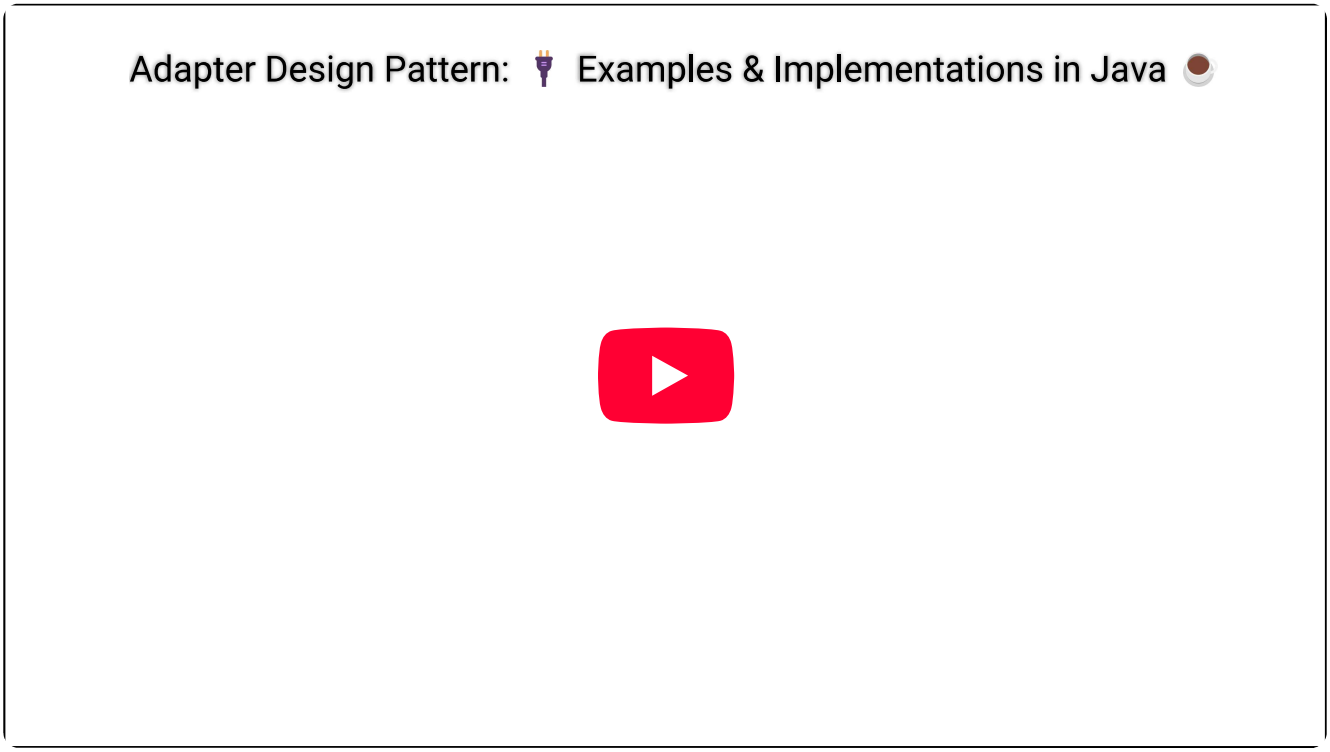
Search articles...

Adapter Design Pattern



Topic Tags:

System Design LLD

Problem Statement: Connecting the Unconnectable

Imagine you're designing a smart home system. Your goal is to create a centralized app that controls various devices such as air conditioners, smart lights, coffee machines, and security cameras.

Each device comes from a different manufacturer, and they all communicate differently:
• Air Conditioners use Bluetooth for communication.
• Smart Lights operate over Wi-Fi.
• Coffee Machines use Zigbee.
• Security Cameras rely on their own custom API.

Now, your app needs to seamlessly control all these devices, regardless of the communication protocol.

The Problem: Each device uses a unique communication protocol, and your app would become a mess if you hard-code the logic for each device. It will be difficult to maintain and extend as more devices are added.
The Challenge: How can you create a clean, scalable solution to connect all these devices?

Solving It the Traditional Way: A Messy Solution 🛠️

Let's look at how you might solve this problem in a straightforward but inflexible way:

Java

```java
import java.util.Scanner;
public class SmartHomeController {
  // Method to control devices based on their type
  public void controlDevice(String deviceType) {
    if (deviceType.equalsIgnoreCase("AirConditioner")) {
      System.out.println("Connecting to Air Conditioner via Bluetooth...")
    } else if (deviceType.equalsIgnoreCase("SmartLight")) {
      System.out.println("Connecting to Smart Light via Wi-Fi...");
    } else if (deviceType.equalsIgnoreCase("CoffeeMachine")) {
      System.out.println("Connecting to Coffee Machine via Zigbee...");
    } else {
      System.out.println("Device type not supported!");
    }
  }

  // Main method to test the SmartHomeController
  public static void main(String[] args) {
    SmartHomeController controller = new SmartHomeController();
    Scanner scanner = new Scanner(System.in);
    System.out.println("Welcome to the Smart Home Controller!");
    System.out.println(
        "Available devices: AirConditioner, SmartLight, CoffeeMachine");
    while (true) {
      System.out.print(
          "nEnter the device you want to control (or type 'exit' to quit):
      String deviceType = scanner.nextLine();
      if (deviceType.equalsIgnoreCase("exit")) {
        System.out.println("Exiting the Smart Home Controller. Goodbye!");
        break;
      }
      controller.controlDevice(deviceType);
```

```
    }
    scanner.close();
  }
}
```

In the current implementation, the Main class handles device operations directly by identifying the device type (e.g., AirConditioner, SmartLight, CoffeeMachine) and calling the appropriate methods. While this works for a small system, it quickly becomes unmanageable as more devices are added or existing devices are updated.

Interviewer's Follow-up Questions: Can We Improve the Code? 🤔

An interviewer might ask:
• What if we need to add more devices in the future? For example, a new SmartSpeaker or SecurityCamera.
• What if the logic for interacting with devices changes? For instance, what if the protocol for controlling a SmartLight switches from Wi-Fi to a cloud-based API?

In such scenarios, managing the operations for each device type in the Main class becomes complex. The code grows fragile, and adding or modifying device types requires changes in multiple places, increasing the risk of introducing bugs.

Ugly Code: When We Realize the Code Needs Restructuring 🛠️

Let's say the logic for controlling devices becomes more complex. For instance:
1. User Input: The user decides which device to control.
2. Protocol-Specific Behavior: Each device has its own proprietary communication protocol (e.g., Bluetooth, Wi-Fi, Zigbee).
3. Dynamic Changes: The implementation for a device might evolve over time (e.g., a CoffeeMachine might integrate with a new IoT standard).

If this complexity isn't addressed early, the Main class quickly becomes a mess with hardcoded, tightly coupled logic. 🤦‍♂️

It might look something like this:

Java

```java
public class SmartHomeController {
  public static void main(String[] args) {
    String deviceType = "SmartLight"; // Imagine this value is dynamic
    if (deviceType.equals("AirConditioner")) {
```

```
        AirConditioner airConditioner = new AirConditioner();
        airConditioner.connectViaBluetooth();
        airConditioner.startCooling();
      } else if (deviceType.equals("SmartLight")) {
        SmartLight smartLight = new SmartLight();
        smartLight.connectToWiFi();
        smartLight.switchOn();
      } else if (deviceType.equals("CoffeeMachine")) {
        CoffeeMachine coffeeMachine = new CoffeeMachine();
        coffeeMachine.initializeZigbeeConnection();
        coffeeMachine.startBrewing();
      } else {
        System.out.println("Device type not supported!");
      }
    }
}
```

This approach tightly couples the SmartHomeController to the device classes and their specific protocols. Any new device or protocol change requires updating the controller, leading to a cascade of maintenance issues.

The Savior: Adapter Design Pattern 🦸

The Adapter Pattern is designed to solve this exact problem. It acts as a bridge between two incompatible interfaces, allowing them to work together seamlessly without modifying their code.
In our SmartHomeController example, the adapter provides a common interface that the controller can use to interact with devices, regardless of their specific communication protocols or implementation details.

How the Adapter Pattern Works 🔧

The Adapter Pattern achieves this by introducing a new class (the Adapter) that implements the interface expected by the client (e.g., the SmartHomeController) and translates its requests into commands that the incompatible class (the device) understands.

In essence, the adapter hides the complexity of device-specific protocols from the client, ensuring smooth interaction between the SmartHomeController and devices like AirConditioner, SmartLight, or CoffeeMachine. This makes the system more flexible and maintainable.

Solving the Problem with Adapter Design Pattern

Here's how we can solve the problem using the Adapter Design Pattern, enabling seamless integration of devices with different communication protocols into the SmartHomeController system.

Step 1: Define a Common Interface

The first step is to define a common interface for all devices. This ensures that the SmartHomeController can interact with any device using the same methods, regardless of their internal protocols.

Java

```java
// SmartDevice.java - Common interface for all smart devices
public interface SmartDevice {
    void turnOn(); // method to turn on a specific Device
    void turnOff(); // method to turn off a specific Device
}
```

Step 2 : Create Concrete classes for Each Device

- **AirConditioner.java**

Java

```java
// AirConditioner.java - Device using Bluetooth for communication
public class AirConditioner {
  // Method to connect to the Air Conditioner via Bluetooth
  public void connectViaBluetooth() {
    System.out.println("Air Conditioner connected via Bluetooth.");
  }

  // Method to start the cooling process
  public void startCooling() {
    System.out.println("Air Conditioner is now cooling.");
  }

  // Method to stop the cooling process
  public void stopCooling() {
    System.out.println("Air Conditioner stopped cooling.");
```

```
  }

  // Method to disconnect Bluetooth connection
  public void disconnectBluetooth() {
    System.out.println("Air Conditioner disconnected from Bluetooth.");
  }
}
```

SmartLight.java

Java

```
// SmartLight.java - Device using Wi-Fi for communication
public class SmartLight {
  // Method to connect the Smart Light to Wi-Fi
  public void connectToWiFi() {
    System.out.println("Smart Light connected to Wi-Fi.");
  }

  // Method to turn the Smart Light on
  public void switchOn() {
    System.out.println("Smart Light is now ON.");
  }

  // Method to turn the Smart Light off
  public void switchOff() {
    System.out.println("Smart Light is now OFF.");
  }

  // Method to disconnect Wi-Fi connection
  public void disconnectWiFi() {
    System.out.println("Smart Light disconnected from Wi-Fi.");
  }
}
```

- **CoffeeMachine.java**

Java

```java
// CoffeeMachine.java - Device using Zigbee for communication
public class CoffeeMachine {
  // Method to initialize the Zigbee connection
  public void initializeZigbeeConnection() {
    System.out.println("Coffee Machine connected via Zigbee.");
  }

  // Method to start brewing coffee
  public void startBrewing() {
    System.out.println("Coffee Machine is now brewing coffee.");
  }

  // Method to stop brewing coffee
  public void stopBrewing() {
    System.out.println("Coffee Machine stopped brewing coffee.");
  }

  // Method to terminate the Zigbee connection
  public void terminateZigbeeConnection() {
    System.out.println("Coffee Machine disconnected from Zigbee.");
  }
}
```

Step 3 : Create Adapters for Each Device

Each adapter implements the SmartDevice interface and translates the controller's requests into commands specific to the underlying device.

Java

```java
// Adapter for Air Conditioner
public class AirConditionerAdapter implements SmartDevice {
  private AirConditioner airConditioner;
  // Constructor
  public AirConditionerAdapter(AirConditioner airConditioner) {
    this.airConditioner = airConditioner;
  }

  @Override
```

```
  public void turnOn() {
    airConditioner.connectViaBluetooth();
    airConditioner.startCooling();
  }

  @Override
  public void turnOff() {
    airConditioner.stopCooling();
    airConditioner.disconnectBluetooth();
  }
}

// Adapter for Smart Light
public class SmartLightAdapter implements SmartDevice {
  private SmartLight smartLight;
  public SmartLightAdapter(SmartLight smartLight) {
    this.smartLight = smartLight;
  }

  @Override
  public void turnOn() {
    smartLight.connectToWiFi();
    smartLight.switchOn();
  }

  @Override
  public void turnOff() {
    smartLight.switchOff();
    smartLight.disconnectWiFi();
  }
}

// Adapter for Coffee Machine
public class CoffeeMachineAdapter implements SmartDevice {
  private CoffeeMachine coffeeMachine;
  public CoffeeMachineAdapter(CoffeeMachine coffeeMachine) {
    this.coffeeMachine = coffeeMachine;
  }

  @Override
  public void turnOn() {
```

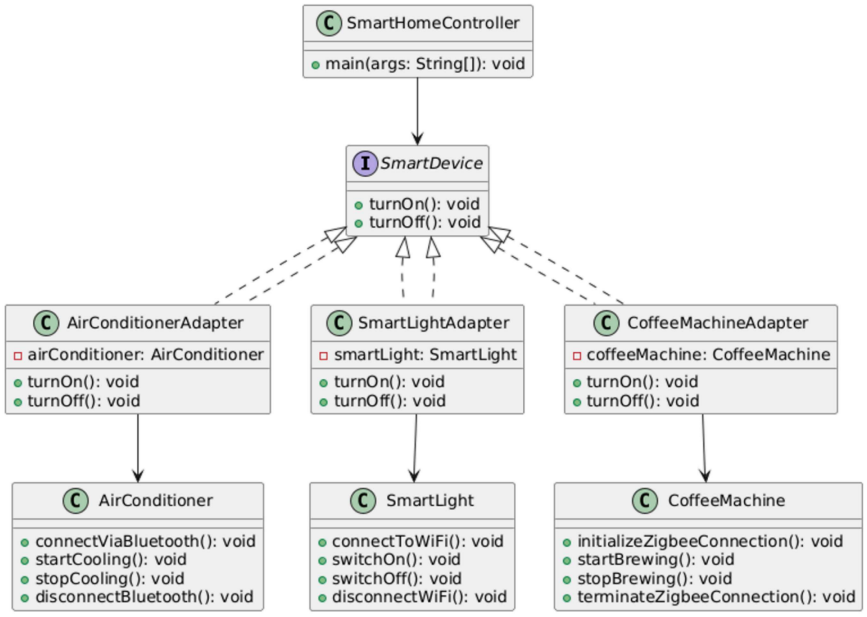
```
    coffeeMachine.initializeZigbeeConnection();
    coffeeMachine.startBrewing();
  }

  @Override
  public void turnOff() {
    coffeeMachine.stopBrewing();
    coffeeMachine.terminateZigbeeConnection();
  }
}
```

Step 4: Use Adapters in the SmartHomeController

The SmartHomeController no longer needs to handle device-specific logic. It interacts with the devices through the SmartDevice interface, allowing the adapters to manage the communication.

Java

```
public class SmartHomeController {
  public static void main(String[] args) {
    // Create adapters for each device
    SmartDevice airConditioner =
        new AirConditionerAdapter(new AirConditioner());
    SmartDevice smartLight = new SmartLightAdapter(new SmartLight());
    SmartDevice coffeeMachine = new CoffeeMachineAdapter(new CoffeeMachine
    // Control devices through the unified interface
    airConditioner.turnOn();
    smartLight.turnOn();
    coffeeMachine.turnOn();
    airConditioner.turnOff();
    smartLight.turnOff();
    coffeeMachine.turnOff();
  }
}
```



Advantages of Using the Adapter Design Pattern 🏆

Let's review how the Adapter Pattern improves our solution:

1. Seamless Integration:

The Adapter Pattern enables the SmartHomeController to interact with devices using different protocols (Bluetooth, Wi-Fi, Zigbee, etc.) without worrying about their implementation details.

2. Scalability:

Adding a new device type (e.g., a SmartSpeaker or SecurityCamera) only requires creating a new adapter. The SmartHomeController doesn't need any changes.

3. Decoupling:

The controller is decoupled from the specific implementations of devices, making the system more modular and maintainable.

4. Flexibility:

If a device's protocol changes (e.g., the SmartLight switches from Wi-Fi to a cloud API), only the adapter needs to be updated, leaving the rest of the system unaffected.

Real-life Use Cases and Examples of the Adapter Pattern 🌎

The Adapter Pattern is widely used in real-world scenarios, especially in systems where components with incompatible interfaces need to work together:

1. Smart Home Systems:

Just like in this example, adapters are used to integrate devices from various manufacturers with different communication protocols into a unified controller.

2. Payment Gateways:
Adapters are used to unify APIs from different payment gateways (e.g., PayPal, Stripe, Razorpay), allowing a single payment interface in the application.

3. Database Drivers:
Adapters enable applications to interact with various databases (e.g., MySQL, PostgreSQL, MongoDB) using a consistent set of commands.

4. Media Players:
In multimedia applications, adapters allow a single player to support multiple file formats by translating file-specific operations into a common interface.

Conclusion 🎯

The Adapter Design Pattern simplifies integration by acting as a translator between incompatible interfaces. In our SmartHomeController example, it provides a unified way to interact with diverse devices, regardless of their protocols or underlying implementations.

By centralizing and abstracting communication logic in adapters, the pattern makes the system cleaner, more maintainable, and highly extensible. The Adapter Pattern is an essential tool for building flexible, scalable systems where different components need to work together seamlessly. 🌟