Lecture - 5

1- Structural Design Patterns
2- Adapter

Structural Design Patterns

What is a Structural Pattern?

- Structural patterns describe how classes and objects can be combined to form larger structures.
- The Structural patterns are:
  - The **Adapter pattern** can be used to make one class interface match another to make programming easier.
  - The **Composite pattern** is a composition of objects, each of which may be either simple or itself a composite object.
  - The **Proxy pattern** is frequently a simple object that takes the place of a more complex object that may be invoked later.

Structural Pattern cont…

- The **Flyweight pattern** is a pattern for sharing objects, where each instance does not contain its own state, but stores it externally. This allows efficient sharing of objects to save space, when there are many instances, but only a few different types.
- The **Façade pattern** is used to make a single class represent an entire subsystem.
- The **Bridge pattern** separates an object’s interface from its implementation, so you can vary them separately.
- The **Decorator pattern**, which can be used to add responsibilities to objects dynamically.

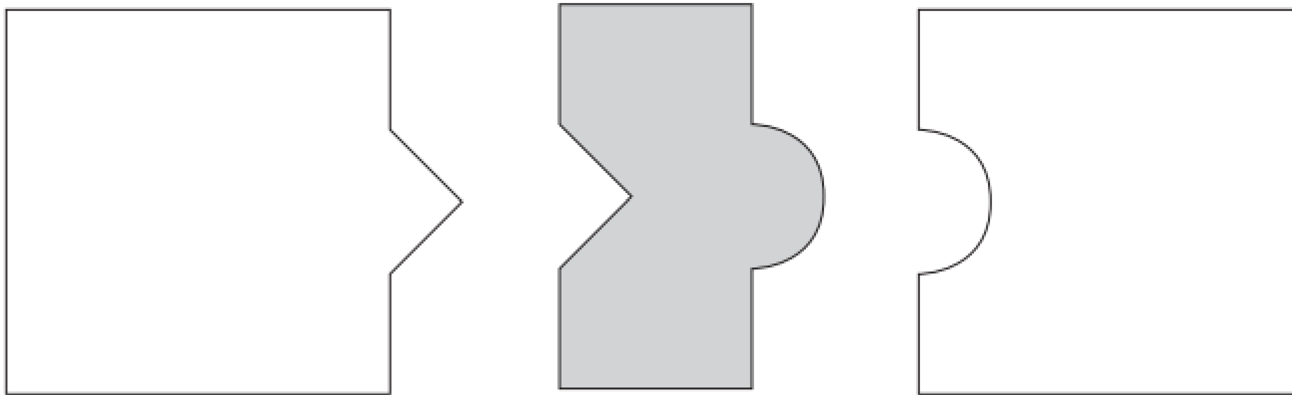
Adapter Pattern

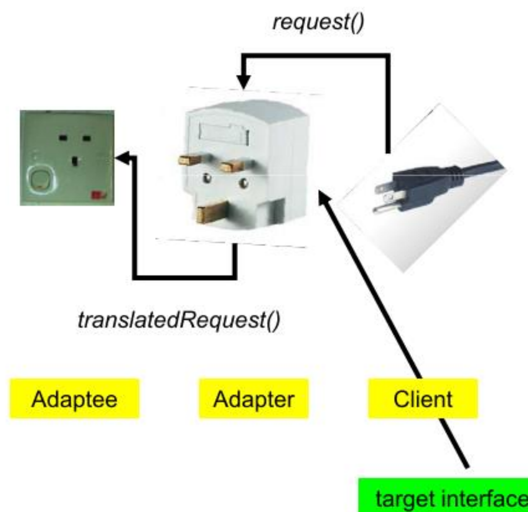
Definition:

"Convert the interface of a class into another interface that clients expect. The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces."

Concept

- The core concept is best described by the examples given below.





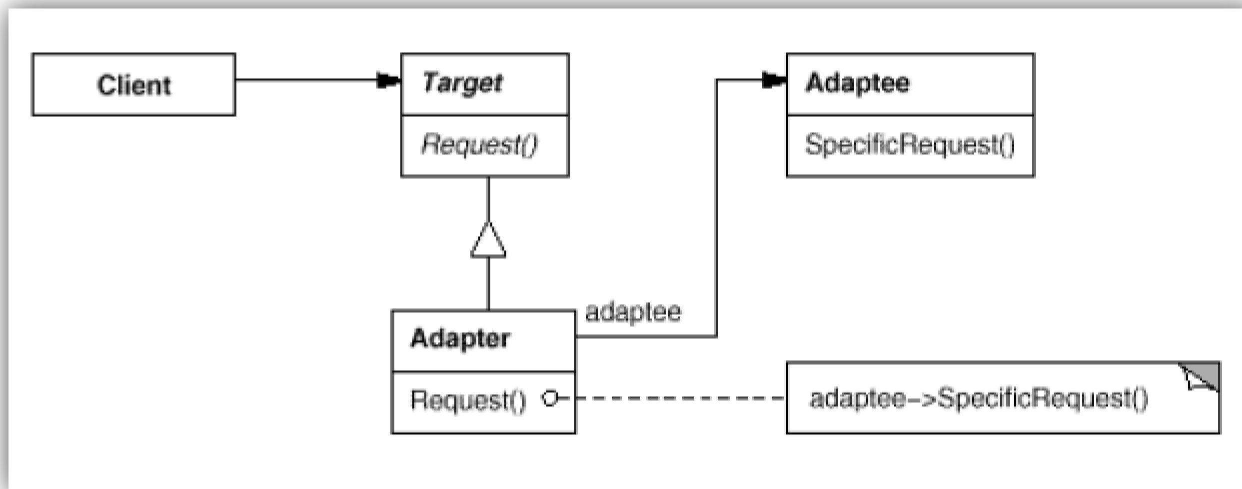
Real-Life Examples

- The most common example of this type can be found with mobile charging devices. If our charger is not supported by a particular kind of switchboard, we need to use an adapter.
- Even the translator who is translating language for one person is following this pattern in real life.

Where to use

- When you want to use an existing class, and its interface does not match the one you need.
- When you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- When you want to increase transparency of classes.

Structure



Participants

- **Target**
  - defines the domain-specific interface that Client uses.
- **Client**
  - collaborates with objects conforming to the Target interface.
- **Adaptee**
  - defines an existing interface that needs adapting.
- **Adapter**
  - adapts the interface of Adaptee to the Target interface.

Example:

- In this example, we can calculate the area of a rectangle easily using the Calculator class and its getArea() method that uses a rectangle as an input .
- Now suppose we want to calculate the area of a triangle, but we need to get the area of the triangle through the getArea() method of Calculator. How can we do that?
- To do that we have made a CalculatorAdapter for the triangle and passed a triangle in its getArea() method.
- The method will translate the triangle input to rectangle input and in turn, it will call the getArea() of Calculator to get the area of it.
- From the user's point of view, it seems to the user that he is passing a triangle to get the area of that triangle.

Example:

```java
public class AdapterPattern {

    public static void main(String[] args) {
        System.out.println("***Adapter Pattern Demo***");
        CalculatorAdapter cal = new CalculatorAdapter();
        Triangle t = new Triangle(20, 10);
        System.out.println("\nAdapter Pattern Example\n");
        System.out.println("Area of Triangle is :" + cal.getArea(t));
    }
}
```

Output:

```
run:
***Adapter Pattern Demo***

Adapter Pattern Example

Area of Triangle is :100.0
```

Adaptee

```
class Calculator {

    Rect rectangle;

    public double getArea(Rect r) {
        rectangle = r;
        return rectangle.l * rectangle.w;
    }
}
```

Adapter

```
class CalculatorAdapter {

    Calculator calculator;
    Triangle triangle;

    public double getArea(Triangle t) {
        calculator = new Calculator();
        triangle = t;
        Rect r = new Rect();
        //Area of Triangle=0.5*base*height
        r.l = triangle.b;
        r.w = 0.5 * triangle.h;
        return calculator.getArea(r);
    }

}
```

Target

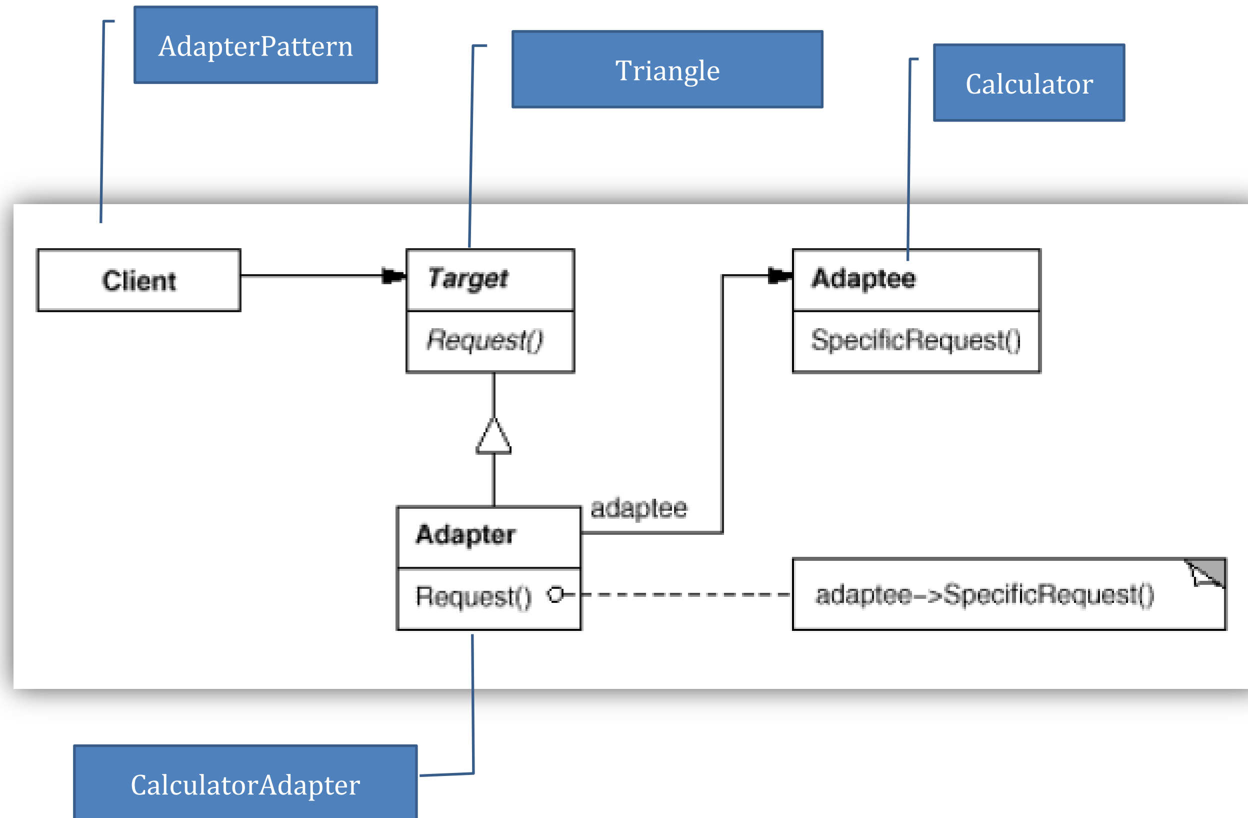
```
class Triangle
{
    public double b;//base
    public double h;//height
    public Triangle(int b, int h)
    {
        this.b = b;
        this.h = h;
    }
}
```

Client

```
public class AdapterPattern {

    public static void main(String[] args) {
        System.out.println("***Adapter Pattern Demo***");
        CalculatorAdapter cal = new CalculatorAdapter();
        Triangle t = new Triangle(20, 10);
        System.out.println("\nAdapter Pattern Example\n");
        System.out.println("Area of Triangle is :" + cal.getArea(t));
    }
}
```

Participants:



Participants

Participant	Class Or Interface
Adapter	CalculatorAdapter class
Adaptee	Calculator class
Target	Triangle class
Client	AdapterPattern class

Thanks !