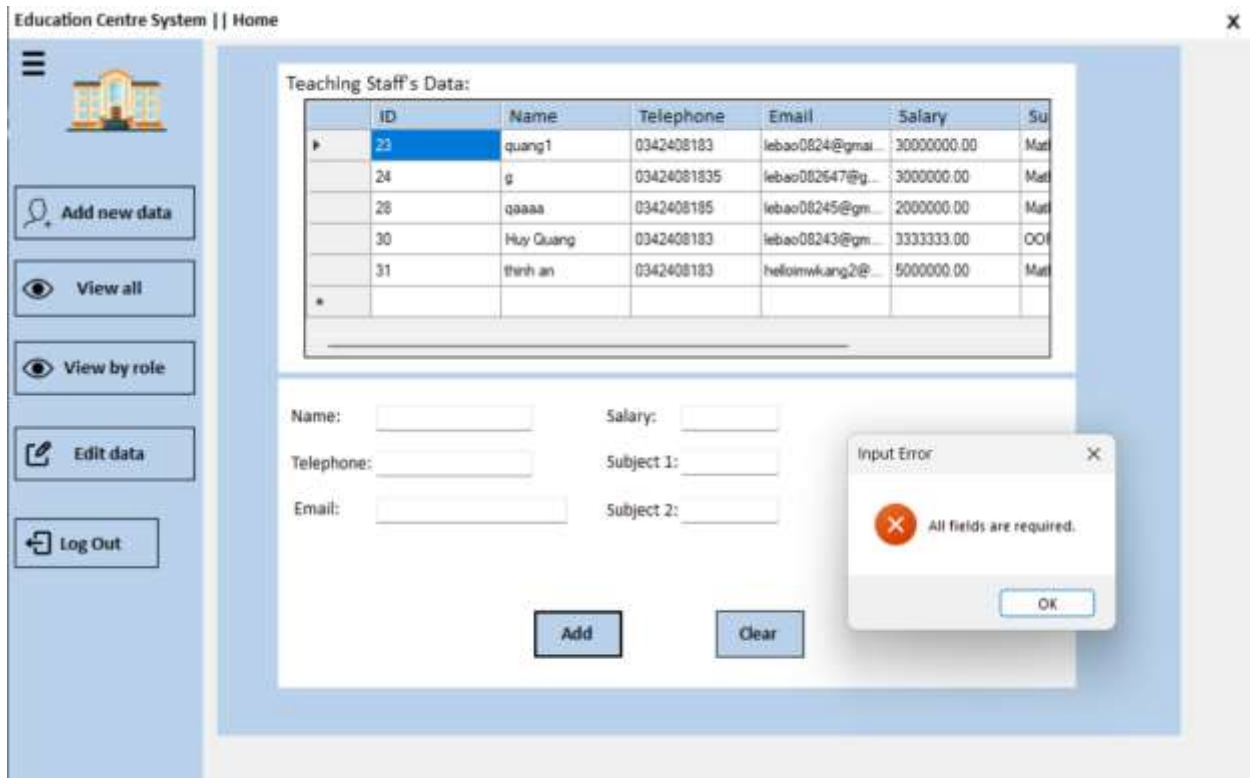


Figure 14: Blank Input Error in Adding Data

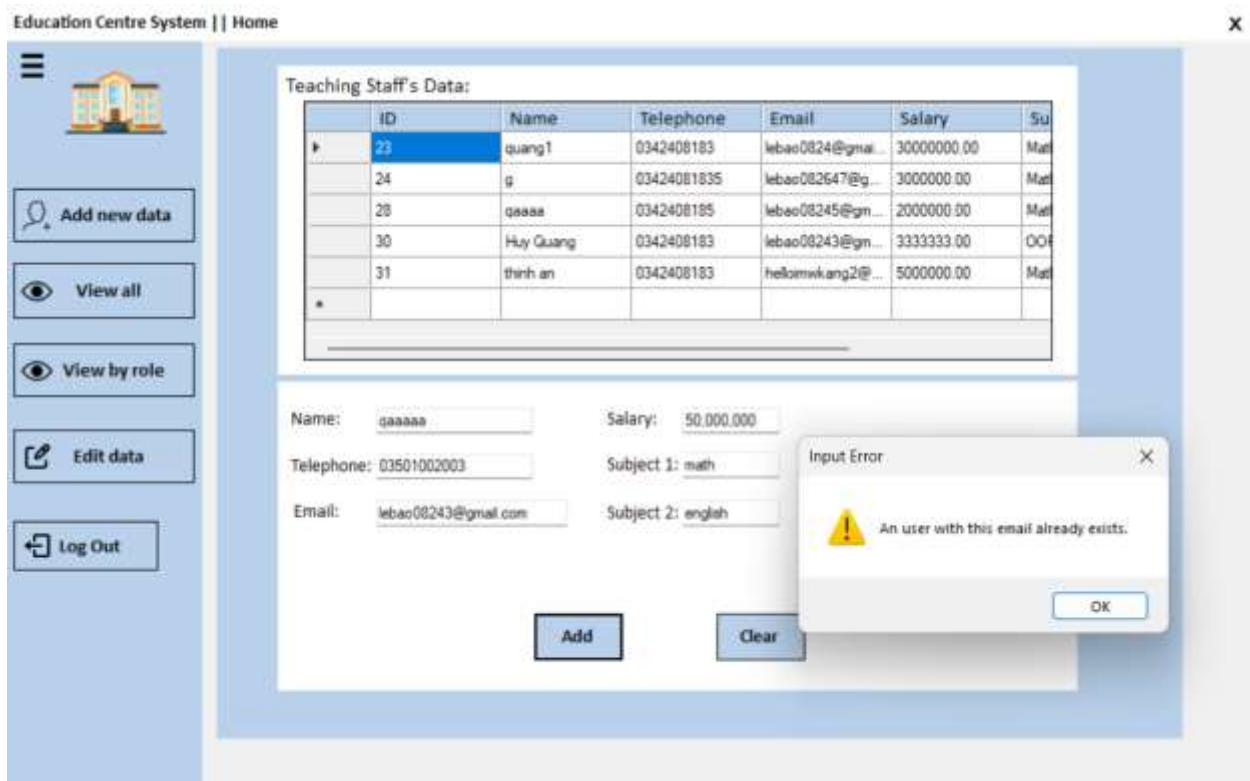


Figure 15: Check Duplicate User



The screenshot displays the 'Education Centre System' interface. On the left is a sidebar with a menu icon and a building icon, followed by buttons: 'Add new data', 'View all', 'View by role', 'Edit data', and 'Log Out'. The main area shows a table titled 'Teaching Staff's Data' with columns: ID, Name, Telephone, Email, Salary, and Subject. The table contains six rows of data. Below the table is a form with fields for Name, Salary, Telephone, Subject 1, and Subject 2, along with 'Add' and 'Clear' buttons. A 'Confirmation Message' dialog box is open, asking 'Are you sure you want to log out?' with 'Yes' and 'No' buttons.

ID	Name	Telephone	Email	Salary	Subject
23	quang1	0342408183	lebao0824@gmail...	3000000.00	Mat
24	g	03424081835	lebao082647@g...	3000000.00	Mat
28	qaaaa	0342408185	lebao08245@gm...	2000000.00	Mat
30	Huy Quang	0342408183	lebao08243@gm...	3333333.00	OO
31	thinh an	0342408183	helloimkwang2@...	5000000.00	Mat
*					

Name: Salary:
Telephone: Subject 1:
Email: Subject 2:

Confirmation Message
Are you sure you want to log out?

Figure 16: Exit the System