1. Implement the Singleton Pattern to create a thread-safe LoggerService class. The class should have a private constructor and a public static method to get the single instance.
2. Refactor a shape area calculation logic to adhere to the Open/Closed Principle (OCP). Create an abstract Shape and concrete classes for Circle, Square, and Triangle. The area calculation logic should be in the concrete classes, ensuring new shapes can be added without modifying existing code.
3. Refactor a monolithic Student class to adhere to the Single Responsibility Principle (SRP). Separate the logic into three classes: Student (data only), GradeCalculator (calculates average/grade), and StudentRepository (simulates saving data to a local file).
4. Implement the Prototype Pattern for deep cloning of objects. Create a Shape interface with a clone() method. Create concrete Circle and Rectangle classes that implement cloning. The client should create new instances by cloning existing ones.
5. Refactor a large Worker interface (with work(), eat(), sleep(), get_paid()) to adhere to the Interface Segregation Principle (ISP). Break it into smaller interfaces (Workable, Feedable, Restable, Payable) and implement them in HumanWorker (implements all) and RobotWorker (implements only Workable and Payable).
6. Implement the Abstract Factory Pattern to create families of related objects for a mobile phone. Create an abstract factory for MobileCamera and concrete factories for HighResolutionCamera and StandardCamera, each creating related products like PhotoCapturer and VideoRecorder.
7. Implement the Memento Pattern for a simple in-memory TextEditor. The editor should have a save() method that captures its current state (text), and an undo() method that restores the last saved state using a separate History (Caretaker) object.
8. Implement the Strategy Pattern for processing payments in an e-commerce application. Define a PaymentStrategy interface and concrete classes like CreditCardPayment, PayPalPayment, and UpiPayment, allowing the client to switch payment methods at runtime.
9. You have a Bird class with a fly() method. You then create a Penguin class that inherits from Bird but cannot fly. The fly() method in Penguin throws an error or does nothing, which breaks the expectation that any Bird can fly.

   Refactor the code to adhere to the Liskov Substitution Principle. The Penguin class should not inherit directly from Bird. Create a more general base class or interface for animals that can have different behaviors.

10. Write a java program to implement Proxy Pattern which efficiently manages the loading and displaying of images by introducing a proxy that controls access to the real image object.

11. Write a Java Program to implement Iterator Pattern for Designing Menu like Breakfast, Lunch or Dinner Menu.

12.

1. Spring API : Design a simple Task Management API using Spring Boot Web. Implement a REST Controller to handle GET (list all tasks), POST (add task), and DELETE (remove task) requests. (You can Use an in-memory Java List or Map to store task objects.)

Spring Security (Basic)

2. Develop a simple User Registration and Login Module using Spring Boot. Implement basic validation logic (e.g., password length) in the registration service. (You can Store user credentials in an in-memory HashMap for simplicity.)
3. Perform the Full SOLID Refactoring on a monolithic OrderProcessor class. The refactored solution must show separate classes/interfaces for Validation (SRP/ISP), Price Calculation (SRP/ISP), Persistence (SRP/ISP/DIP), and Order Type Handling (OCP/LSP), using dependency injection (DIP).
4. Implement the Bridge Design Pattern to separate the abstraction of a Vehicle from its implementation of a Manufacture. Create Car and Bike abstractions, and Produce and Assemble implementations. Create vehicles by linking them with a specific manufacture process.
5. Create a basic Employee Details Microservice using Spring Boot Web. Implement an API endpoint that accepts employee details and persists them to a local text file (employee_data.txt), bypassing a database.
6. Implement the Decorator Design Pattern for a coffee ordering system. Define a Coffee interface. Create concrete components like BasicCoffee. Then create decorators like MilkDecorator, SugarDecorator, and SyrupDecorator to dynamically add costs and ingredients to the coffee object.
7. Implement the Factory Method Pattern for a PizzaStore. Create an abstract PizzaStore with a createPizza() factory method. Create concrete stores like NYPizzaStore and ChicagoPizzaStore, each creating their respective styles of CheesePizza, VeggiePizza, etc.
8. Implement the Observer Design Pattern for a weather monitoring system. Create a WeatherStation (Subject) that maintains a list of Display (Observer) objects. When the weather data changes in the subject, all observer displays are automatically notified and updated.
9. Design a simple HR Application focusing on Spring's Inversion of Control (IoC) and Dependency Injection (DI). Create service and repository classes and demonstrate DI using Constructor Injection or Setter Injection to manage dependencies between components without manual instantiation.
10. Develop a Student Management Microservice using Spring Boot Web. The API should allow the creation (POST), retrieval (GET by ID), and listing (GET all) of student records. (You can Use an in-memory collection to store the student data.)
11. Implement the Command Design Pattern for a simple home automation system's universal remote. Create command objects (TurnOnLightCommand, TurnOffLightCommand, StartFanCommand) and a RemoteControl (Invoker) that can execute, queue, and support a basic undo() operation.

12. Implement the Adapter Design Pattern for a Media Player application. You have an old AdvancedMediaPlayer (with playMp4() and playVlc()) and a new MediaPlayer interface (with play(type, filename)). Create a MediaAdapter to make the old advanced player compatible with the new interface.

Programs (Core Design Patterns & SOLID)

These programs can be implemented as standard Java applications in Eclipse. **No external libraries or special configurations are needed.**

Program : Singleton Pattern (Thread-Safe Logger)

The **Singleton** pattern ensures a class has only one instance and provides a global point of access to it. We use a **double-checked locking** mechanism to ensure thread safety and lazy initialization.

File

LoggerService.java

```
public class LoggerService {

// 1. Volatile static instance ensures visibility across threads

private static volatile LoggerService instance;

// 2. Private constructor prevents external instantiation

private LoggerService() {

    if (instance != null) {

        // Protect against reflection-based instantiation

        throw new IllegalStateException("Singleton already initialized.");

    }

}


// 3. Public static method for global access (Thread-Safe)

public static LoggerService getInstance() {

    if (instance == null) {

        // First check: Reduces synchronization overhead

        synchronized (LoggerService.class) {

            if (instance == null) {

                // Second check: Initializes only if null

                instance = new LoggerService();

            }

        }

    }
```

```
    return instance;
}

public void log(String message) {
    System.out.println("LOG [" + System.currentTimeMillis() + "]: " + message);
}
}
// Client Test:
class SingletonTest {
public static void main(String[] args) {
// Both objects refer to the SAME instance
LoggerService logger1 = LoggerService.getInstance();
LoggerService logger2 = LoggerService.getInstance();
    logger1.log("Application started.");
    logger2.log("Configuration loaded.");

    // Proof of Singleton: They must have the same hashcode
    System.out.println("\nProof of Singleton: " + (logger1 == logger2));
}
}
``` | The getInstance() method uses double-checked locking with the `volatile` keyword, which is the standard way to create a thread-safe, lazily initialized singleton in Java. |

Program B2: Open/Closed Principle (OCP)

The **OCP** states that software entities should be **open for extension**, but **closed for modification**. We achieve this by defining an interface and extending it, avoiding changes to the main processing logic (AreaCalculator).

File

Shape.java

```
// 1. Abstraction (Closed for Modification)
interface Shape {
double calculateArea();
```

File

```
}
// 2. Concrete Implementation (Open for Extension)
class Circle implements Shape {
private double radius;
public Circle(double r) { this.radius = r; }
@Override
public double calculateArea() {
return Math.PI * radius * radius;
}
}
// 3. Concrete Implementation (Open for Extension)
class Square implements Shape {
private double side;
public Square(double s) { this.side = s; }
@Override
public double calculateArea() {
return side * side;
}
}
// 4. Client Logic (Closed for Modification)
class AreaCalculator {
public double getTotalArea(Shape[] shapes) {
double totalArea = 0;
for (Shape shape : shapes) {
// The calculator uses the abstraction, not the concrete type
totalArea += shape.calculateArea();
}
return totalArea;
}
}
```

// Client Test:

class OCPTest {

public static void main(String[] args) {

Shape[] shapes = new Shape[]{

new Circle(5),

new Square(4)

};

AreaCalculator calculator = new AreaCalculator();

System.out.println("Total Area: " + calculator.getTotalArea(shapes));

// If a new shape (e.g., Triangle) is added, AreaCalculator.java

// does NOT need to be modified. This satisfies OCP.

}

}

``` | The AreaCalculator class does not need to know about `Circle` or `Square` specifically. If you add a `Triangle` class, you only extend the `Shape` interface; the `AreaCalculator` logic remains unchanged. |

Program : Single Responsibility Principle (SRP)

The **SRP** states that a class should have only one reason to change, meaning it should have a single responsibility. We break down a monolithic Student class into separate, focused classes.

File

Student.java

// 1. Responsibility: Data Holder

class Student {

private int rollNo;

private String name;

private double marks; // Example: Average marks

public Student(int rollNo, String name, double marks) {

this.rollNo = rollNo;

this.name = name;

this.marks = marks;

}
```

```
// Getters and Setters...
public double getMarks() { return marks; }
@Override
public String toString() {
   return "Student [RollNo=" + rollNo + ", Name=" + name + "]";
}
}
// 2. Responsibility: Grading Logic
class GradeCalculator {
public String calculateGrade(Student student) {
double m = student.getMarks();
if (m >= 75) return "A+";
if (m >= 60) return "A";
return "B";
}
}
// 3. Responsibility: Persistence (Simulated)
class StudentRepository {
public void save(Student student) {
// Simulating saving data to a file or DB
System.out.println("SAVED: " + student.getName() +
" record to file successfully.");
}
}
// Client Test:
class SRPTest {
public static void main(String[] args) {
Student s1 = new Student(101, "Amit Sharma", 82.5);
GradeCalculator grader = new GradeCalculator();
StudentRepository repo = new StudentRepository();
   System.out.println(s1);
```

```
    System.out.println("Grade: " + grader.calculateGrade(s1));
    repo.save(s1);

    // If the grading policy changes, only GradeCalculator needs modification.
    // If the persistence mechanism changes, only StudentRepository needs modification.
}
}
```

``` | The original class is split: Student holds data, `GradeCalculator` handles calculation, and `StudentRepository` handles I/O. This separation means a change in grading logic won't affect the data or persistence logic. |

Program : Prototype Pattern (Deep Cloning)

The **Prototype** pattern creates new objects by copying an existing object (the prototype). It uses the clone() method. Since deep cloning is required, we must ensure all mutable fields are also cloned.

File

Shape.java

```
// Marker interface for cloning
interface CloneableShape extends Cloneable {
CloneableShape clone();
}
class Dimension implements Cloneable {
private int value;
public Dimension(int v) { this.value = v; }
public int getValue() { return value; }
// Must override clone() to enable deep copy
@Override
protected Object clone() throws CloneNotSupportedException {
return super.clone();
}
}
class Circle implements CloneableShape {
private int x;
```
```

```
private Dimension radius;
public Circle(int x, int r) {
   this.x = x;
   this.radius = new Dimension(r);
}

// Implementing deep copy
@Override
public CloneableShape clone() {
   try {
      Circle clonedCircle = (Circle) super.clone();
      // Crucial: Deep copy the mutable Dimension object
      clonedCircle.radius = (Dimension) this.radius.clone();
      return clonedCircle;
   } catch (CloneNotSupportedException e) {
      // Should not happen as we implement Cloneable
      throw new RuntimeException(e);
   }
}

// Setter for radius to show deep copy effect
public void setRadiusValue(int val) { this.radius = new Dimension(val); }
public int getRadiusValue() { return radius.getValue(); }
}
// Client Test:
class PrototypeTest {
public static void main(String[] args) {
Circle original = new Circle(10, 5); // x=10, radius=5
Circle cloned = (Circle) original.clone();
   System.out.println("Original Radius: " + original.getRadiusValue()); // 5
```

```
    // Change the clone's internal mutable state
    cloned.setRadiusValue(15);

    // Check if the original object was affected (Deep Copy Check)
    System.out.println("Cloned Radius: " + cloned.getRadiusValue());   // 15
    System.out.println("Original Radius (After change): " + original.getRadiusValue()); // Still 5
    // If the original was 15, it would be a shallow copy.
}
}
```
``` | We create a separate Dimension class to represent a mutable part of `Circle`. The `clone()` method in `Circle` must explicitly call `clone()` on its `Dimension` field (`clonedCircle.radius = (Dimension) this.radius.clone();`) to ensure a deep copy, preventing changes in the clone from affecting the original. |

Program : Interface Segregation Principle (ISP)

The **ISP** states that clients should not be forced to depend on interfaces they do not use. We split one large, "fat" interface into smaller, role-specific interfaces.

File

Worker.java

```
// 1. Segregated Interfaces (ISP adherence)
interface Workable {
void work();
// No other methods like eat() or getPaid()
}
interface Feedable {
void eat();
}
interface PayAble {
void getPaid();
}
// 2. HumanWorker implements all relevant interfaces
class HumanWorker implements Workable, Feedable, PayAble {
```
```

```
@Override
public void work() { System.out.println("Human working on task..."); }
@Override
public void eat() { System.out.println("Human eating lunch..."); }
@Override
public void getPaid() { System.out.println("Human receiving paycheck..."); }
}
// 3. RobotWorker implements only the relevant interfaces
class RobotWorker implements Workable, PayAble { // Robot doesn't need to eat/sleep
@Override
public void work() { System.out.println("Robot working 24/7..."); }
@Override
// Robots are paid for maintenance/utility, even if not salary
public void getPaid() { System.out.println("Robot needs maintenance budget..."); }
}
// Client Test:
class ISPTest {
public static void manageWork(Workable w) { w.work(); }
public static void manageLunch(Feedable f) { f.eat(); }
public static void main(String[] args) {
   HumanWorker human = new HumanWorker();
   RobotWorker robot = new RobotWorker();

   manageWork(human);
   manageWork(robot); // Client (manager) can only call work() on the robot

   manageLunch(human);
   // manageLunch(robot); // ERROR: Robot does not implement Feedable, demonstrating ISP
}
}
```

``` | The large interface is split into Workable, `Feedable`, and `Payable`. The `RobotWorker` only implements what it needs (`Workable`, `PayAble`), thereby avoiding dependency on the irrelevant `Feedable` interface, satisfying ISP. |

Program: Abstract Factory Pattern

The **Abstract Factory** pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Here, the families are Camera Types (HighResolutionCamera and StandardCamera) and their related components (PhotoCapturer and VideoRecorder).

File

MobileFactory.java

```
// Abstract Products
interface PhotoCapturer {
void capture();
}
interface VideoRecorder {
void record();
}
// Abstract Factory
interface MobileCameraFactory {
PhotoCapturer createPhotoCapturer();
VideoRecorder createVideoRecorder();
}
// Concrete Product Family 1 (High Resolution)
class HDPhotoCapturer implements PhotoCapturer {
@Override
public void capture() { System.out.println("Captured photo in 48MP resolution."); }
}
class HDVideoRecorder implements VideoRecorder {
@Override
public void record() { System.out.println("Recording video in 4K UHD."); }
}
// Concrete Factory 1
```
```

```
class HighResolutionCameraFactory implements MobileCameraFactory {
@Override
public PhotoCapturer createPhotoCapturer() { return new HDPhotoCapturer(); }
@Override
public VideoRecorder createVideoRecorder() { return new HDVideoRecorder(); }
}
// Concrete Factory 2 (Standard)
class StandardPhotoCapturer implements PhotoCapturer {
@Override
public void capture() { System.out.println("Captured photo in 12MP resolution."); }
}
class StandardVideoRecorder implements VideoRecorder {
@Override
public void record() { System.out.println("Recording video in 1080p."); }
}
class StandardCameraFactory implements MobileCameraFactory {
@Override
public PhotoCapturer createPhotoCapturer() { return new StandardPhotoCapturer(); }
@Override
public VideoRecorder createVideoRecorder() { return new StandardVideoRecorder(); }
}
// Client Test:
class MobileClient {
public void testCamera(MobileCameraFactory factory) {
PhotoCapturer photo = factory.createPhotoCapturer();
VideoRecorder video = factory.createVideoRecorder();
photo.capture();
video.record();
}
}
class AbstractFactoryTest {
```

```
public static void main(String[] args) {
MobileClient client = new MobileClient();
   System.out.println("--- Using High Resolution Camera ---");
   client.testCamera(new HighResolutionCameraFactory());

   System.out.println("\n--- Using Standard Resolution Camera ---");
   client.testCamera(new StandardCameraFactory());
}
}
```
``` | The client code uses the abstract factory (MobileCameraFactory) to create products, meaning it doesn't need to know if it's getting an HD or Standard component. It guarantees that all components created by one factory instance belong to the same family. |

Program : Memento Pattern

The **Memento** pattern allows an object to restore its state to a previous state. It involves three roles: **Originator** (the object whose state is saved), **Memento** (the saved state), and **Caretaker** (manages the history of Mementos).

File

Memento.java

```
// 1. Memento: The saved state (Immutable)
class EditorMemento {
private final String state;
public EditorMemento(String state) {
this.state = state;
}
public String getSavedState() {
return state;
}
}
// 2. Originator: The object whose state is being tracked
class TextEditor {
private String text = "";
```
```

```
public void type(String words) {
    text += words;
    System.out.println("Current Text: " + text);
}

// Creates a Memento (Save State)
public EditorMemento save() {
    return new EditorMemento(text);
}

// Restores state from a Memento (Undo)
public void restore(EditorMemento memento) {
    this.text = memento.getSavedState();
}
}
// 3. Caretaker: Manages history
class History {
private EditorMemento lastMemento;
public void save(EditorMemento memento) {
    this.lastMemento = memento;
}

public EditorMemento undo() {
    return lastMemento;
}
}
// Client Test:
class MementoTest {
public static void main(String[] args) {
TextEditor editor = new TextEditor();
History history = new History();
```

```
    editor.type("Hello ");
    history.save(editor.save()); // Save 1

    editor.type("World! ");
    history.save(editor.save()); // Save 2

    editor.type("This is new text.");

    System.out.println("\n--- UNDO ---");
    editor.restore(history.undo()); // Restore to Save 2
    System.out.println("Text after undo: " + editor.text);
}
}
``` | The TextEditor (Originator) creates `EditorMemento` objects containing its state. The `History` (Caretaker) holds the last memento. When `restore()` is called, the editor sets its internal state using the data from the memento. |

Program : Strategy Pattern

The **Strategy** pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows the client to switch payment methods seamlessly at runtime.

File

Payment.java

```
// 1. Strategy Interface
interface PaymentStrategy {
void pay(double amount);
}
// 2. Concrete Strategy 1
class CreditCardPayment implements PaymentStrategy {
private String cardNumber;
public CreditCardPayment(String card) { this.cardNumber = card; }
@Override
public void pay(double amount) {
```

```
System.out.println("Paid Rs." + amount + " using Credit Card ending in " +
cardNumber.substring(12));
}
}
// 3. Concrete Strategy 2
class PayPalPayment implements PaymentStrategy {
private String email;
public PayPalPayment(String email) { this.email = email; }
@Override
public void pay(double amount) {
System.out.println("Paid Rs." + amount + " using PayPal account: " + email);
}
}
// 4. Context: The class that holds a reference to the strategy
class ShoppingCart {
private PaymentStrategy paymentMethod;
private double totalAmount;
public ShoppingCart(double amount) {
   this.totalAmount = amount;
}

public void setPaymentStrategy(PaymentStrategy strategy) {
   this.paymentMethod = strategy;
}

public void checkout() {
   if (paymentMethod == null) {
      System.out.println("Error: No payment method selected.");
      return;
   }
   System.out.println("\nProcessing order of Rs." + totalAmount);
```

```
    paymentMethod.pay(totalAmount);
}
}
// Client Test:
class StrategyTest {
public static void main(String[] args) {
ShoppingCart cart = new ShoppingCart(1500.75);
    // Strategy 1: Use Credit Card
    cart.setPaymentStrategy(new CreditCardPayment("4560123456789012"));
    cart.checkout();

    // Strategy 2: Switch to PayPal at runtime
    cart.setPaymentStrategy(new PayPalPayment("user@example.com"));
    cart.checkout();
}
}
``` | The ShoppingCart (Context) holds a reference to the `PaymentStrategy`. By calling `setPaymentStrategy()`, the application can dynamically change its behavior (how it processes payment) without modifying the `ShoppingCart` class. |

Programs (Advanced Patterns & Spring)

Program : Spring DI/IoC (HR Application)

1. Design a simple HR Application focusing on Spring's Inversion of Control (IoC) and Dependency Injection (DI). Create service and repository classes and demonstrate DI using Constructor Injection or Setter Injection to manage dependencies between components without manual instantiation.

This program demonstrates **Dependency Injection (DI)** using **Constructor Injection**, which is the preferred method in modern Spring Boot development.

File

HRApp.java

File

```
// 1. Component 1: Repository (Dependency)
@Component
class CandidateRepository {
public String findCandidateEmail(int candidateId) {
// Simulating data lookup
return "candidate" + candidateId + "@mail.com";
}
}
// 2. Component 2: Service (The client that requires the dependency)
@Service
class MessageService {
private final CandidateRepository repository;
// CONSTRUCTOR INJECTION: Spring automatically injects CandidateRepository
public MessageService(CandidateRepository repository) {
    this.repository = repository;
    System.out.println("MessageService initialized with CandidateRepository.");
}

public void sendMessage(int candidateId) {
    String email = repository.findCandidateEmail(candidateId);
    String message = "Dear Candidate, You have been selected for an interview.";
    // Simulating sending the message
    System.out.println("--- Sending Message ---");
    System.out.println("To: " + email);
    System.out.println("Message: " + message);
    System.out.println("-----------------------");
}
}
// 3. Main Application Class (Entry point)
```

```
@SpringBootApplication

public class HrApplication {

public static void main(String[] args) {

// This command starts the Spring IoC Container

ApplicationContext context = SpringApplication.run(HrApplication.class, args);

   // Retrieve the service bean from the context (IoC)

   MessageService service = context.getBean(MessageService.class);


   service.sendMessage(101);

}

}
```
``` | The @Component and `@Service` annotations mark classes as Spring Beans. By using the constructor in `MessageService`, we tell Spring (the IoC Container) to automatically create and inject an instance of `CandidateRepository` when it creates `MessageService`. This is Dependency Injection. |

Program : Spring Security (Basic In-Memory Authentication)

This is a simplified security module. Due to the 2-hour limit and "no database" constraint, we use Spring Boot's built-in **In-Memory User Details** for authentication.

File

SecurityConfig.java

```
// 1. Security Configuration

@Configuration

@EnableWebSecurity

public class SecurityConfig {

// 2. Configure In-Memory User Details

@Bean

public UserDetailsService userDetailsService() {

   UserDetails user = User.builder()

      .username("user")

      .password(passwordEncoder().encode("pass123")) // Password must be encoded

      .roles("USER")
```
```

```
        .build();

    UserDetails admin = User.builder()
        .username("admin")
        .password(passwordEncoder().encode("admin123"))
        .roles("USER", "ADMIN")
        .build();

    return new InMemoryUserDetailsManager(user, admin);
}

// 3. Password Encoder (REQUIRED for any modern Spring Security setup)
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(); // Strong password hashing algorithm
}

// 4. Configure Access Rules (Authorization)
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for simplicity in API
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/public").permitAll() // Anyone can access
            .requestMatchers("/api/admin/**").hasRole("ADMIN") // Only ADMIN role
            .anyRequest().authenticated() // All other requests need login
        )
        .httpBasic(withDefaults()); // Use basic HTTP authentication (popup box)
    return http.build();
}
}
```

```
// 5. Simple Controller to test access
@RestController
@RequestMapping("/api")
class TestController {
@GetMapping("/public")
public String publicAccess() {
return "This is a public, unauthenticated endpoint.";
}
@GetMapping("/user")
public String userAccess() {
   return "This is for authenticated users (USER or ADMIN).";
}

@GetMapping("/admin/data")
public String adminAccess() {
   return "This is SECRET admin data! (Only ADMIN can see).";
}
}
```
``` | In-Memory Users: We define two users, user and `admin`, in `userDetailsService()`. Authorization: The `securityFilterChain` defines access rules. Any request to `/api/admin/**` requires the `ADMIN` role. All other endpoints are locked unless explicitly set to `permitAll()`. |

Program: Full SOLID Refactoring (DIP Focus)

This implements the **Dependency Inversion Principle (DIP)**, where high-level modules (OrderProcessor) should not depend on low-level modules (DatabaseSaver). Both should depend on abstractions (OrderRepository).

File

OrderProcessor.java

```
// 1. DIP: Abstraction (High and Low level depend on this)
interface OrderRepository {
void save(Order order);
}
```
```

```
// 2. Low-Level Module (Depends on Abstraction)
class DatabaseSaver implements OrderRepository {
@Override
public void save(Order order) {
System.out.println("DIP: Low-level DatabaseSaver saved order " + order.id);
}
}
// 3. SRP/OCP: Dedicated class for a single rule
interface IPriceCalculator {
double calculate(double basePrice);
}
class StandardPriceCalculator implements IPriceCalculator {
@Override
public double calculate(double basePrice) {
return basePrice * 1.05; // 5% tax
}
}
// 4. High-Level Module (Depends on Abstraction)
class Order {
public int id;
public double basePrice;
public Order(int id, double price) { this.id = id; this.basePrice = price; }
}
class OrderProcessor {
private final OrderRepository repository; // Depends on ABSTRACTION
private final IPriceCalculator calculator;
// Dependency Injection (Constructor)
public OrderProcessor(OrderRepository repository, IPriceCalculator calculator) {
    this.repository = repository;
    this.calculator = calculator;
}
```

```
public void processOrder(Order order) {
    // High-level logic
    double finalPrice = calculator.calculate(order.basePrice);
    System.out.println("SRP: Calculated final price: " + finalPrice);

    repository.save(order); // Call the abstraction
}
}
// Client Test:
class SOLIDTest {
public static void main(String[] args) {
// Instantiate low-level module
OrderRepository saver = new DatabaseSaver();
IPriceCalculator calculator = new StandardPriceCalculator();
    // Inject the dependency into the high-level module
    OrderProcessor processor = new OrderProcessor(saver, calculator);

    processor.processOrder(new Order(101, 500.00));

    // If persistence changes (e.g., to FileSaver),
    // OrderProcessor class remains untouched (DIP, OCP).
}
}
```
``` | The OrderProcessor (high-level) depends on the `OrderRepository` interface (abstraction), not the concrete `DatabaseSaver` (low-level). This adheres to DIP. Other SOLID principles like SRP and OCP are also demonstrated by the specialized `IPriceCalculator`. |

Program : Bridge Pattern

The **Bridge** pattern decouples an abstraction from its implementation so that the two can vary independently. Here, the **Abstraction** is the Vehicle type, and the **Implementation** is the ManufactureProcess.
```

File

Bridge.java

```
// 1. Implementation Interface (The "Bridge")
interface ManufactureProcess {
void produce();
void assemble();
}
// 2. Concrete Implementations
class CarManufacture implements ManufactureProcess {
@Override
public void produce() { System.out.print("Producing Car Body..."); }
@Override
public void assemble() { System.out.println("Assembling Car Engine and Chassis."); }
}
class BikeManufacture implements ManufactureProcess {
@Override
public void produce() { System.out.print("Producing Bike Frame..."); }
@Override
public void assemble() { System.out.println("Assembling Bike Wheels and Handlebars."); }
}
// 3. Abstraction (The Vehicle)
abstract class Vehicle {
protected ManufactureProcess manufacture; // Reference to the Implementation
public Vehicle(ManufactureProcess manufacture) {
    this.manufacture = manufacture;
}

public abstract void build();
}
// 4. Refined Abstractions
```

```
class Sedan extends Vehicle {
public Sedan(ManufactureProcess manufacture) { super(manufacture); }
@Override
public void build() {
System.out.print("Building Sedan: ");
manufacture.produce();
manufacture.assemble();
}
}
class SportsBike extends Vehicle {
public SportsBike(ManufactureProcess manufacture) { super(manufacture); }
@Override
public void build() {
System.out.print("Building Sports Bike: ");
manufacture.produce();
manufacture.assemble();
}
}
// Client Test:
class BridgeTest {
public static void main(String[] args) {
// Abstraction (Sedan) connected to Implementation (CarManufacture)
Vehicle sedan = new Sedan(new CarManufacture());
sedan.build();
   // Abstraction (SportsBike) connected to Implementation (BikeManufacture)
   Vehicle bike = new SportsBike(new BikeManufacture());
   bike.build();

   // Bridge allows Abstractions to change independently of Implementations.
   // E.g., You could easily create an SUV (new abstraction)
   // that still uses the CarManufacture (existing implementation).
```

}

}

``` | The Vehicle classes (Abstraction) and the `ManufactureProcess` classes (Implementation) are connected by the `manufacture` reference (the Bridge). This separation allows new vehicle types (e.g., `SUV`) or new processes (e.g., `LuxuryManufacture`) to be introduced without modifying the other side. |

Program : Decorator Pattern

The **Decorator** pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

File

Coffee.java

```
// 1. Component Interface (The core object)
interface Coffee {
String getDescription();
double getCost();
}
// 2. Concrete Component
class BasicCoffee implements Coffee {
@Override
public String getDescription() {
return "Simple Black Coffee";
}
@Override
public double getCost() {
return 50.0;
}
}
// 3. Abstract Decorator (Must implement the Component interface)
abstract class CoffeeDecorator implements Coffee {
protected Coffee decoratedCoffee;
public CoffeeDecorator(Coffee coffee) {
```
```

```
    this.decoratedCoffee = coffee;
}

@Override
public String getDescription() {
    return decoratedCoffee.getDescription();
}
@Override
public double getCost() {
    return decoratedCoffee.getCost();
}
}
// 4. Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
public MilkDecorator(Coffee coffee) { super(coffee); }
@Override
public String getDescription() {
return super.getDescription() + ", with Milk";
}
@Override
public double getCost() {
return super.getCost() + 15.0; // Added cost
}
}
class SugarDecorator extends CoffeeDecorator {
public SugarDecorator(Coffee coffee) { super(coffee); }
@Override
public String getDescription() {
return super.getDescription() + ", with Sugar";
}
@Override
```

```
public double getCost() {
return super.getCost() + 5.0;
}
}
// Client Test:
class DecoratorTest {
public static void main(String[] args) {
// 1. Order a basic coffee
Coffee coffee1 = new BasicCoffee();
System.out.println("Order 1: " + coffee1.getDescription() + " | Cost: Rs." + coffee1.getCost());
   // 2. Order a coffee with milk and sugar (wrapped sequentially)
   Coffee coffee2 = new BasicCoffee();
   coffee2 = new MilkDecorator(coffee2);   // Decorate with Milk
   coffee2 = new SugarDecorator(coffee2); // Decorate with Sugar

   System.out.println("Order 2: " + coffee2.getDescription() + " | Cost: Rs." + coffee2.getCost());

   // Final cost for Order 2: 50 + 15 + 5 = 70.0
}
}
```
``` | The MilkDecorator wraps a `Coffee` object. When its `getDescription()` is called, it first calls the wrapped object's method (`super.getDescription()`) and then adds its own description ("with Milk"). This allows for flexible, recursive combination of features. |

Program : Factory Method Pattern

The **Factory Method** pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. The concrete factory classes (NYPizzaStore, ChicagoPizzaStore) are responsible for defining *how* to create the specific products (NYCheesePizza, ChicagoCheesePizza).

File

Pizza.java

```
// 1. Product Abstraction
abstract class Pizza {
```
```

File

```
protected String name;

public abstract void prepare();

public void bake() { System.out.println("Bake for 25 mins..."); }

// Other common methods...

}

// 2. Concrete Products

class NYCheesePizza extends Pizza {

public NYCheesePizza() { this.name = "NY Style Sauce and Cheese Pizza"; }

@Override

public void prepare() { System.out.println("Preparing " + name + ": Thin crust dough, fresh mozzarella..."); }

}

class ChicagoVeggiePizza extends Pizza {

public ChicagoVeggiePizza() { this.name = "Chicago Style Deep Dish Veggie Pizza"; }

@Override

public void prepare() { System.out.println("Preparing " + name + ": Thick crust, bell peppers, spinach..."); }

}

// 3. Creator Abstraction (The Factory Method is declared here)

abstract class PizzaStore {

// The Factory Method (Subclasses implement this)

protected abstract Pizza createPizza(String type);

public Pizza orderPizza(String type) {

    Pizza pizza = createPizza(type); // The factory call

    pizza.prepare();

    pizza.bake();

    return pizza;

}

}

// 4. Concrete Creators (Decides which product to instantiate)
```

```
class NYPizzaStore extends PizzaStore {
@Override
protected Pizza createPizza(String type) {
if (type.equals("cheese")) {
return new NYCheesePizza();
} else if (type.equals("pepperoni")) {
return new NYCheesePizza(); // Simplified
} else return null;
}
}
class ChicagoPizzaStore extends PizzaStore {
@Override
protected Pizza createPizza(String type) {
if (type.equals("cheese")) {
return new ChicagoVeggiePizza(); // Deep dish cheese is a type of veggie pizza
} else if (type.equals("veggie")) {
return new ChicagoVeggiePizza();
} else return null;
}
}
// Client Test:
class FactoryMethodTest {
public static void main(String[] args) {
PizzaStore nyStore = new NYPizzaStore();
PizzaStore chicagoStore = new ChicagoPizzaStore();
   System.out.println("--- Ordering from NY Store ---");
   nyStore.orderPizza("cheese");

   System.out.println("\n--- Ordering from Chicago Store ---");
   chicagoStore.orderPizza("veggie");
```

```
    // The client uses the store, not the pizza factory itself.
}
}
``` | The abstract PizzaStore defines the `orderPizza()` workflow, but delegates the actual object creation to the abstract `createPizza()` method. Concrete stores (`NYPizzaStore`, `ChicagoPizzaStore`) implement `createPizza()`, choosing the specific pizza class to instantiate. |

Program : Observer Pattern

The **Observer** pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

File

Weather.java

```
import java.util.ArrayList;
import java.util.List;
// 1. Observer Interface (The dependents)
interface Display {
void update(float temperature, float humidity);
}
// 2. Subject Interface (The source of data)
interface Subject {
void registerObserver(Display d);
void removeObserver(Display d);
void notifyObservers();
}
// 3. Concrete Subject (The Weather Station)
class WeatherStation implements Subject {
private List<Display> observers = new ArrayList<>();
private float temperature;
private float humidity;
@Override
public void registerObserver(Display d) {
    observers.add(d);
```

```
}
@Override
public void removeObserver(Display d) {
   observers.remove(d);
}
@Override
public void notifyObservers() {
   for (Display display : observers) {
      display.update(temperature, humidity);
   }
}

// Method to change the state and notify
public void setMeasurements(float temp, float hum) {
   this.temperature = temp;
   this.humidity = hum;
   notifyObservers();
}
}
// 4. Concrete Observer 1
class CurrentConditionsDisplay implements Display {
@Override
public void update(float temp, float hum) {
System.out.println("Current Conditions: Temp=" + temp + "C, Humidity=" + hum + "%");
}
}
// 5. Concrete Observer 2
class ForecastDisplay implements Display {
@Override
public void update(float temp, float hum) {
// Simplified forecast logic
```

```
if (temp > 30) {
System.out.println("Forecast: Next day will be HOT.");
} else {
System.out.println("Forecast: Next day will be pleasant.");
}
}
}
// Client Test:
class ObserverTest {
public static void main(String[] args) {
WeatherStation station = new WeatherStation();
    Display current = new CurrentConditionsDisplay();
    Display forecast = new ForecastDisplay();

    // Register observers
    station.registerObserver(current);
    station.registerObserver(forecast);

    System.out.println("--- First update ---");
    station.setMeasurements(32.5f, 65.0f); // All observers notified

    System.out.println("\n--- Second update ---");
    station.setMeasurements(25.0f, 70.0f); // All observers notified again

    // Remove one observer
    station.removeObserver(forecast);

    System.out.println("\n--- Third update (Forecast removed) ---");
    station.setMeasurements(40.0f, 50.0f); // Only CurrentConditions is notified
}
}
```

``` | The WeatherStation (Subject) maintains a list of `Display` objects (Observers). When `setMeasurements()` is called, it changes its state and then calls `notifyObservers()`, which iterates through the list, calling the `update()` method on every registered observer. |

Program: Command Pattern

The **Command** pattern encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

File

Command.java

```
import java.util.Stack;

// 1. Command Interface (Encapsulates the action)
interface Command {
void execute();
void undo();
}
// 2. Receiver (The object that knows how to perform the action)
class Light {
public void turnOn() { System.out.println("Light is ON"); }
public void turnOff() { System.out.println("Light is OFF"); }
}
// 3. Concrete Command 1 (Turn On)
class TurnOnLightCommand implements Command {
private Light light;
public TurnOnLightCommand(Light light) { this.light = light; }
@Override
public void execute() { light.turnOn(); }
@Override
public void undo() { light.turnOff(); } // Undo means turning off
}
// 4. Concrete Command 2 (Turn Off)
class TurnOffLightCommand implements Command {
private Light light;
public TurnOffLightCommand(Light light) { this.light = light; }
```
```

```
@Override
public void execute() { light.turnOff(); }
@Override
public void undo() { light.turnOn(); } // Undo means turning on
}
// 5. Invoker (The remote control, queues and executes commands)
class RemoteControl {
private Command slot; // Single slot for simplicity
private Stack<Command> history = new Stack<>(); // For Undo
public void setCommand(Command command) {
   this.slot = command;
}

public void pressButton() {
   slot.execute();
   history.push(slot); // Save command to history for undo
}

public void pressUndo() {
   if (!history.isEmpty()) {
      Command lastCommand = history.pop();
      System.out.print("UNDO: ");
      lastCommand.undo();
   } else {
      System.out.println("Nothing to undo.");
   }
}
}
// Client Test:
class CommandTest {
public static void main(String[] args) {
```

```
RemoteControl remote = new RemoteControl();
Light livingRoomLight = new Light();
    // 1. Turn ON the light
    Command onCommand = new TurnOnLightCommand(livingRoomLight);
    remote.setCommand(onCommand);
    remote.pressButton();

    // 2. Turn OFF the light
    Command offCommand = new TurnOffLightCommand(livingRoomLight);
    remote.setCommand(offCommand);
    remote.pressButton();

    // 3. Undo the last action (Turn OFF)
    remote.pressUndo(); // Output: UNDO: Light is ON

    // 4. Undo the previous action (Turn ON)
    remote.pressUndo(); // Output: UNDO: Light is OFF
}
}
```
``` | The RemoteControl (Invoker) holds a generic `Command` object. The specific command (`TurnOnLightCommand`) holds a reference to the `Light` (Receiver). The invoker triggers `execute()` without knowing the receiver or the specific action, and uses the `undo()` method for reversal. |

Program : Adapter Pattern

The **Adapter** pattern converts the interface of a class into another interface that clients expect. It allows classes with incompatible interfaces to work together. We adapt an old AdvancedMediaPlayer to the new MediaPlayer interface.

File

Adapter.java

```
// 1. Target Interface (The interface the client expects)
interface MediaPlayer {
void play(String audioType, String fileName);
```
```

File

```
}
// 2. Adaptee (The existing, incompatible class/library)
class AdvancedMediaPlayer {
public void playMp4(String fileName) {
System.out.println("Playing MP4 file: " + fileName);
}
public void playVlc(String fileName) {
System.out.println("Playing VLC file: " + fileName);
}
}
// 3. Adapter (Converts the Adaptee to the Target)
class MediaAdapter implements MediaPlayer {
private AdvancedMediaPlayer advancedPlayer;
public MediaAdapter() {
   this.advancedPlayer = new AdvancedMediaPlayer();
}

// The adapter provides the expected interface method
@Override
public void play(String audioType, String fileName) {
   if (audioType.equalsIgnoreCase("mp4")) {
      advancedPlayer.playMp4(fileName); // Call the Adaptee's specific method
   } else if (audioType.equalsIgnoreCase("vlc")) {
      advancedPlayer.playVlc(fileName);
   } else {
      System.out.println("Invalid format. Cannot play " + audioType);
   }
}
}
// 4. Client (Uses the Target Interface)
```

```
class AudioPlayer implements MediaPlayer {
private MediaAdapter mediaAdapter;
@Override
public void play(String audioType, String fileName) {
   // Play MP3 directly (compatible format)
   if (audioType.equalsIgnoreCase("mp3")) {
      System.out.println("Playing MP3 file: " + fileName);
   }
   // Use the Adapter for incompatible formats
   else if (audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")) {
      mediaAdapter = new MediaAdapter();
      mediaAdapter.play(audioType, fileName); // Delegates call to the Adapter
   } else {
      System.out.println("Cannot play media type: " + audioType);
   }
}
}
// Client Test:
class AdapterTest {
public static void main(String[] args) {
AudioPlayer player = new AudioPlayer();
   player.play("mp3", "test.mp3");
   player.play("mp4", "movie.mp4"); // Handled by Adapter -> AdvancedMediaPlayer.playMp4
   player.play("vlc", "clip.vlc");  // Handled by Adapter -> AdvancedMediaPlayer.playVlc
   player.play("avi", "video.avi");
}
}
```
``` | The AudioPlayer (Client) expects the `MediaPlayer` interface. To use the features of the incompatible `AdvancedMediaPlayer` (Adaptee), we introduce the `MediaAdapter`. The adapter implements the `MediaPlayer` interface and translates the generic `play()` request into the specific, legacy calls (`playMp4()`, `playVlc()`). |
```