Part A	
Class B Tech CSE 2nd Year Lab	**Sub: Design Pattern**
Aim: Write a program to implement the Adapter Design Pattern	
Prerequisite: Basics of Object Oriented Programming	
Outcome: To impart knowledge of Design Pattern techniques	
Theory: Explain about the Adapter Design Pattern.	

Adapter Pattern

The Adapter pattern is a structural design pattern used in software engineering. Its main purpose is to enable objects with incompatible interfaces to work together. This pattern acts as a bridge between two incompatible interfaces, allowing them to collaborate seamlessly.

Working:

1. Adaptee: This is the existing component or class with an interface that is incompatible with the client's requirements. The Adaptee needs integration into the client's code, but its interface doesn't match what the client expects.

2. Target: This is the interface that the client expects. It defines the methods or operations that the client code will use to interact with the Adaptee or the existing system.

3. Adapter: The Adapter is a class that bridges the gap between the Adaptee and the Target. It implements the Target interface and internally communicates with the Adaptee, translating the calls and parameters as necessary.

4. Client: The client is the code that requires interaction with the Adaptee but expects the Target interface. The client interacts with the Adapter as if it were interacting with the Target directly, unaware of the Adaptee's presence or its incompatible interface.

Key components:

- Object Adapter vs. Class Adapter: There are two common implementations of the Adapter pattern: object adapters and class adapters. Object adapters use composition to connect the Adapter with the Adaptee, while class adapters use multiple inheritance to inherit from both the Target interface and the Adaptee class.

- Wrapper: The Adapter acts as a wrapper around the Adaptee, providing a compatible interface that the client can use. It translates method calls, converts parameters, or performs any necessary operations to ensure seamless communication between the client and the Adaptee.

- Transparency: Ideally, the client should be unaware of the Adapter's presence. It should interact with the Adapter through the Target interface, treating it as if it were the actual Target object. This transparency ensures that the client code remains clean and unaffected by the integration of the Adaptee.

- Flexibility and Reusability: The Adapter pattern promotes flexibility and reusability by allowing the integration of existing components or classes without requiring modifications to their interfaces. This is particularly useful when working with legacy code or third-party libraries that cannot be modified directly.

Part B
Steps: Write procedure/code here.
Output: Paste Output here.
Observation & Learning: Give your observation and learning about the experiment.
Conclusion: Give your conclusion for the experiment.

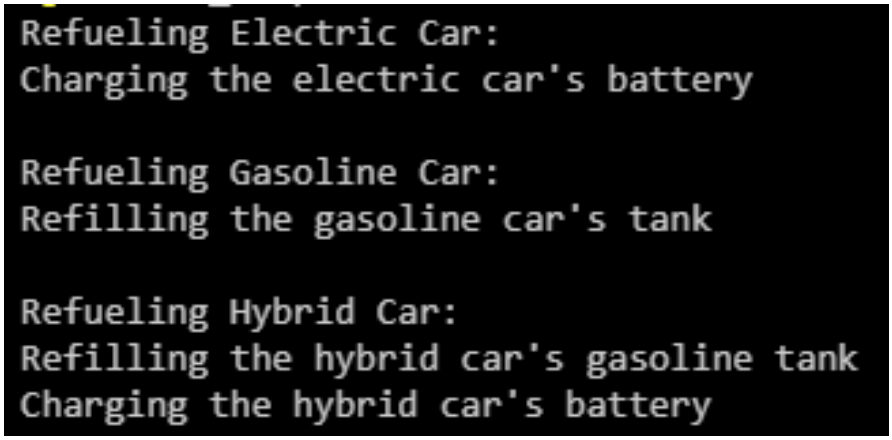
Code:

```
// Target Interface - Represents the interface expected by the client
interface Car {
    void refuel();
}

// Adaptees - Different types of cars with incompatible interfaces

class ElectricCar {
    public void chargeBattery() {
        System.out.println("Charging the electric car's battery");
    }
}

class GasolineCar {
    public void refillGasoline() {
        System.out.println("Refilling the gasoline car's tank");
    }
}

class HybridCar {
    public void refillGasoline() {
        System.out.println("Refilling the hybrid car's gasoline tank");
    }

    public void chargeBattery() {
        System.out.println("Charging the hybrid car's battery");
    }
}

// Adapters - Adapts the different types of cars to the Car interface

class ElectricCarAdapter implements Car {
    private ElectricCar electricCar;

    public ElectricCarAdapter(ElectricCar electricCar) {
        this.electricCar = electricCar;
    }

    @Override
```

```
        public void refuel() {
            electricCar.chargeBattery();
        }
    }

    class GasolineCarAdapter implements Car {
        private GasolineCar gasolineCar;

        public GasolineCarAdapter(GasolineCar gasolineCar) {
            this.gasolineCar = gasolineCar;
        }

        @Override
        public void refuel() {
            gasolineCar.refillGasoline();
        }
    }

    class HybridCarAdapter implements Car {
        private HybridCar hybridCar;

        public HybridCarAdapter(HybridCar hybridCar) {
            this.hybridCar = hybridCar;
        }

        @Override
        public void refuel() {
            hybridCar.refillGasoline();
            hybridCar.chargeBattery();
        }
    }

    // Client
    public class Pr7_Adapter {
        public static void main(String[] args) {
            // Create instances of different types of cars
            ElectricCar electricCar = new ElectricCar();
            GasolineCar gasolineCar = new GasolineCar();
            HybridCar hybridCar = new HybridCar();

            // Create adapters for each type of car
            Car electricCarAdapter = new ElectricCarAdapter(electricCar);
            Car gasolineCarAdapter = new GasolineCarAdapter(gasolineCar);
            Car hybridCarAdapter = new HybridCarAdapter(hybridCar);

            // Now, the client can interact with the cars through the unified
Car interface
            System.out.println("Refueling Electric Car:");
```

```
            electricCarAdapter.refuel();

            System.out.println("\nRefueling Gasoline Car:");
            gasolineCarAdapter.refuel();

            System.out.println("\nRefueling Hybrid Car:");
            hybridCarAdapter.refuel();
        }
    }
```

Output



Observation & Learning:

From the experiment, I noticed that the Adapter pattern helps connect different parts of a system with incompatible interfaces. It makes it easier to reuse code and keeps things organized. There are different types of adapters, like object adapters and class adapters, which offer flexibility in how they're used.

Conclusion:

In summary, the experiment showed that the Adapter pattern is useful for integrating different components in software. It makes systems more flexible and easier to maintain. Using this pattern can lead to better-designed software that's easier to work with in the long run.