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

Facade Pattern

The Facade pattern is a structural design pattern used in software engineering. It is categorized under the Gang of Four (GoF) design patterns, which are a set of commonly used design patterns originally described in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

The main purpose of the Facade pattern is to provide a unified, simplified interface to a complex system of classes, subsystems, or interfaces. It encapsulates the complexity of the system behind a single interface, making it easier for clients to interact with the system without needing to understand its internal details.

Working:

1. Complex Subsystem: In a software system, there may be numerous classes, components, or subsystems that interact with each other to achieve a specific functionality. These subsystems may have intricate relationships and dependencies.

2. Facade: The Facade is a class or interface that acts as a simplified interface to this complex subsystem. It provides a higher-level interface that hides the complexities of the underlying subsystem from the client code.

3. Client: The client is the code that interacts with the Facade to access the functionality provided by the complex subsystem. Instead of directly interacting with multiple classes or subsystems, the client simply uses the Facade, which shields it from the intricacies of the system.

Key benefits of using the Facade pattern include:

- Simplified Interface: Clients only need to interact with a single, well-defined interface provided by the Facade, which abstracts away the complexities of the underlying system.

- Decoupling: Facade promotes loose coupling between clients and the subsystem, as clients are not dependent on the internal structure of the subsystem. This allows for easier maintenance and evolution of the system.

- Encapsulation: The Facade encapsulates the subsystem's implementation details, providing a clear separation of concerns and hiding unnecessary complexity from clients.

- Improved Usability: By providing a simplified interface, the Facade pattern enhances the usability of the system, making it easier for developers to use and understand.

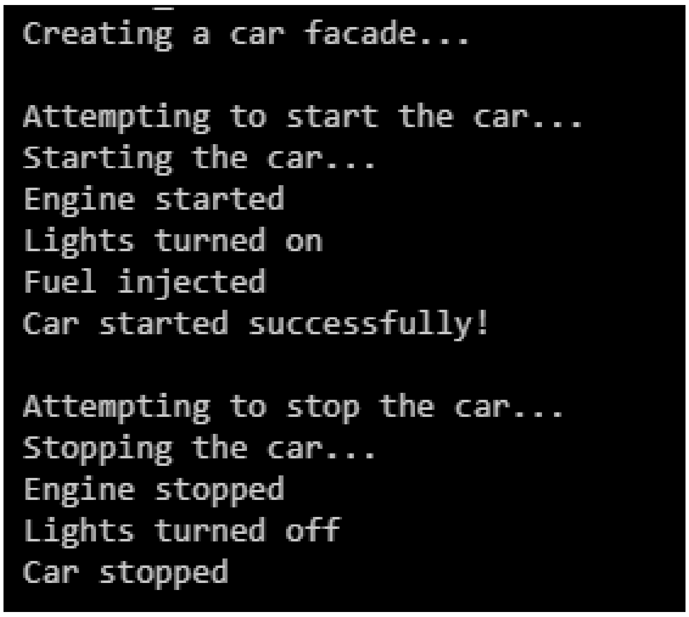
A common real-world analogy for the Facade pattern is a remote control for a home entertainment system. The remote control serves as a Facade, providing a simplified interface for users to interact with various complex components such as the TV, DVD player, sound system, etc., without needing to understand the internal workings of each individual component.

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:

```java
// Subsystem classes
class Engine {
    public void start() {
        System.out.println("Engine started");
    }

    public void stop() {
        System.out.println("Engine stopped");
    }
}

class Lights {
    public void turnOn() {
        System.out.println("Lights turned on");
    }

    public void turnOff() {
        System.out.println("Lights turned off");
    }
}

class FuelInjector {
    public void injectFuel() {
        System.out.println("Fuel injected");
    }
}

// Facade class
class CarFacade {
    private Engine engine;
    private Lights lights;
    private FuelInjector fuelInjector;

    public CarFacade() {
        this.engine = new Engine();
        this.lights = new Lights();
        this.fuelInjector = new FuelInjector();
    }
    public void startCar() {
        System.out.println("Starting the car...");
        engine.start();
        lights.turnOn();
        fuelInjector.injectFuel();
        System.out.println("Car started successfully!");
    }

    public void stopCar() {
        System.out.println("Stopping the car...");
        engine.stop();
        lights.turnOff();
        System.out.println("Car stopped");
    }
}

// Client class
public class Pr8_Facade {
    public static void main(String[] args) {
        System.out.println("Creating a car facade...");
        CarFacade car = new CarFacade();

        // Starting the car
        System.out.println("\nAttempting to start the car...");
        car.startCar();

        // Stopping the car
        System.out.println("\nAttempting to stop the car...");
        car.stopCar();

        System.out.println("\nExiting the program.");
    }
}
```

Output



```
Creating a car facade...

Attempting to start the car...
Starting the car...
Engine started
Lights turned on
Fuel injected
Car started successfully!

Attempting to stop the car...
Stopping the car...
Engine stopped
Lights turned off
Car stopped
```

Observation & Learning:

During the experiment, I noticed that the Facade pattern is really handy for making complicated things simpler in software. It wraps up all the complicated stuff behind a single easy-to-use interface. This makes it much easier for people to work with the system without needing to know all the technical details. It also helps keep different parts of the software separate, which makes it easier to fix and change things later. From this, I learned that the Facade pattern is like a magic trick that makes complex software easier to understand and work with.

Conclusion:

To sum up, the experiment showed me that the Facade pattern is a great tool for simplifying software and making it easier to manage. By hiding the complexity behind a simple interface, it helps keep things organized and makes the software more user-friendly. Using the Facade pattern in software design can make projects more manageable and less confusing for everyone involved.