****TRƯỜNG ĐẠI HỌC ĐÀ LẠT**

**KHOA CÔNG NGHỆ THÔNG TIN**

🙠🙟🕮🙝🙢



**Bài TIỂU LUẬN**

**Môn: Mẫu thiết kế**

**Giáo viên hướng dẫn : ThS. Đoàn Minh Khuê**

**Sinh viên thực hiện : Nguyễn Hữu Quang**

**MSSV : 2212445**

**Lớp: CTK46-PM**

***Đà Lạt: tháng 10 năm 2025***

**LỜI CẢM ƠN**

Trước hết, em xin gửi lời tri ân sâu sắc đến thầy Đoàn Minh Khuê vì đã tận tình hướng dẫn, định hướng chuyên môn và hỗ trợ em trong suốt quá trình thực hiện đề tài. Sự chỉ dẫn tận tâm và kiến thức quý báu của thầy là nền tảng giúp em củng cố vững chắc cả lý thuyết lẫn kỹ năng thực hành.

Thầy đã giúp em tiếp cận và nắm vững các kiến thức quan trọng về công nghệ phần mềm, kiến trúc hệ thống và các mẫu thiết kế phần mềm, tạo điều kiện để em triển khai đề tài một cách hiệu quả. Em cũng xin chân thành cảm ơn thầy vì những góp ý, hỗ trợ thiết thực trong giai đoạn phát triển và kiểm thử sản phẩm.

Đồng thời, em xin cảm ơn cộng đồng lập trình .NET cùng các nguồn tài liệu mở đã chia sẻ nhiều kiến thức hữu ích về ASP.NET Core, mô hình MVC, Dependency Injection và các mẫu thiết kế phần mềm. Những tài liệu này đã giúp em tiết kiệm thời gian nghiên cứu và ứng dụng hiệu quả vào dự án.

Dù đã nỗ lực hoàn thiện, bài báo cáo này chắc chắn vẫn còn những thiếu sót. Em rất mong nhận được những ý kiến đóng góp quý báu từ thầy để có thể hoàn thiện đề tài tốt hơn trong thời gian tới.

MỤC LỤC

[CHƯƠNG 1. Mở Đầu 1](#_Toc211871155)

[1.1. Giới thiệu hệ thống 1](#_Toc211871156)

[1.2. Lý do chọn đề tài 3](#_Toc211871157)

[1.3. Mục tiêu 4](#_Toc211871158)

[1.4. Phạm vi 5](#_Toc211871159)

[1.5. Công nghệ sử dụng 6](#_Toc211871160)

[CHƯƠNG 2. Cơ Sở Lý Thuyết 7](#_Toc211871161)

[2.1. Khái niệm Design Patterns 7](#_Toc211871162)

[2.2. Lợi ích của Design Patterns 7](#_Toc211871163)

[2.3. Phân loại các mẫu thiết kế (theo GOF) 8](#_Toc211871164)

[2.4. Các mẫu thiết kế hỗ trợ kiến trúc hiện đại 9](#_Toc211871165)

[CHƯƠNG 3. CÀI ĐẶT VÀ HƯỚNG DẪN SỬ DỤNG 10](#_Toc211871166)

[3.1. Yêu cầu hệ thống 10](#_Toc211871167)

[3.2. Thiết lập dự án 11](#_Toc211871168)

[CHƯƠNG 4. Giới Thiệu Hệ Thống Website Bán Sách 15](#_Toc211871169)

[4.1. Giao diện 15](#_Toc211871170)

[4.2. Trò chơi 17](#_Toc211871171)

[CHƯƠNG 5. Design Patterns Được Áp Dụng 18](#_Toc211871172)

[5.1. Singleton Pattern 18](#_Toc211871173)

[5.2. Strategy Pattern 32](#_Toc211871174)

[5.3. Decorator Pattern 38](#_Toc211871175)

[5.4. MVC Pattern 42](#_Toc211871176)

[5.5. Repository Pattern 47](#_Toc211871177)

[5.6. Dependency Injection Pattern 53](#_Toc211871178)

[CHƯƠNG 6. Kết Luận 57](#_Toc211871179)

[TÀI LIỆU THAM KHẢO 59](#_Toc211871180)

**DANH MỤC HÌNH ẢNH**

[Hình 4.1.1. Đăng nhập 15](#_Toc211871146)

[Hình 4.1.1. Giao diện chính 16](#_Toc211871147)

[Hình 4.2.1. Trò chơi 17](#_Toc211871148)

# CHƯƠNG 1. Mở Đầu

## 1.1. Giới thiệu hệ thống

Ứng dụng Game 2048 là một trò chơi giải đố số học được phát triển dưới dạng web application, sử dụng kiến trúc Client-Server với ASP.NET Core làm backend và HTML/CSS/JavaScript làm frontend.

**Các tính năng chính của hệ thống:**

- **Hệ thống đăng nhập**: Cho phép người chơi đăng nhập với tài khoản riêng và lưu trữ điểm số cao nhất

- **Đa kích thước bàn chơi**: Hỗ trợ 3 chế độ kích thước khác nhau:

  - 3x3 (đạt 128 để thắng)

  - 4x4 (đạt 2048 để thắng - chế độ chuẩn)

  - 5x5 (đạt 4096 để thắng)

- **Đa chiến lược chơi**: 3 chế độ chơi với cơ chế tính điểm khác nhau:

  - **Standard Mode**: Chế độ thường với tính điểm chuẩn

  - **Fast Mode**: Chế độ nhanh với điểm số cao hơn khi merge

  - **Test Mode**: Chế độ thử nghiệm không tính điểm

- **Lưu trữ thành tích**: Tự động lưu điểm cao nhất của mỗi người chơi theo từng kích thước bàn chơi

- **Giao diện responsive**: Thiết kế đẹp mắt, hỗ trợ animations và hiệu ứng trực quan

**Kiến trúc hệ thống:**

┌─────────────────────────────────────────┐

│           Frontend (Client)             │

│  HTML5 + CSS3 + Vanilla JavaScript       │

│  - UI/UX responsive                     │

│  - Fetch API communication               │

│  - Real-time game rendering             │

└─────────────────┬───────────────────────┘

                  │ HTTP/REST API

┌─────────────────▼───────────────────────┐

│        Backend (ASP.NET Core 7.0)       │

│  ┌─────────────────────────────────┐     │

│  │   Controllers Layer             │    │

│  │  - AuthController               │    │

│  │  - GameController               │    │

│  └──────────────┬──────────────────┘     │

│  ┌──────────────▼──────────────────┐     │

│  │   Models Layer (Design Patterns) │     │

│  │  - Singleton: GameManager       │    │

│  │  - Strategy: Move Algorithms     │     │

│  │  - Decorator: Cell Enhancement   │    │

│  └─────────────────────────────────┘     │

│  ┌─────────────────────────────────┐     │

│  │   Session Management             │     │

│  │  - User authentication           │     │

│  │  - Score persistence            │    │

│  └─────────────────────────────────┘     │

└─────────────────────────────────────────┘

## 1.2. Lý do chọn đề tài

Game 2048 là một ứng dụng lý tưởng để minh họa và áp dụng các mẫu thiết kế (Design Patterns) trong lập trình hướng đối tượng vì:

- Có logic nghiệp vụ rõ ràng, dễ phân tích

- Cấu trúc đủ phức tạp để áp dụng nhiều patterns khác nhau

- Dễ dàng mở rộng và bảo trì khi áp dụng đúng patterns

Đề tài cho phép triển khai và thể hiện rõ ràng 3 Design Patterns cơ bản:

- **Singleton Pattern**: Quản lý trạng thái game toàn cục, đảm bảo tính nhất quán của dữ liệu

- **Strategy Pattern**: Cho phép thay đổi thuật toán tính điểm và di chuyển linh hoạt

- **Decorator Pattern**: Trang trí và mở rộng tính năng của các ô (cells) mà không thay đổi cấu trúc cốt lõi

## 1.3. Mục tiêu

* Giới thiệu tổng quan về **Design Patterns** và vai trò trong phát triển phần mềm.
* Xây dựng một ứng dụng web game 2048 hoàn chỉnh, áp dụng hiệu quả các Design Patterns để tạo ra codebase dễ đọc, dễ bảo trì và dễ mở rộng.

1. Về mặt kỹ thuật:

- Triển khai thành công 3 Design Patterns chính:

* Singleton Pattern cho Game Manager
* Strategy Pattern cho các thuật toán di chuyển
* Decorator Pattern cho các tính năng mở rộng của Cell

- Xây dựng RESTful API với ASP.NET Core

* Thiết kế giao diện người dùng responsive và thân thiện
* Implement session-based authentication
* Xử lý game logic chính xác (merge, move, scoring)

2. Về mặt chức năng:

* - Hệ thống đăng nhập với nhiều tài khoản
* - Hỗ trợ 3 kích thước bàn chơi (3x3, 4x4, 5x5)
* - Hỗ trợ 3 chế độ chơi với cơ chế tính điểm khác nhau
* - Lưu trữ và hiển thị điểm cao nhất (High Score)
* - Phát hiện trạng thái thắng/thua chính xác
* - Cho phép chơi lại (New Game)

3. Về mặt học tập:

* Hiểu sâu về Design Patterns và cách áp dụng vào thực tế

4. Về mặt sản phẩm:

* Tạo ra một game có thể chơi được, mượt mà, không lỗi
* Giao diện đẹp mắt, animations mượt mà
* Trải nghiệm người dùng tốt (UX)
* Performance tối ưu, load nhanh

## 1.4. Phạm vi

Trong phạm vi tiểu luận này, tập trung nghiên cứu và áp dụng các mẫu thiết kế sau:

**Các mẫu thiết kế chính:**

* **Singleton Pattern** – Quản lý trạng thái game toàn cục (board, score, strategy) thông qua `GameManagerSingleton`. Đảm bảo chỉ có một instance duy nhất quản lý game state, tránh xung đột dữ liệu giữa các phiên chơi.
* **Strategy Pattern** – Cho phép thay đổi thuật toán di chuyển và tính điểm một cách linh hoạt. Gồm 3 strategies: `StandardMoveStrategy` (tính điểm chuẩn), `FastMoveStrategy` (điểm gấp đôi khi merge), `TestMoveStrategy` (không tính điểm) mà không cần sửa đổi lớp `GameManagerSingleton`.
* **Decorator Pattern** – Mở rộng chức năng hiển thị của Cell (ô game) một cách động. Sử dụng `SpecialEffectDecorator` (thêm hiệu ứng glow cho ô cao điểm), `NewTileDecorator` (thêm animation cho ô mới), `MergedTileDecorator` (thêm hiệu ứng khi merge) mà không thay đổi cấu trúc Cell cơ bản.

**Các mẫu thiết kế phụ (áp dụng qua framework):**

* + **MVC Pattern:** Tách biệt Model (`GameModels.cs`, `UserModels.cs`), View (`wwwroot/index.html`), Controller (GameController.cs, AuthController.cs) để tổ chức code rõ ràng và dễ bảo trì.
  + **Repository Pattern** – Quản lý truy cập dữ liệu người dùng thông qua `Dictionary<string, User>` trong `AuthController`, tách biệt logic lưu trữ khỏi business logic.
  + **Dependency Injection Pattern** – Sử dụng ASP.NET Core DI Container để inject các services (Controllers, Session, Cache) giảm sự phụ thuộc giữa các components.

## 1.5. Công nghệ sử dụng

Để triển khai và minh họa hệ thống, tiểu luận sử dụng các công nghệ sau:

* **Ngôn ngữ lập trình**: **C#** - Ngôn ngữ chính cho backend, hỗ trợ OOP và Design Patterns
* **Framework kiến trúc**: ASP.NET MVC.
* **Frontend**: HTML, CSS, JavaScript
* **Công cụ hỗ trợ**: GitHub để quản lý mã nguồn.

# CHƯƠNG 2. Cơ Sở Lý Thuyết

## 2.1. Khái niệm Design Patterns

- Design Patterns (hay còn gọi là Mẫu Thiết Kế Phần Mềm) là tập hợp những mẫu giải pháp thiết kế tổng quát, được rút ra từ kinh nghiệm thực tiễn qua nhiều dự án phần mềm, nhằm giải quyết các vấn đề thường gặp trong lập trình hướng đối tượng. Những mẫu này không phải là các đoạn mã cụ thể để lập trình viên sao chép, mà chính là các mô hình, cách tiếp cận hoặc phương pháp tổ chức cấu trúc có thể áp dụng linh hoạt cho nhiều tình huống khác nhau trong quá trình phát triển phần mềm.  
- Nhờ có Design Patterns, lập trình viên có thể thừa hưởng các giải pháp đã được kiểm nghiệm, đóng gói dưới dạng tri thức thực tế, từ đó dễ dàng tái sử dụng kinh nghiệm mà không phải phát minh lại từ đầu. Ví dụ thực tế: Khi muốn đảm bảo toàn bộ hệ thống chỉ có một đối tượng duy nhất chịu trách nhiệm quản lý kết nối cơ sở dữ liệu, thay vì tự phát triển cơ chế kiểm soát riêng, lập trình viên có thể lựa chọn triển khai mẫu Singleton Pattern để giải quyết nhu cầu này một cách hệ thống và hiệu quả.

## 2.2. Lợi ích của Design Patterns

Việc áp dụng các mẫu thiết kế mang lại nhiều lợi ích trong phát triển phần mềm:

* **Tái sử dụng (Reusability):** Giúp sử dụng lại giải pháp quen thuộc cho nhiều dự án.
* **Dễ mở rộng (Extensibility):** Cho phép bổ sung chức năng mới mà ít ảnh hưởng đến phần đã có.
* **Dễ bảo trì (Maintainability):** Mã nguồn rõ ràng, có cấu trúc tốt, giảm chi phí bảo trì.
* **Giao tiếp tốt hơn:** Các mẫu thiết kế được chuẩn hóa, giúp lập trình viên dễ dàng trao đổi bằng ngôn ngữ chung (“chỉ cần nói: dùng Observer Pattern”).
* **Giảm độ phức tạp:** Tách biệt các phần trong hệ thống, giúp quản lý tốt hơn.

## 2.3. Phân loại các mẫu thiết kế (theo GOF)

Theo nhóm Gang of Four (GOF), các mẫu thiết kế được chia thành 3 loại chính:

1. **Creational Patterns (Nhóm khởi tạo):**
   * Mục đích: kiểm soát và đơn giản hóa việc khởi tạo đối tượng.
   * Ví dụ: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
   * Ứng dụng trong website bán sách: Factory Method để tạo ra các loại sản phẩm (sách giấy, ebook, audiobook).
2. **Structural Patterns (Nhóm cấu trúc):**
   * Mục đích: tổ chức mối quan hệ giữa các lớp, đối tượng để xây dựng hệ thống linh hoạt.
   * Ví dụ: Adapter, Facade, Decorator, Composite, Proxy.
   * Ứng dụng trong website bán sách: Facade để gom các bước đặt hàng thành một giao diện đơn giản, Decorator để thêm chức năng gói quà cho sản phẩm.
3. **Behavioral Patterns (Nhóm hành vi):**
   * Mục đích: quản lý cách các đối tượng giao tiếp và cộng tác với nhau.
   * Ví dụ: Observer, Strategy, Command, State, Template Method.
   * Ứng dụng trong website bán sách: Strategy cho các cách tính phí vận chuyển, Command cho thao tác tăng/giảm số lượng giỏ hàng.

## 2.4. Các mẫu thiết kế hỗ trợ kiến trúc hiện đại

Bên cạnh bộ 23 mẫu GOF truyền thống, các hệ thống phần mềm hiện đại như Domain-Driven Design (DDD), Clean Architecture hay .NET Core MVC còn tận dụng thêm nhiều mẫu thiết kế bổ sung, giúp nâng cao khả năng mở rộng, phân tách lớp, và tổ chức code hiệu quả hơn:

* Repository Pattern: Giúp phân tách phần xử lý dữ liệu khỏi lớp business logic, từ đó tạo điều kiện cho kiểm thử, tái sử dụng và bảo trì dễ dàng hơn.
* Unit of Work Pattern: Gom lại nhiều thao tác thay đổi dữ liệu thành một giao dịch duy nhất, đảm bảo tính toàn vẹn và nhất quán cho dữ liệu khi xử lý các nghiệp vụ phức tạp.
* Dependency Injection (DI): Là kỹ thuật giúp giảm sự phụ thuộc chặt chẽ giữa các lớp bằng cách "tiêm" các đối tượng phụ thuộc từ bên ngoài, từ đó tăng tính linh hoạt, dễ kiểm thử và bảo trì hệ thống.
* ViewModel Pattern: Tổng hợp, phối hợp dữ liệu từ nhiều entity khác nhau để đưa ra cho giao diện người dùng, giúp quá trình thiết kế giao diện rõ ràng, dễ quản lý và nâng cấp.

# CHƯƠNG 3. CÀI ĐẶT VÀ HƯỚNG DẪN SỬ DỤNG

## 3.1. Yêu cầu hệ thống

**Dự án yêu cầu**

**Cài đặt .NET SDK 7.0 (Bắt buộc)**

* Tải và cài đặt .NET SDK 7.0 từ: <https://dotnet.microsoft.com/download/dotnet/7.0>
* Sau khi cài đặt, restart terminal và kiểm tra: dotnet --version

**Trình duyệt web (Bắt buộc)**

* Google Chrome (khuyến nghị) - phiên bản 90 trở lên
* Microsoft Edge - phiên bản 90 trở lên
* Firefox - phiên bản 88 trở lên
* Hỗ trợ JavaScript ES6+ và Fetch API

**Công cụ phát triển (Tùy chọn)**

* **Visual Studio Code** - Editor khuyến nghị- Tải từ: <https://code.visualstudio.com/>
* **Extensions đề xuất:**
* C# (Microsoft)
* C# Dev Kit (Microsoft)
* REST Client hoặc Thunder Client

**Git** - Version control- Tải từ: <https://git-scm.com/downloads>

**Postman** - Test API (tùy chọn)

- Tải từ: <https://www.postman.com/downloads/>

## 3.2. Thiết lập dự án

**Clone hoặc tải project từ GitHub**

Cách 1: Sử dụng Git Clone

```bash

git clone https://github.com/nhquang2509/MTK-CuoiKy.git

cd MTK-CuoiKy

Cách 2: Tải ZIP

- Truy cập: https://github.com/nhquang2509/MTK-CuoiKy

- Click vào nút \*\*Code\*\* → \*\*Download ZIP\*\*

- Giải nén file ZIP vào thư mục mong muốn

- Mở terminal tại thư mục vừa giải nén

**Restore dependencies**

Khôi phục các NuGet packages cần thiết:

```bash

dotnet restore

```

Lệnh này sẽ tải về các dependencies được định nghĩa trong `Game2048.csproj`:

- Microsoft.AspNetCore.Session

- Microsoft.Extensions.Caching.Memory

- System.Text.Json

**Build project**

Build dự án để kiểm tra lỗi compilation:

```bash

dotnet build

```

**Kết quả mong đợi:**

Build succeeded.

    0 Warning(s)

    0 Error(s)

Nếu gặp lỗi `NETSDK1045` (target framework mismatch), kiểm tra file `Game2048.csproj` có dòng:

```xml

<TargetFramework>net7.0</TargetFramework>

```

**Chạy ứng dụng**

Khởi động server:

```bash

dotnet run

```

Hoặc chạy với hot reload (tự động restart khi có thay đổi):

```bash

dotnet watch run

```

Server sẽ khởi động tại địa chỉ:

http://localhost:5000

**Truy cập ứng dụng**

- Mở trình duyệt web

- Truy cập: `http://localhost:5000`

- Đăng nhập bằng một trong hai tài khoản demo:

  - Username: player1, Password: pass123

  - Username: player2, Password: pass456

**3.3. Cấu trúc thư mục dự án**

MTK/

├── Controllers/                     # Controller Layer (MVC Pattern)

│   ├── AuthController.cs         # Xử lý đăng nhập, đăng xuất, lưu điểm

│   └── GameController.cs           # Xử lý game logic, move, new game

│

├── Models/                         # Model Layer - Design Patterns

│   ├── GameModels.cs               # Singleton, Strategy, Decorator Patterns

│   │   ├── GameManagerSingleton   # Singleton Pattern

│   │   ├── IMoveStrategy           # Strategy Pattern Interface

│   │   │   ├── StandardMoveStrategy

│   │   │   ├── FastMoveStrategy

│   │   │   └── TestMoveStrategy

│   │   ├── ICell                   # Decorator Pattern Interface

│   │   │   ├── Cell               # Base Component

│   │   │   ├── CellDecorator       # Base Decorator

│   │   │   ├── SpecialEffectDecorator

│   │   │   ├── NewTileDecorator

│   │   │   └── MergedTileDecorator

│   │   └── Direction               # Enum (Up, Down, Left, Right)

│   └── UserModels.cs               # Data Transfer Objects (DTOs)

│       ├── User

│       ├── LoginRequest

│       ├── GameResponse

│       └── MoveRequest

│

├── wwwroot/                         # Static Files (Frontend)

│   ├── index.html                 # Giao diện chính

│   ├── css/

│   │   └── style.css               # Styling, animations, responsive

│   └── js/

│       └── app.js                 # Frontend logic, API calls

│

├── bin/                             # Compiled binaries (auto-generated)

│   └── Debug/

│       └── net7.0/

│

├── obj/                             # Build artifacts (auto-generated)

│   └── Debug/

│       └── net7.0/

│

├── Program.cs                       # Entry point, middleware configuration

└── Game2048.csproj                 # Project file, dependencies

# CHƯƠNG 4. Giới Thiệu Hệ Thống Website Bán Sách

## 4.1. Giao diện

|  |
| --- |
|  |
| Hình 4.1.1. Đăng nhập |

|  |
| --- |
|  |
| Hình 4.1.1. Giao diện chính |

## 4**.2.** Trò chơi

|  |
| --- |
|  |
| Hình 4.2.1. Trò chơi |

# CHƯƠNG 5. Design Patterns Được Áp Dụng

## 5.1. Singleton Pattern

**5.1.1. Giới thiệu**

Singleton là một Creational Design Pattern – thuộc nhóm mẫu thiết kế dùng để khởi tạo đối tượng.

Mục tiêu của mẫu này là đảm bảo một lớp chỉ có duy nhất một thể hiện (instance) trong suốt vòng đời ứng dụng, đồng thời cung cấp một điểm truy cập toàn cục (global access point) đến thể hiện đó. ​

**Ứng dụng:**

- Khi chỉ cần một đối tượng duy nhất để điều phối hoạt động toàn hệ thống (ví dụ: cấu hình ứng dụng, logging, quản lý kết nối, cache, thread pool, game state manager).

- Khi muốn kiểm soát số lượng thể hiện được khởi tạo và truy cập tập trung.

- Khi cần đảm bảo tính nhất quán của dữ liệu chia sẻ giữa các thành phần khác nhau.

**5.1.2. Mục đích ra đời**

Trong quá trình thiết kế hệ thống, đôi khi cần một đối tượng tồn tại duy nhất có thể truy cập từ bất kỳ đâu.

Một giải pháp hiển nhiên là sử dụng **biến toàn cục (global variable)**, nhưng điều này phá vỡ **nguyên tắc đóng gói (encapsulation)** của lập trình hướng đối tượng.

**Singleton** khắc phục điểm yếu này bằng cách:

* Ẩn constructor để ngăn việc tự ý khởi tạo.
* Cung cấp phương thức tĩnh (static) duy nhất để trả về đối tượng toàn cục.
* Đảm bảo chỉ một thể hiện duy nhất được tạo ra

**Vấn đề cụ thể trong Game 2048:**

Trong game 2048, trạng thái game (bàn chơi, điểm số, chiến lược di chuyển) cần được duy trì xuyên suốt các request HTTP từ client. Nếu mỗi request tạo một instance mới của GameManager, sẽ dẫn đến:

- Mất dữ liệu game giữa các lượt di chuyển

- Xung đột dữ liệu khi nhiều người chơi cùng lúc

- Lãng phí bộ nhớ với nhiều instance không cần thiết

**5.1.3. Cấu trúc & Triển khai**

Cách cài đặt mẫu Singleton bao gồm ba bước cơ bản:

1. Đặt **constructor là private** để ngăn tạo đối tượng từ bên ngoài.
2. Tạo một **thuộc tính tĩnh (private static)** chứa duy nhất thể hiện của chính lớp đó.
3. Cung cấp một **phương thức tĩnh công khai (public static)** để trả về thể hiện duy nhất này. ​

**Sơ đồ UML (giản lược)**

A diagram of a computer

AI-generated content may be incorrect.

**Luồng hoạt động:**

Client Request 1                Client Request 2

       ↓                               ↓

GameController.NewGame()        GameController.Move()

       ↓                               ↓

GameManagerSingleton.Instance ← Check if \_instance == null?

       ↓                               ↓

    [null] → Create new instance    [exists] → Return existing

       ↓                               ↓

   lock(\_lock)                   Same instance

       ↓                               ↓

 \_instance = new()                Shared state

       ↓                               ↓

  Return \_instance ─────────────────→ Same Board, Score, Strategy

**5.1.4. Ưu điểm và Hạn chế**

**Ưu điểm**

- Đảm bảo chỉ tồn tại duy nhất một instance của lớp trong toàn bộ ứng dụng.

- Cung cấp truy cập toàn cục (global access) đến thể hiện đó một cách an toàn.

- Khởi tạo trễ (lazy initialization) - Tạo thể hiện chỉ khi cần thiết, tiết kiệm tài nguyên.

- Thread-safe - Đảm bảo an toàn trong môi trường đa luồng với double-checked locking.

- Giảm xung đột trong việc quản lý tài nguyên dùng chung (game state).

- Tính nhất quán - Đảm bảo dữ liệu đồng bộ giữa các thành phần khác nhau.

**Nhược điểm**

- Vi phạm Single Responsibility Principle vì vừa kiểm soát vòng đời vừa cung cấp nghiệp vụ.

- Tight coupling - Các module phụ thuộc trực tiếp vào Singleton, khó thay thế.

- Khó khăn cho Unit Testing - Không thể tạo mock objects dễ dàng, state được chia sẻ giữa các test.

- Ẩn chứa dependencies - Các class sử dụng Singleton không khai báo rõ dependencies trong constructor.

- Concurrency issues - Nếu không implement đúng, có thể gây race condition trong môi trường đa luồng.

**5.1.5. Khi nào nên sử dụng Singleton**

Áp dụng mẫu Singleton khi:

* Cần đảm bảo chỉ tồn tại duy nhất một thể hiện trong toàn hệ thống.
* Cần cung cấp truy cập tập trung đến tài nguyên hay thành phần chia sẻ.
* Đối tượng đại diện cho tài nguyên hệ thống dùng chung (file log, cache, database connection, configuration).

Một số mẫu khác cũng tận dụng Singleton trong quá trình triển khai như: Abstract Factory, Builder, Prototype, Facade. ​

**5.1.6. Ứng dụng trong dự án: GameManagerSingleton**

**5.1.6.1. Mục tiêu thiết kế**

Lớp **GameManagerSingleton** trong namespace `Game2048.Models` được xây dựng theo mẫu Singleton Pattern, nhằm đảm bảo rằng chỉ có một thực thể duy nhấtcủa GameManager tồn tại trong ứng dụng.

**Điều này mang lại lợi ích lớn:**

- Tất cả các controller trong hệ thống chia sẻ cùng một game state

- Đồng bộ hóa giữa các requests (NewGame, Move, SetStrategy)

- Tránh việc tạo nhiều instance trùng lặp gây lãng phí bộ nhớ

- Đảm bảo tính nhất quán của trạng thái game (board, score, strategy)

**5.1.6.2. Cấu trúc chính của Singleton Logger**

**a. Thành phần lõi**

public sealed class GameManagerSingleton

{

    private static GameManagerSingleton? \_instance;

    private static readonly object \_lock = new object();

    public static GameManagerSingleton Instance

    {

        get

        {

            if (\_instance == null)

            {

                lock (\_lock)

                {

                    if (\_instance == null)

                    {

                        \_instance = new GameManagerSingleton();

                    }

                }

            }

            return \_instance;

        }

    }

}

**Giải thích chi tiết:**

- \_instance: Biến static nullable lưu trữ instance duy nhất của GameManager. Khởi tạo là null, chỉ được gán giá trị khi lần đầu truy cập.

- \_lock: Object dùng làm khóa (lock) để đảm bảo thread-safety khi khởi tạo instance trong môi trường đa luồng (multi-threading).

- Instance` Property: Là cổng truy cập toàn cục (Global Access Point), sử dụng Double-Checked Locking pattern:

  - Kiểm tra lần 1 (if (\_instance == null)): Tránh việc lock không cần thiết khi instance đã tồn tại.

  - lock (\_lock): Đảm bảo chỉ một thread được phép khởi tạo instance tại một thời điểm.

  - Kiểm tra lần 2: Trong lock, kiểm tra lại để đảm bảo thread khác chưa tạo instance.

  - Thread Safety: Cơ chế này đảm bảo an toàn 100% trong môi trường ASP.NET Core với nhiều requests đồng thời.

So sánh với Lazy\<T\> (alternative approach):

Cách khác: Sử dụng Lazy<T> (tự động thread-safe)

private static readonly Lazy<GameManagerSingleton> instance =

    new Lazy<GameManagerSingleton>(() => new GameManagerSingleton());

public static GameManagerSingleton Instance => instance.Value;

Dự án sử dụng Double-Checked Locking thay vì Lazy<T> để:

- Kiểm soát rõ ràng hơn quá trình khởi tạo

- Hiểu rõ cơ chế thread-safety

- Phù hợp cho mục đích học tập và minh họa pattern

**5.1.6.3. Cấu trúc class Logger**

public sealed class GameManagerSingleton

{

    private static GameManagerSingleton? \_instance;

    private static readonly object \_lock = new object();

    public int GridSize { get; private set; }

    public int[,] Board { get; private set; }

    public int Score { get; private set; }

    public int BestScore { get; private set; }

    public string Username { get; private set; }

    public IMoveStrategy MoveStrategy { get; set; }

}

**Giải thích các thành phần:**

- sealed: Modifier này ngăn lớp bị kế thừa, đảm bảo không có subclass nào có thể phá vỡ hành vi Singleton. Điều này rất quan trọng vì nếu cho phép kế thừa, class con có thể tạo nhiều instances.

- \_lock: Object dùng để bảo vệ truy cập vào tài nguyên chia sẻ (instance) trong trường hợp nhiều threads cùng lúc gọi Instance property.

- Properties (Game State):

  - GridSize: Kích thước bàn chơi (3, 4, hoặc 5)

  - Board: Ma trận 2D chứa giá trị các ô (0 = ô trống)

  - Score: Điểm số hiện tại của game

  - BestScore: Điểm cao nhất đạt được

  - Username: Tên người chơi hiện tại

  - MoveStrategy: Chiến lược di chuyển (Strategy Pattern) - có thể thay đổi runtime

Hàm khởi tạo (constructor) là private:

private GameManagerSingleton()

{

    GridSize = 4;

    Board = new int[4, 4];

    Score = 0;

    BestScore = 0;

    Username = "";

    MoveStrategy = new StandardMoveStrategy(); // Default strategy

}

Constructor private có mục đích:

- Ngăn tạo instance từ bên ngoài bằng new GameManagerSingleton() (sẽ gây compile error)

- Chỉ cho phép tạo instance thông qua property Instance

- Khởi tạo giá trị mặc định: Bàn chơi 4x4, điểm 0, strategy chuẩn

**5.1.6.4. Các phương thức nghiệp vụ chính**

**a. Khởi tạo game mới**

public void InitializeGame(int gridSize, string username)

{

    GridSize = gridSize;

    Board = new int[gridSize, gridSize];

    Score = 0;

    Username = username;

    AddRandomTile();

    AddRandomTile();

}

**Chức năng:**

- Reset toàn bộ game state với kích thước mới

- Tạo bàn chơi trống (ma trận gridSize × gridSize)

- Reset điểm về 0

- Lưu username của người chơi

- Thêm 2 ô ngẫu nhiên (2 hoặc 4) để bắt đầu game

**b. Thêm ô ngẫu nhiên**

public void AddRandomTile()

{

    var emptyCells = new List<(int, int)>();

    for (int i = 0; i < GridSize; i++)

    {

        for (int j = 0; j < GridSize; j++)

        {

            if (Board[i, j] == 0)

                emptyCells.Add((i, j));

        }

    }

    if (emptyCells.Count > 0)

    {

        var random = new Random();

        var (row, col) = emptyCells[random.Next(emptyCells.Count)];

        Board[row, col] = random.Next(10) < 9 ? 2 : 4; // 90% là 2, 10% là 4

    }

}

**Thuật toán:**

1. Duyệt toàn bộ bàn chơi, tìm các ô trống (giá trị 0)

2. Lưu vị trí các ô trống vào list

3. Chọn ngẫu nhiên một ô từ list

4. Gán giá trị: 90% xác suất là 2, 10% là 4 (theo rule chuẩn của 2048)

**c. Ghi hoạt động liên quan đơn hàng**

public bool Move(Direction direction)

{

    var (newBoard, scoreDelta) = MoveStrategy.ExecuteMove(Board, GridSize, direction);

    bool moved = !BoardsEqual(Board, newBoard);

    if (moved)

    {

        Board = newBoard;

        Score += scoreDelta;

        AddRandomTile();

        if (Score > BestScore)

            BestScore = Score;

    }

    return moved;

}

**Logic xử lý:**

1. Gọi MoveStrategy.ExecuteMove() để thực hiện di chuyển theo chiến lược đã chọn

2. So sánh bàn chơi cũ và mới để xác định có di chuyển thành công không

3. Nếu có di chuyển:

   - Cập nhật bàn chơi mới

   - Cộng điểm delta vào score

   - Thêm 1 ô ngẫu nhiên

   - Cập nhật best score nếu vượt qua

4. Trả về true nếu có di chuyển, false nếu không

**d. Kiểm tra game over**

public bool IsGameOver()

{

    // Kiểm tra có ô trống

    for (int i = 0; i < GridSize; i++)

    {

        for (int j = 0; j < GridSize; j++)

        {

            if (Board[i, j] == 0)

                return false;

        }

    }

    // Kiểm tra có thể merge

    for (int i = 0; i < GridSize; i++)

    {

        for (int j = 0; j < GridSize; j++)

        {

            if (j < GridSize - 1 && Board[i, j] == Board[i, j + 1])

                return false;

            if (i < GridSize - 1 && Board[i, j] == Board[i + 1, j])

                return false;

        }

    }

    return true;

}

**Điều kiện Game Over:**

1. Không còn ô trống (tất cả ô đều có giá trị)

2. Không còn 2 ô kề nhau nào có cùng giá trị (không thể merge)

**e. Kiểm tra thắng cuộc**

public bool HasWon()

{

    int target = GridSize == 3 ? 128 : GridSize == 4 ? 2048 : 4096;

    for (int i = 0; i < GridSize; i++)

    {

        for (int j = 0; j < GridSize; j++)

        {

            if (Board[i, j] >= target)

                return true;

        }

    }

    return false;

}

**Target theo grid size:**

- 3×3: Đạt 128

- 4×4: Đạt 2048 (chuẩn)

- 5×5: Đạt 4096

**5.1.6.5. Cách sử dụng trong Controller**

public class GameController : ControllerBase

{

    // Truy cập Singleton instance

    private GameManagerSingleton \_gameManager = GameManagerSingleton.Instance;

    [HttpPost("new-game")]

    public IActionResult NewGame([FromBody] NewGameRequest request)

    {

        var username = HttpContext.Session.GetString("Username");

        // Khởi tạo game mới

        \_gameManager.InitializeGame(request.GridSize, username);

        return Ok(new GameResponse

        {

            Board = ConvertToJaggedArray(\_gameManager.Board, \_gameManager.GridSize),

            Score = \_gameManager.Score,

            BestScore = \_gameManager.BestScore,

            GameOver = false,

            Won = false

        });

    }

    [HttpPost("move")]

    public IActionResult Move([FromBody] MoveRequest request)

    {

        // Sử dụng cùng một instance

        bool moved = \_gameManager.Move(direction);

        return Ok(new GameResponse

        {

            Board = ConvertToJaggedArray(\_gameManager.Board, \_gameManager.GridSize),

            Score = \_gameManager.Score,

            BestScore = \_gameManager.BestScore,

            GameOver = \_gameManager.IsGameOver(),

            Won = \_gameManager.HasWon()

        });

    }

}

**Lợi ích khi sử dụng:**

- Mọi request đều dùng chung 1 instance → state được duy trì

- Không cần truyền tham số giữa các methods

- Thread-safe khi nhiều requests đồng thời

**5.1.6.6. Tóm tắt cơ chế hoạt động**

|  |  |  |
| --- | --- | --- |
| Thành phần | Mô tả | Lợi ích |
| Double-Checked Locking | Kiểm tra 2 lần với lock ở giữa | Đảm bảo thread-safe, tránh overhead không cần thiết |
| sealed class | Không cho phép kế thừa | Tránh sai lệch hành vi Singleton qua subclass |
| private constructor | Ẩn constructor | Ngăn tạo instance từ bên ngoài, chỉ qua Instance property |
| static Instance property | Global access point | Truy cập duy nhất, nhất quán từ mọi nơi |
| \_lock object | Đồng bộ khởi tạo | Bảo vệ race condition trong multi-threading |
| Game State Properties | GridSize, Board, Score, Strategy | Lưu trữ toàn bộ trạng thái game |
| Business Methods | InitializeGame, Move, IsGameOver | Cung cấp API cho game logic |

## 5.2. Strategy Pattern

**5.2.1. Giới thiệu**

Strategy là một Behavioral Design Pattern – thuộc nhóm mẫu thiết kế hành vi (Behavioral Patterns), tập trung vào cách các đối tượng tương tác và phân chia trách nhiệm.

  Mục tiêu của mẫu này là định nghĩa một họ các thuật toán (algorithms), đóng gói từng thuật toán và làm chúng có thể thay thế lẫn nhau. Strategy cho phép thuật toán thay đổi độc lập với client sử dụng nó.

Ứng dụng:

  - Khi có nhiều cách khác nhau để thực hiện cùng một tác vụ

  - Khi muốn chọn thuật toán tại runtime thay vì compile-time

  - Khi cần tránh sử dụng nhiều câu lệnh điều kiện (if-else, switch-case) để chọn hành vi

  - Khi các biến thể của thuật toán có thể được đóng gói thành các class riêng biệt

Trong Game 2048:

- Cần các cách tính điểm khác nhau cho các chế độ chơi

- Cần thay đổi thuật toán di chuyển/merge mà không sửa GameManager

- Dễ dàng thêm chế độ chơi mới trong tương lai

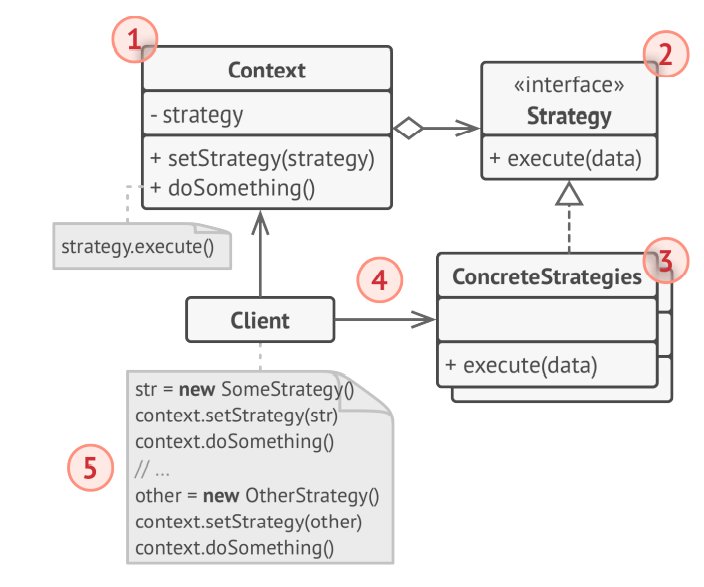
**5.2.2. Mục đích ra đời**

Trong phát triển phần mềm, thường gặp tình huống có nhiều cách để giải quyết cùng một vấn đề. Cách tiếp cận truyền thống là sử dụng câu lệnh điều kiện (if-else hoặc switch-case).

**5.2.3. Kiến trúc**

Cấu trúc chính của Strategy Pattern gồm 3 thành phần:

* **Context**: lớp đóng vai trò "ngữ cảnh", chứa tham chiếu đến một Strategy cụ thể. Nó sẽ gọi phương thức của Strategy thay vì tự triển khai logic.
* **Strategy (Interface)**: định nghĩa giao diện chung cho các chiến lược.
* **Concrete Strategies**: các lớp triển khai cụ thể của Strategy (ví dụ: gói quà, không gói quà, gói quà VIP,…).



**5.2.4. Ưu điểm & Nhược điểm**

**Ưu điểm**

* Mở rộng dễ dàng: thêm chiến lược mới mà không cần chỉnh sửa code cũ.
* Loại bỏ if-else rườm rà.
* Tuân thủ Open/Closed Principle (OCP) trong SOLID.
* Có thể thay đổi chiến lược ngay khi đang chạy chương trình.

**Nhược điểm**

- Mỗi lần thêm thuật toán mới phải sửa code hiện tại (vi phạm Open/Closed Principle)

- Code khó đọc và maintain khi có nhiều điều kiện

- Khó test riêng từng thuật toán

- Không thể thay đổi thuật toán runtime dễ dàng

- Logic bị trộn lẫn, khó tái sử dụng

**5.2.5. Khi nào thì sử dụng**

* Khi có nhiều thuật toán khác nhau cho cùng một hành động, và cần chuyển đổi linh hoạt giữa chúng.
* Khi muốn loại bỏ việc dùng nhiều câu lệnh **if-else/switch-case**.
* Khi một hành vi có thể thay đổi thường xuyên hoặc phụ thuộc vào ngữ cảnh.

Ví dụ:

* Các phương thức thanh toán khác nhau (thẻ, ví điện tử, COD).
* Các thuật toán sắp xếp (QuickSort, MergeSort, BubbleSort).
* Các chiến lược gói quà (cơ bản, cao cấp, không gói).

**5.2.6. Ứng dụng trong dự án: Move Strategies**

public enum Direction

{

    Up,

    Down,

    Left,

    Right

}

public interface IMoveStrategy

{

    (int[,] newBoard, int scoreDelta) ExecuteMove(

        int[,] board,

        int gridSize,

        Direction direction

    );

}

**Giải thích:**

- **Direction Enum**: Định nghĩa 4 hướng di chuyển có thể (Up, Down, Left, Right)

- **IMoveStrategy Interface**: Contract chung cho tất cả strategies

  - **Input**:

    - board: Ma trận hiện tại (2D array)

    - gridSize: Kích thước bàn chơi (3, 4, hoặc 5)

    - direction: Hướng di chuyển

  - **Output**:

    - newBoard: Ma trận mới sau khi di chuyển

    - scoreDelta: Điểm số thay đổi (có thể là 0 với Test mode)

- **Tuple return**: Sử dụng C# tuple để return 2 giá trị cùng lúc

**Concrete Strategy 1: StandardMoveStrategy**

**a. Mô tả:**

StandardMoveStrategy là **thuật toán di chuyển chuẩn** của game 2048 gốc:

- Merge 2 tiles kề nhau có cùng giá trị

- Tính điểm = giá trị sau khi merge (ví dụ: 2+2=4 → +4 điểm)

- Chỉ merge 1 lần mỗi lượt di chuyển

**b. Cấu trúc:**

public class StandardMoveStrategy : IMoveStrategy

{

    public (int[,] newBoard, int scoreDelta) ExecuteMove(

        int[,] board, int gridSize, Direction direction)

    {

        int[,] newBoard = (int[,])board.Clone();

        int scoreDelta = 0;

        switch (direction)

        {

            case Direction.Left:

                scoreDelta = MoveLeft(newBoard, gridSize);

                break;

            case Direction.Right:

                scoreDelta = MoveRight(newBoard, gridSize);

                break;

            case Direction.Up:

                scoreDelta = MoveUp(newBoard, gridSize);

                break;

            case Direction.Down:

                scoreDelta = MoveDown(newBoard, gridSize);

                break;

        }

        return (newBoard, scoreDelta);

    }

}

**c. Thuật toán xử lý (MergeLine):**

private int MergeLine(int[] line)

{

    int score = 0;

    // Bước 1: Slide các giá trị khác 0 về bên trái

    int[] temp = line.Where(x => x != 0).ToArray();

    // Bước 2: Merge các ô kề nhau có cùng giá trị

    for (int i = 0; i < temp.Length - 1; i++)

    {

        if (temp[i] == temp[i + 1] && temp[i] != 0)

        {

            temp[i] \*= 2;        // Nhân đôi giá trị

            score += temp[i];     // Cộng điểm

            temp[i + 1] = 0;      // Xóa ô thứ 2

        }

    }

    // Bước 3: Slide lại sau khi merge

    int[] result = temp.Where(x => x != 0).ToArray();

    // Bước 4: Fill vào line gốc

    for (int i = 0; i < line.Length; i++)

    {

        line[i] = i < result.Length ? result[i] : 0;

    }

    return score;

## 5.3. Decorator Pattern

**5.3.1. Giới thiệu**

**Decorator Pattern** là một mẫu thiết kế thuộc nhóm **Structural Design Pattern**.  
Nó cho phép **bổ sung thêm hành vi (behavior) hoặc tính năng** cho một đối tượng **mà không cần sửa đổi code lớp gốc**.

Ý tưởng chính: thay vì kế thừa (inheritance), ta dùng **thành phần bao bọc (composition)** để “trang trí” thêm tính năng cho object hiện có.

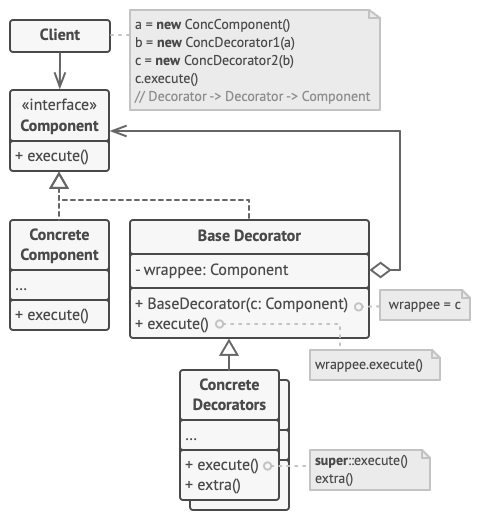
Ví dụ thực tế:

* Một chiếc bánh có thể được thêm kem, thêm trái cây, thêm socola… mỗi lớp phủ là một "decorator".
* Trong hệ thống xử lý đơn hàng: khi OrderProcessor thực hiện, ta có thể “trang trí” nó thêm các bước như log, gửi email, tính phí ship mà không thay đổi logic gốc.

**5.3.2. Mục đích ra đời**

* Cho phép mở rộng hành vi của đối tượng **mà không cần thay đổi lớp gốc** (nguyên tắc **Open/Closed Principle** trong SOLID).
* Thay thế việc tạo ra nhiều lớp kế thừa phức tạp (ví dụ: OrderProcessorWithEmail, OrderProcessorWithLogging, OrderProcessorWithEmailAndLogging...).
* Giúp code **linh hoạt** và **tái sử dụng** hơn.

**5.3.3. Kiến trúc**

**Component**: interface hoặc abstract class định nghĩa hành vi chung.

* **ConcreteComponent**: đối tượng chính, chứa logic gốc.
* **Decorator (abstract)**: lớp trừu tượng hoặc base class, cài đặt Component và có tham chiếu tới Component khác.
* **ConcreteDecorators**: lớp cụ thể, mở rộng hành vi bằng cách override method và gọi tiếp phương thức từ \_component.

**5.3.4. Ưu điểm & Nhược điểm**

**Ưu điểm**

* Dễ mở rộng chức năng cho object mà không sửa code gốc.
* Tránh việc tạo nhiều lớp kế thừa phức tạp.
* Có thể kết hợp nhiều decorator để tạo hành vi mới.

**Nhược điểm**

* Có thể làm tăng độ phức tạp vì nhiều lớp decorator chồng chéo.
* Khó debug khi có nhiều lớp trang trí nối tiếp.
* Nếu dùng quá nhiều decorator có thể làm code khó đọc.

**5.3.5. Khi nào sử dụng**

* Khi cần **mở rộng hành vi động (runtime)** cho một object.
* Khi muốn **tuân thủ Open/Closed Principle**: mở rộng nhưng không sửa đổi lớp gốc.
* Khi có nhiều biến thể của một tính năng, và không muốn tạo nhiều lớp con cho từng trường hợp.

**5.3.6. Mã nguồn trong dự án**

**Interface gốc**

public interface ICell

{

    int Value { get; set; }

    string GetDisplay();

    string GetCssClass();

}

**Concrete Component – Cell**

public class Cell : ICell

{

    public int Value { get; set; }

    public Cell(int value)

    {

        Value = value;

    }

    public virtual string GetDisplay()

    {

        return Value == 0 ? "" : Value.ToString();

    }

    public virtual string GetCssClass()

    {

        return $"cell tile-{Value}";

    }

}

**Base Decorator – CellDecorator**

public abstract class CellDecorator : ICell

{

    protected ICell \_cell;

    public CellDecorator(ICell cell)

    {

        \_cell = cell;

    }

    public int Value

    {

        get => \_cell.Value;

        set => \_cell.Value = value;

    }

    public virtual string GetDisplay()

    {

        return \_cell.GetDisplay();

    }

    public virtual string GetCssClass()

    {

        return \_cell.GetCssClass();

    }

}

## 5.4. MVC Pattern

**5.4.1. Giới thiệu**

MVC (Model-View-Controller) là một architectural pattern thuộc nhóm Presentation Patterns, được sử dụng rộng rãi trong phát triển ứng dụng web và desktop.

Mục đích của MVC là:

- Tách biệt các phần khác nhau của ứng dụng: dữ liệu (Model), giao diện (View), và logic xử lý (Controller).

- Giảm coupling giữa các thành phần, giúp dễ bảo trì, mở rộng và kiểm thử.

- Tái sử dụng các thành phần (một Model có thể được nhiều View hiển thị, một Controller có thể xử lý nhiều request khác nhau).

**5.4.2. Mục đích ra đời**

MVC Pattern ra đời để giải quyết vấn đề này bằng cách tách biệt trách nhiệm (Separation of Concerns):

- Model: Quản lý dữ liệu và business logic

- View: Hiển thị giao diện cho người dùng

- Controller: Nhận request, xử lý logic, điều phối Model và View

**5.4.3. Kiến trúc**

**\*\*Sơ đồ luồng MVC:\*\***

**┌─────────────┐**

**│   Browser   │ (User)**

**└──────┬──────┘**

**│ HTTP Request**

**▼**

**┌─────────────────┐**

**│   Controller     │ ◄──┐**

**│  (GameController │     │**

**│   AuthController)     │**

**└────┬────────────┘     │**

**│                 │ 3. Return Data**

**│ 1. Call         │**

**▼                 │**

**┌─────────────────┐     │**

**│     Model       │ ───┘**

**│  (Game, User,   │**

**│   Cell, Score)   │**

**└─────────────────┘**

**│**

**│ 2. Data**

**▼**

**┌─────────────────┐**

**│      View       │**

**│  (index.html,   │**

**│   app.js, CSS)   │**

**└────┬────────────┘**

**│ HTTP Response**

**▼**

**┌─────────────────┐**

**│    Browser       │**

**└─────────────────┘**

**5.4.4. Ưu điểm & Nhược điểm**

**Ưu điểm**

- Separation of Concerns: Tách biệt rõ ràng giữa data, presentation, và logic

- Testability: Dễ dàng unit test từng thành phần độc lập

- Reusability: Một Model có thể được nhiều Views/Controllers sử dụng

- Parallel Development: Frontend và Backend có thể phát triển song song

- Maintainability: Thay đổi giao diện không ảnh hưởng business logic

**Nhược điểm**

- Complexity: Thêm nhiều files và folders so với monolithic code

- Learning Curve: Developers mới cần thời gian để hiểu luồng MVC

- Over-engineering: Với các ứng dụng nhỏ, MVC có thể quá phức tạp

- Coupling: View và Controller vẫn có thể phụ thuộc lẫn nhau nếu không cẩn thận

**5.4.5. Khi nào thì sử dụng**

- Khi xây dựng web applications với nhiều trang/chức năng khác nhau

- Khi cần phân chia công việc giữa Frontend và Backend teams

- Khi muốn dễ dàng bảo trì và mở rộng\*\* ứng dụng về sau

- Khi cần unit testing cho business logic

- Khi sử dụng frameworks hỗ trợ sẵn (ASP.NET Core MVC, Spring MVC, Ruby on Rails, Django)

**5.4.6. Mã nguồn trong dự án**

public class Game

{

    public int Size { get; set; }                    // 3x3, 4x4, 5x5

    public string Mode { get; set; }                 // Standard, Fast, Test

    public ICell[,] Grid { get; set; }              // 2D grid của cells

    public int Score { get; set; }                   // Điểm hiện tại

    public bool IsGameOver { get; set; }             // Trạng thái game

    // Business logic methods

    public void InitializeGrid() { ... }             // Khởi tạo grid

    public bool Move(Direction direction) { ... }    // Di chuyển tiles

    public bool CanMove() { ... }                    // Kiểm tra có thể move

    public void AddRandomTile() { ... }              // Thêm tile ngẫu nhiên

}

public class Cell : ICell

{

    public int Value { get; set; }

    public int Row { get; set; }

    public int Col { get; set; }

    public string GetCssClass() { ... }

    public string GetDisplay() { ... }

}

**Models/UserModels**

public class User

{

    public string Username { get; set; }

    public string Password { get; set; }

    public int HighScore { get; set; }

    public DateTime LastLogin { get; set; }

}

**Controllers/GameController**

[ApiController]

[Route("api/[controller]")]

public class GameController : ControllerBase

{

    // 1. Nhận HTTP request từ client

    [HttpPost("new")]

    public IActionResult NewGame([FromBody] GameConfig config)

    {

        // 2. Gọi Model để xử lý business logic

        var gameManager = GameManagerSingleton.Instance;

        var game = gameManager.CreateNewGame(config.Size, config.Mode);

        // 3. Trả response về client (View sẽ render)

        return Ok(new {

            grid = game.Grid,

            score = game.Score,

            message = "Game started successfully"

        });

    }

    [HttpPost("move")]

    public IActionResult Move([FromBody] MoveRequest request)

    {

        var gameManager = GameManagerSingleton.Instance;

        var game = gameManager.GetCurrentGame();

        // Business logic trong Model

        bool moved = game.Move(request.Direction);

        if (!moved)

            return BadRequest("Invalid move");

        return Ok(new {

            grid = game.Grid,

            score = game.Score,

            isGameOver = game.IsGameOver

        });

    }

    [HttpGet("state")]

    public IActionResult GetGameState()

    {

        var game = GameManagerSingleton.Instance.GetCurrentGame();

        return Ok(game);

    }

}

## 5.5. Repository Pattern

**5.5.1. Giới thiệu**

Repository Pattern là một mẫu thiết kế thuộc nhóm Data Access Patterns, thường được sử dụng trong các ứng dụng quản lý dữ liệu.

Mục đích của Repository là:

- Tạo một lớp trung gian giữa business logic (service/controller) và data access layer (ORM, database).

- Che giấu chi tiết thao tác dữ liệu, cung cấp API chung cho tầng trên.

- Giúp code dễ đọc, dễ bảo trì, và dễ kiểm thử hơn.

**5.5.2. Mục đích ra đời**

Trong các ứng dụng lớn, việc thao tác trực tiếp với ORM/Database (ví dụ: Entity Framework, Hibernate) sẽ:

- Gây lặp lại code (CRUD tương tự ở nhiều nơi)

- Khó bảo trì khi thay đổi cấu trúc DB hoặc framework ORM

- Gây phụ thuộc mạnh giữa business logic và data access layer

Repository Pattern ra đời để giải quyết vấn đề này: gom các thao tác với dữ liệu vào repository, từ đó controller/service chỉ làm việc với repository thay vì trực tiếp với DbContext/ORM.

**5.5.3. Kiến trúc**

- **IRepository<T>**: Interface chung cho CRUD (GetAll, GetFirstOrDefault, Add, Remove)

- **Repository<T>**: Cài đặt chung cho các thao tác CRUD cơ bản

- **IUserRepository**: Mở rộng IRepository<User> với các method đặc thù (GetByUsername, SaveScore)

- **UserRepository**: Triển khai IUserRepository, viết thêm logic query đặc thù

- **Data Storage**: Dictionary, DbContext (EF Core), File System, v.v.

**5.5.4. Ưu điểm & Nhược điểm**

**Ưu điểm**

- Giảm sự phụ thuộc giữa business logic và ORM/Database

- Gom code thao tác dữ liệu vào một nơi → dễ bảo trì, dễ mở rộng

- Tái sử dụng code (các repository chung cho nhiều entities)

- Thuận tiện khi viết Unit Test (mock repository thay vì DB thật)

- Dễ thay đổi data source (từ SQL → NoSQL, từ DB → File, v.v.)

**Nhược điểm**

- Nếu lạm dụng, có thể dẫn tới tạo quá nhiều lớp repository, làm phức tạp hệ thống

- Repository đôi khi bị trùng lặp logic với ORM (như EF Core vốn đã cung cấp IQueryable)

- Cần kết hợp với Unit of Work Pattern để quản lý transaction hiệu quả

- Thêm layer abstraction → có thể gây overhead cho ứng dụng nhỏ

**5.5.5. Khi nào thì sử dụng**

- Khi ứng dụng có nhiều entity cần CRUD, muốn gom logic xử lý dữ liệu về một chỗ

- Khi muốn tách biệt rõ ràng business logic và data access layer

- Khi cần test code mà không muốn phụ thuộc DB thật (có thể mock repository)

- Khi hệ thống có nguy cơ thay đổi ORM hoặc DB engine trong tương lai

- Khi làm việc trong team lớn, cần chuẩn hóa cách thao tác dữ liệu

**5.5.6. Mã nguồn trong dự án**

**Interface Repository**

public interface IRepository<T> where T : class

{

    T GetFirstOrDefault(Expression<Func<T, bool>> filter);

    IEnumerable<T> GetAll();

    void Add(T entity);

    void Remove(T entity);

    void RemoveRange(IEnumerable<T> entity);

}

**Implementation**

**Controllers/AuthController**

[ApiController]

[Route("api/[controller]")]

public class AuthController : ControllerBase

{

    // In-memory "repository" - Dictionary thay cho Database

    private static Dictionary<string, User> \_users = new Dictionary<string, User>();

    // Method: Add (Register user)

    [HttpPost("register")]

    public IActionResult Register([FromBody] RegisterRequest request)

    {

        // Kiểm tra user đã tồn tại chưa

        if (\_users.ContainsKey(request.Username))

            return BadRequest("Username already exists");

        // Add vào "repository"

        var newUser = new User

        {

            Username = request.Username,

            Password = request.Password, // ⚠️ Trong thực tế phải hash password!

            HighScore = 0,

            LastLogin = DateTime.Now

        };

        \_users.Add(request.Username, newUser);

        return Ok(new { message = "User registered successfully" });

    }

    // Method: GetFirstOrDefault (Login)

    [HttpPost("login")]

    public IActionResult Login([FromBody] LoginRequest request)

    {

        // Query: GetFirstOrDefault(u => u.Username == username)

        if (!\_users.ContainsKey(request.Username))

            return Unauthorized("Invalid username");

        var user = \_users[request.Username];

        // Validate password

        if (user.Password != request.Password)

            return Unauthorized("Invalid password");

        // Set session

        HttpContext.Session.SetString("Username", user.Username);

        user.LastLogin = DateTime.Now;

        return Ok(new

        {

            username = user.Username,

            highScore = user.HighScore,

            message = "Login successful"

        });

    }

    // Method: Get current user

    [HttpGet("current")]

    public IActionResult GetCurrentUser()

    {

        var username = HttpContext.Session.GetString("Username");

        if (string.IsNullOrEmpty(username))

            return Unauthorized("Not logged in");

        // GetFirstOrDefault

        if (!\_users.TryGetValue(username, out var user))

            return NotFound("User not found");

        return Ok(new

        {

            username = user.Username,

            highScore = user.HighScore

        });

    }

    // Method: Update (Save high score)

    [HttpPost("save-score")]

    public IActionResult SaveScore([FromBody] SaveScoreRequest request)

    {

        var username = HttpContext.Session.GetString("Username");

        if (string.IsNullOrEmpty(username))

            return Unauthorized("Not logged in");

        if (!\_users.TryGetValue(username, out var user))

            return NotFound("User not found");

        // Update logic

        if (request.Score > user.HighScore)

        {

            user.HighScore = request.Score;

            return Ok(new { message = "New high score saved!", highScore = user.HighScore });

        }

        return Ok(new { message = "Score not higher than high score", highScore = user.HighScore });

    }

    // Method: Remove (Logout)

    [HttpPost("logout")]

    public IActionResult Logout()

    {

        HttpContext.Session.Remove("Username");

        return Ok(new { message = "Logged out successfully" });

    }

    // Method: GetAll (Leaderboard - tất cả users)

    [HttpGet("leaderboard")]

    public IActionResult GetLeaderboard()

    {

        // GetAll() và sort

        var leaderboard = \_users.Values

            .OrderByDescending(u => u.HighScore)

            .Take(10)

            .Select(u => new

            {

                username = u.Username,

                highScore = u.HighScore

            });

        return Ok(leaderboard);

    }

}

## 5.6. Dependency Injection Pattern

**5.6.1. Giới thiệu**

Dependency Injection (DI) là một Design Pattern thuộc nhóm Inversion of Control (IoC).  
Trong .NET Core, DI được tích hợp sẵn trong framework giúp chúng ta quản lý vòng đời và phụ thuộc của các đối tượng một cách tự động.

Thay vì một class tự khởi tạo phụ thuộc, các phụ thuộc đó sẽ được tiêm từ bên ngoài thông qua constructor, property hoặc method.

**5.6.2. Mục đích ra đời**

* Giảm coupling (sự phụ thuộc chặt chẽ) giữa các lớp.
* Tăng khả năng tái sử dụng code.
* Dễ dàng unit test vì có thể mock/stub phụ thuộc.
* Quản lý lifecycle (singleton, scoped, transient) cho các service.

**5.6.3. Kiến trúc DI trong .NET Core**

Trong ASP.NET Core, DI có 3 thành phần chính:

1. **Service Collection (IServiceCollection)**
   * Là nơi **đăng ký** các service.
   * Ví dụ: builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
2. **Service Provider (IServiceProvider)**
   * Là nơi **tạo instance** của service khi cần dùng.
3. **Service Lifetime**
   * **Transient**: Tạo mới mỗi lần được yêu cầu.
   * **Scoped**: Một instance cho mỗi request.
   * **Singleton**: Duy nhất trong suốt vòng đời ứng dụng.

**5.6.4. Ưu điểm & Nhược điểm**

**Ưu điểm**

* Tách biệt giữa business logic và dependency.
* Test dễ dàng hơn nhờ mock dependency.
* Quản lý tài nguyên hiệu quả (DbContext, HttpClient...).
* Dễ mở rộng khi thay đổi implement mà không ảnh hưởng client.

**Nhược điểm**

* Dễ bị lạm dụng → đăng ký quá nhiều service, khó quản lý.
* Sai lifetime (ví dụ dùng singleton cho DbContext) có thể gây lỗi.

**5.9.5. Khi nào sử dụng**

* Khi cần quản lý phụ thuộc giữa các lớp (Repository, Service, DbContext...).
* Khi ứng dụng có nhiều module tái sử dụng.
* Khi cần test bằng mock object.

**5.6.6. Mã nguồn trong dự án**

**Đăng ký DI trong Program.cs**

var builder = WebApplication.CreateBuilder(args);

// ==================== DEPENDENCY INJECTION CONFIGURATION ====================

// 1. Register Controllers (built-in DI)

builder.Services.AddControllers();

//    ↓ DI Container sẽ tự động resolve các dependencies của Controllers

// 2. Register Session service

builder.Services.AddDistributedMemoryCache(); // Required for session

builder.Services.AddSession(options =>

{

    options.IdleTimeout = TimeSpan.FromMinutes(30);

    options.Cookie.HttpOnly = true;

    options.Cookie.IsEssential = true;

});

//    ↓ Controllers có thể dùng HttpContext.Session

// 3. Register CORS policy

builder.Services.AddCors(options =>

{

    options.AddPolicy("AllowAll",

        policy => policy.AllowAnyOrigin()

                        .AllowAnyMethod()

                        .AllowAnyHeader());

});

// 4. (Optional) Register custom services - Nếu có

// builder.Services.AddSingleton<IGameManager, GameManagerSingleton>();

// builder.Services.AddScoped<IUserRepository, UserRepository>();

// builder.Services.AddTransient<IEmailService, EmailService>();

var app = builder.Build();

// ==================== MIDDLEWARE PIPELINE ====================

app.UseStaticFiles();

app.UseRouting();

app.UseCors("AllowAll");

app.UseSession(); // Enable session middleware

app.MapControllers(); // Map controller routes

app.Run();

# CHƯƠNG 6. Kết Luận

Việc áp dụng Design Patterns trong quá trình phát triển phần mềm mang lại nhiều lợi ích quan trọng, giúp hệ thống vừa dễ duy trì, vừa có khả năng mở rộng trong tương lai:

1. **Tăng khả năng tái sử dụng mã nguồn**
   * Các mẫu thiết kế cung cấp những giải pháp chung cho nhiều vấn đề lặp lại trong phát triển phần mềm.
   * Nhờ đó, cùng một đoạn logic có thể được áp dụng ở nhiều nơi, giảm thiểu việc viết lại mã và tăng hiệu quả phát triển.
2. **Tăng khả năng mở rộng (dễ thêm chức năng mới)**
   * Design Patterns thường khuyến khích lập trình dựa trên nguyên lý mở – đóng (Open/Closed Principle), nghĩa là hệ thống dễ dàng mở rộng thêm tính năng mới mà ít phải chỉnh sửa code cũ.
   * Ví dụ: khi dùng Strategy Pattern, ta có thể thêm một thuật toán mới chỉ bằng cách viết thêm một lớp, không cần thay đổi lớp đang sử dụng.
3. **Cải thiện khả năng bảo trì và đọc hiểu**
   * Code được thiết kế theo các mẫu chuẩn sẽ có cấu trúc rõ ràng, quen thuộc với nhiều lập trình viên.
   * Nhờ đó, việc đọc hiểu và bảo trì hệ thống trở nên dễ dàng hơn, giảm rủi ro khi thay đổi hoặc nâng cấp.
4. **Giảm sự phụ thuộc giữa các thành phần trong hệ thống**
   * Nhiều Design Patterns như Dependency Injection, MVC giúp tách biệt các lớp, giảm sự gắn kết (coupling).
   * Điều này làm cho việc thay đổi hoặc thay thế một thành phần ít ảnh hưởng đến các phần khác, tăng tính linh hoạt và khả năng kiểm thử.

# TÀI LIỆU THAM KHẢO

[1] <https://refactoring.guru/design-patterns/singleton>

[2] <https://refactoring.guru/design-patterns/strategy>

[3] https://refactoring.guru/design-patterns/command

[4] <https://refactoring.guru/design-patterns/strategy>

[5] <https://dev.to/tak089/mvvm-pattern-in-c-15pe>