UnicomTicManagementSystem - Project Report

1. Project Overview

Key Features Implemented

• *Multi-Role User Management System*

- Admin, Student, Staff, and Lecture role-based access control
- Secure login authentication with password management
- Role-specific dashboard and functionality access

• *Comprehensive Student Management*

- Student registration, profile management, and enrollment
- Section-based student organization
- Student-teacher relationship management
- Student dashboard with personalized views

• *Academic Management System*

- Subject and course management with section-based organization
- Exam creation and management linked to subjects
- Mark recording and grade management system
- Attendance tracking with date-based records

• *Resource Management*

- Room allocation and management system
- Timetable scheduling and management
- Staff and teacher profile management
- Section-based organizational structure

• *Advanced Database Operations*

- SQLite database with proper foreign key relationships

- Transaction-based data operations for data integrity
- Async/await pattern implementation for responsive UI
- Comprehensive error handling and validation

Technologies Used

• *Backend Framework*: .NET Framework 4.8 • *Database*: SQLite with System.Data.SQLite • *UI Framework*: Windows Forms (WinForms) • *Architecture Pattern*: Repository Pattern with Service Layer • *Programming Language*: C# • *Design Patterns*:

- Repository Pattern for data access
- Service Layer for business logic
- MVC-like separation of concerns
- Factory Pattern for object creation

Challenges Faced and Solutions

• *Challenge 1: Database Schema Design*

- *Problem*: Complex relationships between students, teachers, subjects, and sections
- *Solution*: Implemented normalized database schema with proper foreign key constraints and unique constraints for data integrity

• *Challenge 2: Role-Based Access Control*

- *Problem*: Different user roles requiring different functionality access
- *Solution*: Implemented dynamic UI control loading based on user roles with centralized access control logic

• *Challenge 3: Async Database Operations*

- *Problem*: UI freezing during database operations
- *Solution*: Implemented async/await pattern throughout the application with proper error handling and transaction management

• *Challenge 4: Data Validation and Error Handling*

- *Problem*: Ensuring data integrity and providing meaningful error messages
- *Solution*: Implemented comprehensive validation in service layer with custom exception handling and user-friendly error messages

• *Challenge 5: Code Maintainability*

- *Problem*: Large codebase with potential for code duplication
- *Solution*: Implemented Repository pattern with base classes, service layer abstraction, and consistent coding standards

2. Code Samples (Best Work)

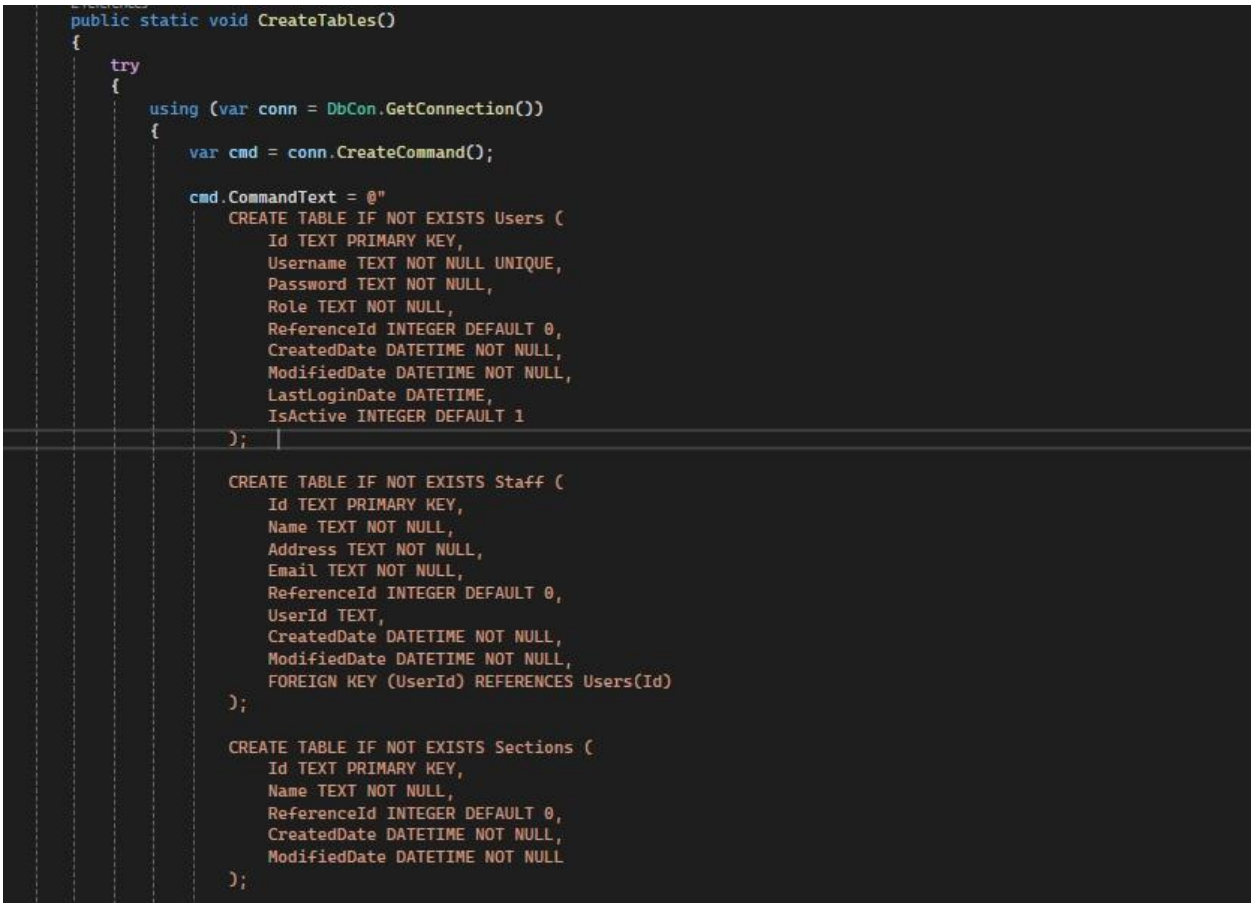
Best Code 1: BaseRepository Pattern Implementation

File: Controllers/Repositories/BaseRepository.cs *Description*: Demonstrates clean abstraction and reusable data access patterns with comprehensive error handling and async support.

```
public abstract class BaseRepository<T> where T : class
{
    protected readonly string _connectionString;

    0 references
    protected BaseRepository()
    {
        _connectionString = DbCon.GetConnectionString();
    }

    6 references
    protected SQLiteConnection GetConnection()
    {
        try
        {
            var connection = new SQLiteConnection(_connectionString);
            connection.Open();
            return connection;
        }
        catch (SQLiteException ex)
        {
            throw new Exception($"Failed to establish database connection: {ex.Message}", ex);
        }
        catch (Exception ex)
        {
            throw new Exception($"Unexpected error while connecting to database: {ex.Message}", ex);
        }
    }
```

Best Code 2: Database Schema Design

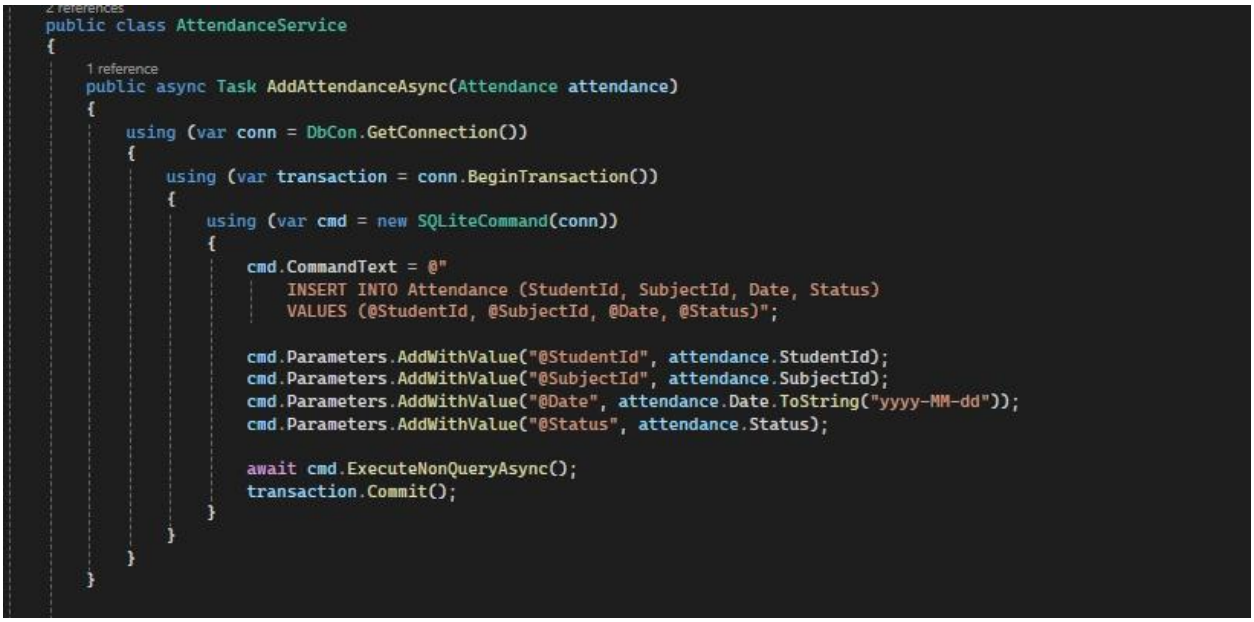
File: Data/DataInitializer.cs *Description*: Shows comprehensive database schema with proper relationships, constraints, and normalization.



```
public static void CreateTables()
{
    try
    {
        using (var conn = DbCon.GetConnection())
        {
            var cmd = conn.CreateCommand();

            cmd.CommandText = @"
                CREATE TABLE IF NOT EXISTS Users (
                    Id TEXT PRIMARY KEY,
                    Username TEXT NOT NULL UNIQUE,
                    Password TEXT NOT NULL,
                    Role TEXT NOT NULL,
                    ReferenceId INTEGER DEFAULT 0,
                    CreatedDate DATETIME NOT NULL,
                    ModifiedDate DATETIME NOT NULL,
                    LastLoginDate DATETIME,
                    IsActive INTEGER DEFAULT 1
                );

                CREATE TABLE IF NOT EXISTS Staff (
                    Id TEXT PRIMARY KEY,
                    Name TEXT NOT NULL,
                    Address TEXT NOT NULL,
                    Email TEXT NOT NULL,
                    ReferenceId INTEGER DEFAULT 0,
                    UserId TEXT,
                    CreatedDate DATETIME NOT NULL,
                    ModifiedDate DATETIME NOT NULL,
                    FOREIGN KEY (UserId) REFERENCES Users(Id)
                );

                CREATE TABLE IF NOT EXISTS Sections (
                    Id TEXT PRIMARY KEY,
                    Name TEXT NOT NULL,
                    ReferenceId INTEGER DEFAULT 0,
                    CreatedDate DATETIME NOT NULL,
                    ModifiedDate DATETIME NOT NULL
                );
```

Best Code 3: Service Layer with Transaction Management

File: Controllers/Services/AttendanceService.cs *Description*: Demonstrates proper service layer implementation with transaction management and complex query building.



```
2 references
public class AttendanceService
{
    1 reference
    public async Task AddAttendanceAsync(Attendance attendance)
    {
        using (var conn = DbCon.GetConnection())
        {
            using (var transaction = conn.BeginTransaction())
            {
                using (var cmd = new SQLiteCommand(conn))
                {
                    cmd.CommandText = @"
                        INSERT INTO Attendance (StudentId, SubjectId, Date, Status)
                        VALUES (@StudentId, @SubjectId, @Date, @Status)";

                    cmd.Parameters.AddWithValue("@StudentId", attendance.StudentId);
                    cmd.Parameters.AddWithValue("@SubjectId", attendance.SubjectId);
                    cmd.Parameters.AddWithValue("@Date", attendance.Date.ToString("yyyy-MM-dd"));
                    cmd.Parameters.AddWithValue("@Status", attendance.Status);

                    await cmd.ExecuteNonQueryAsync();
                    transaction.Commit();
                }
            }
        }
    }
```

Best Code 4: Model Design with Encapsulation

File: Models/Section.cs *Description*: Shows proper model design with encapsulation, factory pattern, and immutable properties.

```
public class Section
{
    private static int _lastReferenceId = 0;

    8 references
    public Guid Id { get; private set; }
    16 references
    public string Name { get; private set; }
    6 references
    public int ReferenceId { get; private set; }
    8 references
    public DateTime CreatedDate { get; private set; }
    7 references
    public DateTime ModifiedDate { get; private set; }

    1 reference
    private Section(Guid id, string name)
    {
        Id = id;
        Name = name;
        CreatedDate = DateTime.UtcNow;
        ModifiedDate = DateTime.UtcNow;
        ReferenceId = ++_lastReferenceId;
    }

    2 references
    public Section(Guid id, string name, DateTime createdDate, DateTime modifiedDate)
    {
        Id = id;
        Name = name;
        CreatedDate = createdDate;
        ModifiedDate = modifiedDate;
    }

    1 reference
    public static Section CreateSection(string name)
    {
        return new Section(Guid.NewGuid(), name);
    }
```

Best Code 5: Role-Based Access Control

File: Views/MainForm.cs *Description*: Demonstrates dynamic UI control based on user roles with clean separation of concerns.

```
private void ApplyRoleAccess()
{
    flowSidebar.Controls.Clear();

    flowSidebar.Controls.Add(lblWelcome);

    if (userRole == "Admin")
    {
        flowSidebar.Controls.Add(button1); // Student
        flowSidebar.Controls.Add(button2); // Lectures
        flowSidebar.Controls.Add(button3); // Section
        flowSidebar.Controls.Add(button4); // Subject
        flowSidebar.Controls.Add(button5); // Staff
        flowSidebar.Controls.Add(button6); // Timetable
        flowSidebar.Controls.Add(button8); // Exam
        flowSidebar.Controls.Add(button7); // Marks
        flowSidebar.Controls.Add(button9); // Room
        flowSidebar.Controls.Add(btnResetPassword); // Reset
        flowSidebar.Controls.Add(button10); // Attendance
    }
    else if (userRole.ToLower() == "staff")
    {
        flowSidebar.Controls.Add(button6); // Timetable
        flowSidebar.Controls.Add(button7); // Marks
        flowSidebar.Controls.Add(button8); // Exam
        flowSidebar.Controls.Add(btnResetPassword); // Reset
        flowSidebar.Controls.Add(button10); // Attendance
    }
    else if (userRole.ToLower() == "lecture")
    {
        flowSidebar.Controls.Add(button6); // Timetable
        flowSidebar.Controls.Add(button7); // Marks
        flowSidebar.Controls.Add(btnResetPassword); // Reset
        flowSidebar.Controls.Add(button10); // Attendance
    }

    flowSidebar.Controls.Add(btnlogout);
}
```

Best Code 6: Controller Pattern with Clean API

File: Controllers/ControllersTic/MarkController.cs *Description*: Shows clean controller implementation with async operations and proper separation of concerns.

```
public class MarkController
{
    private MarkRepository repository = new MarkRepository();

    1 reference
    public async Task<DataTable> GetAllMarksAsync() => await repository.GetAllMarksAsync();
    1 reference
    public async Task AddMarkAsync(int studentId, string subject, string exam, int score) => await repository.AddMarkAsync(studentId, subject, exam, score);
    1 reference
    public async Task DeleteMarkAsync(int markId) => await repository.DeleteMarkAsync(markId);
    1 reference
    public async Task UpdateMarkAsync(int markId, int studentId, string subject, string exam, int score) => await repository.UpdateMarkAsync(markId, studentId, subject, exam, score);
    1 reference
    public async Task<string> GetStudentNameAsync(int studentId) => await repository.GetStudentNameAsync(studentId);
    1 reference
    public async Task<DataTable> GetSubjectsByStudentAsync(int studentId) => await repository.GetSubjectsByStudentAsync(studentId);
    1 reference
    public async Task<DataTable> GetExamsAsync() => await repository.GetAllExamsAsync();
}
```

Installation Note (SQLite Package)

To install the SQLite package in your C# project using Visual Studio:

Go to your project in Visual Studio → Click on Tools → NuGet Package Manager → Package Manager Console

Then paste the following command:

Install-Package System.Data.SQLite

This installs the SQLite package required to handle database operations.

User Role-Based Functionalities

Admin Flow:

The Admin has full control over the system. They can:

Create and manage Students, Lectures, Sections, Subjects, Staff, Timetables, Exams, and Rooms.

View all attendance records.

Edit or delete any attendance entry.

Student Flow:

Students have a limited view. They can:

Log in to view their personal dashboard.

Check their attendance records filtered by date or subject.

Staff Flow:

Staff members can:

View the timetable.

Manage exams and mark attendance.

Update or delete existing attendance records for assigned students.

Lecture Flow:

Lecturers can:

View the timetable.

Mark attendance for their subjects by selecting a date and student.

View or update previously marked attendance.

Summary

This project demonstrates strong software engineering principles including:

- *Clean Architecture* with proper separation of concerns
- *Repository Pattern* for data access abstraction
- *Service Layer* for business logic encapsulation
- *Async/Await* patterns for responsive UI
- *Proper Error Handling* and validation
- *Role-Based Security* implementation
- *Database Design* with normalization and constraints
- *Code Reusability* through base classes and inheritance