CRICKET MANAGEMENT SYSTEM

Session 2024 - 2028

Submitted by:

Ali Hassan Siddiqui 2024-CS-5

Submitted To:

Prof. Dr. Muhammad Awais Hassan

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Cricket Management System

Project Documentation

---

1. Introduction

The *Cricket Management System* is a structured desktop application designed to efficiently manage cricket players, their performance statistics, and user roles. The system is based on a role-based access control model with two user types: *Manager* and *Viewer*. It enables data entry, statistics tracking, and performance evaluation for different player types: *Batsman*, *Bowler*, and *AllRounder*.

Objectives:

- Implement a structured class hierarchy for player roles.
- Design a three-layer architecture: UI, BL, and DL.
- Secure user access through role-based login.
- Provide detailed player information and performance analysis.

System Features by Role:

- *Manager*: Add, update, delete, search, and view player data.
- *Viewer*: Search and view player data and statistics.

---

2. OOP Concepts

The system uses all four major pillars of Object-Oriented Programming:

a. Inheritance

- *CricketPlayer* is an abstract base class.
- *Batsman*, *Bowler*, and *AllRounder* inherit from it.

```
public abstract class CricketPlayer
{
    // Base properties and methods
}

public class Batsman : CricketPlayer
{
    // Batsman-specific properties and method implementations
}

public class Bowler : CricketPlayer
{
    // Bowler-specific properties and method implementations
}

public class AllRounder : CricketPlayer
{
    // AllRounder-specific properties and method implementations
}
```

b. Polymorphism

- Method overriding for CalculatePerformanceIndex() and GetPlayerInfo() in derived classes.

```
// Abstract method in base class
public abstract double CalculatePerformanceIndex();

// Different implementations in derived classes
public override double CalculatePerformanceIndex()
{
    // Batsman-specific calculation
}

// Different implementation in Bowler class
public override double CalculatePerformanceIndex()
{
    // Bowler-specific calculation
}
```

c. Encapsulation

- Use of private fields and public properties to control access.

```
private int _centuries;  // Private field

public int GetCenturies() { return _centuries; }  // Getter
public void SetCenturies(int value) { _centuries = value; }  // Setter

// C# Property that encapsulates the field
public int Centuries
{
    get { return GetCenturies(); }
    set { SetCenturies(value); }
}
```

d. Abstraction

- Abstract classes and interfaces like IPlayerService define contracts.

```
// Abstract class defining a contract
public abstract class CricketPlayer
{
    // Common implementation
    public virtual string GetPlayerInfo()
    {
        // Base implementation
    }

    // Abstract method that derived classes must implement
    public abstract double CalculatePerformanceIndex();
}

// Interface defining a contract
public interface IPlayerService
{
    DataTable GetAllPlayers();
    DataTable SearchPlayersByName(string name);
    bool AddPlayer(CricketPlayer player);
    bool UpdatePlayer(CricketPlayer player);
    bool DeletePlayer(int id);
}
```

e. Association

- Layer communication via object references.

```
// Abstract class defining a contract
public abstract class CricketPlayer
{
    // Common implementation
    public virtual string GetPlayerInfo()
    {
        // Base implementation
    }

    // Abstract method that derived classes must implement
    public abstract double CalculatePerformanceIndex();
}

// Interface defining a contract
public interface IPlayerService
{
    DataTable GetAllPlayers();
    DataTable SearchPlayersByName(string name);
    bool AddPlayer(CricketPlayer player);
    bool UpdatePlayer(CricketPlayer player);
    bool DeletePlayer(int id);
}
```

Comparison with Procedural Programming:

- OOP promotes modularity, reusability, and easier maintenance.
- Procedural programming would require duplicating logic for each player type.

---

3. Design Pattern Implementation

This project uses a **Three-Layer Architecture**:

UI Layer (Presentation):

- Manages forms (e.g., login, dashboard)
- Handles user input and displays data

```
// Example from UI Layer (ManagerDashboard.cs)
private void btnAdd_Click(object sender, EventArgs e)
{
    if (string.IsNullOrEmpty(txtName.Text.Trim()))
    {
        MessageBox.Show("Please enter a player name", "Validation Error",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    CricketPlayer player = GetPlayerFromForm();

    if (_playerService.AddPlayer(player))
    {
        MessageBox.Show("Player added successfully", "Success",
                        MessageBoxButtons.OK, MessageBoxIcon.Information);
        LoadAllPlayers();
        ClearForm();
    }
    else
    {
        MessageBox.Show("Player Addition Failed", "Error",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

BL Layer (Business Logic):

- Contains core rules, services, and models

```
// Example from BL Layer (PlayerBL.cs)
public bool AddPlayer(CricketPlayer player)
{
    if (string.IsNullOrEmpty(player.Name) || player.Age <= 0)
    {
        return false;
    }

    int centuries = 0, halfCenturies = 0, fours = 0, sixes = 0,
        ballsBowled = 0, fiveWicketHauls = 0;

    if (player is Batsman batsman)
    {
        centuries = batsman.Centuries;
        halfCenturies = batsman.HalfCenturies;
        fours = batsman.Fours;
        sixes = batsman.Sixes;
    }
    // Additional type checks...

    PlayerDL.AddPlayer(player.Name, player.Age, player.Role,
                       player.BattingStyle, player.BowlingStyle,
                       player.Matches, player.Runs, player.Wickets,
                       centuries, halfCenturies, fours, sixes,
                       ballsBowled, fiveWicketHauls);
    return true;
}
```

DL Layer (Data Access):

- Manages all database interactions

```
// Example from BL Layer (PlayerBL.cs)
public bool AddPlayer(CricketPlayer player)
{
    if (string.IsNullOrEmpty(player.Name) || player.Age <= 0)
    {
        return false;
    }

    int centuries = 0, halfCenturies = 0, fours = 0, sixes = 0,
        ballsBowled = 0, fiveWicketHauls = 0;

    if (player is Batsman batsman)
    {
        centuries = batsman.Centuries;
        halfCenturies = batsman.HalfCenturies;
        fours = batsman.Fours;
        sixes = batsman.Sixes;
    }
    // Additional type checks...

    PlayerDL.AddPlayer(player.Name, player.Age, player.Role,
                       player.BattingStyle, player.BowlingStyle,
                       player.Matches, player.Runs, player.Wickets,
                       centuries, halfCenturies, fours, sixes,
                       ballsBowled, fiveWicketHauls);
    return true;
}
```

Advantages:

- Separation of concerns
- Easier debugging and testing
- Clear modular boundaries for future updates

---

4. Class Details

Base Class: CricketPlayer

- Common properties: name, age, role, batting style, bowling style
- Abstract method: CalculatePerformanceIndex()

Derived Classes:

Batsman

- Additional fields: centuries, half-centuries, fours, sixes

- Performance index based on average and boundaries

Bowler

- Additional fields: balls bowled, five-wicket hauls
- Performance index based on wickets and bowling efficiency

AllRounder

- Combination of batting and bowling metrics
- Complex index calculation using both aspects

---

5. Conclusion

The Cricket Management System successfully demonstrates core OOP principles and multi-layer architecture. It effectively separates UI, business logic, and data handling, ensuring maintainability and scalability. Key achievements include:

- Clean and modular architecture
- Effective use of inheritance and polymorphism
- Role-based functionality and security
- Clear abstraction and encapsulation in class designs

Challenges Faced:

- Designing balanced performance index formulas for different player types
- Managing data flow between layers while keeping interfaces clean

Lessons Learned:

- Importance of abstraction and contracts (interfaces)
- Separation of concerns improves testability and reusability
- OOP enables clean, extendable designs that scale well with additional features

Class Relationships

Inheritance Relationships

- Batsman ← CricketPlayer (Batsman inherits from CricketPlayer)
- Bowler ← CricketPlayer (Bowler inherits from CricketPlayer)
- AllRounder ← CricketPlayer (AllRounder inherits from CricketPlayer)

Implementation Relationships

- PlayerBL ⟹ IPlayerService (PlayerBL implements IPlayerService)
- UserBL ⟹ IUserService (UserBL implements IUserService)

Association Relationships

- PlayerBL → PlayerDL (PlayerBL uses PlayerDL)
- UserBL → UserDL (UserBL uses UserDL)
- PlayerDL → SqlHelper (PlayerDL uses SqlHelper)
- UserDL → SqlHelper (UserDL uses SqlHelper)
- LoginForm → IUserService (LoginForm uses IUserService)
- SignupForm → IUserService (SignupForm uses IUserService)
- ManagerDashboard → IPlayerService (ManagerDashboard uses IPlayerService)
- ViewerDashboard → IPlayerService (ViewerDashboard uses IPlayerService)