Composite Report

* Mục đích: Cho phép cấu trúc các đối tượng thành dạng cây và làm việc với chúng như thể chúng là các đối tượng đơn lẻ.
* Vấn đề ban đầu: Khi chúng ta có các đối tượng như “Product” và “Box”, trong đó “Box” có thể chứ nhiều “Product” hoặc các “Box” nhỏ hơn, việc tính toán tổng giá trị trở nên phức tạp do cấu trúc lồng nhau.
* Giải phép xử lý: Ta có thể áp dụng mẫu thiết kế Composite bằng cách:
  + Định nghĩa một giao diện chung Component với phương thức getPrice()
  + Product triển khai getPrice() để trả về giá của chính nó.
  + Box triển khai getPrice() bằng cách duyệt qua các phần tử con và cộng dồn kết quả.

Điều này cho phép xử lý đệ quy qua toàn bộ cấu trúc cây mà không cần quan tâm đến loại cụ thể của từng phần tử.

* Ví dụ thực tế: Cấu trúc quân độ với các cấp bậc như: Quân đội, sư đoàn, lữ đoàn, tiểu đoàn, trung đội, binh sĩ. Mệnh lệch truyền từ cấp cap nhất xuống từng cấp dưới.
* Code demo:
  + Component: Giao diện chung cho cả Leaf và Composite
  + Leaf: Thành phần không có con
  + Composite: Thành phần có thể chứ các Component khác
  + Client: Làm việc với các Component thông qua giao diện chung

using System;

using System.Collections.Generic;

// Component

public interface IItem

{

int GetPrice();

}

// Leaf

public class Product : IItem

{

private int price;

public Product(int price) => this.price = price;

public int GetPrice() => price;

}

// Composite

public class Box : IItem

{

private List<IItem> items = new List<IItem>();

public void Add(IItem item) => items.Add(item);

public int GetPrice()

{

int total = 0;

foreach (var item in items)

total += item.GetPrice();

return total;

}

}

// Client

class Program

{

static void Main()

{

var p1 = new Product(100);

var p2 = new Product(200);

var box1 = new Box();

box1.Add(p1);

box1.Add(p2);

var p3 = new Product(300);

var box2 = new Box();

box2.Add(box1);

box2.Add(p3);

Console.WriteLine("Total Price: " + box2.GetPrice()); // Output: 600

}

}

Adapter Report

* Mục đích: Chuyển đổi giao diện của một lớp thành một giao diện khác mà client mong đợi. Adapter cho phép các lớp với giao diện không tương thích cùng làm việc với nhau.
* Vấn đề ban đầu: Khi ta muốn sử dụng một thành phần có sẵn nhưng giao diện của nó không phù hợp với hệ thống hiện tại.
* Giải pháp: Tạo một lớp Adapter đóng vai trò là cầu nối giữa client và thành phần cần thích nghi. Adapter sẽ chuyển đổi các lời gọi phương thức từ client sang định dạng và adaptee hiểu được.
* Cấu trúc thực tế:
  + Client: Thành phần sử dụng giao diện mục tiêu.
  + Target: Giao diện mà client mong đợi.
  + Adaptee: Lớp có giao diện không tương thích cần được thích nghi.
  + Adapter: Lớp triển khai giao diện Target và chuyển đổi các lời gọi đến Adaptee.

using System;

// Target interface

public interface ITarget

{

void Request();

}

// Adaptee class

public class Adaptee

{

public void SpecificRequest()

{

Console.WriteLine("Called SpecificRequest()");

}

}

// Adapter class

public class Adapter : ITarget

{

private readonly Adaptee \_adaptee;

public Adapter(Adaptee adaptee)

{

\_adaptee = adaptee;

}

public void Request()

{

\_adaptee.SpecificRequest();

}

}

// Client code

class Program

{

static void Main()

{

ITarget target = new Adapter(new Adaptee());

target.Request(); // Output: Called SpecificRequest()

}

}

Bridge Report

* Mục đích: Tách rời phần trừu tượng (abstraction) khỏi phần triển khai (implementation) để cả hai có thể phát triển độc lập.
* Vấn đề: Khi sử dụng kế thừa để tạo ra các lớp con cho từng kết hợp của trừu tượng và khiển khai, số lượng lớp có thể tăng lên theo cấp số nhân. Điều này dẫn đến hệ thống trở nên cồng kềnh và khó bảo trì.
* Giải pháp: Bridge Pattern đề xuất phân tách hệ thống thành 2 phần độc lập:
  + Abstraction: Định nghĩa giao diện trừu tượng và duy trì một tham chiếu đối tượng Implementor.
  + Implementor: Định nghĩa giao diện cho các lớp triển khai cụ thể.

Abstraction ủy quyền công việc thực tế cho Implementor, cho phép thay đổi hoặc mở rộng triển khai mà không ảnh hượng đến phần trừu tượng.

* Cấu trúc:
  + Abstraction: Giao diện trừu tượng chứa tham chiếu đến Implementor.
  + RefinedAbstraction: Mở rộng Abstraction và có thể thêm các chức năng bổ sung.
  + Implementor: Giao diện cho các lớp triển khai cụ thể.
  + ConcreteImplementor: Triển khai cụ thể của Implementor.
* Code demo:

using System;

// Implementor

public interface IDevice

{

void TurnOn();

void TurnOff();

}

// ConcreteImplementor

public class Light : IDevice

{

public void TurnOn() => Console.WriteLine("Light is ON");

public void TurnOff() => Console.WriteLine("Light is OFF");

}

public class Fan : IDevice

{

public void TurnOn() => Console.WriteLine("Fan is ON");

public void TurnOff() => Console.WriteLine("Fan is OFF");

}

// Abstraction

public abstract class RemoteControl

{

protected IDevice device;

public RemoteControl(IDevice device) => this.device = device;

public abstract void TogglePower();

}

// RefinedAbstraction

public class BasicRemote : RemoteControl

{

private bool isOn = false;

public BasicRemote(IDevice device) : base(device) { }

public override void TogglePower()

{

if (isOn)

{

device.TurnOff();

isOn = false;

}

else

{

device.TurnOn();

isOn = true;

}

}

}

// Client

class Program

{

static void Main()

{

IDevice light = new Light();

RemoteControl remote = new BasicRemote(light);

remote.TogglePower(); // Output: Light is ON

remote.TogglePower(); // Output: Light is OFF

IDevice fan = new Fan();

remote = new BasicRemote(fan);

remote.TogglePower(); // Output: Fan is ON

}

}