Lab No.: 05

Experiment Name : Design Adapter design pattern using UML.

Objectives:

1. Gain a comprehensive understanding of the Adapter design pattern's intent, structure, and behavior.

2. Develop detailed UML diagrams, including class and sequence diagrams, to visually represent the structure and interactions of the Adapter pattern.

3. Translate UML diagrams into code, focusing on designing adaptable interfaces, implementing concrete adapters, and integrating them into the existing software system.

Theory:

The Adapter design pattern serves as a bridge between incompatible interfaces, allowing objects to collaborate seamlessly. In the context of UML, this pattern comprises several key elements. Firstly, there's the Target Interface, representing the desired interface that the client expects to interact with. It defines the operations or methods that the client code utilizes. Then, there's the Client, which represents the entity that interacts with the Target interface. On the other side, there's the Adaptee, embodying the existing interface that needs adaptation. It's the interface that the Adapter wraps and translates calls to. Finally, there's the Adapter itself, a class that implements the Target interface and wraps the Adaptee. It translates calls from the Target interface into calls to the Adaptee's interface. Within the UML diagram, these components are depicted clearly, showcasing how the Adapter pattern facilitates interaction between the client and the Adaptee. This pattern proves invaluable in scenarios where integration between incompatible interfaces is necessary, ensuring seamless collaboration and system functionality.

Implementation of Design Adapter design pattern using UML:

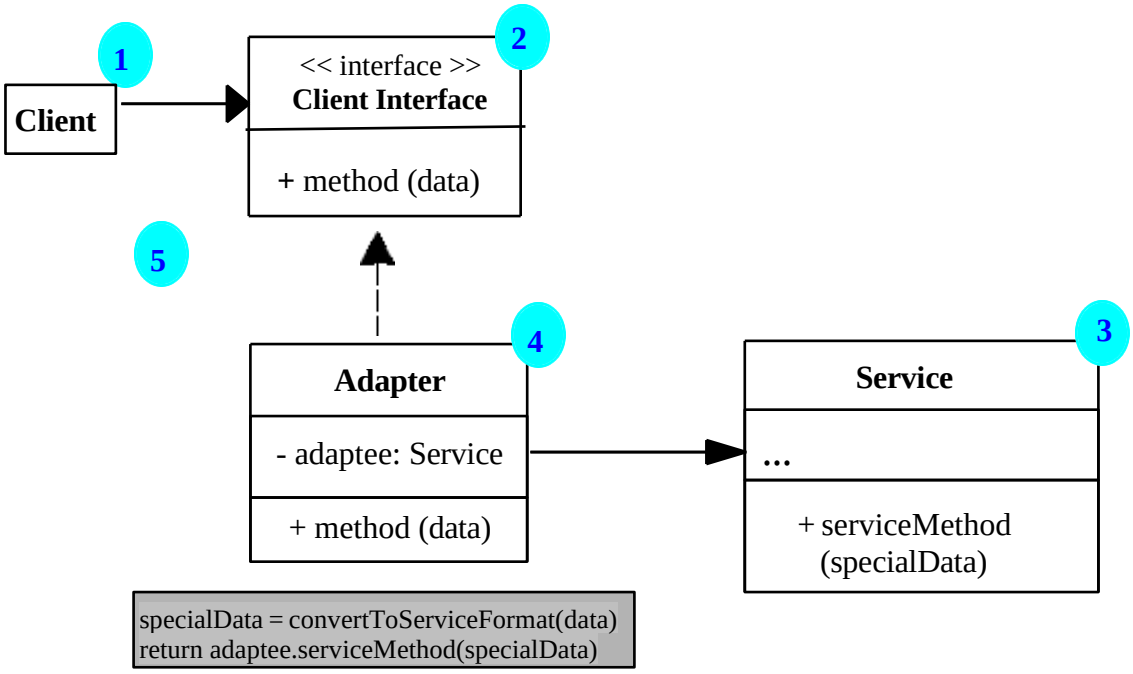

Fig 1 : Object adapter

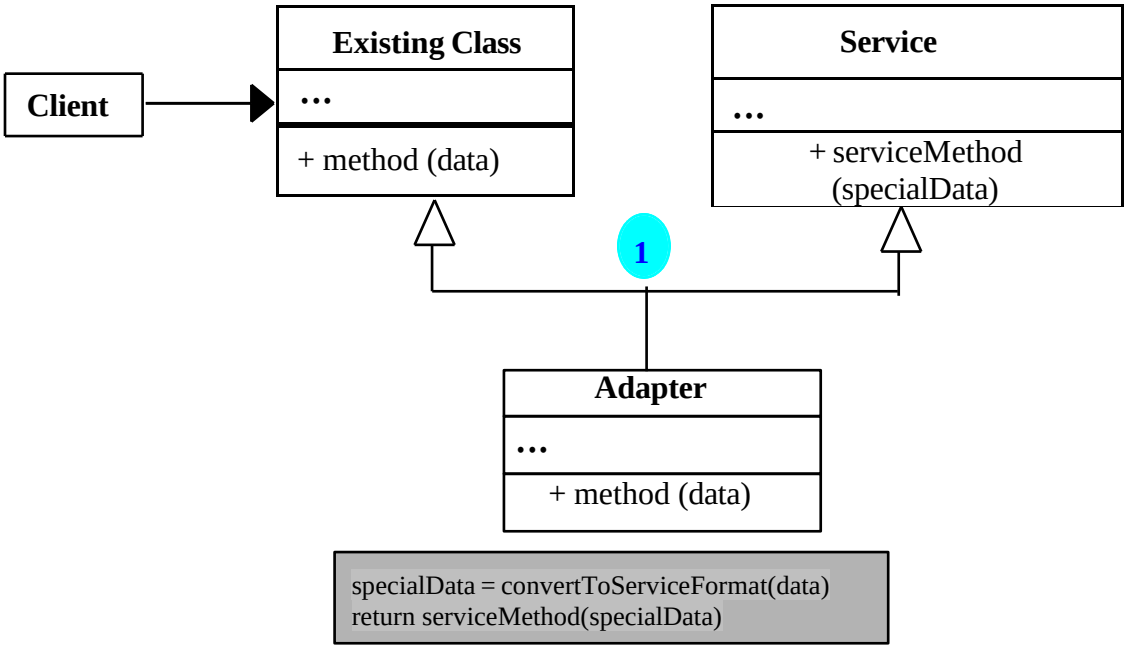

Fig 2 : Class Adapter

Implementation in Java:

```
// Say we have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Return the radius of the hole.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Return the radius of the peg.


// But there's an incompatible class: SquarePeg.
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Return the square peg width.


// An adapter class lets fit square pegs into round holes.
// It extends the RoundPeg class to let the adapter objects act
// as round pegs.
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2
```

```
// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

Result and Discussion:

The adapter acts as an intermediary, facilitating communication between objects with incompatible interfaces. By wrapping one object and converting its interface to match that of another object, the adapter enables seamless integration without disrupting existing codebases. Additionally, adapters can support bidirectional communication and promote modularity and extensibility in software systems. Overall, the adapter pattern plays a vital role in promoting interoperability and reusability in software development.