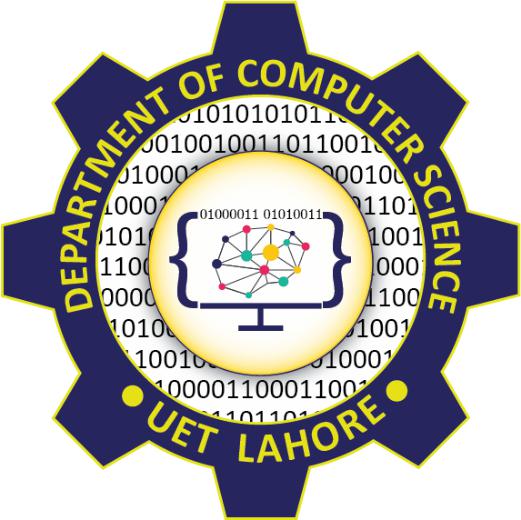
**CRICKET MANAGEMENT SYSTEM**



**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**

Contents

1. [Project Documentation 3](#_Toc197948467)

[2. OOP Concepts 3](#_Toc197948468)

[3. Design Pattern Implementation 6](#_Toc197948469)

[BL Layer (Business Logic): 7](#_Toc197948470)

[DL Layer (Data Access): 8](#_Toc197948471)

[Advantages: 8](#_Toc197948472)

[4. Class Details 8](#_Toc197948473)

[5. Conclusion 9](#_Toc197948474)

[Class Relationships 10](#_Toc197948475)

[Inheritance Relationships 10](#_Toc197948476)

[Implementation Relationships 10](#_Toc197948477)

[Association Relationships 10](#_Toc197948478)

**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.

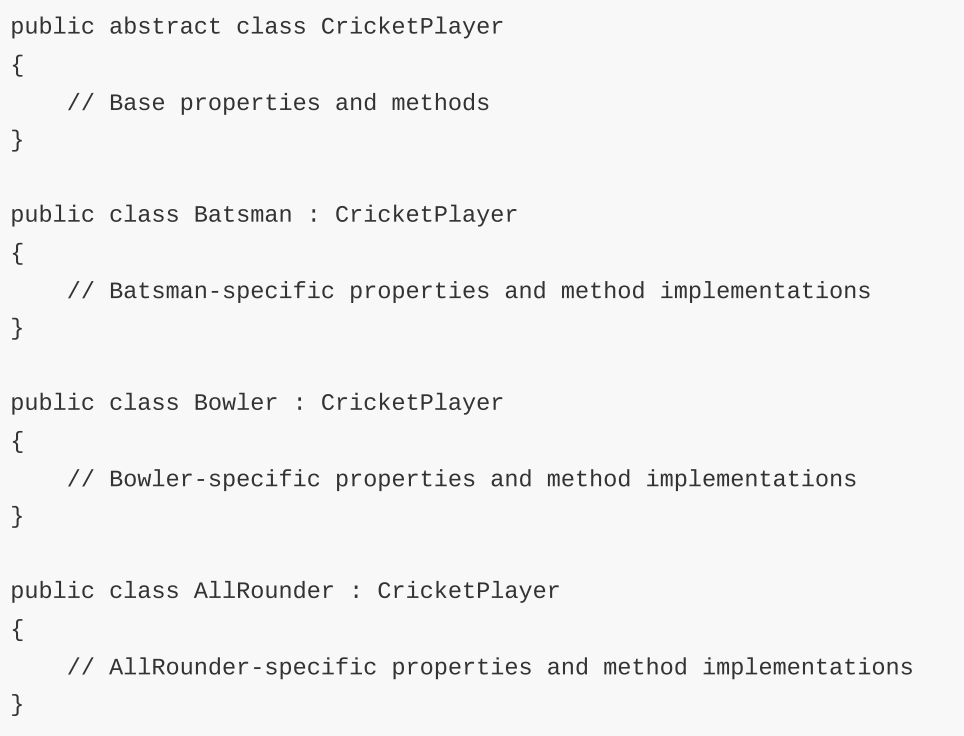
**System Features by Role:**

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

# 2. OOP Concepts

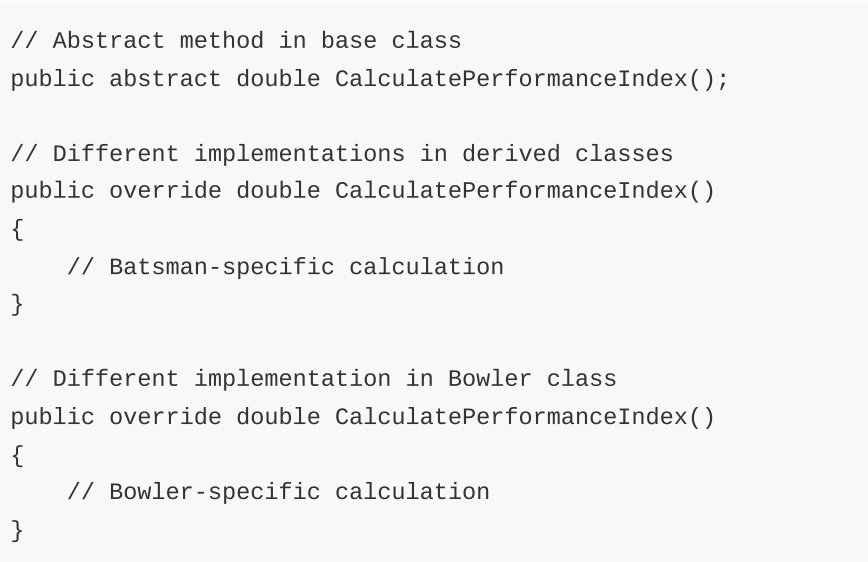
The system uses all four major pillars of Object-Oriented Programming:

**a. Inheritance**

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

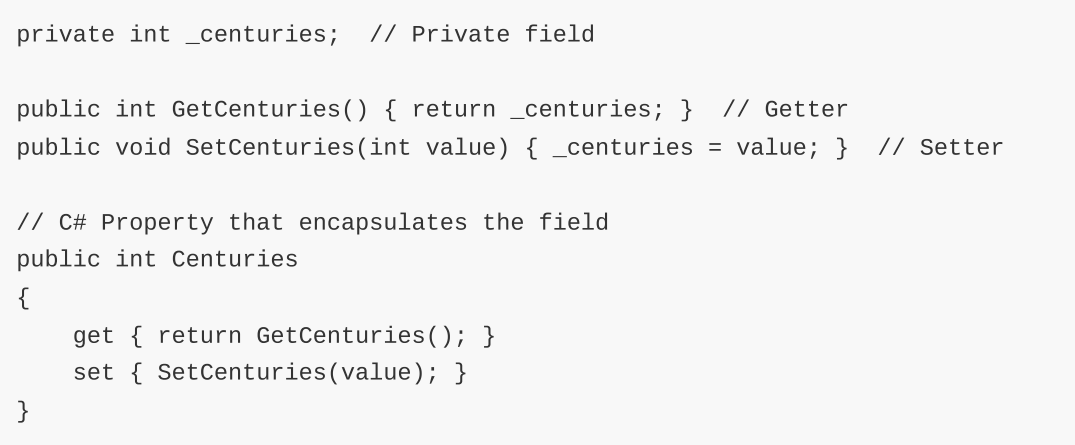
**b. Polymorphism**

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

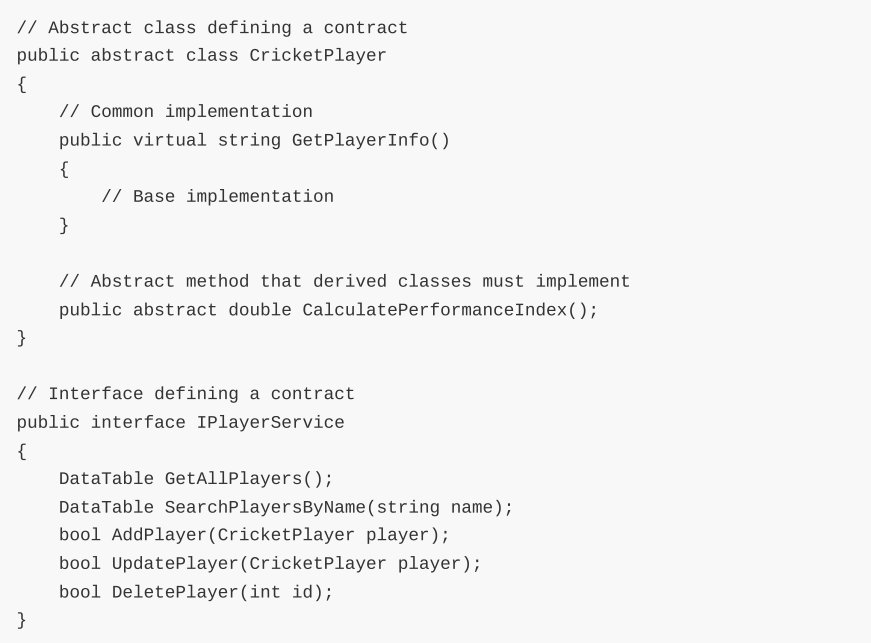


**c. Encapsulation**

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

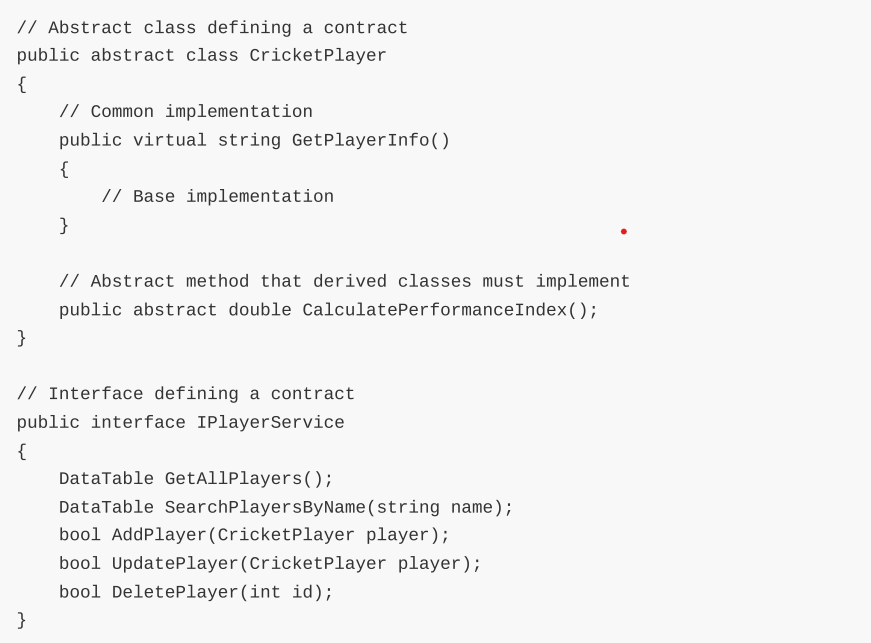
**d. Abstraction**

* Abstract classes and interfaces like IPlayerService define contracts.



**e. Association**

* Layer communication via object references.

*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**:

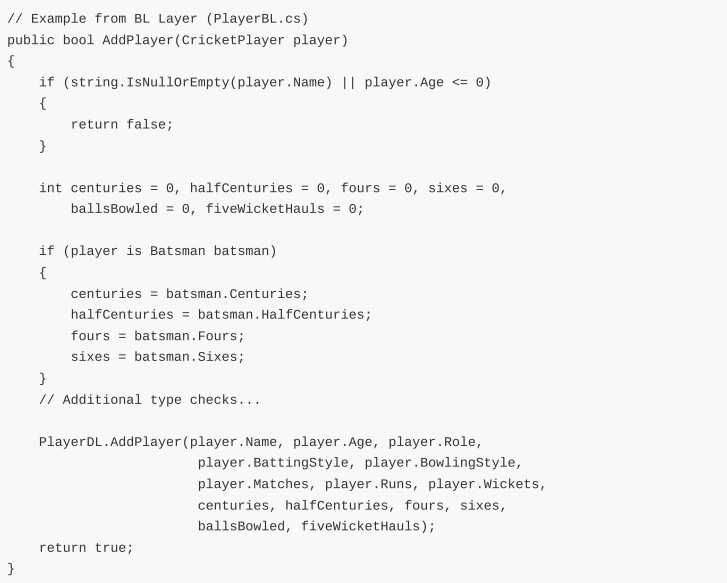
**UI Layer (Presentation):**

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



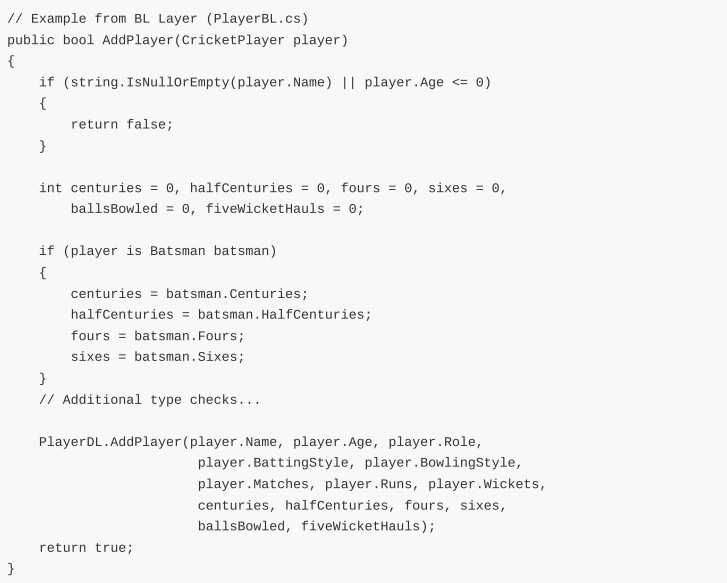
## BL Layer (Business Logic):

* Contains core rules, services, and models



## DL Layer (Data Access):

* Manages all database interactions



## 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)