Week 2 Additional Tutorial Solution

Table of contents

1 SOLID

1. **Single Responsibility Principle (SRP):**

```java
// User data handling class
public class UserData {
    private String name;
    private String email;

    public UserData(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void updateEmail(String newEmail) {
        this.email = newEmail;
    }
}

// User persistence logic
public class UserPersistence {
    public void saveUser(UserData userData) {
```

```
        System.out.println("User saved: " + userData.getName());
        // Logic to save user to a database
    }
}
```

The `UserProfile` class has been refactored into two separate classes: `UserData` and `UserPersistence`. `UserData` handles user data-related responsibilities, while `UserPersistence` handles user persistence logic.

2. **Open/Closed Principle (OCP):**

```
public interface DiscountStrategy {
    double calculateDiscount(double price);
}

public class FixedDiscount implements DiscountStrategy {
    private final double discountRate;

    public FixedDiscount(double discountRate) {
        this.discountRate = discountRate;
    }

    @Override
    public double calculateDiscount(double price) {
        return price * discountRate;
    }
}

public class DiscountCalculator {
    private final DiscountStrategy discountStrategy;

    public DiscountCalculator(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public double calculateDiscount(double price) {
        return discountStrategy.calculateDiscount(price);
    }
}
```

The `DiscountCalculator` class has been refactored to accept a `DiscountStrategy` interface. New discount strategies can be added by implementing the `DiscountStrategy` interface without modifying the existing `DiscountCalculator` class.

3. **Liskov Substitution Principle (LSP):**

```
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}

class Ostrich implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Ostrich is swimming");
    }
}
```

Instead of inheriting from a common `Bird` class, separate interfaces (`Flyable` and `Swimmable`) have been introduced. The `Duck` class implements both interfaces, while the `Ostrich` class only implements the `Swimmable` interface, avoiding invalid method calls.

4. **Interface Segregation Principle (ISP):**

```
interface Printer {
    void print();
}

interface Scanner {
    void scan();
}
```

```
interface Fax {
    void fax();
}

class SmartPrinter implements Printer, Scanner, Fax {
    public void print() {
        // Print logic
    }

    public void scan() {
        // Scan logic
    }

    public void fax() {
        // Fax logic
    }
}
```

The `SmartDevice` interface has been split into three separate interfaces: `Printer`, `Scanner`, and `Fax`. The `SmartPrinter` class implements only the relevant interfaces, avoiding the need to implement irrelevant methods.

5. **Dependency Inversion Principle (DIP):**

```
interface Database {
    void store(Object data);
}

class MySQLDatabase implements Database {
    public void store(Object data) {
        System.out.println("Storing data in MySQL database");
    }
}

class UserManager {
    private Database database;

    public UserManager(Database database) {
        this.database = database;
    }
```

```
    public void saveUser(Object user) {
        database.store(user);
    }
}
```

The `UserManager` class now depends on the `Database` interface instead of the concrete `MySQLDatabase` class. This allows for easier substitution of different database implementations.

6. **Single Responsibility Principle (SRP):**

```
public class OrderProcessor {
    private ErrorLogger errorLogger;

    public OrderProcessor(ErrorLogger errorLogger) {
        this.errorLogger = errorLogger;
    }

    public void processOrder(Order order) {
        try {
            // Process order logic
        } catch (Exception e) {
            errorLogger.logError(e);
        }
    }
}

class ErrorLogger {
    void logError(Exception e) {
        // Log error to a file
    }
}
```

The `OrderProcessor` class has been refactored to handle order processing only, while the `ErrorLogger` class handles error logging responsibilities.

7. **Liskov Substitution Principle (LSP):**

```
abstract class Payment {
    abstract void initiatePayments();
    abstract boolean validatePayment();
}
```

```
class CreditCardPayment extends Payment {
    @Override
    void initiatePayments() {
        // Credit card payment logic
    }

    @Override
    boolean validatePayment() {
        // Credit card validation logic
        return true;
    }
}

class PayPalPayment extends Payment {
    @Override
    void initiatePayments() {
        // PayPal payment logic
    }

    @Override
    boolean validatePayment() {
        // PayPal validation logic
        return true;
    }
}
```

The `Payment` class has been made abstract, and the `initiatePayments` and `validatePayment` methods have been made abstract. Derived classes (`CreditCardPayment` and `PayPalPayment`) provide their own implementations of these methods.

8. **Interface Segregation Principle (ISP):**

```
interface PrintingDevice {
    void printDocument();
}

interface ScanningDevice {
    void scanDocument();
}

interface FaxingDevice {
```

```
    void faxDocument();
}

class MultifunctionPrinter implements PrintingDevice, ScanningDevice, FaxingDevice {
    public void printDocument() {
        // Print document logic
    }

    public void scanDocument() {
        // Scan document logic
    }

    public void faxDocument() {
        // Fax document logic
    }
}
```

The `Printer` interface has been split into three separate interfaces: `PrintingDevice`, `ScanningDevice`, and `FaxingDevice`. The `MultifunctionPrinter` class implements only the relevant interfaces.

9. **Single Responsibility Principle (SRP):**

```
public class UserPreferences {
    public void changeSetting(String setting, String value) {
        // Change settings logic
    }
}

public class UserAuthenticator {
    public boolean login(String username, String password) {
        // Login logic
    }
}
```

The `UserSettings` class has been split into two separate classes: `UserPreferences` and `UserAuthenticator`. `UserPreferences` handles user preference-related responsibilities, while `UserAuthenticator` handles user authentication.

10. **Open/Closed Principle (OCP):**

```
interface ProductFilter {
    Stream<Product> filter(List<Product> products);
}

class ColorFilter implements ProductFilter {
    private final Color color;

    public ColorFilter(Color color) {
        this.color = color;
    }

    @Override
    public Stream<Product> filter(List<Product> products) {
        return products.stream().filter(p -> p.getColor() == color);
    }
}

class SizeFilter implements ProductFilter {
    private final Size size;

    public SizeFilter(Size size) {
        this.size = size;
    }

    @Override
    public Stream<Product> filter(List<Product> products) {
        return products.stream().filter(p -> p.getSize() == size);
    }
}

class ProductFilteringService {
    public Stream<Product> filterProducts(List<Product> products, List<ProductFilter> filt
        return filters.stream()
                .flatMap(filter -> filter.filter(products))
                .distinct();
    }
}
```

The `ProductFilter` interface has been introduced, and new filters (`ColorFilter` and `SizeFilter`) implement this interface. The `ProductFilteringService` class uses these filters to filter products without modifying the existing code.

11. **Liskov Substitution Principle (LSP):**

```
interface Vehicle {
    void refuel();
}

class Car implements Vehicle {
    @Override
    public void refuel() {
        // Refueling logic
    }
}

class ElectricCar implements Vehicle {
    @Override
    public void refuel() {
        // Electric charging logic
    }
}
```

The `Car` and `ElectricCar` classes implement the `Vehicle` interface, and both provide their own implementation of the `refuel` method. This allows the `ElectricCar` class to be substituted for the `Car` class without breaking the expected behavior.

12. **Interface Segregation Principle (ISP):**

```
interface Worker {
    void work();
}

interface Manager extends Worker {
    void manage();
}

class ManagerImpl implements Manager {
    public void work() {
        // Manager-specific work
    }

    public void manage() {
        // Management tasks
    }
}
```

```
class Technician implements Worker {
    public void work() {
        // Technical tasks
    }
}
```

The `Worker` interface has been split into two interfaces: `Worker` and `Manager`. The `ManagerImpl` class implements both interfaces, while the `Technician` class implements only the `Worker` interface, avoiding the need to implement irrelevant methods.

2 GRASP

1. **Information Expert Principle:**

```
class Order {
    Item[] items;

    public Order(Item[] items) {
        this.items = items;
    }

    double calculateTotalPrice() {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice();
        }
        return total;
    }
}

class Item {
    private double price;

    public Item(double price) {
        this.price = price;
    }

    public double getPrice() {
        return price;
    }
```

```
}
```

The `calculateTotalPrice` method has been moved to the `Order` class, as it has the required information (list of items) to perform the calculation.

2. **Creator Principle:**

```
class Task {
    String description;

    public Task(String description) {
        this.description = description;
    }
}

class Project {
    List<Task> tasks = new ArrayList<>();

    void addTask(String description) {
        Task newTask = new Task(description);
        tasks.add(newTask);
    }
}

class User {
    // User details
}
```

The creation of `Task` instances has been moved to the `Project` class, as it is responsible for managing tasks within a project.

3. **Controller Principle:**

```
class UserView {
    private UserController userController;

    public UserView(UserController userController) {
        this.userController = userController;
    }

    void onRegisterButtonClicked() {
        System.out.println("Registering a user...");
```

```
        userController.registerUser();
    }
}

class UserController {
    void registerUser() {
        // Logic to register a user
    }
}
```

The `UserView` class now has a reference to the `UserController` class, which handles the user registration logic. The `UserView` class simply delegates the user registration request to the `UserController`.

4. **Low Coupling Principle:**

```
interface PaymentGateway {
    void makePayment(double amount);
}

class OrderManager {
    private PaymentGateway paymentGateway;

    public OrderManager(PaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
    }

    void processOrder() {
        paymentGateway.makePayment(100); // Example amount
    }
}

class PaypalGateway implements PaymentGateway {
    public void makePayment(double amount) {
        System.out.println("Processing payment of: " + amount + " via PayPal");
    }
}
```

The `OrderManager` class now depends on the `PaymentGateway` interface instead of the concrete `PaymentGateway` class. This reduces coupling and allows for easier substitution of different payment gateway implementations.

5. **High Cohesion Principle:**

```
class ActivityManager {
    void startActivity() {
        System.out.println("Activity started.");
    }

    void stopActivity() {
        System.out.println("Activity stopped.");
    }
}

class ActivityLogger {
    void logActivity() {
        System.out.println("Activity logged.");
    }
}

class NotificationService {
    void sendActivityNotifications() {
        System.out.println("Activity notification sent.");
    }
}
```

The responsibilities of the `ActivityManager` class have been split into three separate classes: `ActivityManager`, `ActivityLogger`, and `NotificationService`. Each class now has a cohesive set of responsibilities.

6. **Polymorphism Principle:**

```
interface Account {
    double calculateInterest();
}

class SavingsAccount implements Account {
    private double balance;

    public SavingsAccount(double balance) {
        this.balance = balance;
    }

    @Override
    public double calculateInterest() {
        return balance * 0.03;
```

```
    }
}

class CheckingAccount implements Account {
    private double balance;

    public CheckingAccount(double balance) {
        this.balance = balance;
    }

    @Override
    public double calculateInterest() {
        return balance * 0.01;
    }
}
```

The `Account` interface has been introduced, and concrete classes (`SavingsAccount` and `CheckingAccount`) implement this interface. This eliminates the need for conditional logic based on account type in the `calculateInterest` method.

7. **Pure Fabrication Principle:**

```
class ProductService {
    private Logger logger;

    public ProductService(Logger logger) {
        this.logger = logger;
    }

    void addProduct(Product product) {
        System.out.println("Product added: " + product.getName());
        logger.log("Product added: " + product.getName());
        // Add product logic
    }
}

class Logger {
    void log(String message) {
        System.out.println("Logging: " + message);
    }
}
```

```
class Product {
    private String name;

    public Product(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

The `Logger` class has been introduced as a pure fabrication to handle logging responsibilities for the `ProductService` class. This separates logging concerns from the core functionality of the `ProductService` class.

8. **Indirection Principle:**

```
interface EmailService {
    void sendEmail(String message);
}

class CustomerManager {
    private EmailService emailService;

    public CustomerManager(EmailService emailService) {
        this.emailService = emailService;
    }

    void sendEmailToCustomer() {
        emailService.sendEmail("Thank you for your purchase!");
    }
}

class EmailClient implements EmailService {
    public void sendEmail(String message) {
        System.out.println("Email sent: " + message);
    }
}
```

The `EmailService` interface has been introduced, and the `CustomerManager` class now depends on this interface instead of the concrete `EmailClient` class. This introduces an indirection layer and reduces coupling between the two classes.

9. **Information Expert Principle:**

```
class Customer {
    Order[] orders;

    public Customer(Order[] orders) {
        this.orders = orders;
    }

    int countOrders() {
        return orders.length;
    }
}

class Order {
    // Order details
}
```

The `countOrders` method has been moved to the `Customer` class, as it has the required information (list of orders) to perform the counting operation.

3 Other Principles

1. **DRY (Don't Repeat Yourself) Principle:**

```
class DiscountCalculator {
    private static final double DISCOUNT_RATE = 0.9;

    double calculateDiscount(double price, ProductType type) {
        return price * DISCOUNT_RATE; // 10% discount
    }
}

enum ProductType {
    BOOK,
    TOY,
    // Other product types
}
```

The repeated logic for calculating discounts has been refactored into a single `calculateDiscount` method in the `DiscountCalculator` class. The discount rate is now a constant, and the method accepts the product type as a parameter.

2. **DRY Principle:**

```
class BankAccount {
    private static final double TRANSACTION_FEE = 2;
    double balance;

    void deposit(double amount) {
        logTransaction("Deposit", amount);
        balance += amount - TRANSACTION_FEE;
    }

    void withdraw(double amount) {
        logTransaction("Withdraw", amount);
        balance -= amount + TRANSACTION_FEE;
    }

    private void logTransaction(String type, double amount) {
        System.out.println(type + ": " + amount);
    }
}
```

The repeated logic for logging transactions and applying transaction fees has been refactored into separate methods (`logTransaction` and `TRANSACTION_FEE` constant). The `deposit` and `withdraw` methods now call these shared methods.

3. **KISS (Keep It Simple, Stupid) Principle:**

```
class UserSession {
    Boolean isLoggedIn;

    boolean checkIfUserIsLoggedIn() {
        return isLoggedIn != null && isLoggedIn;
    }
}
```

The `checkIfUserIsLoggedIn` method has been simplified by using a concise conditional expression, making the code more readable and easier to understand.

4. **KISS Principle:**

```
class ReportGenerator {
    void generateReport(String title, String data, ReportOptions options) {
        // Report generation logic
    }
}

class ReportOptions {
    boolean includeGraphics;
    boolean includeTables;
    boolean isDraft;

    // Constructor, getters, and setters
}
```

The `generateReport` method signature has been simplified by introducing a `ReportOptions` class that encapsulates the various options required for report generation.

5. **Separation of Concerns Principle:**

```
class UserAuthenticator {
    void loginUser(String username, String password) {
        // Login logic
    }
}

class UserDataManager {
    void saveUser(String username) {
        // Save user data
    }
}
```

The responsibilities of user authentication and user data management have been separated into two classes: `UserAuthenticator` and `UserDataManager`.

6. **Principle of Least Surprise:**

```
class Order {
    double total;
    List<Item> items;

    double calculateTotal() {
        double calculatedTotal = 0;
```

```
        for (Item item : items) {
            calculatedTotal += item.price;
        }
        return calculatedTotal;
    }
}
```

The `calculateTotal` method has been refactored to avoid the side effect of modifying the `total` property of the `Order` class. Instead, it calculates and returns the total without modifying the existing state.

7. **Law of Demeter Principle:**

```
class ShoppingSession {
    Cart cart;

    void checkout() {
        double total = cart.getTotalPrice();
        // Checkout logic
    }
}

class Cart {
    List<Item> items;

    double getTotalPrice() {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice();
        }
        return total;
    }
}
```

The `ShoppingSession` class now accesses the `getTotalPrice` method directly on the `Cart` instance, instead of navigating through the `getItems` method to calculate the total price.

8. **Law of Demeter Principle:**

```
class Employee {
    Manager manager;
```

```
    void sendReport(Report report) {
        manager.submitReport(report);
    }
}

class Manager {
    Department department;

    void submitReport(Report report) {
        department.receiveReport(report);
    }
}

class Department {
    void receiveReport(Report report) {
        System.out.println("Report received: " + report);
    }
}
```

The `Employee` class now sends the report directly to the `Manager` instance, instead of navigating through the `Department` instance. The `Manager` class handles the submission of the report to the `Department`.