Structural Design Pattern

Lecture

by

Maliha Bushra Hoque (MBH)

- Structural design patterns are a category of design patterns in software engineering that focus on the organization and composition of classes or objects to form larger structures.
- These patterns help in defining how different components or objects should be structured to create more efficient and maintainable software systems.
- They often address concerns related to the composition of objects and the relationships between them.

Some common Structural Design Patterns include - Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Flyweight Pattern, Proxy Pattern.

Adapter Pattern –

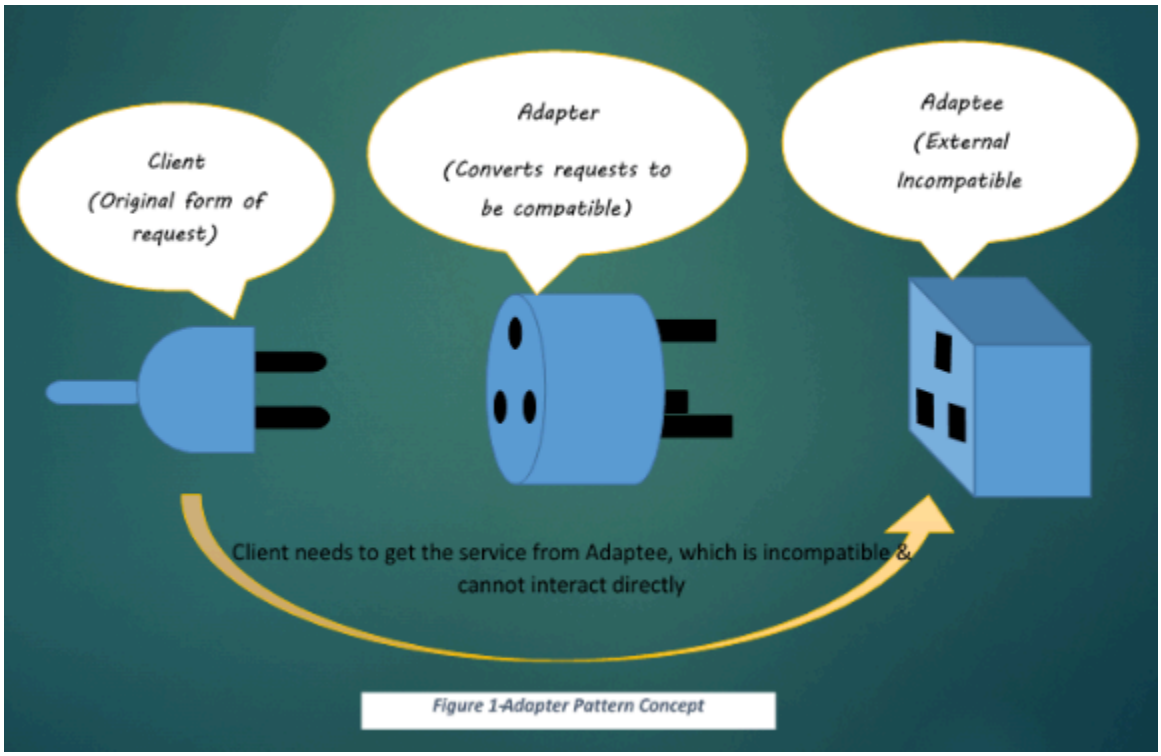

Figure 1-Adapter Pattern Concept

- A Structural Pattern (Structural patterns are concerned with how classes and objects are composed to form larger structures.)
- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As **Wrapper.**

The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to work together.

It acts as a bridge between two incompatible interfaces, making them compatible without changing their source code.

The primary purpose of the Adapter Pattern is to make different classes or components work together smoothly and continuously.

Participants:

Target: defines the domain-specific interface that the Client uses. (ex: Pizza)

Client: collaborates with objects conforming to the Target interface.

Adaptee: defines an existing interface that needs adapting (ex: ChittagongPizza)

Adapter: adapts the interface of Adaptee to the Target interface. (ex: ChittagongClassAdapter).

Class Adapter

Scenario: I have a pizza-making store that creates different pizzas based on the choices of people from different locations. For example – people of Dhaka like DhakaStylePizza, people of Sylhet like SylhetStylePizza.

Solution: To meet the scenario, we can declare a Pizza interface, and at different location, people can make their own style of pizza by implementing the same interface.

```
Public Interface Pizza{
    abstract void toppings();
    abstract void bun();
}
```

```
Public class DhakaStylePizza implements Pizza{
        public void toppings(){
            print("Dhaka cheese toppings");
        }
        public void bun(){
            print("Dhaka bread bun");
        }
    }
```

Now we want to support ChittagongStylePizza.

The customers of Chittagong are rigid. They want to use the authentic existing class, ChittagongPizza

But we can not call it directly, as it's not name same as our Pizza interface.

```
public class ChittagongPizza{
    public void sausage(){
        print("Ctg pizza");
    }
    public void bread(){
        print("Ctg bread");
    }

}
```

```
Public Interface Pizza{
    abstract void toppings();
    abstract void bun();
}
```

- We want to adapt the existing ChittagongPizza, so it's a Adaptee.
- To do so, introduce a Class Adapater, ChittagongClassAdapter

```
Public class ChittagongClassAdapter extends ChittagongPizza implements
Pizza{
        public void toppings(){
            this.sausage();
        }
         public void bun(){
            this.bread();
        }

}
```

```
public class ChittagongPizza{
    public void sausage(){
        print("Ctg pizza");
    }
    public void bread(){
        print("Ctg bread");
    }

}
```

From main method, customer call –

```
Pizza adaptedPizza = new ChittagongClassAdapter();
adaptedPizza.toppings();
adaptedPizza.bun();
```