Unicom TIC Management System - Project Report

Student: S.M.Suhaib-(UT010664)
Project: Unicom TIC Management System (UMS)

---

1. Project Overview

This report details the development of the Unicom TIC Management System, a C# WinForms desktop application designed to handle the core operational needs of a school. The system follows a Model-View-Controller (MVC) architecture and utilizes a local SQLite database for data persistence.

Key Features Implemented:

- **Secure, Role-Based Authentication System:**
    - A dedicated login screen authenticates four distinct user roles: Admin, Staff, Lecturer, and Student.
    - The system implements robust role-based access control (RBAC), where the user interface dynamically changes after login. Each role only sees the functions and modules they are permitted to access, ensuring data security and a tailored user experience.
- **Comprehensive Management Modules:**
    - **Course & Subject Management:** Admins can perform full CRUD (Create, Read, Update, Delete) operations on courses and their associated subjects in a master-detail view.
    - **Student Management:** Admins have full control over student records, including creation, updates, and deletion.
    - **Timetable & Room Allocation:** A powerful feature allowing Admins to schedule classes by assigning subjects to specific time slots and rooms (differentiated by "Lab" or "Hall" types). All other roles have read-only access to the timetable.
    - **Exams & Marks Management:** A unified, tabbed interface allows authorized users (Admin, Staff) to create exams and for authorized users (Admin, Staff, Lecturer) to enter and update student marks. Students are provided with a secure, read-only view of their own marks.

- **Modern User Interface & User Experience:**
    - The application features a consistent, modern UI with a professional color scheme, clear layouts, and responsive controls.
    - UI elements like the TabControl are used to logically group related functions (like managing exams and marks), improving workflow.
    - Client-side validation provides immediate, user-friendly feedback with clear error messages, preventing bad data submission.

Technologies and Architecture Used:

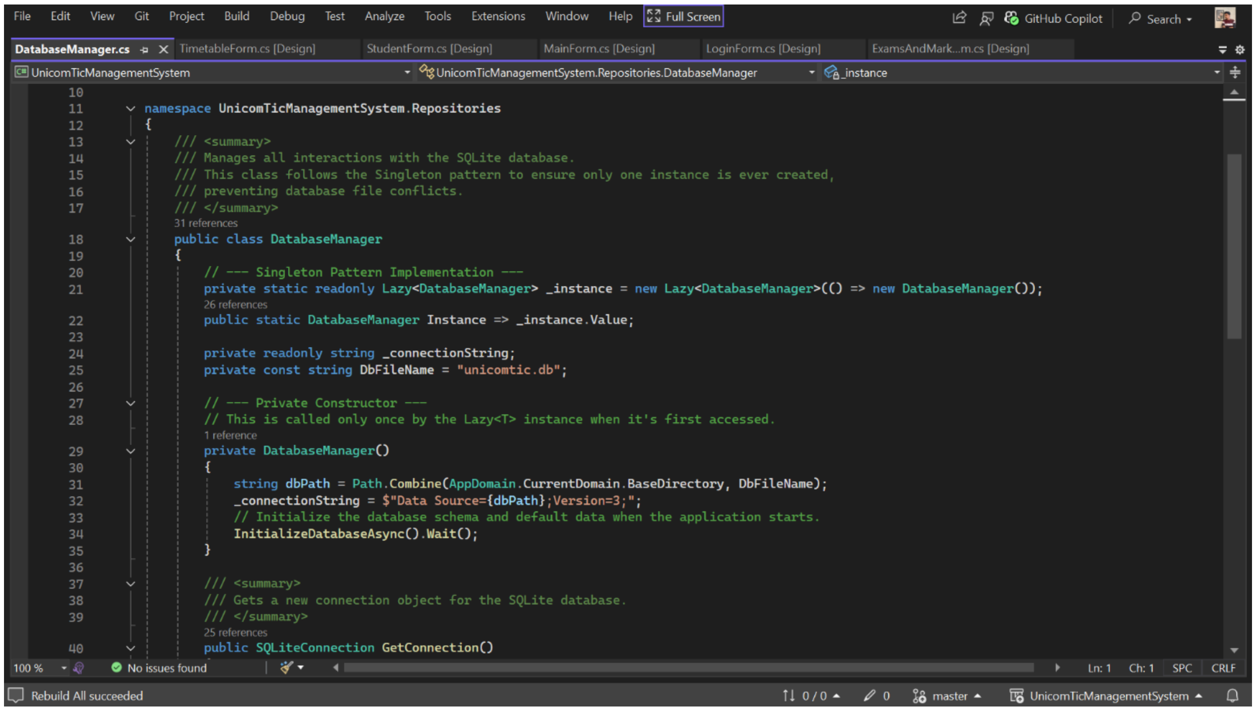
- **Language & Framework:** C# with .NET Framework.
- **User Interface:** Windows Forms (WinForms).
- **Database:** SQLite, a lightweight, file-based database perfect for desktop applications.
- **Database Driver:** System.Data.SQLite NuGet package.
- **Core Architecture:** Model-View-Controller (MVC)
    - **Model:** Plain C# objects representing database entities (e.g., Student.cs, Course.cs).
    - **View:** The WinForms themselves (e.g., LoginForm.cs, MainForm.cs).
    - **Controller:** Logic classes (e.g., StudentController.cs) that mediate between the View and the Model/Repository.
- **Design Patterns:**
    - **Singleton Pattern:** Used for the DatabaseManager to ensure a single, stable connection point to the database throughout the application's lifecycle.
    - **Repository Pattern:** The DatabaseManager acts as a repository, abstracting all data access logic away from the rest of the application.
- **Key C# Features:**
    - **async/await:** Implemented for all database operations to ensure the user interface remains responsive and never freezes during data-intensive tasks.
    - **using statement:** Used consistently to ensure proper disposal of database connections and other resources, preventing memory leaks.

Challenges Faced and Solutions:

- **Challenge 1: Application Freezing During Database Calls.**
    - **Problem:** Initially, when loading large amounts of data (like the list of students or marks), the application UI would become unresponsive until the database query was complete.
    - **Solution:** I refactored all data access methods in the DatabaseManager and the corresponding calls in the controllers and forms to be asynchronous using async/await. This moved the database work off the UI thread, resulting in a smooth, non-freezing user experience.
- **Challenge 2: The SQLite.Interop.dll Not Found Error on New Machines.**
    - **Problem:** After cloning the project from GitHub onto a new computer, the application would crash on startup with a DllNotFoundException because it couldn't find the essential SQLite.Interop.dll file.
    - **Solution:** I diagnosed this as a NuGet package configuration issue. The solution involved cleaning the project, manually editing the .csproj file to remove incorrect references, and performing a clean reinstall of the correct System.Data.SQLite meta-package. This forced Visual Studio to correctly add the necessary build targets, which now automatically copy the required native DLLs (for both x86 and x64 platforms) to the output directory during the build process, making the project portable and fixing the issue permanently.
- **Challenge 3: Complex UI Logic for Different Roles.**
    - **Problem:** Passing user information through multiple form constructors to control UI visibility was becoming complex and difficult to maintain.
    - **Solution:** I implemented a static UserSession class. Upon a successful login, the user's details (ID, Role, Username) are stored globally in this static class. Any form can then access this information directly (UserSession.Role) without needing it to be passed through its constructor. This significantly decoupled the forms, simplified the code, and made the role-based logic much cleaner and more robust.

---

2. Code Samples (Screenshots):


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
GitHub Copilot
Search
DatabaseManager.cs
TimetableForm.cs [Design]
StudentForm.cs [Design]
MainForm.cs [Design]
LoginForm.cs [Design]
ExamsAndMark...m.cs [Design]
UnicomTicManagementSystem
UnicomTicManagementSystem.Repositories.DatabaseManager
_instance
100 %
No issues found
Ln: 1 Ch: 1 SPC CRLF
Rebuild All succeeded
0 / 0
0
master
UnicomTicManagementSystem

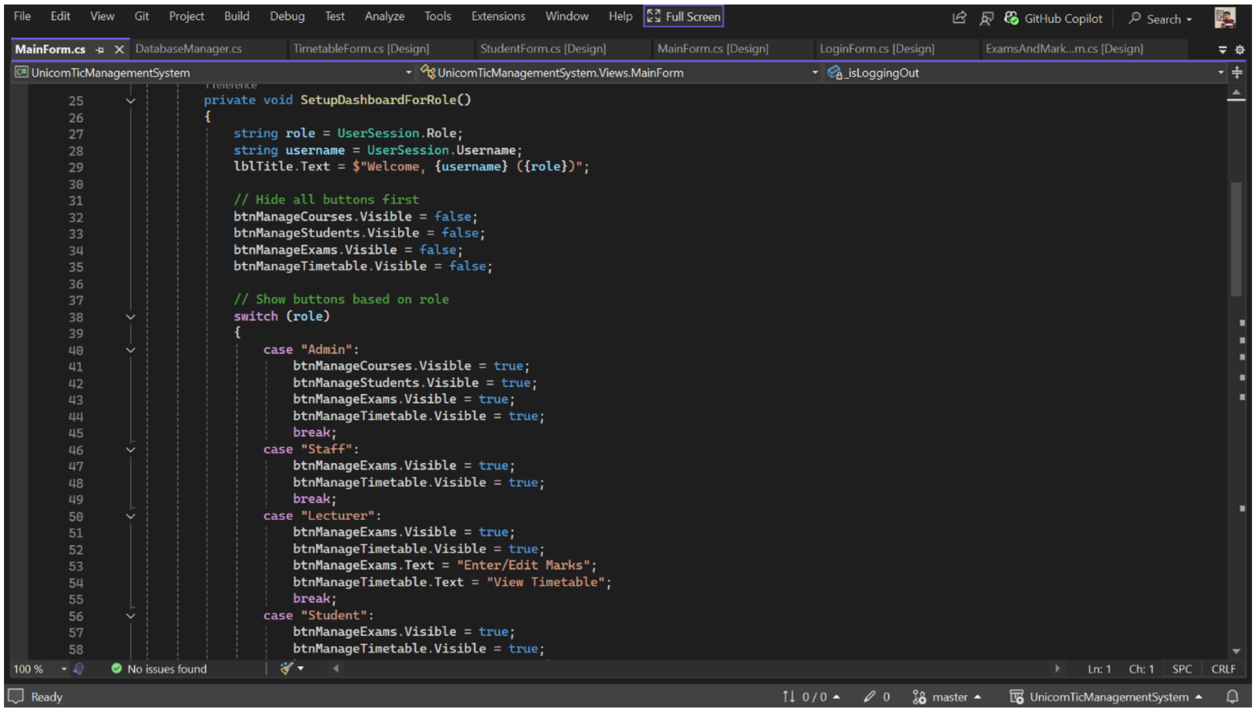
```
namespace UnicomTicManagementSystem.Repositories
{
    /// <summary>
    /// Manages all interactions with the SQLite database.
    /// This class follows the Singleton pattern to ensure only one instance is ever created,
    /// preventing database file conflicts.
    /// </summary>
    31 references
    public class DatabaseManager
    {
        // --- Singleton Pattern Implementation ---
        private static readonly Lazy<DatabaseManager> _instance = new Lazy<DatabaseManager>(() => new DatabaseManager());
        26 references
        public static DatabaseManager Instance => _instance.Value;

        private readonly string _connectionString;
        private const string DbFileName = "unicomtic.db";

        // --- Private Constructor ---
        // This is called only once by the Lazy<T> instance when it's first accessed.
        1 reference
        private DatabaseManager()
        {
            string dbPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, DbFileName);
            _connectionString = $"Data Source={dbPath};Version=3;";
            // Initialize the database schema and default data when the application starts.
            InitializeDatabaseAsync().Wait();
        }

        /// <summary>
        /// Gets a new connection object for the SQLite database.
        /// </summary>
        25 references
        public SQLiteConnection GetConnection()
```

Figure 1:The DatabaseManager Singleton and Robust Initialization. This code demonstrates two key architectural patterns. The Singleton pattern (Lazy<T>) ensures only one instance of the database manager exists, preventing file conflicts

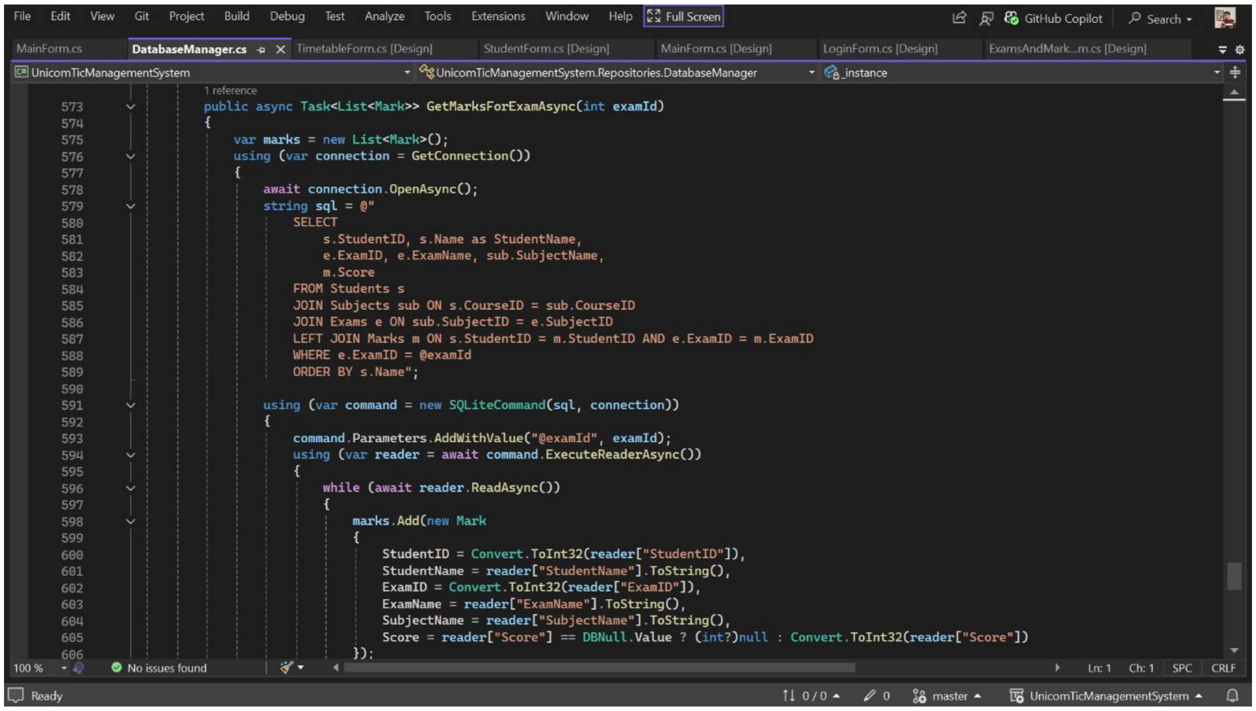


```
private void SetupDashboardForRole()
{
    string role = UserSession.Role;
    string username = UserSession.Username;
    lblTitle.Text = $"Welcome, {username} ({role})";

    // Hide all buttons first
    btnManageCourses.Visible = false;
    btnManageStudents.Visible = false;
    btnManageExams.Visible = false;
    btnManageTimetable.Visible = false;

    // Show buttons based on role
    switch (role)
    {
        case "Admin":
            btnManageCourses.Visible = true;
            btnManageStudents.Visible = true;
            btnManageExams.Visible = true;
            btnManageTimetable.Visible = true;
            break;
        case "Staff":
            btnManageExams.Visible = true;
            btnManageTimetable.Visible = true;
            break;
        case "Lecturer":
            btnManageExams.Visible = true;
            btnManageTimetable.Visible = true;
            btnManageExams.Text = "Enter/Edit Marks";
            btnManageTimetable.Text = "View Timetable";
            break;
        case "Student":
            btnManageExams.Visible = true;
            btnManageTimetable.Visible = true;
```

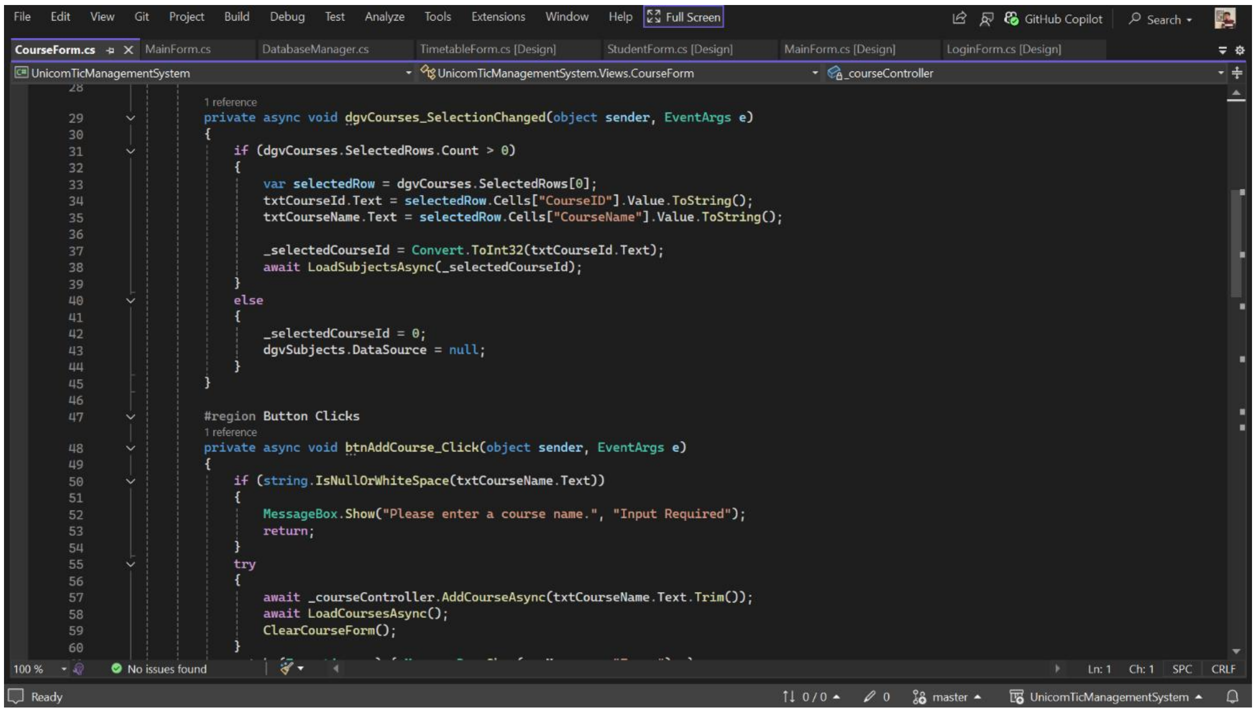
Figure 2: Dynamic UI with Role-Based Access Control. This method in the MainForm showcases the implementation of role-based security.



```
public async Task<List<Mark>> GetMarksForExamAsync(int examId)
{
    var marks = new List<Mark>();
    using (var connection = GetConnection())
    {
        await connection.OpenAsync();
        string sql = @"
            SELECT
                s.StudentID, s.Name as StudentName,
                e.ExamID, e.ExamName, sub.SubjectName,
                m.Score
            FROM Students s
            JOIN Subjects sub ON s.CourseID = sub.CourseID
            JOIN Exams e ON sub.SubjectID = e.SubjectID
            LEFT JOIN Marks m ON s.StudentID = m.StudentID AND e.ExamID = m.ExamID
            WHERE e.ExamID = @examId
            ORDER BY s.Name";

        using (var command = new SQLiteCommand(sql, connection))
        {
            command.Parameters.AddWithValue("@examId", examId);
            using (var reader = await command.ExecuteReaderAsync())
            {
                while (await reader.ReadAsync())
                {
                    marks.Add(new Mark
                    {
                        StudentID = Convert.ToInt32(reader["StudentID"]),
                        StudentName = reader["StudentName"].ToString(),
                        ExamID = Convert.ToInt32(reader["ExamID"]),
                        ExamName = reader["ExamName"].ToString(),
                        SubjectName = reader["SubjectName"].ToString(),
                        Score = reader["Score"] == DBNull.Value ? (int?)null : Convert.ToInt32(reader["Score"])
                    });
```

Figure 3: Efficient Data Retrieval with a Complex SQL JOIN. This method demonstrates an efficient database query that joins four different tables (Students, Subjects, Exams, Marks) using a combination of JOIN and LEFT JOIN.

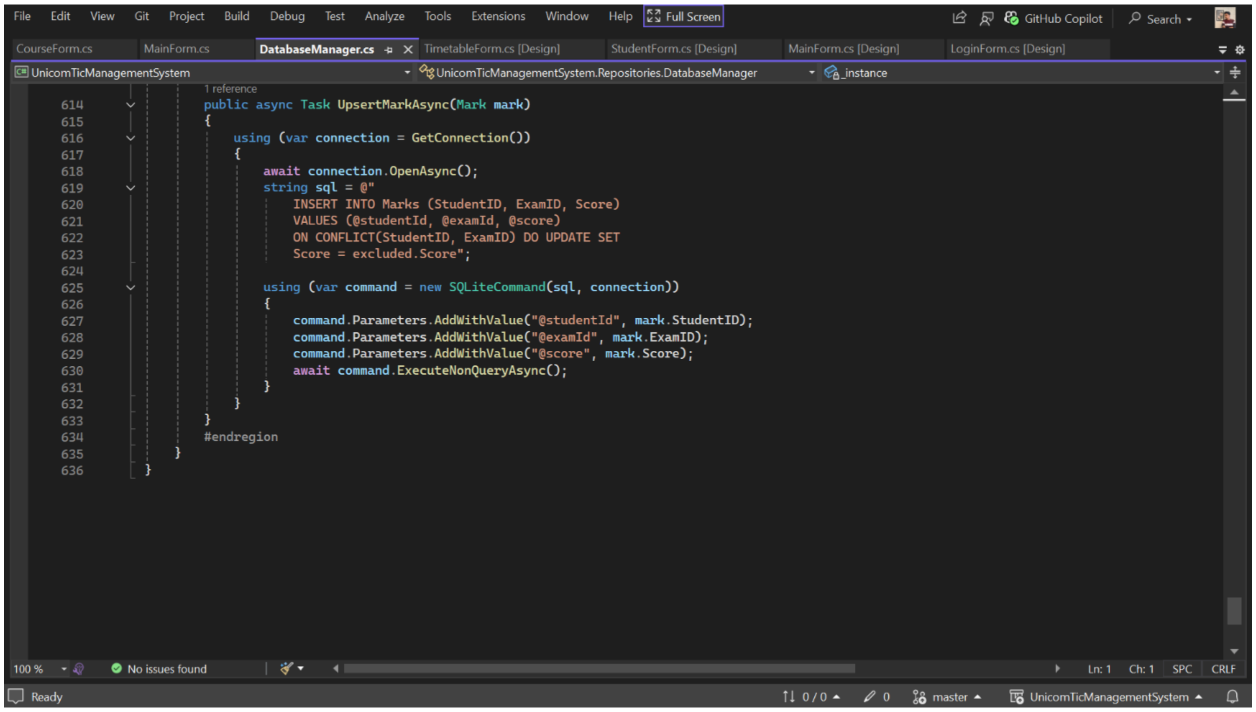


```csharp
        1 reference
        private async void dgvCourses_SelectionChanged(object sender, EventArgs e)
        {
            if (dgvCourses.SelectedRows.Count > 0)
            {
                var selectedRow = dgvCourses.SelectedRows[0];
                txtCourseId.Text = selectedRow.Cells["CourseID"].Value.ToString();
                txtCourseName.Text = selectedRow.Cells["CourseName"].Value.ToString();

                _selectedCourseId = Convert.ToInt32(txtCourseId.Text);
                await LoadSubjectsAsync(_selectedCourseId);
            }
            else
            {
                _selectedCourseId = 0;
                dgvSubjects.DataSource = null;
            }
        }

        #region Button Clicks
        1 reference
        private async void btnAddCourse_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrWhiteSpace(txtCourseName.Text))
            {
                MessageBox.Show("Please enter a course name.", "Input Required");
                return;
            }
            try
            {
                await _courseController.AddCourseAsync(txtCourseName.Text.Trim());
                await LoadCoursesAsync();
                ClearCourseForm();
            }
```

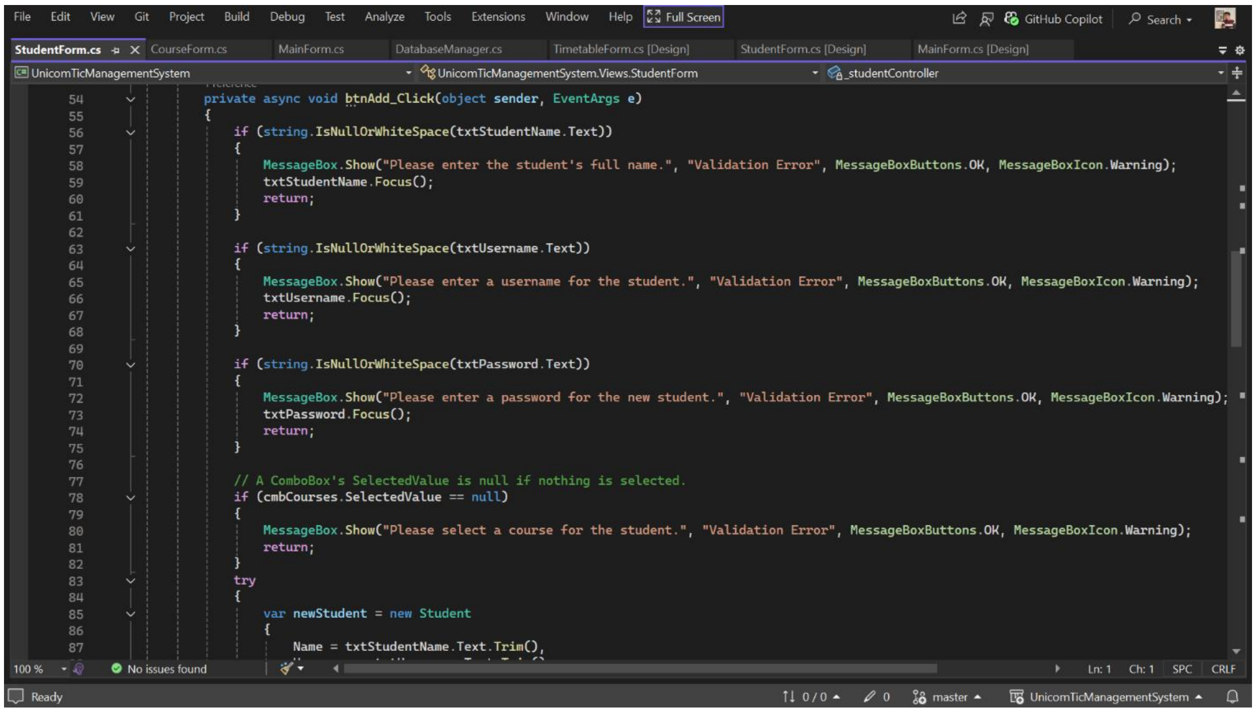
Figure 4: Implementing a "Master-Detail" User Interface. This event handler from the CourseForm creates an interactive master-detail view. When the user selects a course from the main "master" grid (dgvCourses), this code is triggered.



```csharp
        1 reference
        public async Task UpsertMarkAsync(Mark mark)
        {
            using (var connection = GetConnection())
            {
                await connection.OpenAsync();
                string sql = @"
                    INSERT INTO Marks (StudentID, ExamID, Score)
                    VALUES (@studentId, @examId, @score)
                    ON CONFLICT(StudentID, ExamID) DO UPDATE SET
                    Score = excluded.Score";

                using (var command = new SQLiteCommand(sql, connection))
                {
                    command.Parameters.AddWithValue("@studentId", mark.StudentID);
                    command.Parameters.AddWithValue("@examId", mark.ExamID);
                    command.Parameters.AddWithValue("@score", mark.Score);
                    await command.ExecuteNonQueryAsync();
                }
            }
        }
        #endregion
    }
}
```

Figure 5: Efficient Database Writes with an UPSERT Operation. This method demonstrates an advanced and highly efficient database technique.


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
GitHub Copilot Search
StudentForm.cs CourseForm.cs MainForm.cs DatabaseManager.cs TimetableForm.cs [Design] StudentForm.cs [Design] MainForm.cs [Design]
UnicomTicManagementSystem
UnicomTicManagementSystem.Views.StudentForm
_studentController
100 % No issues found Ln: 1 Ch: 1 SPC CRLF
Ready 0 / 0 0 master UnicomTicManagementSystem

```
        private async void btnAdd_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrWhiteSpace(txtStudentName.Text))
            {
                MessageBox.Show("Please enter the student's full name.", "Validation Error", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                txtStudentName.Focus();
                return;
            }

            if (string.IsNullOrWhiteSpace(txtUsername.Text))
            {
                MessageBox.Show("Please enter a username for the student.", "Validation Error", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                txtUsername.Focus();
                return;
            }

            if (string.IsNullOrWhiteSpace(txtPassword.Text))
            {
                MessageBox.Show("Please enter a password for the new student.", "Validation Error", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                txtPassword.Focus();
                return;
            }

            // A ComboBox's SelectedValue is null if nothing is selected.
            if (cmbCourses.SelectedValue == null)
            {
                MessageBox.Show("Please select a course for the student.", "Validation Error", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                return;
            }
            try
            {
                var newStudent = new Student
                {
                    Name = txtStudentName.Text.Trim(),
```

Figure 6: Proactive Client-Side Validation. This code block from the "Add Student" button click demonstrates proactive UI validation. Before any attempt is made to contact the controller or database, the code checks each required input field.