INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA PIAUÍ	**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ** **Curso: ADS** **Disciplina: Engenharia de Software III** **Professor: Ely**

Exercícios 03

Para cada uma das questões abaixo implemente o que se pede e disponibilize uma classe que executa o que foi pedido instanciando, atribuindo valores e gerendo as saídas necessárias.

1. A classe Post abaixo fere o Princípio da Responsabilidade Única (SRP) por ter duas responsabilidades: Ser uma classe de “modelo” além disso ter a responsabilidade de persistir dados.

```
import java.util.Date;
import java.io.FileWriter;
import java.io.IOException;

public class Post {
    private int id;
    private String texto;
    private Autor autor;
    private Date data;
    private int quantidadeDeLikes;
    private String filePath; // Caminho do arquivo - responsabilidade extra

    public Post(int id, String texto, Autor autor, Date data, String filePath) {
        this.id = id;
        this.texto = texto;
        this.autor = autor;
        this.data = new Date(data.getTime());
        this.quantidadeDeLikes = 0;
        this.filePath = filePath;
    }

    public int getId() {
        return id;
    }

    public String getTexto() {
```

```
        return texto;
    }

    public Autor getAutor() {
        return autor;
    }

    public Date getData() {
        return new Date(data.getTime());
    }

    public int getQuantidadeDeLikes() {
        return quantidadeDeLikes;
    }

    // Responsabilidade extra: Manipulação de Arquivo
    public void saveToFile() {
        try (FileWriter writer = new FileWriter(filePath)) {
            writer.write("ID: " + id + "\n");
            writer.write("Texto: " + texto + "\n");
            writer.write("Autor: " + autor.getNome() + "\n");
            writer.write("Data: " + data + "\n");
            writer.write("Quantidade de Likes: " + quantidadeDeLikes + "\n");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public class Autor {
    private int id;
    private String nome;
    private String email;

    public Autor(int id, String nome, String email) {
       this.id = id;
        this.nome = nome;
        this.email = email;
    }

    // Métodos de acesso (getters) aqui.
}
```

Refaça a implementação delegando a uma classe RepositorioDePosts a responsabilidade de persistir os dados. Crie uma classe para testes que instancie autores, posts e faça a escrita e adicionalmente a leitura de um post em arquivo.

2. A classe Calculadora abaixo possui o método calcular com possibilidade de crescer “infinitamente”, ferido ao princípio Open Closed (OCP):

```
import java.util.List;

public class Calculadora {
    private final double a;
    private final double b;

    public Calculadora(double a, double b) {
        this.a = a;
        this.b = b;
    }

    public List<Double> calcular(List<String> operacoes) {

        List<Double> resultados = new ArrayList<>();

        for (String operacao : operacoes) {
            switch (operacao) {
                case "soma":
                    resultados.add(this.a + this.b);
                    break;
                case "subtracao":
                    resultados.add(this.a - this.b);
                    break;
                // case "multiplicacao":
                //     resultados.add(this.a * this.b);
                //     break;
                default:
                    throw new IllegalArgumentException("Operação não suportada: " +
operacao);
            }
        }
        return resultados;
    }
}
```

Dessa forma, refatore a implementação da seguinte forma:

a) Crie uma interface Operacao que possui um método executar() onde dois parâmetros são recebidos;
b) Implemente a interface para cada uma das operações possíveis;
c) Refatore o método calcular para coerente com o OCP recebendo uma lista de operações e adicionando à lista de resultados o resultado de cada método executar da Operacao em questão.

3. Considere a classe abaixo:

```
import java.util.List;

class ImpostoDeRenda {
    private String cpfContribuinte;
    private List<Double> rendimentos;
    private List<Double> despesas;

     public ImpostoDeRenda(String cpfContribuinte,
                           List<Double> rendimentos,
                           List<Double> despesas) {
        this.cpfContribuinte = cpfContribuinte;
        this.rendimentos = rendimentos;
        this.despesas = despesas;
    }

    public double calcularImposto() {
        double rendaTotal =
                        rendimentos.stream().mapToDouble(Double::doubleValue).sum();
        double despesaTotal =
                        despesas.stream().mapToDouble(Double::doubleValue).sum();
        double baseCalculo = rendaTotal - despesaTotal;

        if (baseCalculo <= 1903.98) {
            return 0.0;
        }
        if (baseCalculo <= 2826.65) {
            return baseCalculo * 0.075 - 142.80;
        }
        // E assim por diante, para outros intervalos...

        return baseCalculo * 0.275 - 869.36; // Para bc acima de 4664.68
    }

    public void gerarRelatorio() {
        System.out.println("CPF: " + cpfContribuinte);
        System.out.println("Rendimentos: " + rendimentos);
        System.out.println("Despesas: " + despesas);
        System.out.println("Imposto Devido: " + calcularImposto());
    }
}
```

A classe viola o princípio SRP em dois pontos onde efetua cálculos e na geração de dados de saída. Proponha então uma refatoração da classe para que:

a) Exista uma classe chamada CalculadoraImpostoDeRenda que possua um método chamado calcular. Nesse método, são passados os rendimentos e as despesas e é feito o cálculo para diferentes valores da base de cálculo;
b) Crie outra classe para gerar a saída de tela onde são passadas a classe atual (this) ou os seus atributos e o cálculo do imposto já efetuado;
c) O método da classe ImpostoDeRenda.calcular() seja renomeado para processar() e que na sua definição sejam instanciados uma calculadora e relatório conforme abaixo:

```
public class ImpostoDeRenda {
    private String cpfContribuinte;
    private List<Double> rendimentos;
    private List<Double> despesas;

    public ImpostoDeRenda(String cpfContribuinte, List<Double> rendimentos,
                          List<Double> despesas) {
        this.cpfContribuinte = cpfContribuinte;
        this.rendimentos = rendimentos;
        this.despesas = despesas;
    }

    public void processar() {
        CalculadoraImpostoDeRenda calculadora = new CalculadoraImpostoDeRenda();
        double impostoDevido = calculadora.calcularImposto(rendimentos, despesas);

        RelatorioImpostoRenda relatorio = new RelatorioImpostoRenda();
        relatorio.gerarRelatorio(this, impostoDevido);
    }
}
```

4. Suponha agora a classe ImpostoDeRenda possua as seguintes validações executadas antes do cálculo no método processar:
    - CPF não pode ser vazio;
    - CPF tem que ter 11 caracteres;
    - Os rendimentos e despesas não podem ser negativos;
    - Não podem existir mais que 5 rendimentos;

    a. Implemente individualmente dentro do método processar essas validações;
    b. Refatore a classe ImpostoDeRenda novamente criando uma classe específica chamada ValidacaoImpostoDerenda ter um método com as validações conforme mostrado em sala de aula como forma de não ferir diretamente o SRP.

c. Considerando que o número de validações pode ser “infinito” à medida que a necessidade de novas, refatore a classe acima para receber uma lista de validações e processá-las de acordo com o princípio Open Closed.

5. As classes abaixo violam o SRP conforme comentários:

```
import java.util.List;

enum TipoInvestimento {
    RENDA_FIXA, RENDA_VARIAVEL
}

enum TipoTransacao {
    CREDITO, DEBITO
}

class ContaCorrente {
    private String numero;
    private double saldo;
    private List<Transacao> transacoes;

    public ContaCorrente(String numero, double saldo) {
        this.numero = numero;
        this.saldo = saldo;
    }

    // Método que viola o SRP, pois filtrar transações não é responsabilidade de
ContaCorrente
    public List<Transacao> filtrarTransacoesInvalidas() {
        // ... implemente a lógica para filtrar transações inválidas
        return transacoesInvalidas;
    }

    public String getNumero() {
        return numero;
    }

    public double getSaldo() {
        return saldo;
    }
}

class Investimento {
    private int id;
    private double valor;
    private TipoInvestimento tipo;
    private String statusRisco;

    public Investimento(int id, double valor, TipoInvestimento tipo) {
```

```
        this.id = id;
        this.valor = valor;
        this.tipo = tipo;
    }

    // Método que viola o SRP, pois avaliar risco não é responsabilidade do
investimento
    public String avaliarRisco() {
        // ... implemente a lógica para avaliar o risco do investimento
        return statusRisco;
    }

    public int getId() {
        return id;
    }

    public double getValor() {
        return valor;
    }

    public TipoInvestimento getTipo() {
        return tipo;
    }
}

class Transacao {
    private int id;
    private double valor;
    private TipoTransacao tipo;

    public Transacao(int id, double valor, TipoTransacao tipo) {
        this.id = id;
        this.valor = valor;
        this.tipo = tipo;
    }

    // Método que viola o SRP, pois verificar fraude não é responsabilidade da
transação
    public boolean verificarFraude() {
        // ... implemente a lógica para verificar se a transação é fraudulenta
        return isFraudulenta;
    }

    public int getId() {
        return id;
    }

    public double getValor() {
        return valor;
```

```
    }

    public TipoTransacao getTipo() {
        return tipo;
    }
}
```

Com base nas implementações, proponha a codificação dos métodos que ferem o SRP e segregue tais alterações/responsabilidades em métodos de 3 classes: ContaCorrenteService, InvestimentoService e TransacaoService.

6. Considerando ainda as classes anteriores, a classe abaixo fere o OCP:

```
class AuditoriaFinanceiraService {
     public void executar(List<ContaCorrente> contas,
                          List<Investimento> investimentos,
                          List<Transacao> transacoes) {
        // Auditoria para Conta Corrente
        for(ContaCorrente conta : contas) {
            // Lógica de auditoria para Conta Corrente...
        }

        // Auditoria para Investimento
        for(Investimento investimento : investimentos) {
            // Lógica de auditoria para Investimento...
        }

        // Auditoria para Transacao
        for(Transacao transacao : transacoes) {
            // Lógica de auditoria para Transacao...
        }
    }
}
```

Implemente de forma livre as lógicas de auditorias das 3 classes. Refatore posteriormente a implementação com o que for necessário para que seja passada apenas uma lista de “auditáveis” no método executar, deixando assim a classe de acordo com o OCP.