Day 2 : Applying SOLID Principles, DRY, and YAGNI in Java

Type	Assignment
Attachments	https://students.masaischool.com/assignments/41296?tab=assignmentDetails
Date	@June 29, 2024
Status	In progress

Submission Guidelines:

1. Push your code to GitHub.
2. Submit the Repository link.

Note: Ensure to make the repository public.

Questions

Part 1: Single Responsibility Principle (SRP)

Question 1:
Consider the following class:

```
class Invoice {
    private int invoiceNumber;
```

```
    private double amount;

    public Invoice(int invoiceNumber, double amount) {
        this.invoiceNumber = invoiceNumber;
        this.amount = amount;
    }

    public void printInvoice() {
        // Code to print the invoice
    }

    public void saveToDatabase() {
        // Code to save the invoice to the database
    }
}
```

Task:
Refactor the above class to adhere to the Single Responsibility Principle.

Part 2: Open/Closed Principle (OCP)

Question 2:
Consider the following class:

```
class Employee {
    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public double calculateBonus() {
        return salary * 0.1;
```

```
    }
}
```

Task:
Refactor the above class so that it adheres to the Open/Closed Principle, allowing the bonus calculation to be extended without modifying the `Employee` class.

Part 3: Liskov Substitution Principle (LSP)

Question 3:
Consider the following class hierarchy:

```
class Rectangle {
    private double width;
    private double height;

    public void setWidth(double width) {
        this.width = width;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(double width) {
        super.setWidth(width);
        super.setHeight(width);
    }
```

```
    @Override
    public void setHeight(double height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

Task:
Identify the violation of the Liskov Substitution Principle in the above code and refactor the classes to adhere to LSP.

Part 4: Interface Segregation Principle (ISP)

Question 4:
Consider the following interface:

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

Task:
Refactor the above interface and its implementations to adhere to the Interface Segregation Principle.

Part 5: Dependency Inversion Principle (DIP)

Question 5:
Consider the following classes:

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

```
class Switch {
    private Lamp lamp;

    public Switch(Lamp lamp) {
        this.lamp = lamp;
    }

    public void operate() {
        lamp.turnOn();
    }
}
```

Task:
Refactor the above classes to adhere to the Dependency Inversion Principle.

Part 6: DRY (Don't Repeat Yourself)

Question 6:
Consider the following classes:

```
class Square {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public double calculateArea() {
        return side * side;
    }
}

class Circle {
    private double radius;
```

```
    public Circle(double radius) {
        this.radius = radius;
    }

    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

Task:
Refactor the above classes to avoid code duplication, adhering to the DRY principle.

Part 7: YAGNI (You Aren't Gonna Need It)

Question 7:
Consider the following class:

```
class Product {
    private String name;
    private double price;

    // Prematurely added methods
    public void convertPriceToDifferentCurrency(String curren
cy) {
        // Code to convert price
    }

    public String toJson() {
        return "{\\"name\\":\\"" + name + "\\", \\"price\\":"
+ price + "}";
    }

    // Constructor, getters, and setters
    public Product(String name, double price) {
        this.name = name;
```

```
        this.price = price;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price;
}
}
```

Task:
Identify the YAGNI violation in the above code and refactor the class accordingly.

Submission:

1. Provide the refactored code for each task in separate files or sections.
2. Include comments explaining how your refactored code adheres to the respective principles

Bug Bash:

Questions

Question 1: User Authentication

Task:
Consider the following `UserAuthenticator` class:

```
class UserAuthenticator {
    public boolean authenticate(String username, String passw
ord) {
        // Authentication logic
        return true;
    }
```

```
    public void logAuthenticationAttempt(String username, boo
lean success) {
        // Logging logic
    }

    public void sendAuthenticationNotification(String usernam
e) {
        // Notification logic
    }
}
```

Requirement:
Identify the violation in the
`UserAuthenticator` class and refactor the code to adhere to the correct principles.

Question 2: Shape Area Calculation

Task:
Consider the following
`Shape` and `AreaCalculator` classes:

```
class Shape {
    public String type;
    public double length;
    public double width;
    public double radius;

    public Shape(String type, double length, double width, do
uble radius) {
        this.type = type;
        this.length = length;
        this.width = width;
        this.radius = radius;
    }
}
```

```
class AreaCalculator {
    public double calculateArea(Shape shape) {
        if ("rectangle".equalsIgnoreCase(shape.type)) {
            return shape.length * shape.width;
        } else if ("circle".equalsIgnoreCase(shape.type)) {
            return Math.PI * shape.radius * shape.radius;
        }
        return 0;
    }
}
```

Requirement:
Identify the violation in the `Shape` and `AreaCalculator` classes and refactor the code to adhere to the correct principles.

Question 3: Employee Payment

Task:
Consider the following `Employee` class:

```
class Employee {
    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public double getSalary() {
        return salary;
    }
```

```
    public void printPaySlip() {
        // Code to print payslip
    }

    public void calculateBonus() {
        // Code to calculate bonus
    }
}
```

Requirement:
Identify the violation in the
`Employee` class and refactor the code to adhere to the correct principles.

Question 4: Animal Sounds

Task:
Consider the following
`Animal` and `SoundMaker` classes:

```
class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

class SoundMaker {
    public void playSound(Animal animal) {
        if (animal instanceof Dog) {
            System.out.println("Playing dog sound");
```

```
        } else {
            animal.makeSound();
        }
    }
}
```

Requirement:
Identify the violation in the
`Animal` and `SoundMaker` classes and refactor the code to adhere to the correct principles.

Question 5: Order Processing

Task:
Consider the following
`OrderProcessor` class:

```
class OrderProcessor {
    public void processOrder(String orderId) {
        // Order processing logic
    }

    public void sendOrderConfirmation(String orderId) {
        // Order confirmation logic
    }

    public void updateOrderStatus(String orderId, String stat
us) {
        // Order status update logic
    }
}
```

Requirement:
Identify the violation in the
`OrderProcessor` class and refactor the code to adhere to the correct principles.

Submission:

1. Provide the refactored code for each task in separate files or sections.
2. Include comments explaining the identified violations and how your refactored code adheres to the correct principles.

Detailed Question:

Question 1: Online Shopping System

Task:
Design a small subsystem for an online shopping application that includes the following features:

1. A `Product` class with attributes `name`, `price`, and `category`.
2. An `Order` class that can contain multiple products and calculate the total price of the order.
3. A `Discount` mechanism that can be applied to the order to adjust the total price.
4. A `NotificationService` that sends notifications to customers about their order status.

Requirements:

- Ensure that your design allows for easy addition of new types of products and discounts without modifying existing code.
- Ensure that each class has a clear and single responsibility.
- Use interfaces or abstract classes where appropriate to promote flexibility and extension.
- Avoid duplicating code across classes.
- Only implement features that are necessary for the current requirements.

Provide the complete implementation of all the necessary classes.

Question 2: Library Management System

Task:
Design a small subsystem for a library management application that includes the following features:

1. A `Book` class with attributes `title`, `author`, `ISBN`, and `genre`.
2. A `Member` class that represents a library member and can borrow multiple books.
3. A `Librarian` class that can add new books to the library's collection and manage member borrowings.
4. A `FineCalculator` that calculates fines for overdue books based on the number of days overdue.

Requirements:

- Ensure that your design allows for easy addition of new types of genres and fine calculation rules without modifying existing code.
- Ensure that each class has a clear and single responsibility.
- Use interfaces or abstract classes where appropriate to promote flexibility and extension.
- Avoid duplicating code across classes.
- Only implement features that are necessary for the current requirements.

Provide the complete implementation of all the necessary classes.

Submission:

1. Provide the full implementation of the classes for each task.
2. Include comments in your code explaining your design choices.