Project Report: Unicom TIC Management System

Author: Mishari

---

1. Project Overview

This document outlines the key features, technologies, and development journey of the Unicom TIC Management System, a desktop application designed to manage academic and administrative data for an educational institution.

Key Features Implemented:

- **Secure Role-Based Login System:** The application features a robust login screen that authenticates users and assigns them one of four roles: Admin, Student, Staff, or Lecturer.
- **Dynamic, Role-Based Dashboards:** The main dashboard is customized based on the user's role. Admins have access to a full "Manage" menu, while Students have a restricted view with access only to their personal information.
- **Full CRUD Functionality:** Admins have complete Create, Read, Update, and Delete (CRUD) capabilities for all core entities through dedicated management forms.
- **Modular Management System:** The application includes separate, easy-to-use modules for managing:
    - User Accounts (including role assignment)
    - Courses and Subjects (with relational links)
    - Student Records (with course enrollment)
    - Rooms and Timetables
- **Advanced Marks Entry System:** A master-detail form allows administrators to manage exams and efficiently enter marks for all students enrolled in the relevant subject.
- **Personalized Student Views:** Students can log in to view their own academic timetable and a personalized list of their exam marks, ensuring data privacy.
- **Portable Database:** Utilizes a file-based SQLite database (unicomtic.db), allowing the entire application to be run from a single folder with no complex setup.

Technologies Used:

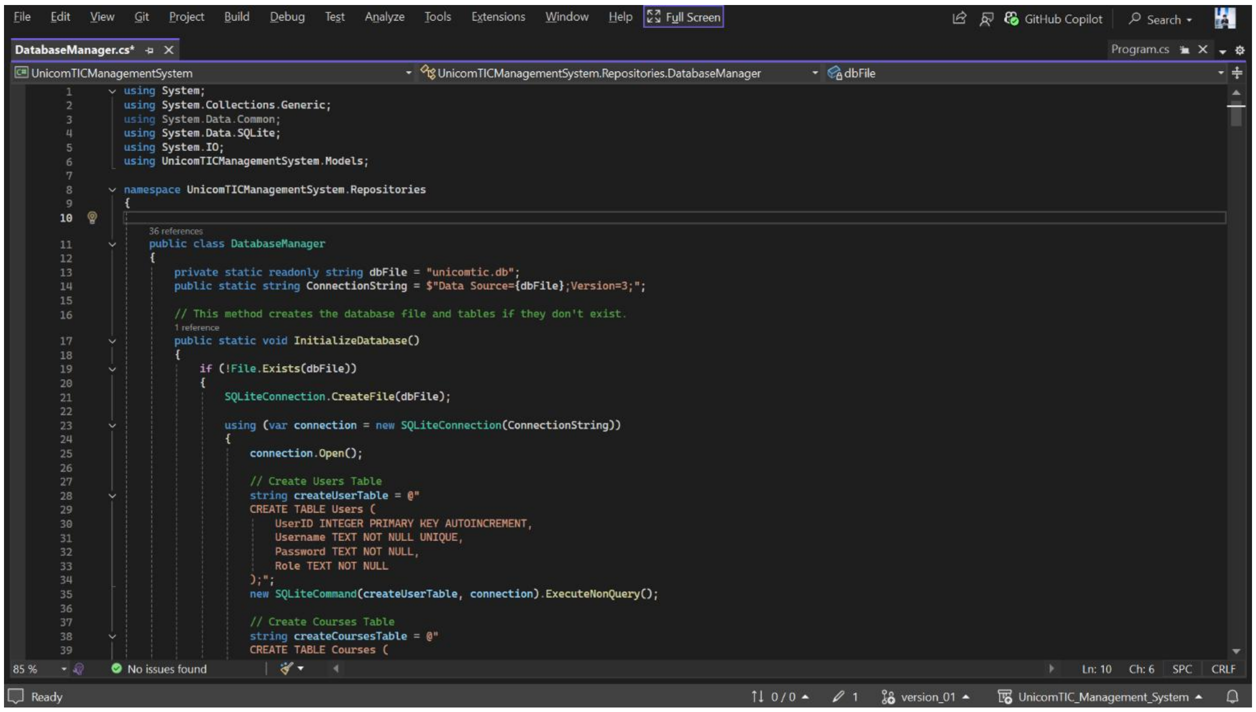
- **Programming Language:** C#
- **Framework:** .NET Framework
- **Application Type:** Windows Forms (WinForms) Desktop Application
- **Database:** SQLite
- **IDE:** Microsoft Visual Studio
- **Version Control:** Git & GitHub

Challenges Faced and Solutions:

- **Challenge: UI Elements not updating correctly after database changes.**
    - **Solution:** Implemented dedicated LoadData() methods (e.g., LoadUsersGrid()). After any action like adding or deleting, this method is called to refresh the DataSource of the DataGridView, ensuring the UI always reflects the current state of the database.
- **Challenge: Application crashing when moved to a new computer (SQLite.Interop.dll not found).**
    - **Solution:** Diagnosed the issue as a platform architecture mismatch. The problem was permanently fixed by setting the project's **Platform Target** to **x86** in the project properties. This ensures compatibility between the application and the SQLite library on any machine.
- **Challenge: Forms crashing in the Visual Studio Designer after being moved.**
    - **Solution:** Identified that complex code (like creating a controller) in a form's constructor can cause the designer to fail. The logic was refactored by moving all data-loading and controller-initialization code from the constructor into the Form_Load event, which is not run by the designer.

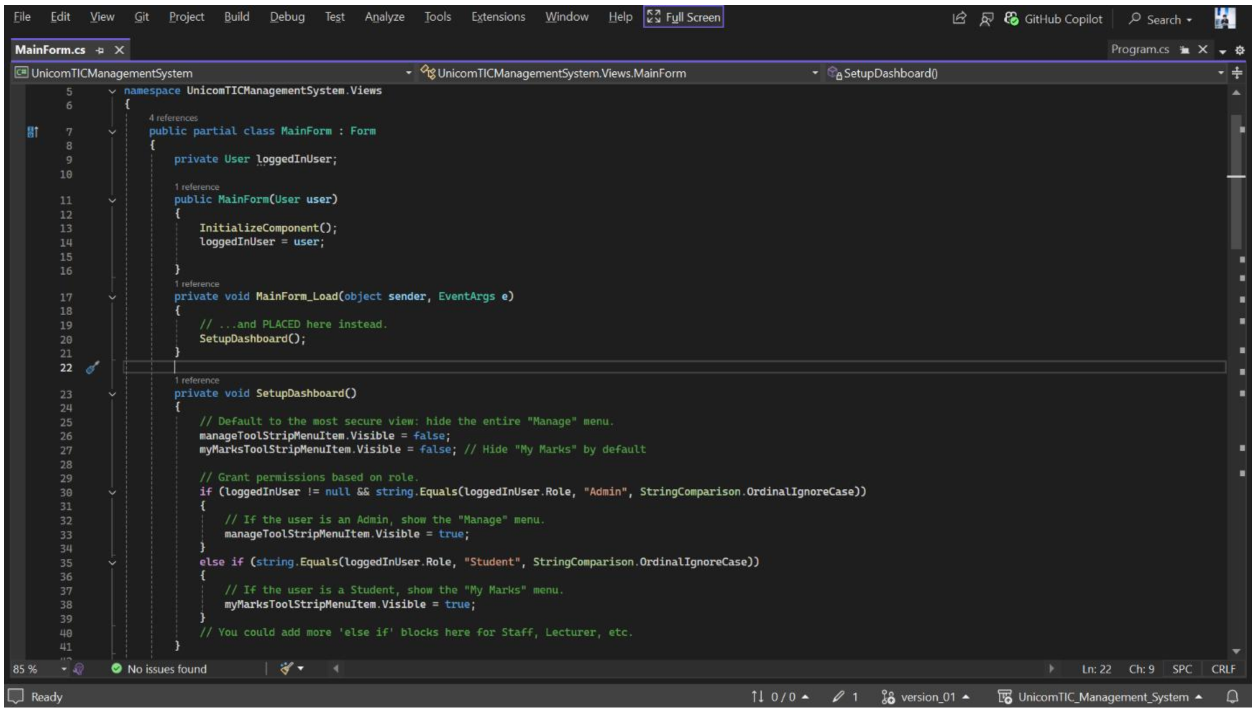
---

2. Code Samples (Screenshots)


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
GitHub Copilot
Search
DatabaseManager.cs*
Program.cs
UnicomTICManagementSystem
UnicomTICManagementSystem.Repositories.DatabaseManager
dbFile
85 %
No issues found
Ln: 10 Ch: 6 SPC CRLF
Ready
0 / 0
1
version_01
UnicomTIC_Management_System

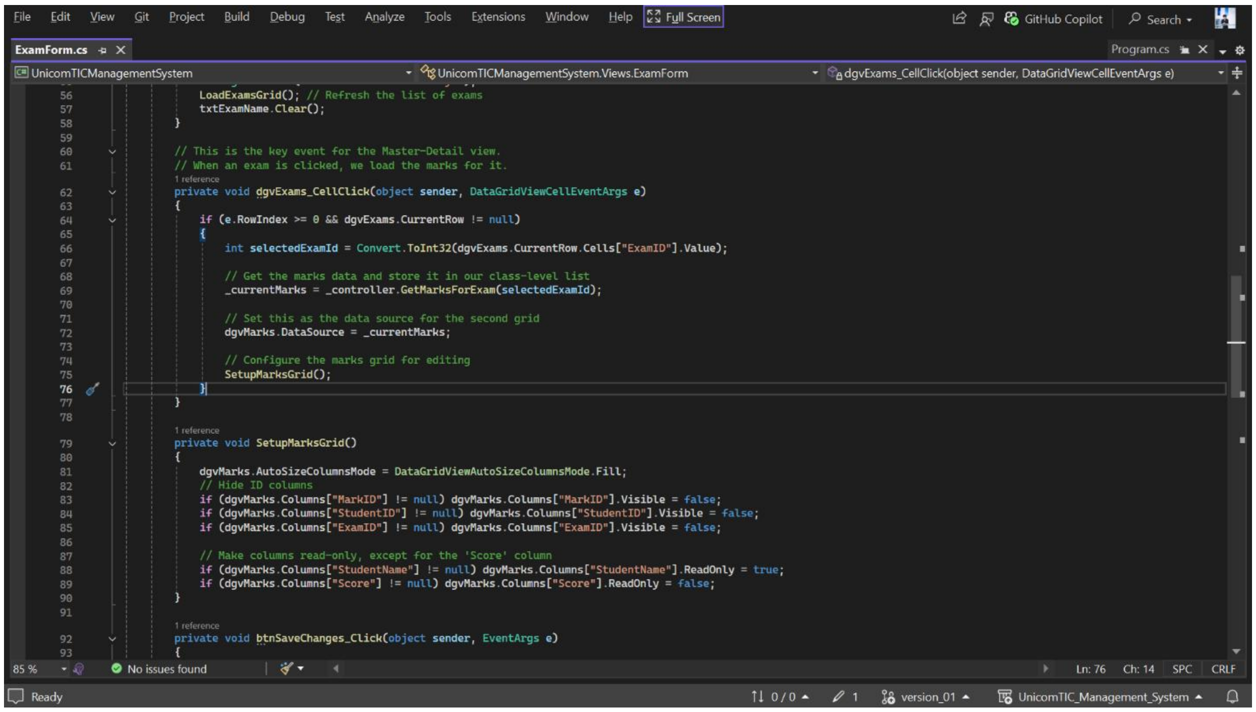
```csharp
using System;
using System.Collections.Generic;
using System.Data.Common;
using System.Data.SQLite;
using System.IO;
using UnicomTICManagementSystem.Models;

namespace UnicomTICManagementSystem.Repositories
{

    36 references
    public class DatabaseManager
    {
        private static readonly string dbFile = "unicomtic.db";
        public static string ConnectionString = $"Data Source={dbFile};Version=3;";

        // This method creates the database file and tables if they don't exist.
        1 reference
        public static void InitializeDatabase()
        {
            if (!File.Exists(dbFile))
            {
                SQLiteConnection.CreateFile(dbFile);

                using (var connection = new SQLiteConnection(ConnectionString))
                {
                    connection.Open();

                    // Create Users Table
                    string createUserTable = @"
                    CREATE TABLE Users (
                        UserID INTEGER PRIMARY KEY AUTOINCREMENT,
                        Username TEXT NOT NULL UNIQUE,
                        Password TEXT NOT NULL,
                        Role TEXT NOT NULL
                    );";
                    new SQLiteCommand(createUserTable, connection).ExecuteNonQuery();

                    // Create Courses Table
                    string createCoursesTable = @"
                    CREATE TABLE Courses (
```

Figure 1: The InitializeDatabase method, which creates the entire database schema and seeds it with default users.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
GitHub Copilot
Search
MainForm.cs
Program.cs
UnicomTICManagementSystem
UnicomTICManagementSystem.Views.MainForm
SetupDashboard()
85 %
No issues found
Ln: 22 Ch: 9 SPC CRLF
Ready
0 / 0
1
version_01
UnicomTIC_Management_System

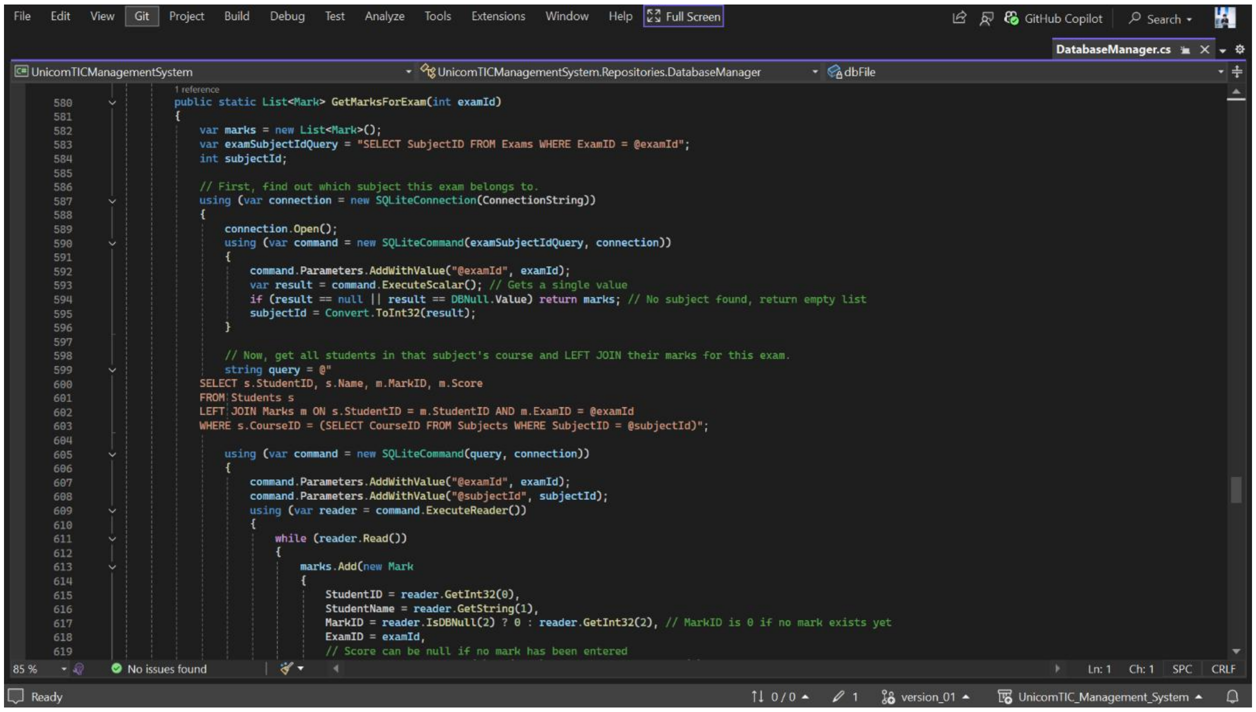
```csharp
namespace UnicomTICManagementSystem.Views
{
    4 references
    public partial class MainForm : Form
    {
        private User loggedInUser;

        1 reference
        public MainForm(User user)
        {
            InitializeComponent();
            loggedInUser = user;

        }
        1 reference
        private void MainForm_Load(object sender, EventArgs e)
        {
            // ...and PLACED here instead.
            SetupDashboard();
        }

        1 reference
        private void SetupDashboard()
        {
            // Default to the most secure view: hide the entire "Manage" menu.
            manageToolStripMenuItem.Visible = false;
            myMarksToolStripMenuItem.Visible = false; // Hide "My Marks" by default

            // Grant permissions based on role.
            if (loggedInUser != null && string.Equals(loggedInUser.Role, "Admin", StringComparison.OrdinalIgnoreCase))
            {
                // If the user is an Admin, show the "Manage" menu.
                manageToolStripMenuItem.Visible = true;
            }
            else if (string.Equals(loggedInUser.Role, "Student", StringComparison.OrdinalIgnoreCase))
            {
                // If the user is a Student, show the "My Marks" menu.
                myMarksToolStripMenuItem.Visible = true;
            }
            // You could add more 'else if' blocks here for Staff, Lecturer, etc.
        }
```

Figure 2: The SetupDashboard method, which implements role-based access control.



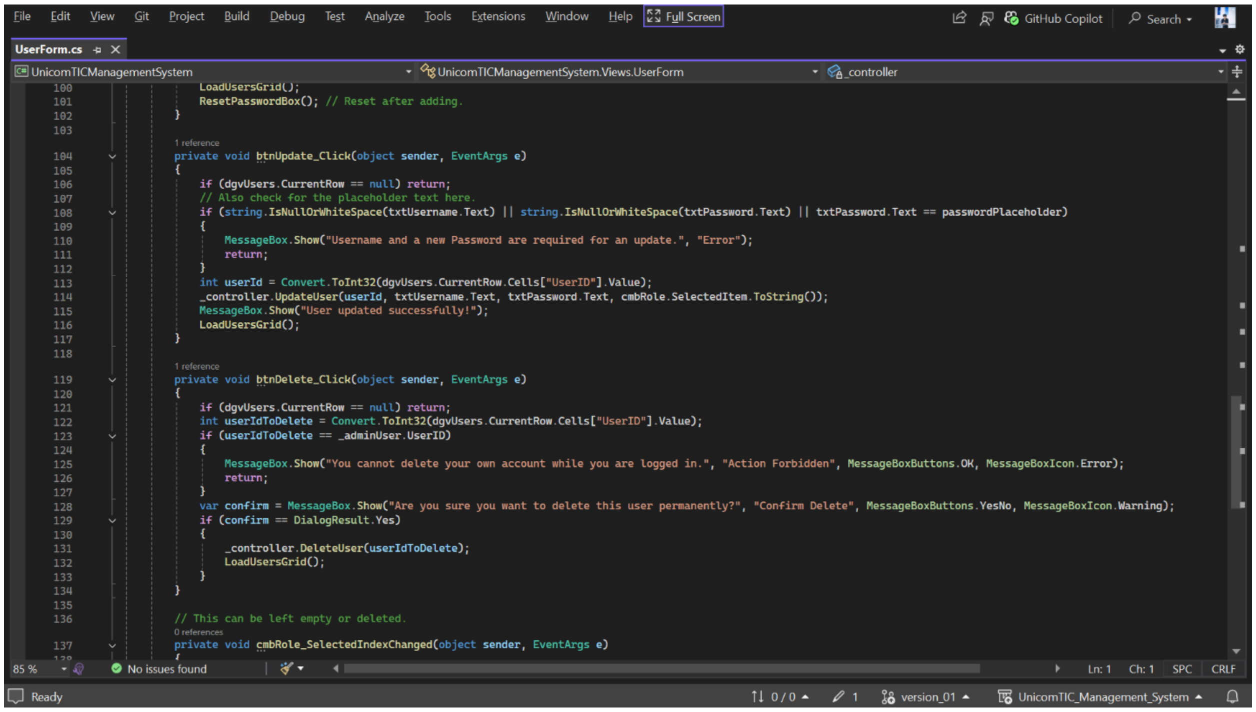
```
            LoadExamsGrid(); // Refresh the list of exams
            txtExamName.Clear();
        }

        // This is the key event for the Master-Detail view.
        // When an exam is clicked, we load the marks for it.
        1 reference
        private void dgvExams_CellClick(object sender, DataGridViewCellEventArgs e)
        {
            if (e.RowIndex >= 0 && dgvExams.CurrentRow != null)
            {
                int selectedExamId = Convert.ToInt32(dgvExams.CurrentRow.Cells["ExamID"].Value);

                // Get the marks data and store it in our class-level list
                _currentMarks = _controller.GetMarksForExam(selectedExamId);

                // Set this as the data source for the second grid
                dgvMarks.DataSource = _currentMarks;

                // Configure the marks grid for editing
                SetupMarksGrid();
            }
        }

        1 reference
        private void SetupMarksGrid()
        {
            dgvMarks.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;
            // Hide ID columns
            if (dgvMarks.Columns["MarkID"] != null) dgvMarks.Columns["MarkID"].Visible = false;
            if (dgvMarks.Columns["StudentID"] != null) dgvMarks.Columns["StudentID"].Visible = false;
            if (dgvMarks.Columns["ExamID"] != null) dgvMarks.Columns["ExamID"].Visible = false;

            // Make columns read-only, except for the 'Score' column
            if (dgvMarks.Columns["StudentName"] != null) dgvMarks.Columns["StudentName"].ReadOnly = true;
            if (dgvMarks.Columns["Score"] != null) dgvMarks.Columns["Score"].ReadOnly = false;
        }

        1 reference
        private void btnSaveChanges_Click(object sender, EventArgs e)
        {
```

Figure 3: The event handler for the master exams grid (dgvExams). This code demonstrates a master-detail UI pattern where selecting an exam in one grid triggers a database query to populate a second grid with a detailed list of students and their marks.



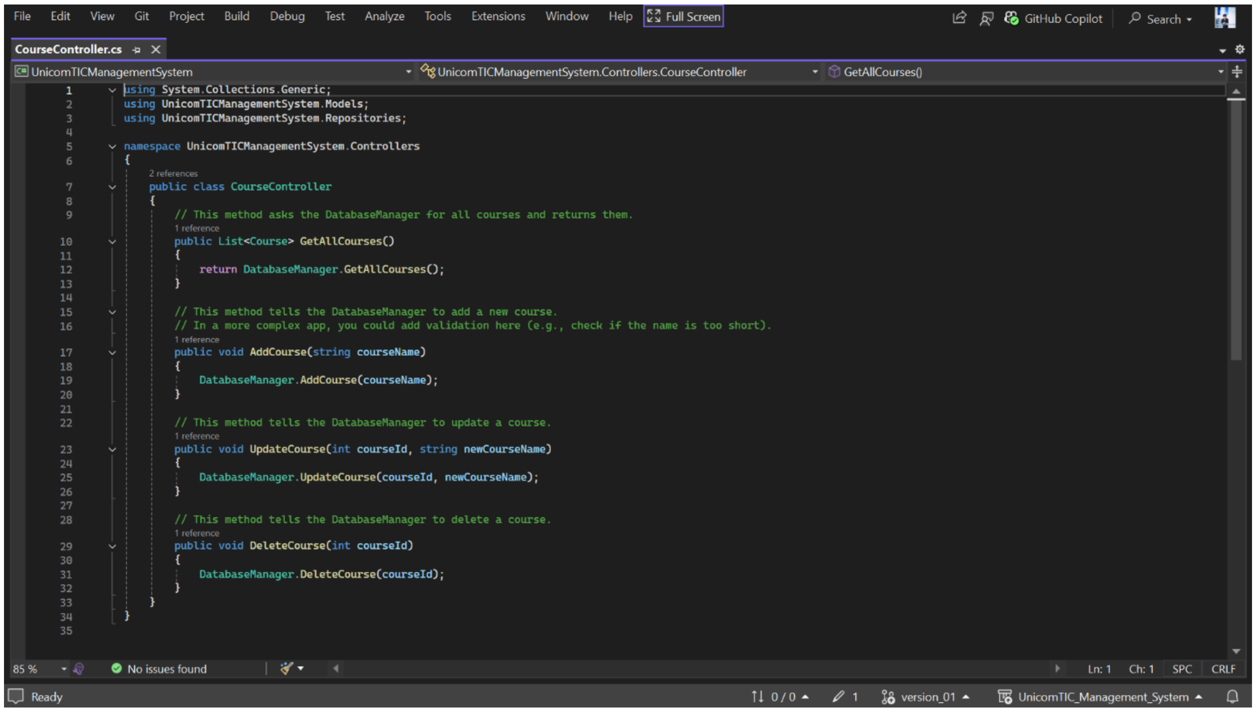
```
        1 reference
        public static List<Mark> GetMarksForExam(int examId)
        {
            var marks = new List<Mark>();
            var examSubjectIdQuery = "SELECT SubjectID FROM Exams WHERE ExamID = @examId";
            int subjectId;

            // First, find out which subject this exam belongs to.
            using (var connection = new SQLiteConnection(ConnectionString))
            {
                connection.Open();
                using (var command = new SQLiteCommand(examSubjectIdQuery, connection))
                {
                    command.Parameters.AddWithValue("@examId", examId);
                    var result = command.ExecuteScalar(); // Gets a single value
                    if (result == null || result == DBNull.Value) return marks; // No subject found, return empty list
                    subjectId = Convert.ToInt32(result);
                }

                // Now, get all students in that subject's course and LEFT JOIN their marks for this exam.
                string query = @"
            SELECT s.StudentID, s.Name, m.MarkID, m.Score
            FROM Students s
            LEFT JOIN Marks m ON s.StudentID = m.StudentID AND m.ExamID = @examId
            WHERE s.CourseID = (SELECT CourseID FROM Subjects WHERE SubjectID = @subjectId)";

                using (var command = new SQLiteCommand(query, connection))
                {
                    command.Parameters.AddWithValue("@examId", examId);
                    command.Parameters.AddWithValue("@subjectId", subjectId);
                    using (var reader = command.ExecuteReader())
                    {
                        while (reader.Read())
                        {
                            marks.Add(new Mark
                            {
                                StudentID = reader.GetInt32(0),
                                StudentName = reader.GetString(1),
                                MarkID = reader.IsDBNull(2) ? 0 : reader.GetInt32(2), // MarkID is 0 if no mark exists yet
                                ExamID = examId,
                                // Score can be null if no mark has been entered
```

Figure 4: A repository method featuring a complex SQL LEFT JOIN query. This query efficiently combines data from multiple tables (Students and Marks) to build a comprehensive list for the marks entry screen.



```
                LoadUsersGrid();
                ResetPasswordBox(); // Reset after adding.
            }

            private void btnUpdate_Click(object sender, EventArgs e)
            {
                if (dgvUsers.CurrentRow == null) return;
                // Also check for the placeholder text here.
                if (string.IsNullOrWhiteSpace(txtUsername.Text) || string.IsNullOrWhiteSpace(txtPassword.Text) || txtPassword.Text == passwordPlaceholder)
                {
                    MessageBox.Show("Username and a new Password are required for an update.", "Error");
                    return;
                }
                int userId = Convert.ToInt32(dgvUsers.CurrentRow.Cells["UserID"].Value);
                _controller.UpdateUser(userId, txtUsername.Text, txtPassword.Text, cmbRole.SelectedItem.ToString());
                MessageBox.Show("User updated successfully!");
                LoadUsersGrid();
            }

            private void btnDelete_Click(object sender, EventArgs e)
            {
                if (dgvUsers.CurrentRow == null) return;
                int userIdToDelete = Convert.ToInt32(dgvUsers.CurrentRow.Cells["UserID"].Value);
                if (userIdToDelete == _adminUser.UserID)
                {
                    MessageBox.Show("You cannot delete your own account while you are logged in.", "Action Forbidden", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    return;
                }
                var confirm = MessageBox.Show("Are you sure you want to delete this user permanently?", "Confirm Delete", MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
                if (confirm == DialogResult.Yes)
                {
                    _controller.DeleteUser(userIdToDelete);
                    LoadUsersGrid();
                }
            }

            // This can be left empty or deleted.
            private void cmbRole_SelectedIndexChanged(object sender, EventArgs e)
```

Figure 5: A security check within the User Management form that prevents an administrator from deleting their own account. This demonstrates defensive programming by anticipating and preventing potentially harmful user actions.



```
using System.Collections.Generic;
using UnicomTICManagementSystem.Models;
using UnicomTICManagementSystem.Repositories;

namespace UnicomTICManagementSystem.Controllers
{
    public class CourseController
    {
        // This method asks the DatabaseManager for all courses and returns them.
        public List<Course> GetAllCourses()
        {
            return DatabaseManager.GetAllCourses();
        }

        // This method tells the DatabaseManager to add a new course.
        // In a more complex app, you could add validation here (e.g., check if the name is too short).
        public void AddCourse(string courseName)
        {
            DatabaseManager.AddCourse(courseName);
        }

        // This method tells the DatabaseManager to update a course.
        public void UpdateCourse(int courseId, string newCourseName)
        {
            DatabaseManager.UpdateCourse(courseId, newCourseName);
        }

        // This method tells the DatabaseManager to delete a course.
        public void DeleteCourse(int courseId)
        {
            DatabaseManager.DeleteCourse(courseId);
        }
    }
}
```

Figure 6: The CourseController.cs class, demonstrating the MVC pattern. This controller handles all business logic for course management, connecting the user interface to the database repository.