

**TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM  
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG  
KHOA CÔNG NGHỆ THÔNG TIN**



**HỒ NGUYỄN MINH KHOA - 52300038  
PHẠM TIẾN LỰC - 52300042  
TRẦN KHẢI TÂN - 52200119**

**ÁP DỤNG DESIGN PATTERN CHO  
ỨNG DỤNG XẾP LỊCH THI ĐẤU GIẢI  
BÓNG ĐÁ**

**BÁO CÁO CUỐI KỲ  
MẪU THIẾT KẾ**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025**

**TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM  
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG  
KHOA CÔNG NGHỆ THÔNG TIN**



**HỒ NGUYỄN MINH KHOA - 52300038  
PHẠM TIẾN LỰC - 52300042  
TRẦN KHẢI TÂN - 52200119**

**ÁP DỤNG DESIGN PATTERN CHO  
ỨNG DỤNG XẾP LỊCH THI ĐẤU GIẢI  
BÓNG ĐÁ**

**BÁO CÁO CUỐI KỲ  
MẪU THIẾT KẾ**

Người hướng dẫn  
**ThS. Vũ Đình Hồng**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025**

## LỜI CẢM ƠN

Lời đầu tiên, chúng em xin bày tỏ lòng ơn đến Ban giám hiệu trường Đại Học Tôn Đức Thắng đã tạo điều kiện thuận lợi về cơ sở vật chất hiện đại, đặc biệt là hệ thống thư viện phong phú, đa dạng tài liệu học thuật, góp phần hỗ trợ đắc lực cho quá trình học tập và nghiên cứu của sinh viên.

Chúng em cũng xin chân thành cảm ơn quý giảng viên bộ môn, đặc biệt là thầy Vũ Đình Hồng, người đã tận tâm giảng dạy và truyền đạt những kiến thức quý báu, giúp chúng em từng bước tiếp cận và hiểu rõ hơn về môn học. Nhờ sự hướng dẫn nhiệt tình của thầy trong suốt thời gian qua, chúng em đã tích lũy được nhiều kiến thức bổ ích và thực tiễn – những hành trang quý giá cho con đường học tập và nghề nghiệp sau này.

Dù đã cố gắng hoàn thành bài báo cáo với tinh thần nghiêm túc và trách nhiệm, nhưng do còn hạn chế về kinh nghiệm và kiến thức, chắc chắn không thể tránh khỏi những thiếu sót nhất định. Chúng em rất mong nhận được sự góp ý, nhận xét và phê bình từ quý thầy cô để có thể hoàn thiện hơn trong những lần thực hiện sau.

Cuối cùng, chúng em xin kính chúc quý thầy cô dồi dào sức khỏe, luôn thành công trong sự nghiệp giảng dạy và công tác, tiếp tục truyền cảm hứng tri thức đến các thế hệ sinh viên.

Trân trọng.

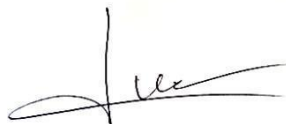
*TP. Hồ Chí Minh, ngày 30 tháng 04 năm 2025*

*Tác giả*

*(Ký tên và ghi rõ họ tên)*



*Trần Khải Tấn*



*Phạm Tiến Lực*



*Hồ Nguyễn Minh Khoa*

## CÔNG TRÌNH ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Chúng tôi xin cam đoan đây là công trình nghiên cứu của riêng chúng tôi và được sự hướng dẫn khoa học của ThS. Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong Dự án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

**Nếu phát hiện có bất kỳ sự gian lận nào, chúng tôi xin hoàn toàn chịu trách nhiệm về nội dung Dự án của mình.** Trường Đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do chúng tôi gây ra trong quá trình thực hiện (nếu có).

*TP. Hồ Chí Minh, ngày 30 tháng 04 năm 2025*

*Tác giả*

*(Ký tên và ghi rõ họ tên)*



*Trần Khải Tấn*



*Phạm Tiến Lực*



*Hồ Nguyễn Minh Khoa*

## TÓM TẮT

Đề tài tập trung nghiên cứu và xây dựng một hệ thống quản lý giải đấu bóng đá, bao gồm các chức năng chính như: quản lý giải đấu, quản lý đội bóng, sắp xếp lịch thi đấu (hình thức vòng tròn 2 lượt), quản lý trọng tài, phân công trọng tài và cập nhật kết quả trận đấu và quản lý bảng xếp hạng.

Để đảm bảo hệ thống đạt được các tiêu chí về tính linh hoạt, dễ mở rộng, dễ bảo trì và tuân thủ các nguyên lý thiết kế phần mềm hướng đối tượng, việc áp dụng các mẫu thiết kế (Design Pattern) trở nên cần thiết. Các mẫu thiết kế giúp chuẩn hóa cấu trúc phần mềm, tách biệt rõ ràng các trách nhiệm, giảm sự phụ thuộc giữa các thành phần và hỗ trợ khả năng mở rộng khi hệ thống phát triển thêm chức năng mới.

Bằng cách lựa chọn và áp dụng phù hợp các mẫu thiết kế trong từng tình huống cụ thể của hệ thống, đề tài hướng tới việc không chỉ hoàn thành yêu cầu chức năng mà còn xây dựng một nền tảng kiến trúc phần mềm ổn định, hiệu quả, dễ thích nghi với thay đổi, đáp ứng tốt như cầu thực tế và khả năng phát triển lâu dài.

## MỤC LỤC

<b>DANH MỤC HÌNH VẼ .....</b>	<b>4</b>
<b>CHƯƠNG 1. SINGLETON PATTERN CHO LỚP LOGGER.....</b>	<b>5</b>
1.1 Lý do áp dụng.....	5
1.2 Sơ đồ lớp: .....	6
1.3 Mã nguồn của Logger: .....	6
1.4 Ứng dụng thực tế trong hệ thống .....	7
1.5 Kết luận .....	8
<b>CHƯƠNG 2. STRATEGY PATTERN TRONG THUẬT TOÁN XẾP LỊCH.....</b>	<b>8</b>
2.1 Lý do áp dụng: .....	8
2.2 Sơ đồ lớp: .....	9
2.3 Code áp dụng: .....	10
<b>CHƯƠNG 3. COMMAND PATTERN ĐÓNG GÓI CÁC CHỨC NĂNG INSERT, UPDATE, DELETE VÀ UNDO .....</b>	<b>13</b>
3.1 Lý do áp dụng: .....	13
3.2 Sơ đồ lớp: .....	13
3.3 Code áp dụng: .....	14
<b>CHƯƠNG 4. TEMPLATE METHOD PATTERN CHUẨN HÓA CÁC THAO TÁC CỦA COMMAND (DÙNG CHUNG VỚI COMMAND PATTERN).....</b>	<b>21</b>
4.1 Lý do áp dụng: .....	21
4.2 Sơ đồ lớp: .....	21
4.3 Code áp dụng: .....	21
4.4 Kết luận: .....	23
<b>CHƯƠNG 5. FACTORY METHOD PATTERN TẠO CÁC COMMAND .....</b>	<b>23</b>

5.1 Lý do áp dụng: .....	23
5.2 Sơ đồ lớp: .....	24
5.3 Code áp dụng: .....	25
<b>CHƯƠNG 6. OBSERVER PATTERN - TỰ ĐỘNG GỬI THÔNG BÁO KHI CẬP NHẬT TRẬN ĐÁU .....</b>	<b>26</b>
6.1 Lý do áp dụng: .....	26
6.2 Sơ đồ lớp: .....	27
6.3 Code áp dụng: .....	27
<b>CHƯƠNG 7. DECORATOR PATTERN - MỞ RỘNG CHỨC NĂNG CỦA HỆ THỐNG .....</b>	<b>27</b>
7.1 Lý do áp dụng: .....	27
7.2 Sơ đồ lớp: .....	28
7.3 Code áp dụng: .....	28
<b>TÀI LIỆU THAM KHẢO .....</b>	<b>29</b>

## DANH MỤC HÌNH VẼ

Hình 1.1 : Sơ đồ lớp Logger .....	6
Hình 1.2 : Mã nguồn Logger áp dụng Singleton Pattern .....	7
Hình 1.3 : Đoạn code áp dụng Logger (Program.cs) .....	8
Hình 2.1 : Sơ đồ lớp Strategy Pattern .....	9
Hình 2.2 : Lớp MatchManger, nơi sử dụng Strategy .....	10
Hình 2.3 : Mã nguồn của ISchedulerStrategy .....	11
Hình 2.4 : Mã nguồn của RoundRobinScheduler .....	12
Hình 3.1 : Sơ đồ lớp Command Pattern .....	14
Hình 3.2 : Mã nguồn ICrudCommand .....	15
Hình 3.3 : Mã nguồn CrudCommandBase .....	17
Hình 3.4 : Mã nguồn TeamCommand .....	19
Hình 3.5 : Mã nguồn của CommandInvoker .....	19
Hình 3.6 : Lớp UcGridTools, nơi sử dụng Command .....	20
Hình 4.1 : Phần code áp dụng Template Method trong CrudCommandBase .....	23
Hình 5.1 : Sơ đồ lớp Factory Method (tối giản bớt các hàm và thuộc tính) .....	24
Hình 5.2 : Mã nguồn lớp CommandFactory .....	25
Hình 5.3 : Phần code UcGridTools dùng CommandFactory .....	25



# CHƯƠNG 1. SINGLETON PATTERN CHO LỚP LOGGER

## 1.1 Lý do áp dụng

Trong quá trình thiết kế và phát triển hệ thống, việc ghi log đóng vai trò rất quan trọng để hỗ trợ theo dõi, kiểm soát và xử lý lỗi trong thời gian thực. Để đảm bảo việc ghi log được thực hiện hiệu quả, đồng bộ và nhất quán trên toàn bộ ứng dụng, nhóm đã lựa chọn áp dụng mẫu thiết kế **Singleton Pattern** cho lớp **Logger** với các lý do sau:

### ***Đảm bảo chỉ có một thể hiện Logger duy nhất:***

- Singleton đảm bảo rằng toàn bộ hệ thống chỉ sử dụng một đối tượng **Logger** duy nhất.
- Điều này giúp tránh việc nhiều đối tượng cùng ghi vào file log một cách không kiểm soát, gây xung đột hoặc trùng lặp nội dung.

### ***Quản lý ghi log một cách tập trung:***

- Mọi thao tác ghi log từ các thành phần khác nhau trong ứng dụng đều đi qua **Logger.Instance**.
- Nhờ đó, định dạng, cấu trúc và đường dẫn file log được thống nhất và dễ bảo trì.

### ***Tiết kiệm tài nguyên hệ thống:***

- Việc mở/ghi file log là một thao tác tốn tài nguyên I/O.
- Sử dụng **Singleton** giúp giảm thiểu việc khởi tạo lặp lại các thể hiện **Logger**, từ đó tiết kiệm tài nguyên và tăng hiệu suất.

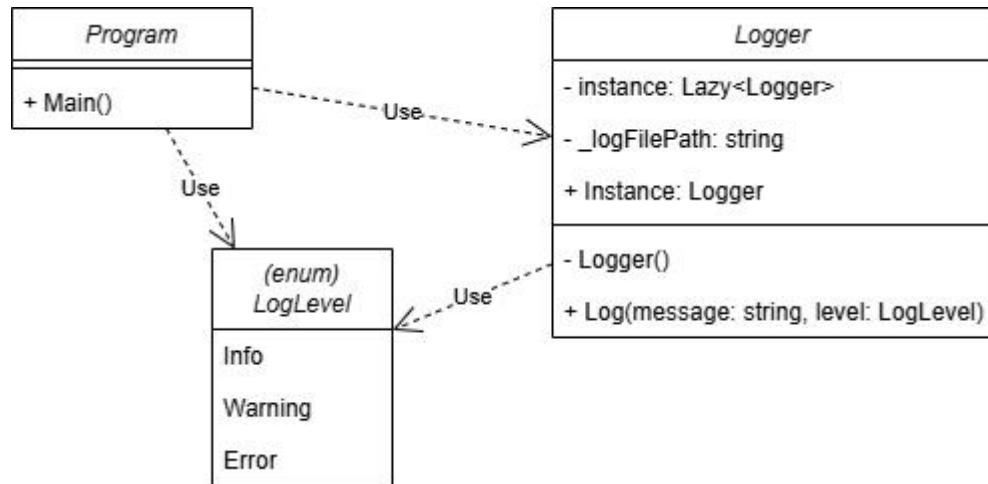
### ***An toàn trong môi trường đa luồng:***

- Lớp **Logger** sử dụng **Lazy<T>** để đảm bảo rằng thể hiện duy nhất được khởi tạo theo cách **thread-safe** -- tức là không bị lỗi khi nhiều luồng truy cập đồng thời.

### ***Dễ dàng mở rộng và tích hợp:***

- Khi hệ thống phát triển, có thể mở rộng **Logger** để ghi log vào nhiều nơi như: cơ sở dữ liệu, dịch vụ giám sát, hoặc cloud log
- Việc thay đổi chỉ cần thực hiện tại một nơi duy nhất, không ảnh hưởng đến các phần còn lại của ứng dụng.

## 1.2 Sơ đồ lớp:



Hình 1.1: Sơ đồ lớp Logger

## 1.3 Mã nguồn của Logger:

```

using System;
using System.IO;

namespace CORE
{
    public sealed class Logger
    {
        private static readonly Lazy<Logger> _instance = new Lazy<Logger>(() => new Logger());

        private readonly string _logFilePath;

        public static Logger Instance => _instance.Value;

        private Logger()
        {
            var logDirectory = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Logs");
            Directory.CreateDirectory(logDirectory);

            _logFilePath = Path.Combine(logDirectory, $"log_{DateTime.Now:yyyyMMdd}.txt");
        }

        public void Log(string message, LogLevel level = LogLevel.Info)
        {
            var logMessage = $"[{DateTime.Now:yyyy-MM-dd HH:mm:ss}] [{level}] {message}";

            try
            {
                File.AppendAllText(_logFilePath, logMessage + Environment.NewLine);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"[Logger] Không thể ghi file log: {ex.Message}");
            }
        }

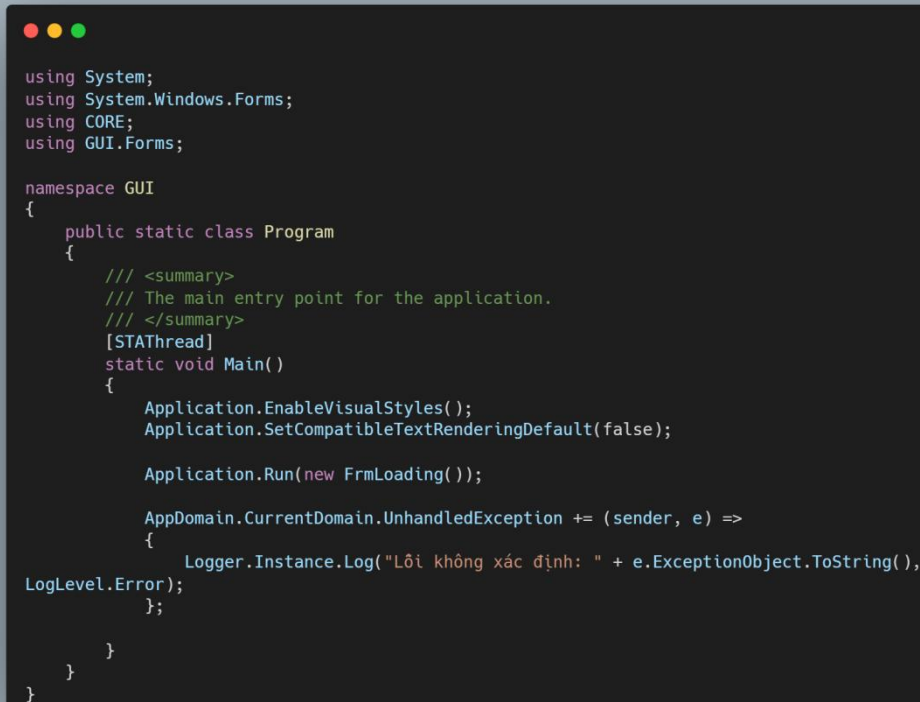
        public enum LogLevel
        {
            Info,
            Warning,
            Error
        }
    }
}

```

Hình 1.2: Mã nguồn Logger áp dụng Singleton Pattern

## 1.4 Ứng dụng thực tế trong hệ thống

Trong dự án, **Logger** được tích hợp tại **Program.cs** để ghi lại các lỗi không xác định thông qua **AppDomain.CurrentDomain.UnhandledException**. Ngoài ra, các phần khác trong hệ thống cũng có thể sử dụng **Logger.Instance.Log(...)** để ghi log hoạt động, cảnh báo hoặc lỗi.



```

using System;
using System.Windows.Forms;
using CORE;
using GUI.Forms;

namespace GUI
{
    public static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);

            Application.Run(new FrmLoading());

            AppDomain.CurrentDomain.UnhandledException += (sender, e) =>
            {
                Logger.Instance.Log("Lỗi không xác định: " + e.ExceptionObject.ToString(),
                LogLevel.Error);
            }
        }
    }
}

```

Hình 1.3: Đoạn code áp dụng Logger (Program.cs)

## 1.5 Kết luận

Việc áp dụng **Singleton Pattern** cho lớp **Logger** không chỉ giúp hệ thống có một cơ chế ghi log thống nhất, tiết kiệm tài nguyên, mà còn dễ bảo trì và mở rộng. Đây là một lựa chọn phù hợp trong bối cảnh ứng dụng desktop hoặc Windows Forms cần sự ổn định và hiệu quả khi ghi log.

## CHƯƠNG 2. STRATEGY PATTERN TRONG THUẬT TOÁN XẾP LỊCH

### 2.1 Lý do áp dụng:

Trong quá trình thiết kế hệ thống quản lý giải đấu, nhóm nhận thấy rằng việc tạo lịch thi đấu là một phần có thể phát sinh thay đổi tùy theo từng yêu cầu cụ thể của từng giải. Mặc dù ở giai đoạn hiện tại chỉ sử dụng một chiến lược tạo lịch (Round Robin), nhóm vẫn lựa chọn áp dụng mẫu thiết kế **Strategy Pattern** cho lớp **MatchManager** với các lý do sau:

### ***Chuẩn bị cho khả năng mở rộng trong tương lai:***

- Mỗi giải đấu có thể có thể thức thi đấu riêng, ví dụ: **vòng tròn một lượt, hai lượt (Round Robin)**; **loại trực tiếp (Knockout)**; hay **ghép cặp theo xếp hạng (Swiss System)**.
- Thay vì gắn chặt logic tạo lịch vào **MatchManager**, việc trừu tượng hóa thông qua interface **ISchedulerStrategy** cho phép dễ dàng bổ sung hoặc thay đổi chiến lược mà không cần sửa lại logic sẵn có.

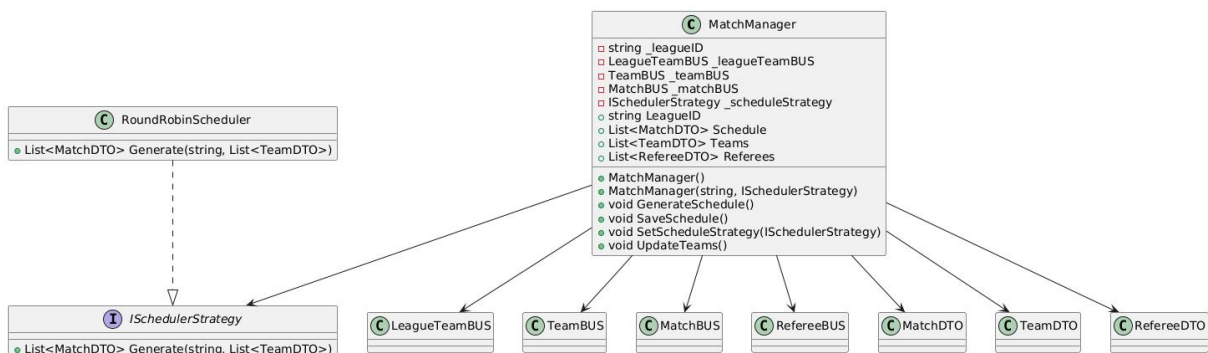
### ***Giảm sự phụ thuộc, tăng tính linh hoạt***

- **MatchManager** không cần biết cụ thể chiến lược tạo lịch nào đang được dùng, chỉ cần gọi qua interface **ISchedulerStrategy**.
- Điều này giúp giảm sự phụ thuộc giữa các lớp, tuân thủ nguyên tắc **Dependency Inversion** trong SOLID, đồng thời thuận lợi khi kiểm thử đơn vị (Unit test).

### ***Để bảo trì và quản lý***

- Việc tách riêng các chiến lược thành các lớp riêng biệt giúp dễ dàng sửa lỗi, điều chỉnh thuật toán mà không ảnh hưởng đến các chức năng khác.
- Nếu sau này cần thêm điều kiện đặc biệt cho từng kiểu lịch thi đấu (ví dụ: tránh trùng sân, giới hạn giờ thi đấu...), việc chỉnh sửa sẽ tập trung trong từng chiến lược cụ thể thay vì rải rác trong **MatchManager**.

## **2.2 Sơ đồ lớp:**



Hình 2.1: Sơ đồ lớp Strategy Pattern

## 2.3 Code áp dụng:

Mã nguồn của MatchManager (nơi sử dụng ISchedulerStrategy):

```
using System.Collections.Generic;
using BUS.Others;
using BUS.Services;
using BUS.Strategy;
using DTO;
using System;

namespace BUS.Managers
{
    public class MatchManager
    {
        private string _leagueID;
        private readonly LeagueTeamBUS _leagueTeamBUS = new LeagueTeamBUS();
        private readonly TeamBUS _teamBUS = new TeamBUS();
        private readonly MatchBUS _matchBUS = new MatchBUS();

        public string LeagueID
        {
            get => _leagueID;
            set
            {
                if (_leagueID != value)
                {
                    _leagueID = value;
                    UpdateTeams(); // Cập nhật lại đội bóng khi LeagueID thay đổi
                }
            }
        }

        public List<MatchDTO> Schedule { get; private set; }
        public List<TeamDTO> Teams { get; private set; } = new List<TeamDTO>();
        public List<RefereeDTO> Referees { get; private set; } = new List<RefereeDTO>();

        private ISchedulerStrategy _scheduleStrategy;

        public MatchManager() : this("", new RoundRobinScheduler()) { }

        public MatchManager(string leagueID, ISchedulerStrategy scheduleStrategy)
        {
            _scheduleStrategy = scheduleStrategy;
            LeagueID = leagueID; // Khởi tạo và gọi setter để cập nhật đội bóng ngay
            LoadReferees();
        }

        public void GenerateSchedule()
        {
            Schedule = _scheduleStrategy.Generate(LeagueID, Teams);

            // Load danh sách trọng tài
            Referees = (new RefereeBUS()).GetAll();
        }

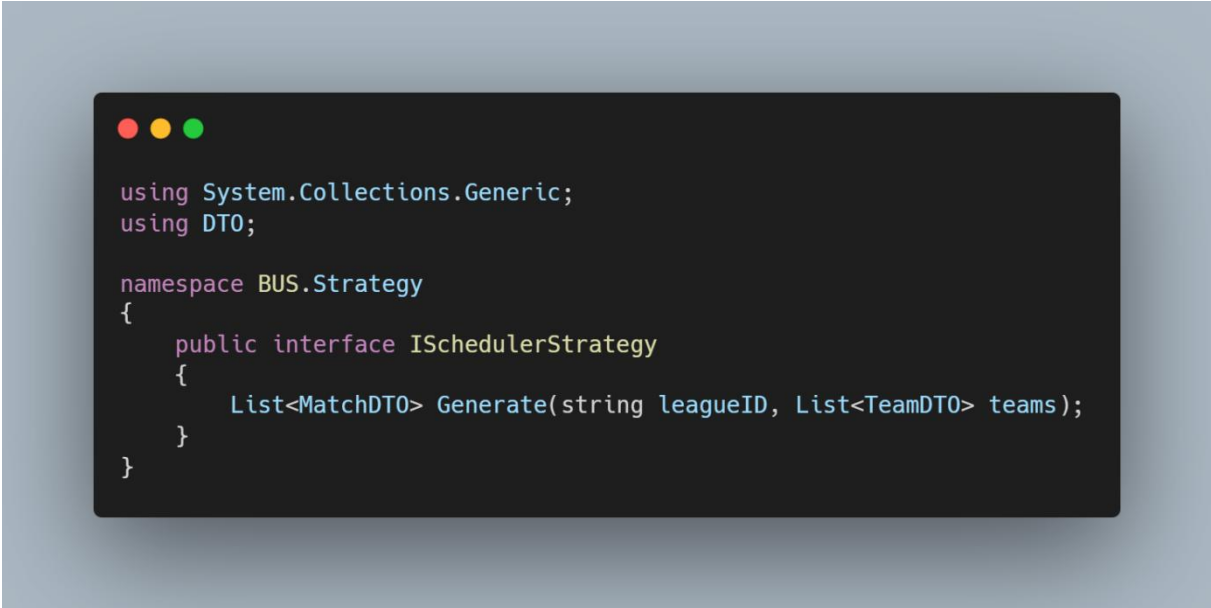
        public void SaveSchedule()
        {
            _matchBUS.InsertRange(Schedule);
        }

        public void SetScheduleStrategy(ISchedulerStrategy newStrategy) =>
            _scheduleStrategy = newStrategy;

        // Các hàm hỗ trợ khác thầy có thể xem thêm trong File Code (BUS.Managers.MatchManager.cs),
        // do code khác dài nên sẽ làm mờ hình khi xuất ra, mong thầy thông cảm ạ.
    }
}
```

Hình 2.2: Lớp MatchManger, nơi sử dụng Strategy

**Mã nguồn của ISchedulerStrategy (lớp Interface):**



```
using System.Collections.Generic;
using DTO;

namespace BUS.Strategy
{
    public interface ISchedulerStrategy
    {
        List<MatchDTO> Generate(string leagueID, List<TeamDTO> teams);
    }
}
```

Hình 2.3: Mã nguồn của ISchedulerStrategy

**Mã nguồn của RoundRobinScheduler (lớp Strategy cụ thể cho thuật toán xếp lịch vòng tròn):**

```

using System;
using System.Collections.Generic;
using System.Linq;
using DTO;

namespace BUS.Strategy
{
    public class RoundRobinScheduler : ISchedulerStrategy
    {
        private const int MatchDayInterval = 7; // Khoảng cách giữa các vòng đấu
        private const int DefaultStartHour = 8; // Giờ bắt đầu mặc định (8h sáng)

        public List<MatchDTO> Generate(string leagueID, List<TeamDTO> teams)
        {
            var schedule = new List<MatchDTO>();
            var workingTeams = new List<TeamDTO>(teams); // Sao chép danh sách để không ảnh hưởng danh
sách gốc

            if (workingTeams.Count % 2 != 0)
            {
                workingTeams.Add(null); // Thêm đội rỗng nếu số lượng đội lẻ
            }

            DateTime startDate = DateTime.Now.Date.AddDays(3).AddHours(DefaultStartHour);
            byte numRounds = (byte)(workingTeams.Count - 1);

            for (byte round = 1; round <= numRounds; round++)
            {
                for (int i = 0; i < workingTeams.Count / 2; i++)
                {
                    var home = workingTeams[i];
                    var away = workingTeams[workingTeams.Count - 1 - i];

                    if (home != null && away != null)
                    {
                        DateTime firstLegDate = startDate.AddDays(round * MatchDayInterval);
                        DateTime secondLegDate = firstLegDate.AddDays(numRounds * MatchDayInterval);

                        // Lượt đi
                        schedule.Add(new MatchDTO(leagueID, round, home.TeamID, away.TeamID,
firstLegDate, home.HomeStadiumID));
                        // Lượt về
                        schedule.Add(new MatchDTO(leagueID, (byte)(round + numRounds), away.TeamID,
home.TeamID, secondLegDate, away.HomeStadiumID));
                    }
                }

                // Xoay vòng đội
                var lastTeam = workingTeams[workingTeams.Count - 1];
                workingTeams.RemoveAt(workingTeams.Count - 1);
                workingTeams.Insert(1, lastTeam);
            }

            return schedule.OrderBy(m => m.RoundNumber)
                            .ThenBy(m => m.KickoffDateTime)
                            .ToList();
        }
    }
}

```

Hình 2.4: Mã nguồn của RoundRobinScheduler



## CHƯƠNG 3. COMMAND PATTERN ĐÓNG GÓI CÁC CHỨC NĂNG INSERT, UPDATE, DELETE VÀ UNDO

### 3.1 Lý do áp dụng:

Trong quá trình thiết kế và phát triển hệ thống, nhóm đã lựa chọn áp dụng mẫu thiết kế **Command Pattern** để xử lý các thao tác CRUD (Create, Read, Update, Delete) vì những lý do sau:

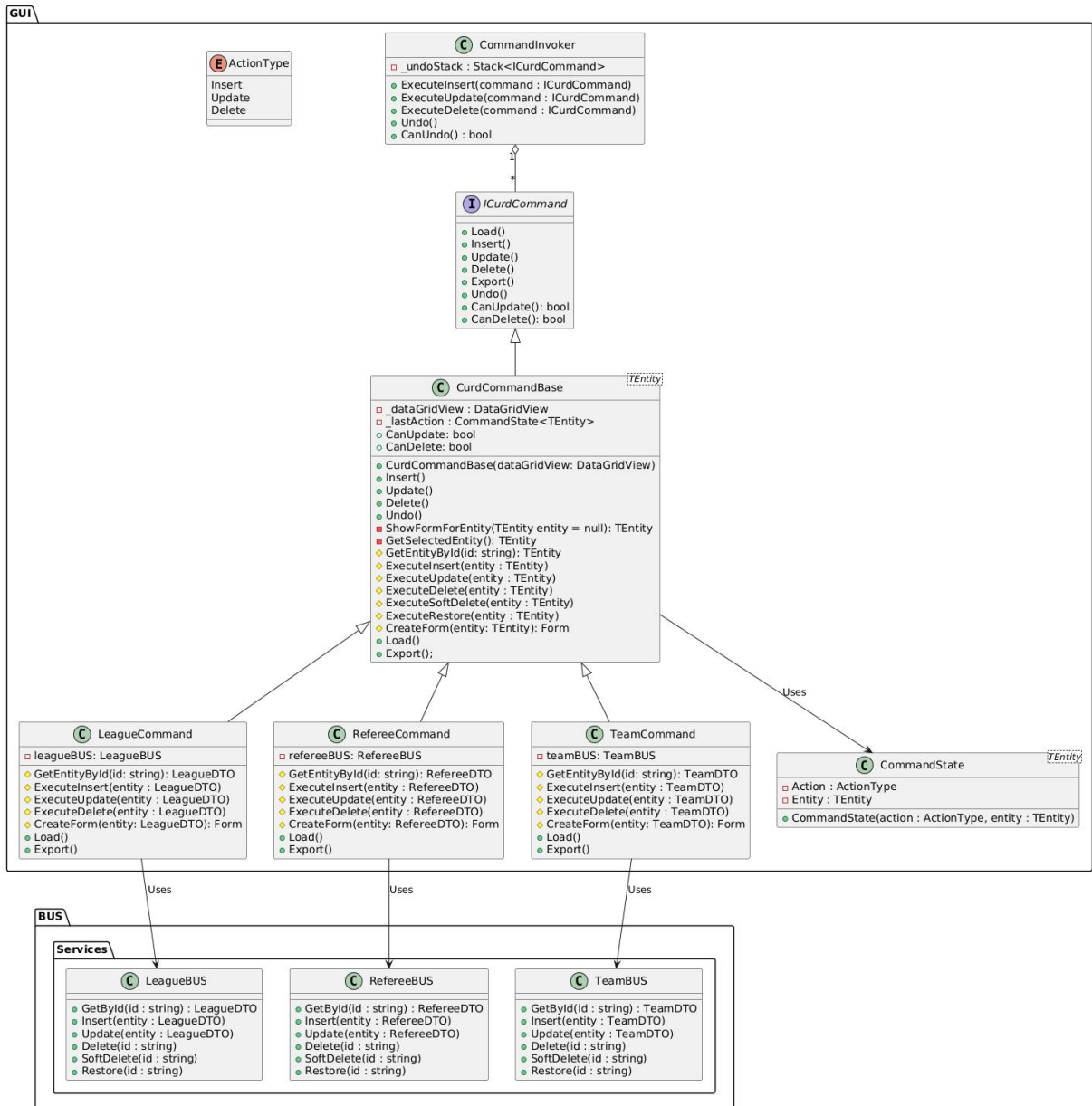
#### *Tách biệt giữa các thành phần:*

- Command Pattern giúp tách biệt logic thực hiện các thao tác CRUD khỏi giao diện người dùng và các lớp xử lý khác (BUS, DTO). Mỗi thao tác CRUD sẽ được đóng gói thành một hàm riêng biệt.

#### *Khả năng Undo dễ dàng:*

- Undo thao tác là một tính năng quan trọng trong việc quản lý và chỉnh sửa dữ liệu. Command Pattern giúp dễ dàng lưu lại các thao tác gần nhất và phục hồi chúng nếu cần, nhờ vào việc lưu trữ trạng thái của đối tượng trước khi thao tác (Undo).

### 3.2 Sơ đồ lớp:



Hình 3.1: Sơ đồ lớp Command Pattern

### 3.3 Code áp dụng:

Mã nguồn của ICurdCommand (Interface):



```
namespace GUI.Commands
{
    public interface ICurdCommand
    {
        void Load();
        void Insert();
        void Update();
        void Delete();
        void Export();
        void Undo();

        bool CanUpdate { get; }
        bool CanDelete { get; }
    }
}
```

Hình 3.2: Mã nguồn ICrudCommand

**Mã nguồn CurdCommandBase:**

```

using System;
using System.Windows.Forms;

namespace GUI.Commands
{
    /// <summary>
    /// Lớp cơ sở cho các thao tác thêm, sửa, xóa (CRUD), có thể hoàn tác thao tác cuối cùng, Load và
    Export.
    /// </summary>
    public abstract class CurdCommandBase<TEntity> : ICurdCommand where TEntity : class
    {
        protected readonly DataGridView _dataGridView;
        private CommandState<TEntity> _lastAction;

        // Xác định điều kiện cho phép thực hiện các thao tác
        public virtual bool CanUpdate => _dataGridView?.CurrentRow != null;
        public virtual bool CanDelete => _dataGridView?.CurrentRow != null;

        public CurdCommandBase(DataGridView dgv)
        {
            _dataGridView = dgv;
        }

        public void Insert()
        {
            var entity = ShowFormForEntity();
            if (entity != null)
            {
                ExecuteInsert(entity);
                _lastAction = new CommandState<TEntity>(ActionType.Insert, entity);
                Load();
            }
        }

        public void Update()
        {
            if (!CanUpdate) return;

            var original = GetSelectedEntity();
            if (original != null)
            {
                var updated = ShowFormForEntity(original);
                if (updated != null)
                {
                    _lastAction = new CommandState<TEntity>(ActionType.Update, original);
                    ExecuteUpdate(updated);
                    Load();
                }
            }
        }

        public void Delete()
        {
            if (!CanDelete) return;

            var entity = GetSelectedEntity();
            if (entity != null)
            {
                _lastAction = new CommandState<TEntity>(ActionType.Delete, entity);
                ExecuteSoftDelete(entity); // Sử dụng soft delete thay vì xóa vĩnh viễn
                Load();
            }
        }

        public void Undo()
        {
            if (_lastAction == null)
            {
                MessageBox.Show("Không có thao tác nào để hoàn tác.", "Thông báo",
                    MessageBoxButtons.OK, MessageBoxIcon.Information);
                return;
            }

            switch (_lastAction.Action)
            {
                case ActionType.Insert:
                    ExecuteSoftDelete(_lastAction.Entity); // Nếu vừa thêm => ấn đi
                    break;
                case ActionType.Update:
                    ExecuteUpdate(_lastAction.Entity); // Nếu vừa sửa => phục hồi bản gốc
                    break;
                case ActionType.Delete:
                    ExecuteRestore(_lastAction.Entity); // Nếu vừa xóa => khôi phục lại
                    break;
            }

            _lastAction = null;
            Load();
        }

        private TEntity ShowFormForEntity(TEntity entity = null)
        {
            using (var form = CreateForm(entity))
            {
                if (form.ShowDialog() == DialogResult.OK && form.Tag is TEntity resultEntity)
                {
                    return resultEntity;
                }
            }
            return null;
        }

        private TEntity GetSelectedEntity()
        {
            var row = _dataGridView?.CurrentRow;
            if (row == null)
            {
                throw new InvalidOperationException("Không có dòng nào được chọn.");
            }

            var id = row.Cells[0].Value?.ToString();
            return id != null ? GetEntityById(id) : null;
        }

        protected abstract TEntity GetEntityById(string id);
        protected abstract void ExecuteInsert(TEntity entity);
        protected abstract void ExecuteUpdate(TEntity entity);
        protected abstract void ExecuteDelete(TEntity entity);
        protected abstract void ExecuteSoftDelete(TEntity entity);
        protected abstract void ExecuteRestore(TEntity entity);

        protected abstract Form CreateForm(TEntity entity = null);
        public abstract void Load();
        public abstract void Export();

        private enum ActionType
        {
            Insert,
            Update,
            Delete
        }

        private class CommandState<T>
        {
            public ActionType Action { get; }
            public T Entity { get; }

            public CommandState(ActionType action, T entity)
            {
                Action = action;
                Entity = entity;
            }
        }
    }
}

```

Hình 3.3: Mã nguồn CrudCommandBase

**Mã nguồn của TeamCommand:**

```

using System.Collections.Generic;
using System;
using System.Windows.Forms;
using BUS.Services;
using DTO;
using GUI.Forms;
using GUI.Helpers;

namespace GUI.Commands
{
    public class TeamCommand : CurdCommandBase<TeamDTO>
    {
        private readonly TeamBUS _teamBUS = new TeamBUS();

        // Constructor nhận vào DataGridView
        public TeamCommand(DataGridView dgv)
            : base(dgv) { }

        #region CRUD Actions

        protected override TeamDTO GetEntityById(string id)
        {
            return _teamBUS.GetById(id);
        }

        protected override void ExecuteInsert(TeamDTO entity)
        {
            _teamBUS.Insert(entity);
        }

        protected override void ExecuteUpdate(TeamDTO entity)
        {
            _teamBUS.Update(entity);
        }

        protected override void ExecuteDelete(TeamDTO entity)
        {
            _teamBUS.Delete(entity.TeamID);
        }

        protected override void ExecuteSoftDelete(TeamDTO entity)
        {
            _teamBUS.SoftDelete(entity.TeamID);
        }

        protected override void ExecuteRestore(TeamDTO entity)
        {
            _teamBUS.Restore(entity.TeamID);
        }

        protected override Form CreateForm(TeamDTO entity = null)
        {
            return new FrmTeamInfo(entity); // Trả về form chỉnh sửa hoặc tạo mới cho Team
        }

        #endregion

        #region Tải và xuất dữ liệu

        public override void Load()
        {
            _dataGridView.DataSource = null; // Xóa dữ liệu cũ
            _dataGridView.DataSource = _teamBUS.GetAll(); // Lấy lại dữ liệu mới
        }

        public override void Export()
        {
            var teams = _teamBUS.GetAll(); // Lấy danh sách đội bóng từ cơ sở dữ liệu

            // Cung cấp tiêu đề và các thông tin cột cần xuất ra PDF
            PdfExportHelper.ExportToPdf(
                teams, // Dữ liệu cần xuất ra
                "DANH SÁCH ĐỘI BÓNG", // Tiêu đề
                new List<string> { "Mã Đội Bóng", "Tên Đội Bóng", "Đại diện", "Số Điện Thoại", "Email" }, // Các tiêu đề cột
                new List<Func<TeamDTO, string>> // Các extractor để lấy dữ liệu từ đối tượng TeamDTO
                {
                    team => team.TeamID,
                    team => team.TeamName,
                    team => team.CoachName,
                    team => team.Phone,
                    team => team.Email
                }
            );
        }

        #endregion
    }
}

```

Hình 3.4: Mã nguồn TeamCommand

Các mã nguồn của LeagueCommand và RefereeCommand tương tự như TeamCommand.

Mã nguồn của CommandInvoker:

```
using System.Collections.Generic;
using GUI.Commands;

namespace GUI
{
    public class CommandInvoker
    {
        // Danh sách các lệnh đã thực thi
        private readonly Stack<ICurdCommand> _undoStack = new Stack<ICurdCommand>
        ();

        // Thực thi thao tác Insert
        public void ExecuteInsert(ICurdCommand command)
        {
            command.Insert();
            _undoStack.Push(command); // Lưu lại lệnh để hoàn tác sau này
        }

        // Thực thi thao tác Update
        public void ExecuteUpdate(ICurdCommand command)
        {
            command.Update();
            _undoStack.Push(command); // Lưu lại lệnh để hoàn tác sau này
        }

        // Thực thi thao tác Delete
        public void ExecuteDelete(ICurdCommand command)
        {
            command.Delete();
            _undoStack.Push(command); // Lưu lại lệnh để hoàn tác sau này
        }

        // Hoàn tác thao tác gần nhất
        public void Undo()
        {
            if (_undoStack.Count > 0)
            {
                var lastCommand = _undoStack.Pop();
                lastCommand.Undo(); // Hoàn tác thao tác cuối cùng
            }
        }

        public bool CanUndo() => _undoStack.Count > 0;
    }
}
```

Hình 3.5: Mã nguồn của CommandInvoker

### Client (nơi gọi và sử dụng các Command):

```

using System;
using System.Windows.Forms;
using GUI.Commands;

namespace GUI.UserControls
{
    public partial class UcGridTools : UserControl
    {
        private readonly CommandInvoker _invoker = new CommandInvoker();
        private readonly ICurdCommand _command;

        public UcGridTools(string type)
        {
            InitializeComponent();
            _command = CommandFactory.CreateCommand(type, dgv);
            Load += (s, e) => _command.Load(); // Tải dữ liệu khi control được load
            dgv.CellClick += (s, e) => UpdateButtonState(); // Cập nhật nút khi chọn dòng khác
            UpdateButtonState();
        }

        private void UpdateButtonState()
        {
            btnEdit.Enabled = _command.CanUpdate;
            btnDelete.Enabled = _command.CanDelete;
            btnUndo.Enabled = _invoker.CanUndo();
        }

        private void btnExport_Click(object sender, EventArgs e) => _command.Export();

        private void btnInsert_Click(object sender, EventArgs e)
        {
            _invoker.ExecuteInsert(_command);
            UpdateButtonState();
        }

        private void btnEdit_Click(object sender, EventArgs e)
        {
            _invoker.ExecuteUpdate(_command);
            UpdateButtonState();
        }

        private void btnDelete_Click(object sender, EventArgs e)
        {
            _invoker.ExecuteDelete(_command);
            UpdateButtonState();
        }

        private void btnUndo_Click(object sender, EventArgs e)
        {
            _invoker.Undo();
            UpdateButtonState();
        }
    }
}

```

Hình 3.6: Lớp UcGridTools, nơi sử dụng Command



## CHƯƠNG 4. TEMPLATE METHOD PATTERN CHUẨN HÓA CÁC THAO TÁC CỦA COMMAND (DÙNG CHUNG VỚI COMMAND PATTERN)

### 4.1 Lý do áp dụng:

Trong quá trình thiết kế và phát triển hệ thống, nhóm đã lựa chọn áp dụng **Template Method Pattern** kết hợp với **Command Pattern** để chuẩn hóa và tối ưu các thao tác CRUD (Create, Read, Update, Delete). Dưới đây là các lý do khi áp dụng **Template Method Pattern** cùng với **Command Pattern** trong hệ thống:

#### *Tính nhất quán trong quy trình thao tác*

- **Template Method Pattern** giúp chuẩn hóa các bước thực hiện thao tác CRUD. Các thao tác như thêm, sửa, xóa dữ liệu sẽ tuân theo một quy trình chuẩn đã được định nghĩa trong lớp cơ sở (**CrudCommandBase**).
- Quy trình cụ thể, chẳng hạn như xác định đối tượng nào được chọn trong DataGridView, mở form thông tin nếu thực hiện các thao tác thêm, sửa (các thao tác này sẽ được các lớp con cụ thể thực hiện), load lại DataGridView sau khi thực hiện thao tác, và giữ lại đối tượng trước khi thao tác để phục vụ chức năng Undo.

#### *Dễ dàng bảo trì và mở rộng*

- Quy trình chung được định nghĩa ở lớp cơ sở, chỉ cần mở rộng các thao tác cụ thể trong các lớp con mà không cần thay đổi logic chung.

#### *Giảm trùng lặp mã nguồn:*

- Các bước chung (như hiển thị form, load lại dữ liệu) được tái sử dụng, giảm sự trùng lặp mã nguồn.

### 4.2 Sơ đồ lớp:

// Dùng chung sơ đồ lớp với Command Pattern

### 4.3 Code áp dụng:

**Phần code áp dụng Template Method trong CurdCommandBase:**

```

using System;
using System.Windows.Forms;

namespace GUI.Commands
{
    public abstract class CurdCommandBase<TEntity> : ICurdCommand where TEntity : class
    {
        #region Thao tác CRUD - Template Method

        public void Insert()
        {
            var entity = ShowFormForEntity();
            if (entity != null)
            {
                ExecuteInsert(entity);
                SaveAction(ActionType.Insert, entity);
                Load();
            }
        }

        public void Update()
        {
            if (!CanUpdate) return;

            var original = GetSelectedEntity();
            if (original != null)
            {
                var updated = ShowFormForEntity(original);
                if (updated != null)
                {
                    SaveAction(ActionType.Update, original);
                    ExecuteUpdate(updated);
                    Load();
                }
            }
        }

        public void Delete()
        {
            if (!CanDelete) return;

            var entity = GetSelectedEntity();
            if (entity != null)
            {
                SaveAction(ActionType.Delete, entity);
                ExecuteSoftDelete(entity);
                Load();
            }
        }

        private TEntity ShowFormForEntity(TEntity entity = null)
        {
            using (var form = CreateForm(entity))
            {
                if (form.ShowDialog() == DialogResult.OK && form.Tag is TEntity resultEntity)
                    return resultEntity;
            }
            return null;
        }

        private void SaveAction(ActionType actionType, TEntity entity)
        {
            _lastAction = new CommandState<TEntity>(actionType, entity);
        }

        protected abstract TEntity GetEntityById(string id);
        protected abstract void ExecuteInsert(TEntity entity);
        protected abstract void ExecuteUpdate(TEntity entity);
        protected abstract void ExecuteSoftDelete(TEntity entity);
        protected abstract Form CreateForm(TEntity entity = null);
        protected abstract void Load();
    }
}

```

Hình 4.1: Phần code áp dụng Template Method trong CrudCommandBase

#### 4.4 Kết luận:

Template Method định nghĩa các thao tác chung như: **Insert()**, **Update()**, **Delete()**.

Trong đó các hành động như **ExecuteInsert(TEntity entity)**, **ExecuteUpdate(TEntity entity)**, **ExecuteSoftDelete(TEntity entity)**, và **Load()** sẽ được các lớp con như **LeagueCommand**, **TeamCommand** và **RefereeCommand** định nghĩa.

## CHƯƠNG 5. FACTORY METHOD PATTERN TẠO CÁC COMMAND

### 5.1 Lý do áp dụng:

Trong quá trình thiết kế và phát triển hệ thống, nhóm đã lựa chọn sử dụng **Factory Method Pattern** để giúp việc tạo và quản lý các đối tượng lệnh CRUD (Create, Read, Update, Delete) trở nên linh hoạt và dễ dàng mở rộng. Dưới đây là các lý do chi tiết tại sao **Factory Method** được áp dụng trong hệ thống:

#### *Quản lý tạo đối tượng lệnh một cách tập trung*

- **Factory Method** giúp tách biệt việc tạo đối tượng lệnh khỏi các phần khác trong hệ thống. Thay vì phải trực tiếp tạo đối tượng lệnh trong từng lớp hoặc phương thức xử lý, **Factory Method** tập trung việc tạo đối tượng ở một chỗ duy nhất thông qua **CommandFactory**. Điều này giúp cải thiện tính tổ chức của mã nguồn và dễ dàng theo dõi, bảo trì.
- Ví dụ, khi có một yêu cầu thao tác CURD cho một đối tượng cụ thể như “Team”, “League” hoặc “Referee”, thay vì phải trực tiếp khởi tạo đối tượng **TeamCommand**, **LeagueCommand** hay **RefereeCommand** ở nhiều nơi khác nhau, ta chỉ cần gọi phương thức **CreateCommand** của **CommandFactory** để nhận đối tượng lệnh phù hợp.

**ICurdCommand command = CommandFactory.CreateCommand(“Team”, dg);**

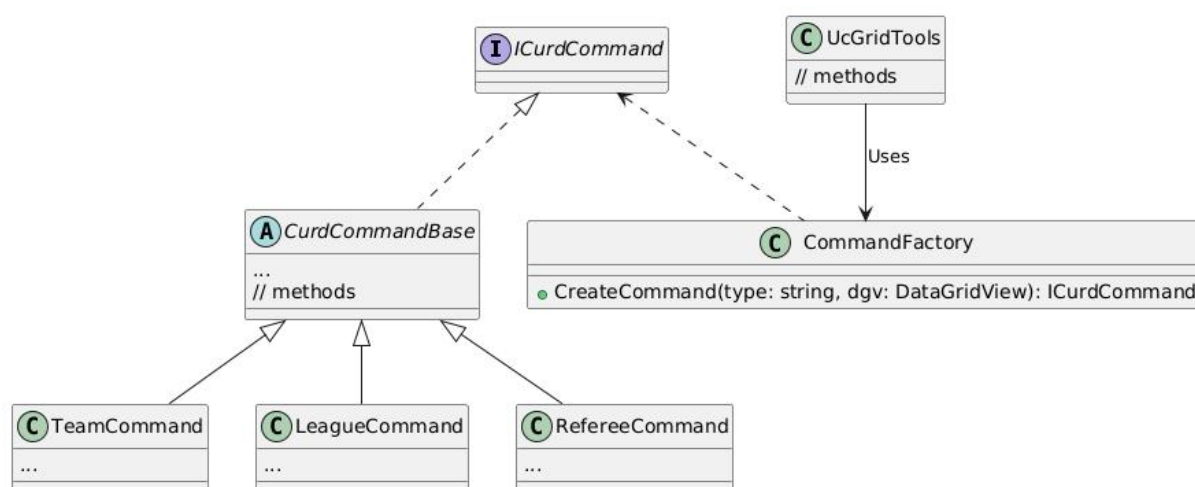
*Tăng tính mở rộng và linh hoạt*

- Khi áp dụng **Factory Method**, việc mở rộng hệ thống trở nên dễ dàng hơn rất nhiều. Nếu cần thêm một loại lệnh CRUD mới (ví dụ: thao tác với một đối tượng khác như “Coach”), chỉ cần thêm một lớp mới (**CoachCommand**) kế thừa từ **CrudCommandBase**. Sau đó, chỉ cần cập nhật **CommandFactory** để nhận diện và tạo đối tượng mới, mà không cần thay đổi những thành phần khác trong hệ thống.
- Cụ thể, ta chỉ cần thêm một case mới trong hàm **CreateCommand** của **CommandFactory** mà không cần phải thay đổi bất kỳ mã nguồn nào trong các lớp sử dụng lệnh CRUD.

#### *Ẩn giấu chi tiết khởi tạo khỏi client:*

- Khi sử dụng **Factory Method**, **client** (nơi gọi) không cần biết cụ thể lớp nào sẽ được tạo ra (ví dụ: **TeamCommand**, **LeagueCommand**, **RefereeCommand**) hay cách đối tượng đó được khởi tạo.
- Client chỉ cần truyền vào thông tin cần thiết (ví dụ như loại đối tượng), và nhận về một đối tượng thực hiện thao tác CRUD phù hợp.
- Điều này giúp giảm sự phụ thuộc giữa client và các lớp cụ thể, tăng tính linh hoạt và tuân thủ nguyên lý “**Lập trình hướng giao diện, không hướng cài đặt**” (**Program to interface, not implementation**).

## 5.2 Sơ đồ lớp:



Hình 5.1: Sơ đồ lớp Factory Method (tối giản bớt các hàm và thuộc tính)

### 5.3 Code áp dụng:

#### Mã nguồn lớp CommandFactory:

```
using System;
using System.Windows.Forms;

namespace GUI.Commands
{
    public static class CommandFactory
    {
        public static ICurdCommand CreateCommand(string type, DataGridView dgv)
        {
            switch (type)
            {
                case "Team":
                    return new TeamCommand(dgv);
                case "League":
                    return new LeagueCommand(dgv);
                case "Referee":
                    return new RefereeCommand(dgv);
                default:
                    throw new InvalidOperationException($"Chức năng '{type}' chưa được hỗ trợ :(");
            }
        }
    }
}
```

Hình 5.2: Mã nguồn lớp CommandFactory

#### Phần code dùng CommandFactory bên Client (UcGridTools)

```
using System;
using System.Windows.Forms;
using GUI.Commands;

namespace GUI.UserControls
{
    public partial class UcGridTools : UserControl
    {
        private readonly CommandInvoker _invoker = new CommandInvoker();
        private readonly ICurdCommand _command;

        public UcGridTools(string type)
        {
            InitializeComponent();

            // Dùng CommandFactory để khởi tạo Command tương ứng với type;
            _command = CommandFactory.CreateCommand(type, dgv);

            Load += (s, e) => _command.Load(); // Tải dữ liệu khi control được load
            dgv.CellClick += (s, e) => UpdateButtonState(); // Cập nhật nút khi chọn dòng khác
            UpdateButtonState();
        }
        // Các phương thức khác ...
    }
}
```

Hình 5.3: Phần code UcGridTools dùng CommandFactory

## CHƯƠNG 6. OBSERVER PATTERN - TỰ ĐỘNG GỬI THÔNG BÁO KHI CẬP NHẬT TRẬN ĐẤU

### 6.1 Lý do áp dụng:

Trong ứng dụng xếp lịch thi đấu giải bóng đá, việc tự động gửi thông báo khi có sự thay đổi về thông tin trận đấu (như thay đổi lịch thi đấu, địa điểm thi đấu hay kết quả trận đấu được cập nhật) là một yêu cầu quan trọng để đảm bảo rằng tất cả các bên liên quan (như đội bóng, trọng tài, huấn luyện viên, người hâm mộ) đều nhận được thông tin kịp thời và chính xác. Để đạt được điều này, **Observer Pattern** được lựa chọn với các lý do sau:

#### *Cập nhật tự động và đồng bộ hóa toàn bộ hệ thống:*

- Khi kết quả một trận đấu được cập nhật, các thành phần hệ thống như bảng xếp hạng, giao diện người dùng, và thông báo đều cần được tự động cập nhật mà không yêu cầu người dùng thực hiện bất kỳ thao tác nào.
- **Observer Pattern** cho phép hệ thống tự động cập nhật tất cả các phần liên quan (như bảng xếp hạng đội bóng, thông báo cho các bên liên quan) ngay khi kết quả trận đấu thay đổi mà không cần thao tác thủ công.

#### *Quản lý và đồng bộ các thay đổi trong hệ thống:*

- Hệ thống bóng đá có nhiều thành phần cần thay đổi khi kết quả trận đấu thay đổi: bảng xếp hạng, lịch thi đấu, thông báo cho trọng tài và đội bóng.
- **Observer Pattern** giúp tách biệt các thành phần này, khiến cho mỗi phần có thể chỉ quan tâm đến sự thay đổi kết quả mà không cần biết chi tiết cách thức xử lý, giúp dễ dàng duy trì và nâng cấp hệ thống.

#### *Tiết kiệm tài nguyên hệ thống:*

- Thay vì phải liên tục kiểm tra và cập nhật bảng xếp hạng hay gửi thông báo khi thông tin trận đấu được cập nhật, **Observer Pattern** chỉ thực hiện các hành động này khi có sự thay đổi thực tế (thông tin trận đấu thay đổi).

- Điều này giúp tiết kiệm tài nguyên hệ thống và tăng hiệu suất ứng dụng.

***Gửi thông báo và cập nhật bảng xếp hạng một cách linh hoạt:***

- Sau mỗi trận đấu, các bên liên quan (đội bóng, trọng tài, ban tổ chức...) cần nhận thông báo về kết quả trận đấu và sự thay đổi bảng xếp hạng.
- **Observer Pattern** cho phép gửi thông báo và cập nhật bảng xếp hạng đến các đối tượng liên quan một cách đồng bộ và tự động.

***Dễ dàng mở rộng và bảo trì hệ thống:***

- Nếu hệ thống cần thêm các kênh thông báo mới (ví dụ: email, SMS, thông báo qua ứng dụng di động), việc mở rộng sẽ rất dễ dàng mà không cần thay đổi các phần còn lại của hệ thống.
- Các Observer có thể được thêm vào mà không ảnh hưởng đến các chức năng chính của hệ thống, giúp cho việc bảo trì và mở rộng ứng dụng trở nên đơn giản hơn.

## 6.2 Sơ đồ lớp:

## 6.3 Code áp dụng:

# CHƯƠNG 7. DECORATOR PATTERN - MỞ RỘNG CHỨC NĂNG CỦA HỆ THỐNG

## 7.1 Lý do áp dụng:

Trong hệ thống xếp lịch thi đấu giải bóng đá, một số tính năng như gửi thông báo qua các kênh khác nhau (email, SMS, thông báo push), cập nhật trạng thái của trận đấu, hoặc thêm phần thưởng cho đội chiến thắng có thể được mở rộng mà không cần thay đổi các lớp hiện tại của hệ thống. Để giải quyết vấn đề này, Decorator Pattern được lựa chọn với các lý do sau:

***Mở rộng chức năng mà không thay đổi mã nguồn cũ:***

- **Decorator Pattern** cho phép bạn mở rộng các chức năng của các đối tượng hiện có mà không thay đổi mã nguồn của chúng. Điều này đặc biệt hữu ích khi cần thêm các tính năng mới mà không làm ảnh hưởng đến cấu trúc hệ thống.

***Giảm sự phụ thuộc giữa các đối tượng:***

- Thay vì thay đổi các đối tượng gốc để thêm tính năng mới, bạn có thể tạo ra các lớp Decorator mới và áp dụng chúng vào đối tượng mà không làm thay đổi các phần còn lại của hệ thống.

***Dễ dàng thay đổi và bảo trì:***

- Với Decorator Pattern, việc thay đổi hoặc mở rộng chức năng của các đối tượng sẽ trở nên đơn giản, giúp giảm bớt sự phức tạp và dễ dàng bảo trì hệ thống trong dài hạn.

**7.2 Sơ đồ lớp:**

**7.3 Code áp dụng:**



## **TÀI LIỆU THAM KHẢO**

Tiếng Việt

...

Tiếng Anh

Các slide bài giảng môn Mẫu Thiết Kế của trường Đại Học Tôn Đức Thắng.