Chương 3: Giới thiệu về Software Design Pattern và Một Số Design Pattern Hay Gặp

3.1. Giới Thiệu Chung về Software Design Patterns

3.1.1. Định Nghĩa Design Pattern

Giả sử một bài toán thực tế

Giả sử bạn đang phát triển một ứng dụng quản lý thư viện, nơi bạn cần quản lý các loại tài liệu khác nhau như **sách** (Book), **tạp chí** (Magazine) và **báo** (Newspaper). Mỗi loại tài liệu có các đặc điểm và chức năng riêng biệt. Ví dụ, sách có thông tin về tác giả, số trang, trong khi tạp chí có thông tin về số xuất bản và ngày phát hành. Tuy nhiên, tất cả các loại tài liệu này lại có một số hành vi chung như **mượn** (Borrow), **trả** (Return), và **tìm kiếm** (Search).

Nếu bạn không có một phương pháp tổ chức mã nguồn hiệu quả, việc mở rộng và bảo trì hệ thống sẽ trở nên khó khăn. Ví dụ, bạn có thể bắt đầu bằng cách tạo các lớp "Book" (Sách), "Magazine" (Tạp chí) với các phương thức tương tự như `Borrow()` và `Return()`. Ban đầu, bạn có thể thấy mã nguồn hoạt động tốt, nhưng khi cần thêm nhiều loại tài liệu mới, bạn sẽ phải lặp lại và sao chép lại mã cho các phương thức chung. Điều này không chỉ gây ra trùng lặp mà còn khiến việc bảo trì trở nên phức tạp. Nếu có bất kỳ thay đổi nào trong quy trình mượn/trả, bạn phải cập nhật tất cả các lớp, dẫn đến nguy cơ phát sinh lỗi.

Ví dụ mã nguồn ban đầu như sau:

```
using System;

public class Book // Lớp đại diện cho Sách
{
    public string Title { get; set; }
    public string Author { get; set; }

    public void Borrow()
    {
        Console.WriteLine($"Borrowing the book: {Title} by {Author}...");
    }

    public void Return()
    {
        Console.WriteLine($"Returning the book: {Title}...");
    }
}

public class Magazine // Lớp đại diện cho Tạp chí
{
    public string Title { get; set; }
    public string Issue { get; set; }

    public void Borrow()
```

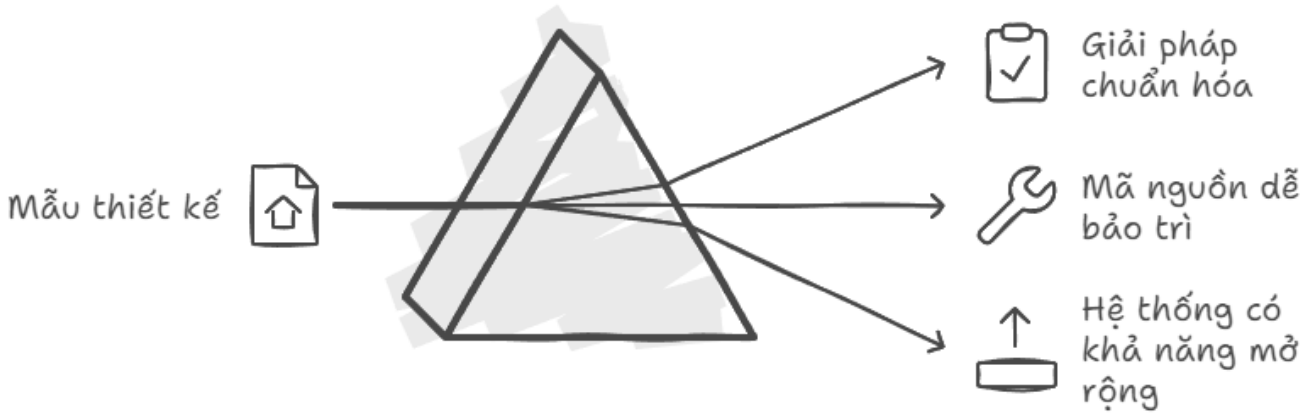
```
    {
        Console.WriteLine($"Borrowing the magazine: {Title}, Issue: {Issue}...");
    }

    public void Return()
    {
        Console.WriteLine($"Returning the magazine: {Title}...");
    }
}

// Lớp quản lý thư viện
public class LibraryManager
{
    public void ManageLibrary(string documentType)
    {
        if (documentType.Equals("book", StringComparison.OrdinalIgnoreCase))
        {
            // Giả lập quy trình cho sách
            Book book = new Book { Title = "C# Programming", Author = "John Doe"
};
            book.Borrow();
            book.Return();
        }
        else if (documentType.Equals("magazine",
StringComparison.OrdinalIgnoreCase))
        {
            // Giả lập quy trình cho tạp chí
            Magazine magazine = new Magazine { Title = "Tech Monthly", Issue =
"October 2024" };
            magazine.Borrow();
            magazine.Return();
        }
        else
        {
            Console.WriteLine("Unknown document type.");
        }
    }
}

// Chương trình chính
public class Program
{
    public static void Main()
    {
        LibraryManager libraryManager = new LibraryManager();

        // Quản lý sách
        libraryManager.ManageLibrary("book");

        // Quản lý tạp chí
        libraryManager.ManageLibrary("magazine");

        // Quản lý tài liệu không xác định
        libraryManager.ManageLibrary("newspaper");
```

```
    }
}
```

Với cách làm trên, nếu bạn cần thêm loại tài liệu mới như báo (Newspaper) hoặc cập nhật logic cho việc mượn/trả tài liệu, bạn sẽ phải chỉnh sửa nhiều phần mã nguồn ở mỗi lớp, điều này rất dễ gây lỗi và mất nhiều thời gian.

Khái niệm cơ bản về Design Pattern

Design Pattern (Mẫu thiết kế) là những giải pháp được chuẩn hóa để giải quyết các vấn đề thường gặp trong thiết kế phần mềm. Nó không phải là những đoạn mã cụ thể, mà là các mô hình, giải pháp tổ chức mã nguồn đã được kiểm chứng. Việc áp dụng Design Pattern giúp bạn tạo ra mã nguồn có cấu trúc rõ ràng, dễ bảo trì và mở rộng khi hệ thống phát triển.



Trong ví dụ ban đầu, thay vì viết lại hành vi mượn và trả cho từng loại tài liệu, bạn có thể sử dụng Design Pattern "Strategy" (một mẫu thiết kế hướng chiến lược) để tách rời hành vi này khỏi lớp cụ thể, giúp mã nguồn linh hoạt và dễ mở rộng hơn.

Ví dụ áp dụng một design pattern phù hợp

Dưới đây là cách giải quyết vấn đề trên bằng cách áp dụng Strategy Pattern:

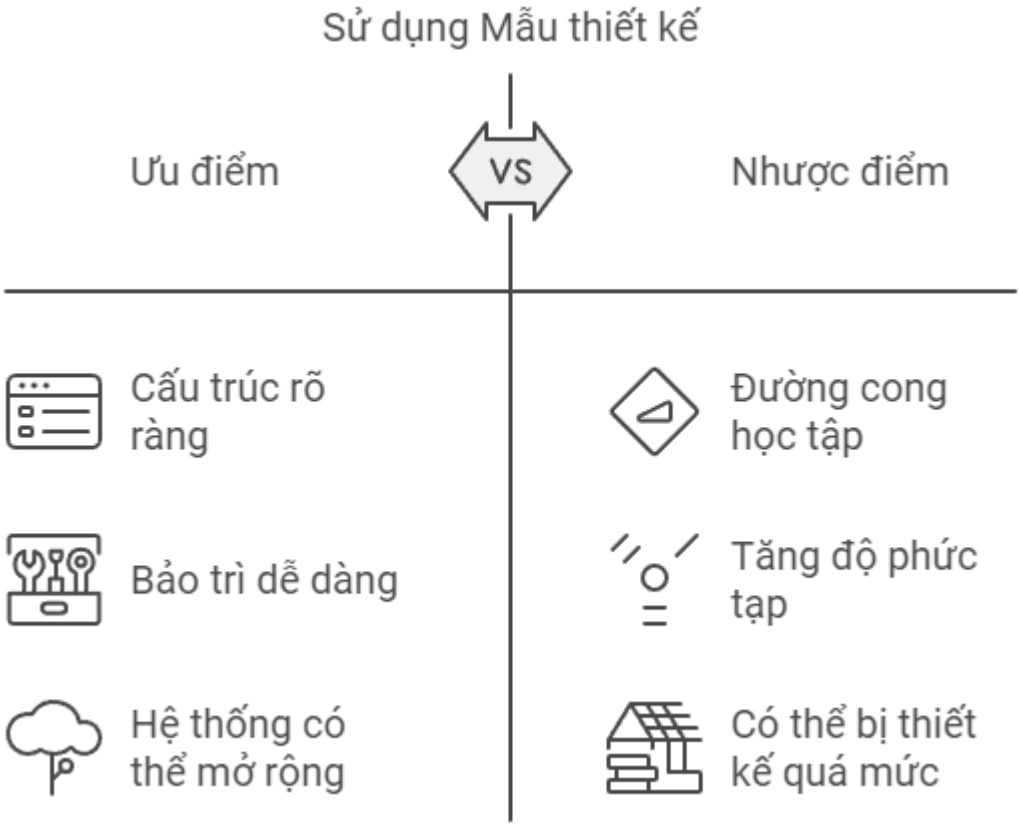
```
// Định nghĩa interface chung cho hành vi mượn tài liệu
public interface IBorrowable // Interface đại diện cho hành vi mượn tài liệu
{
    void Borrow();
}

// Cài đặt hành vi mượn cho sách (Book)
public class BookBorrow : IBorrowable
{
    public void Borrow()
    {
        Console.WriteLine("Borrowing a book...");
    }
}

// Cài đặt hành vi mượn cho tạp chí (Magazine)
```

```
public class MagazineBorrow : IBorrowable
{
    public void Borrow()
    {
        Console.WriteLine("Borrowing a magazine...");
    }
}

// Lớp Document đại diện cho việc quản lý thư viện, sử dụng interface IBorrowable
để quản lý hành vi mượn
public class LibraryManager
{
    private readonly IBorrowable _borrowable;

    public LibraryManager(IBorrowable borrowable)
    {
        _borrowable = borrowable;
    }

    //giả sử một hàm sử dụng hàm mượn
    public void BorrowDocument()
    {
        /**
        * Các logic khác
        */

        _borrowable.Borrow();

        /**
        * Các logic khác
        */
    }
}

// Sử dụng trong chương trình
class Program
{
    static void Main(string[] args)
    {
        LibraryManager book = new LibraryManager(new BookBorrow()); // Tài liệu là
một quyển sách
        book.BorrowDocument(); // Output: Borrowing a book...

        LibraryManager magazine = new LibraryManager(new MagazineBorrow()); // Tài
liệu là một tạp chí
        magazine.BorrowDocument(); // Output: Borrowing a magazine...
    }
}
```

Trong ví dụ trên, hành vi mượn được tách biệt và đóng gói thành các lớp riêng biệt (`BookBorrow` và `MagazineBorrow`) dựa trên interface `IBorrowable`. Điều này giúp bạn dễ dàng mở rộng thêm các loại tài liệu

khác mà không cần thay đổi mã nguồn của lớp `Document`. Khi cần thêm loại tài liệu mới (như báo), bạn chỉ cần tạo một lớp cài đặt mới mà không cần chỉnh sửa bất kỳ đoạn mã nào khác.

Ưu điểm và nhược điểm



Ưu điểm:

1. **Cấu trúc rõ ràng**:
    - Design Patterns cung cấp các giải pháp có cấu trúc rõ ràng, nhất quán cho các vấn đề phổ biến trong lập trình. Khi áp dụng hợp lý mã nguồn sẽ được tổ chức một cách rành mạch, dễ hiểu và theo các nguyên tắc thiết kế tốt. Điều này giúp các lập trình viên dễ dàng nắm bắt cấu trúc của dự án, làm cho mã dễ đọc và dễ duy trì hơn.
2. **Bảo trì dễ dàng**:
    - Nhờ vào việc phân tách trách nhiệm và tổ chức mã theo nguyên tắc hướng đối tượng, các Design Patterns giúp cho việc bảo trì hệ thống trở nên dễ dàng hơn. Khi cần thay đổi hoặc sửa chữa, các lập trình viên có thể nhanh chóng xác định vị trí cần chỉnh sửa mà không ảnh hưởng nhiều đến các phần khác của hệ thống, giảm thiểu rủi ro khi bảo trì.
3. **Khả năng mở rộng**:
    - Design Patterns giúp xây dựng hệ thống với khả năng mở rộng tốt hơn. Các mẫu như **Factory**, **Strategy**, hay **Observer** cho phép thêm tính năng mới hoặc thay đổi hành vi của hệ thống mà không cần phải thay đổi cấu trúc hiện tại. Điều này giúp hệ thống linh hoạt hơn và dễ dàng đáp ứng các yêu cầu mới khi mở rộng hoặc nâng cấp trong tương lai.

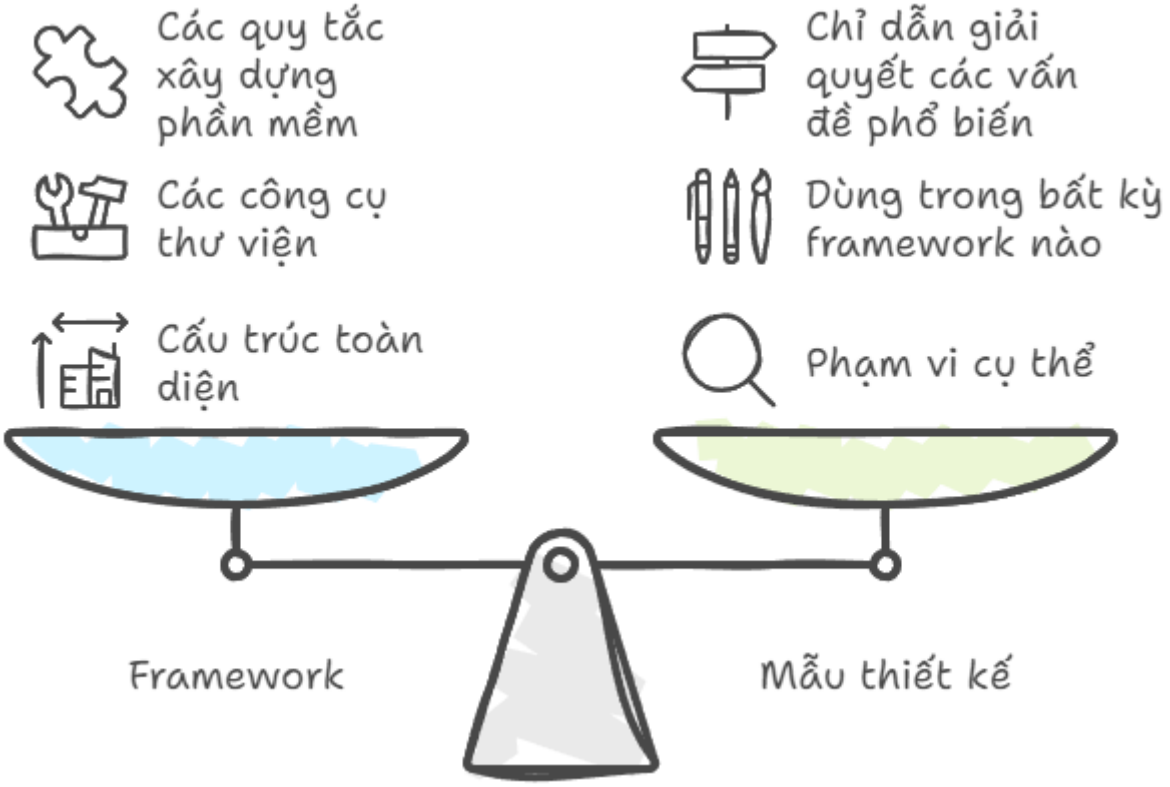
Nhược điểm:

1. **Đường cong học tập**:
    - Sử dụng các Design Patterns đòi hỏi lập trình viên phải có kiến thức và kinh nghiệm về lập trình hướng đối tượng, các nguyên tắc thiết kế phần mềm, và hiểu sâu các mẫu thiết kế. Với các lập trình viên mới, điều này có thể gây ra một đường cong học tập dốc, làm chậm quá trình phát triển ban đầu vì họ phải dành thời gian để học và hiểu cách các mẫu thiết kế hoạt động.
2. **Tăng độ phức tạp**:
    - Một số Design Patterns có thể làm cho mã trở nên phức tạp hơn với nhiều lớp và thành phần trừu tượng không cần thiết. Ví dụ, các mẫu như **Abstract Factory** hay **Decorator** có thể làm cho hệ thống trở nên khó đọc và bảo trì hơn, đặc biệt khi vấn đề thực tế không quá phức tạp. Điều này có thể làm tăng sự phức tạp trong việc theo dõi luồng xử lý hoặc debug.
3. **Thiết kế quá mức (Over-engineering)**:
    - Lạm dụng Design Patterns có thể dẫn đến **over-engineering** – nghĩa là bạn áp dụng các mẫu thiết kế cho những vấn đề không đủ phức tạp, dẫn đến việc hệ thống trở nên cồng kềnh và khó quản lý. Thay vì giải quyết vấn đề một cách đơn giản, việc thiết kế quá mức sẽ làm giảm hiệu suất phát triển và bảo trì, do có quá nhiều cấu trúc không cần thiết.

Sự khác biệt giữa Design Pattern và các khái niệm thiết kế khác

1. **Design Pattern vs. Algorithm (Thuật toán):**

    - **Algorithm**: Là một quy trình hoặc tập hợp các bước cụ thể để giải quyết một bài toán, chẳng hạn như thuật toán sắp xếp hoặc tìm kiếm. Thuật toán thường tập trung vào cách xử lý dữ liệu.
    - **Design Pattern**: Là một phương pháp thiết kế mã nguồn, tập trung vào cách tổ chức và cấu trúc mã để giải quyết các vấn đề thiết kế phần mềm một cách hiệu quả.
2. **Design Pattern vs. Framework:**

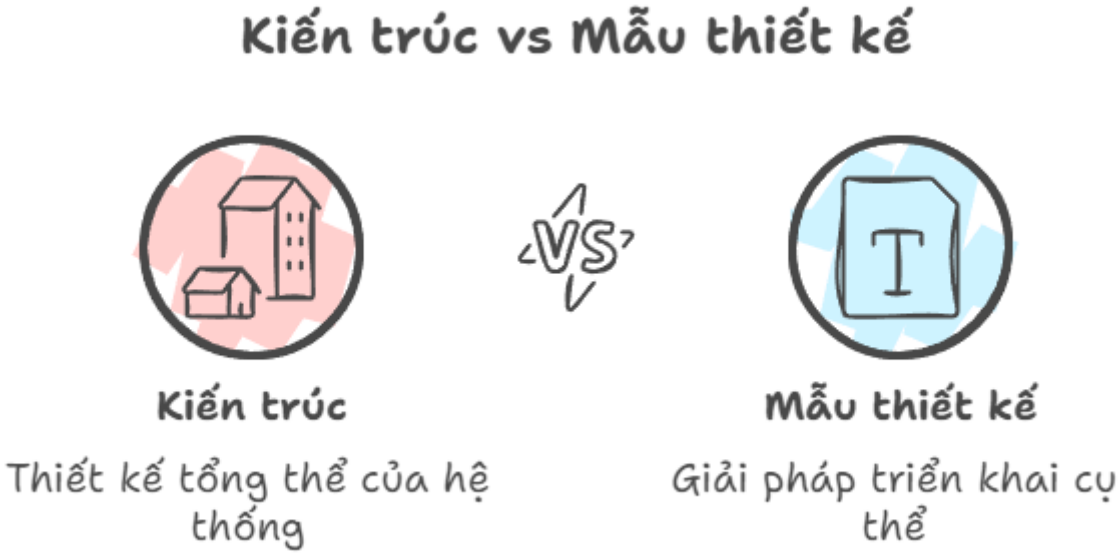


- **Framework**: Là một bộ khung phát triển ứng dụng, cung cấp các công cụ, thư viện và quy tắc để xây dựng phần mềm, ví dụ như ASP.NET hay Angular. Framework hướng dẫn cách xây dựng toàn bộ ứng dụng.
- **Design Pattern**: Là các mẫu giải pháp nhỏ hơn, độc lập, có thể được sử dụng bên trong bất kỳ framework nào để giải quyết các vấn đề cụ thể về thiết kế.

3. **Design Pattern vs. Best Practices:**

- **Best Practices**: Là những quy tắc hoặc phương pháp tốt nhất được chứng minh là hiệu quả, như viết mã dễ hiểu, sử dụng kiểm thử đơn vị. Đây là các nguyên tắc hướng dẫn giúp cải thiện chất lượng mã nguồn.
- **Design Patterns**: Là các mẫu thiết kế cụ thể để giải quyết các vấn đề thường gặp, giúp hệ thống phần mềm dễ bảo trì và mở rộng.

4. **Design Pattern vs. Architecture (Kiến trúc):**



- **Architecture**: Là thiết kế tổng thể của hệ thống phần mềm, bao gồm cách các thành phần tương tác với nhau (ví dụ: kiến trúc Microservices, MVC).
- **Design Pattern**: Tập trung vào các vấn đề cụ thể ở mức thấp hơn trong kiến trúc, giúp thực hiện chi tiết các thành phần của hệ thống một cách hiệu quả.

3.1.2. Lịch Sử Phát Triển của Design Patterns

Nguồn gốc và sự phát triển qua các phiên bản

Design patterns ban đầu xuất phát từ kiến trúc xây dựng. Ý tưởng về design patterns được nhà kiến trúc sư Christopher Alexander giới thiệu vào thập niên 1970. Alexander đã viết bộ sách nổi tiếng "A Pattern Language," trong đó ông miêu tả các mô hình (patterns) lặp lại mà ông quan sát thấy trong quá trình thiết kế đô thị và kiến trúc. Những mô hình này nhằm giải quyết các vấn đề thiết kế thường gặp và mang lại các giải pháp hiệu quả.

Sang lĩnh vực phần mềm, các nhà khoa học máy tính nhận thấy rằng nhiều vấn đề thiết kế phần mềm cũng có tính chất lặp lại tương tự. Đặc biệt là vào thập niên 1990, khái niệm design patterns trong phần mềm được chính thức hóa qua cuốn sách "Design Patterns: Elements of Reusable Object-Oriented Software" (1994) của nhóm tác giả gồm Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides, thường được gọi là **Gang of Four (GoF)**.

Cuốn sách này giới thiệu 23 design patterns phổ biến trong lập trình hướng đối tượng, được phân loại thành ba nhóm chính: **Creational Patterns** (nhóm tạo lập), **Structural Patterns** (nhóm cấu trúc), và **Behavioral Patterns** (nhóm hành vi). Đây là cột mốc quan trọng giúp các lập trình viên dễ dàng nhận diện và áp dụng các giải pháp thiết kế phần mềm tái sử dụng.

Các công trình tiêu biểu

- **Christopher Alexander**: Nguồn gốc của khái niệm patterns bắt đầu với Alexander qua các công trình về kiến trúc, đáng chú ý nhất là cuốn sách "A Pattern Language" (1977). Mặc dù không trực tiếp liên quan đến lập trình, nhưng tư duy của ông đã truyền cảm hứng cho các lĩnh vực khác, bao gồm phần mềm.

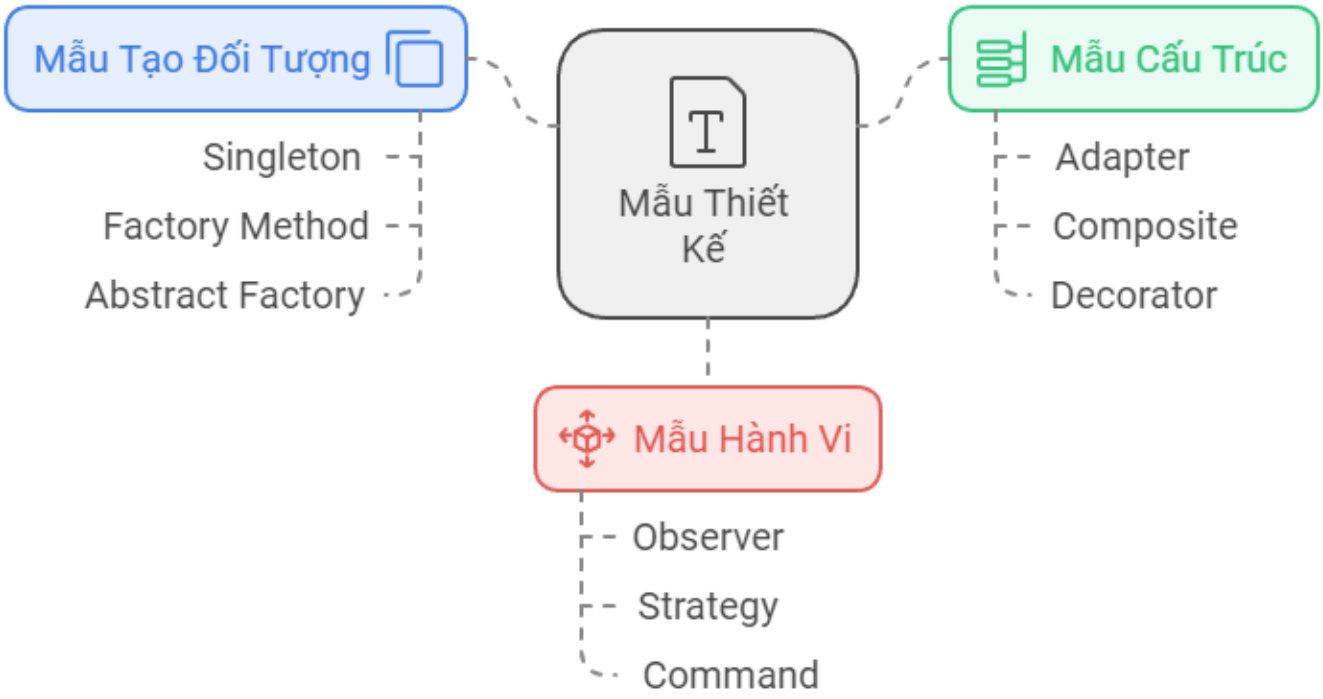
- **Gang of Four (GoF)**: Cuốn sách "Design Patterns: Elements of Reusable Object-Oriented Software" (1994) của nhóm GoF được xem là công trình kinh điển, chính thức hóa khái niệm design patterns trong phần mềm. Các thành viên nhóm GoF bao gồm:
    - **Erich Gamma**
    - **Richard Helm**
    - **Ralph Johnson**
    - **John Vlissides**

  Cuốn sách của họ đã định hình nền tảng cho cách tiếp cận thiết kế phần mềm hướng đối tượng và vẫn giữ nguyên tầm quan trọng trong lập trình hiện đại.

Ngoài GoF, còn có nhiều công trình và tác giả khác đã đóng góp vào sự phát triển của design patterns, như **Martin Fowler** với các cuốn sách về **Enterprise Application Patterns** hay **Refactoring**, và **Robert C. Martin** với các nguyên tắc thiết kế SOLID và clean architecture.

Qua thời gian, design patterns không ngừng phát triển và thích ứng với các xu hướng công nghệ mới như lập trình chức năng, microservices và các mô hình kiến trúc hiện đại khác.

3.2. Phân Loại Design Patterns



Design patterns được chia thành ba nhóm chính:

- **Creational Patterns (Mẫu Tạo Đối Tượng)**: Tập trung vào cách tạo đối tượng, giúp tách rời quá trình khởi tạo đối tượng khỏi phần còn lại của hệ thống, đảm bảo sự linh hoạt và tái sử dụng.
- **Structural Patterns (Mẫu Cấu Trúc)**: Xử lý cách tổ chức và cấu trúc các đối tượng, giúp xây dựng hệ thống phức tạp từ các thành phần đơn giản theo một cách hiệu quả và có tổ chức.
- **Behavioral Patterns (Mẫu Hành Vi)**: Định nghĩa cách các đối tượng giao tiếp và tương tác với nhau, tối ưu hóa các quy trình điều phối và trao đổi thông tin trong hệ thống.

3.3. Giới thiệu một số Design Patterns hay gặp

Decorator Pattern

Định nghĩa

Decorator Pattern cho phép mở rộng hoặc thêm tính năng vào đối tượng một cách linh hoạt mà không làm thay đổi lớp ban đầu. Mẫu này giúp tránh tạo ra nhiều lớp con để mở rộng chức năng. Các đối tượng được bọc (decorated) một cách tuần tự, mỗi lớp decorator bổ sung thêm một chức năng.

Ưu điểm

- Tăng tính mở rộng của hệ thống: Dễ dàng thêm mới hoặc thay đổi chức năng mà không làm thay đổi lớp ban đầu.
- **Open/Closed Principle**: Cho phép mở rộng hệ thống mà không sửa đổi mã hiện có.

Nhược điểm

- Nếu sử dụng nhiều lớp decorator, có thể tạo ra chuỗi dài các đối tượng bọc, khiến mã nguồn khó theo dõi.

Ví dụ trong thương mại điện tử

Tình huống: Trong hệ thống thương mại điện tử, bạn có sản phẩm cơ bản, nhưng khách hàng có thể thêm dịch vụ gói quà và bảo hiểm. Ban đầu hệ thống chỉ có sản phẩm cơ bản, sau đó cần thêm chức năng gói quà mà không sửa đổi logic sản phẩm.

Trước khi áp dụng Decorator Pattern:
Hệ thống ban đầu chỉ có lớp sản phẩm đơn giản:

```
public class Product
{
    public string Name { get; set; }
    public double Price { get; set; }

    public Product(string name, double price)
    {
        Name = name;
        Price = price;
    }

    public double GetTotalPrice() => Price;
}
```

Khi khách hàng yêu cầu thêm gói quà, bạn phải thêm logic vào lớp `Product`:

```
public double GetTotalPrice(bool includeGiftWrap)
{
```

```
    double totalPrice = Price;
    if (includeGiftWrap)
    {
        totalPrice += 5; // Giá gói quà
    }
    return totalPrice;
}
```

Nhược điểm:

- Mã trở nên phức tạp khi có thêm nhiều dịch vụ (ví dụ: bảo hiểm, giao hàng nhanh).
- Vi phạm nguyên tắc **Single Responsibility Principle**, vì `Product` phải lo cả về tính năng bổ sung.

Sau khi áp dụng Decorator Pattern:
Thay vì thay đổi lớp `Product`, ta sử dụng **Decorator Pattern** để thêm chức năng gói quà và bảo hiểm mà không làm thay đổi lớp gốc.

```
// Giao diện sản phẩm
public interface IProduct
{
    string GetName();
    double GetPrice();
}

// Lớp sản phẩm cơ bản
public class BasicProduct : IProduct
{
    private string _name;
    private double _price;

    public BasicProduct(string name, double price)
    {
        _name = name;
        _price = price;
    }

    public string GetName() => _name;
    public double GetPrice() => _price;
}

// Decorator cơ bản
public abstract class ProductDecorator : IProduct
{
    protected IProduct _product;

    public ProductDecorator(IProduct product)
    {
        _product = product;
    }

    public virtual string GetName() => _product.GetName();
```

```
    public virtual double GetPrice() => _product.GetPrice();
}

// Decorator gói quà
public class GiftWrapDecorator : ProductDecorator
{
    public GiftWrapDecorator(IProduct product) : base(product) { }

    public override double GetPrice() => base.GetPrice() + 5; // Thêm giá gói quà
}

// Decorator bảo hiểm
public class InsuranceDecorator : ProductDecorator
{
    public InsuranceDecorator(IProduct product) : base(product) { }

    public override double GetPrice() => base.GetPrice() + 20; // Thêm giá bảo
hiểm
}
```

Ví dụ sử dụng:

```
public class Program
{
    public static void Main()
    {
        // Sản phẩm cơ bản
        IProduct product = new BasicProduct("Laptop", 1000);

        // Thêm gói quà
        product = new GiftWrapDecorator(product);
        Console.WriteLine(product.GetPrice()); // Output: 1005

        // Thêm bảo hiểm
        product = new InsuranceDecorator(product);
        Console.WriteLine(product.GetPrice()); // Output: 1025
    }
}
```

Kết hợp với Dependency Injection trong ASP.NET Core:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IProduct, BasicProduct>();
    services.AddScoped<IProduct, GiftWrapDecorator>();
    services.AddScoped<IProduct, InsuranceDecorator>();
}
```

Khi nào nên sử dụng Decorator Pattern:

1. **Khi cần mở rộng tính năng của đối tượng một cách linh hoạt**:
    - Nếu hệ thống của bạn có các đối tượng cần được bổ sung các tính năng hoặc hành vi mới mà không cần thay đổi lớp gốc. **Decorator Pattern** cho phép bạn "bọc" đối tượng hiện tại với các lớp bổ sung để thêm hành vi mà không phải chỉnh sửa lớp gốc.

   Ví dụ: Trong hệ thống thương mại điện tử, bạn có thể thêm tính năng gói quà, bảo hiểm hoặc giao hàng nhanh cho một sản phẩm mà không cần thay đổi lớp `Product` gốc.

2. **Khi cần giữ nguyên sự đơn giản của lớp gốc**:
    - Khi lớp gốc đại diện cho một đối tượng cơ bản nhưng bạn không muốn làm lớp này trở nên phức tạp bởi nhiều chức năng bổ sung. Thay vì thêm các thuộc tính và phương thức mới vào lớp gốc, bạn có thể sử dụng **Decorator Pattern** để tách biệt các tính năng mở rộng.

   Ví dụ: Nếu lớp `Product` chỉ chứa các thuộc tính cơ bản như tên và giá, các tính năng như gói quà hoặc bảo hiểm có thể được thêm thông qua các lớp decorator mà không làm phức tạp lớp gốc.

3. **Khi bạn cần tuân thủ nguyên tắc Open/Closed Principle (OCP)**:
    - Nếu bạn muốn mở rộng chức năng của đối tượng mà không sửa đổi mã gốc, **Decorator Pattern** là một giải pháp lý tưởng vì bạn chỉ cần thêm các lớp decorator mà không thay đổi lớp ban đầu.

4. **Khi có nhiều hành vi hoặc tính năng bổ sung khác nhau**:
    - Nếu đối tượng có thể có nhiều hành vi bổ sung khác nhau và bạn muốn có khả năng kết hợp các hành vi này một cách linh hoạt mà không tạo ra quá nhiều lớp con. **Decorator Pattern** cho phép bạn kết hợp nhiều lớp decorator khác nhau để tạo ra các tính năng cụ thể.

   Ví dụ: Một sản phẩm có thể vừa có dịch vụ gói quà, vừa có bảo hiểm. Bạn có thể kết hợp hai lớp `GiftWrapDecorator` và `InsuranceDecorator` mà không phải tạo một lớp con mới cho từng tổ hợp tính năng.

5. **Khi bạn không muốn tạo nhiều lớp con**:
    - Nếu hệ thống yêu cầu nhiều biến thể của đối tượng với các tính năng bổ sung khác nhau, thay vì tạo nhiều lớp con cho mỗi biến thể, bạn có thể sử dụng **Decorator Pattern** để tránh sự bùng nổ số lượng lớp.

   Ví dụ: Nếu bạn có nhiều loại sản phẩm với nhiều dịch vụ bổ sung khác nhau, việc sử dụng decorator sẽ giúp bạn tránh việc tạo quá nhiều lớp sản phẩm chỉ để xử lý các dịch vụ khác nhau.

Không nên sử dụng Decorator Pattern trong các trường hợp sau:

- **Khi số lượng lớp decorator quá lớn**: Nếu bạn có quá nhiều lớp decorator và phải kết hợp chúng theo nhiều cách khác nhau, điều này có thể khiến mã trở nên khó quản lý và phức tạp.
- **Khi không cần mở rộng linh hoạt**: Nếu hệ thống không yêu cầu sự thay đổi linh hoạt các tính năng hoặc hành vi, sử dụng **Decorator Pattern** có thể gây ra sự phức tạp không cần thiết.

Strategy Pattern

Định nghĩa

Strategy Pattern tách các thuật toán ra thành các đối tượng riêng biệt, cho phép lựa chọn hoặc thay đổi thuật toán một cách linh hoạt. Mỗi thuật toán là một chiến lược riêng biệt và có thể thay đổi động mà không cần phải chỉnh sửa lớp sử dụng thuật toán đó.

Ưu điểm

- Tách biệt thuật toán ra khỏi đối tượng sử dụng thuật toán, giúp dễ bảo trì và mở rộng.
- Tuân thủ nguyên tắc **Open/Closed Principle**, cho phép thêm mới các thuật toán mà không sửa đổi mã gốc.

Nhược điểm

- Tạo ra nhiều lớp cho mỗi chiến lược.
- Đối tượng sử dụng thuật toán cần biết nhiều về các chiến lược khác nhau.

Ví dụ về tính điểm trong quản lý sinh viên

Tình huống:
Hệ thống quản lý sinh viên cần hỗ trợ nhiều cách tính điểm khác nhau, chẳng hạn tính điểm trung bình hoặc lấy điểm cao nhất. Ban đầu hệ thống chỉ có cách tính điểm trung bình, nhưng sau này yêu cầu thêm cách tính điểm cao nhất.

Không áp dụng Strategy Pattern:
Khi không sử dụng **Strategy Pattern**, mã nguồn sẽ phải xử lý trực tiếp các chiến lược trong lớp `Student`, gây ra sự phức tạp.

```
public class Student
{
    public string Name { get; set; }
    public List<double> Grades { get; set; }

    public double CalculateGrade(string strategy)
    {
        if (strategy == "average")
        {
            return Grades.Average();
        }
        else if (strategy == "highest")
        {
            return Grades.Max();
        }
        return 0;
    }
}
```

Nhược điểm:

- Lớp `Student` phải biết về tất cả các chiến lược tính điểm, vi phạm **Single Responsibility Principle**.
- Khó mở rộng khi cần thêm cách tính điểm mới.

Áp dụng Strategy Pattern:
Với **Strategy Pattern**, các chiến lược tính điểm được tách ra thành các lớp riêng biệt.

```
// Giao diện chiến lược tính điểm
public interface IGradeCalculationStrategy
{
    double CalculateGrade(List<double> grades);
}

// Chiến lược tính điểm trung bình
public class AverageGradeCalculation : IGradeCalculationStrategy
{
    public double CalculateGrade(List<double> grades)
    {
        return grades.Average();
    }
}

// Chiến lược tính điểm cao nhất
public class HighestGradeCalculation : IGradeCalculationStrategy
{
    public double CalculateGrade(List<double> grades)
    {
        return grades.Max();
    }
}
```

Lớp `Student` chỉ cần nhận chiến lược tính điểm thông qua **Dependency Injection**:

```
public class Student
{
    public string Name { get; set; }
    public List<double> Grades { get; set; }
    private IGradeCalculationStrategy _gradeStrategy;

    public Student(string name, List<double> grades, IGradeCalculationStrategy
gradeStrategy)
    {
        Name = name;
        Grades = grades;
        _gradeStrategy = gradeStrategy;
    }

    public double CalculateGrade()
    {
```

```
        return _gradeStrategy.CalculateGrade(Grades);
    }
}
```

Ví dụ sử dụng:

```
public class Program
{
    public static void Main()
    {
        List<double> grades = new List<double> { 85, 90, 78 };

        // Sử dụng chiến lược tính điểm trung bình
        IGradeCalculationStrategy averageStrategy = new AverageGradeCalculation();
        Student student = new Student("John", grades, averageStrategy);
        Console.WriteLine(student.CalculateGrade()); // Output: 84.33

        // Sử dụng chiến lược tính điểm cao nhất
        IGradeCalculationStrategy highestStrategy = new HighestGradeCalculation();
        student = new Student("John", grades, highestStrategy);
        Console.WriteLine(student.CalculateGrade()); // Output: 90
    }
}
```

Kết hợp với Dependency Injection trong ASP.NET Core:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IGradeCalculationStrategy, AverageGradeCalculation>();
    services.AddScoped<IGradeCalculationStrategy, HighestGradeCalculation>();
}
```

Khi nào nên sử dụng Strategy Pattern:

1. **Khi có nhiều giải pháp khác nhau cho cùng một vấn đề và muốn lựa chọn linh hoạt giải pháp**:
    - Nếu có nhiều cách khác nhau để thực hiện một tác vụ và bạn muốn thay đổi cách thực hiện mà không phải sửa đổi mã nguồn của lớp đang sử dụng tác vụ đó. **Strategy Pattern** giúp bạn dễ dàng chuyển đổi giữa các chiến lược khác nhau mà không làm thay đổi đối tượng sử dụng chiến lược.

   Ví dụ: Trong hệ thống quản lý sinh viên, nếu bạn có nhiều cách tính điểm (tính điểm theo hệ số, tính điểm theo thang điểm 10, thang điểm 4) thì mỗi cách tính điểm sẽ là một "strategy" khác nhau. Bạn có thể dễ dàng thay đổi chiến lược tính điểm khi cần mà không phải sửa đổi lớp `Student` hoặc lớp logic xử lý điểm.

2. **Khi bạn muốn tách riêng các thuật toán cụ thể**:

- Nếu bạn có nhiều thuật toán hoặc quy trình cần được áp dụng trong các tình huống khác nhau, việc sử dụng **Strategy Pattern** giúp bạn tách riêng các thuật toán này và giữ cho mã nguồn rõ ràng, dễ bảo trì.

Ví dụ: Trong hệ thống thương mại điện tử, bạn có thể có nhiều chiến lược vận chuyển khác nhau (vận chuyển nhanh, vận chuyển thường, vận chuyển quốc tế). **Strategy Pattern** cho phép bạn tách các thuật toán này thành các lớp riêng, giúp mã dễ hiểu và dễ dàng thay đổi cách vận chuyển.

3. **Khi muốn tuân thủ nguyên tắc Open/Closed Principle (OCP)**:
   - Nếu bạn muốn thêm các chiến lược mới mà không sửa đổi mã hiện tại, **Strategy Pattern** cho phép bạn dễ dàng mở rộng các thuật toán hay hành vi mới bằng cách tạo thêm các lớp chiến lược mà không cần phải thay đổi các lớp hiện có.

   Ví dụ: Khi muốn thêm các loại khuyến mãi mới cho sản phẩm (như giảm giá phần trăm, giảm giá trực tiếp), bạn có thể thêm các chiến lược khuyến mãi mà không phải sửa đổi logic tính giá hiện tại.

4. **Khi cần thay đổi thuật toán hoặc hành vi một cách linh hoạt trong runtime**:
   - **Strategy Pattern** hữu ích khi bạn cần thay đổi thuật toán hoặc hành vi trong khi chương trình đang chạy (runtime), thay vì phải xác định trước khi biên dịch.

   Ví dụ: Trong hệ thống quản lý sinh viên, bạn có thể thay đổi chiến lược tính điểm tùy theo quy định của từng kỳ học, mà không cần phải thay đổi mã nguồn hay tái biên dịch hệ thống.

5. **Khi có nhu cầu loại bỏ sự phức tạp trong việc sử dụng các điều kiện `if-else` hoặc `switch-case`**:
   - Nếu bạn đang sử dụng nhiều điều kiện để chọn một thuật toán cụ thể (ví dụ: nhiều `if-else` hoặc `switch-case`), **Strategy Pattern** giúp bạn loại bỏ những đoạn mã này bằng cách chuyển các điều kiện thành các lớp chiến lược riêng.

   Ví dụ: Thay vì sử dụng `if-else` để quyết định cách tính giá sản phẩm (giá gốc, giảm giá phần trăm, giảm giá trực tiếp), bạn có thể sử dụng **Strategy Pattern** để tách biệt từng chiến lược tính giá và lựa chọn chiến lược phù hợp dựa trên loại sản phẩm.

Không nên sử dụng Strategy Pattern trong các trường hợp sau:

- **Khi bạn chỉ có một thuật toán hoặc hành vi cố định**: Nếu không có nhu cầu thay đổi hay thay thế các thuật toán hay hành vi khác nhau, việc sử dụng **Strategy Pattern** có thể gây ra sự phức tạp không cần thiết.
- **Khi sự khác biệt giữa các thuật toán hoặc hành vi không đáng kể**: Nếu các chiến lược khác nhau chỉ có sự khác biệt rất nhỏ, có thể việc tách thành các lớp chiến lược sẽ gây ra sự thừa thãi và làm tăng độ phức tạp của hệ thống.
- **Khi không cần thay đổi chiến lược linh hoạt**: Nếu hệ thống của bạn không yêu cầu thay đổi chiến lược trong runtime và các chiến lược có thể được xác định trước khi biên dịch, sử dụng **Strategy Pattern** có thể không cần thiết.