Decorator Design Pattern

* Mục đích: là mẫu thiết kế thuộc nhóm cấu trúc (structural pattern), chô phép thêm chức năng mới vào một đối tượng một cách linh hoạt mà không thay đổi cấu trúc bên trong của nó.
* Vấn đề: Khi muốn mở rộng hành vi của một đối tượng mà không muốn haowjc không thể sử dụng kế thừa (do tính cứng và ảnh hưởng đến toàn bộ lớp), Decorator cung cấp giải pháp bằng cách bao bọc đối tượng gốc trong các lớp “trang trí” (decotator) để bổ sung hành vi mới.
* Giải pháp: Sử dụng một giao diện chung cho đối tượng gốc và các decorator. Mỗi decorator chưa một tham chiếu đến đối tượng được bao bọc và triển khai các phương thức bằng cách gọi đến đối tượng đó, đồng thời thêm hành vi mới.
* Ví dụ thực tế: Trang trí cây thông Noel bằng đèn, vòng hoa, kẹo,… Mỗi món đồ trang trí là một decorator, thêm vào cây thông mà không làm thay đổi cây thông gốc.
* Cấu trúc UML:
  + Component: Giao diện chung cho đối tượng gốc và các decorator.
  + ConcreteComponent: Triển khai cụ thể của Component.
  + Decorator: Lớp trừu tượng triển khai Component và chưa tham chiếu đến một Component.
  + ConcreteDecorator: Mở rộng Decorator để thêm hành vi mới.
* Ưu điểm:
  + Tuân thủ nguyên lý Mở - Đóng (Open/Closed Principle): mở rộng hành vi mà không sửa sổi mã nguồn hiện có.
  + Linh hoạt trong việc kết hợp các decorator để tạo ra hành vi phức tạp.
  + Tránh việc tạo ra nhiều lớp con để mở rộng chức năng.
* Nhược điểm:
  + Có thể tạo ra nhiều lớp nhỏ, gây phức tạp trong quản lý.
  + Khó khăn trong việc gỡ lỗi do nhiều lớp bao bọc lẫn nhau.
* Code Demo:

// Giao diện chung

public interface INotifier

{

void Send(string message);

}

// Đối tượng gốc

public class EmailNotifier : INotifier

{

public void Send(string message)

{

Console.WriteLine($"Sending Email: {message}");

}

}

// Lớp Decorator cơ sở

public abstract class NotifierDecorator : INotifier

{

protected INotifier \_notifier;

protected NotifierDecorator(INotifier notifier)

{

\_notifier = notifier;

}

public virtual void Send(string message)

{

\_notifier.Send(message);

}

}

// Decorator cụ thể - Gửi SMS

public class SmsNotifier : NotifierDecorator

{

public SmsNotifier(INotifier notifier) : base(notifier) { }

public override void Send(string message)

{

base.Send(message);

Console.WriteLine($"Sending SMS: {message}");

}

}

// Decorator cụ thể - Gửi Push Notification

public class PushNotifier : NotifierDecorator

{

public PushNotifier(INotifier notifier) : base(notifier) { }

public override void Send(string message)

{

base.Send(message);

Console.WriteLine($"Sending Push Notification: {message}");

}

}

// Sử dụng

class Program

{

static void Main()

{

INotifier notifier = new EmailNotifier();

notifier = new SmsNotifier(notifier);

notifier = new PushNotifier(notifier);

notifier.Send("Hello World!");

}

}

Flyweight Design Parttern

* Mục đích: Là mẫu thiết kế thuộc nhóm cấu trúc, nhằm giảm thiểu việc sử dụng bộ nhớ bằng cách chia sẻ dữ liệu chung giữa nhiều đối tượng tương tự nhau.
* Vấn đề: Khi ứng dụng cần tạo ra số lượng lớn các đối tượng có nhiều phần dữ liệu giống nhau, việc tạo riêng biệt từng đối tượng sẽ tiêu tốn nhiều bộ nhớ và ảnh hưởng đến hiệu suất.
* Giải pháp: Chia trạng thái không thay đổi và có thể chia sẻ giữa các đối tượng.
  + Intrinsic State: Phần trạng thái không thay đổi và có thể chia sẻ giữa các đối tượng.
  + Extrinsic State: Phần trạng thái thay đổi theo ngữ cảnh và được cung cấp bởi client khi sử dụng đối tượng.
  + Việc sử dụng một Factory đề quản lý và tái sử dụng các đối tượng có Instrinsic State giống nhau, giúp giảm số lượng đối tượng cần tạo và tiết kiệm bộ nhớ.
* Cấu trúc UML
  + Flyweight: Giao diện hoặc trừu tượng định nghĩa các phương thức nhận Extrinsic State từ client.
  + ConcreteFlyweight: Triển khai Flyweight, lưu trữ Intrinsic Statr và thực hiện các thao tác với Extrinsic State được cung cấp.
  + FlyweightFactory: Quản lý và cung cấp các đối tượng Flyweight, đảm bảo tái sử dụng các đối tượng có Intrinsic State giống nhau.
  + Client: Sử dụng FlyweightFactory để lấy Flyweight và cung cấp Extrinsic State khi gọi các phương thức.
* Ưu điểm:
  + Giảm thiểu việc sử dụng bộ nhớ bằng cách chia sẻ dữ liệu chung giữa các đối tượng.
  + Tăng hiệu suất ứng dụng khi cần xử lý số lượng lớn đối tượng tương tự nhau.
  + Tách biệt rõ ràng giữa trạng thái có thể chia sẻ và trạng thái phụ thuộc vào ngữ cảnh.
* Nhược điểm:
  + Tăng độ phức tạp trong việc quản lý trạng thái Extrinsic, yêu cầu client cung cấp đầy đủ thông tin khi sử dụng đối tượng.
  + Không phù hợp khii các đối tượng có nhiều trạng thái thay đổi và không thể chia sẻ.
* Code demo:

// Flyweight interface

public interface IShape

{

void Draw(int x, int y);

}

// ConcreteFlyweight

public class Circle : IShape

{

private readonly string \_color; // Intrinsic State

public Circle(string color)

{

\_color = color;

}

public void Draw(int x, int y) // Extrinsic State

{

Console.WriteLine($"Drawing {\_color} circle at ({x}, {y})");

}

}

// FlyweightFactory

public class ShapeFactory

{

private readonly Dictionary<string, IShape> \_shapes = new();

public IShape GetCircle(string color)

{

if (!\_shapes.ContainsKey(color))

{

\_shapes[color] = new Circle(color);

}

return \_shapes[color];

}

}

// Client

class Program

{

static void Main()

{

var factory = new ShapeFactory();

var colors = new[] { "Red", "Green", "Blue" };

var random = new Random();

for (int i = 0; i < 10; i++)

{

var color = colors[random.Next(colors.Length)];

var circle = factory.GetCircle(color);

int x = random.Next(100);

int y = random.Next(100);

circle.Draw(x, y);

}

}

}