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 todas as questões abaixo, crie uma classe para testar as refatorações com um método main a ser executado. C que cada classe está em um arquivo específico.

1) (1,5) Considerando as classes Cliente e Pedido abaixo:

```
import java.util.List;
public class Cliente {
    private String nome;
    private List<Pedido> pedidos;

    public void processarPedido(Pedido pedido) {
        pedidos.add(pedido);
        calcularDesconto(pedido);
    }

    public double calcularDesconto(Pedido pedido) {
        return pedido.getValorTotal() * 0.1;
    }
}

public class Pedido {
    private double valorTotal;
    private Cliente cliente;

    public double getValorTotal() {
        return valorTotal;
    }

    public Cliente getCliente() {
        return cliente;
    }
}
```

A classe Cliente contém métodos relacionados ao processamento e armazenamento de pedidos, enquanto a classe Pedido contém informações do cliente. Refatore as classes para que Cliente tenha apenas uma única responsabilidade e fique em conformidade com o princípio SRP.

2) (1,5) Considere uma classe RedeSocial que valida postagens dos usuários:

```
public class RedeSocial {
    public void postarMensagem(String mensagem) {
        if (mensagem == null || mensagem.trim().equals("")) {
            // espera e tenta novamente
            System.out.println("A mensagem não pode ser vazia...");
        } else {
            System.out.println("Mensagem postada: " + mensagem);
        }
    }
}
```

Refatore o código utilizando o princípio *Fail Fast* ao identificar mensagens inválidas, lançando uma exceção a adequada.

3) (2,0) A classe Carro abaixo possui um método que retorna a quantidade de combustível e outro que retorna a capacidade máxima do tanque. Uma outra classe pega essas informações e decide se é necessário e possível reabastecer o carro ou não. Refatore o código aplicando o princípio *Tell don't ask* para que, em vez de perguntar pelas informações do carro e tomar decisões com base nisso, peça à própria classe Carro para realizar as ações de validação necessárias. Ou seja, se for necessário ou possível reabastecer, o próprio carro deveria ser capaz de decidir e executar essa ação.

```
public class Carro {
    double quantidadeCombustivel;
    double capacidadeMaxima = 50; // em litros

    public double getQuantidadeCombustivel() {
        return quantidadeCombustivel;
    }

    public double getCapacidadeMaxima() {
        return capacidadeMaxima;
    }
}

class TestaCarro {
    public static void reabastecerSeNecessario(Carro c, int quantidade) {
        if (c.getQuantidadeCombustivel() < c.getCapacidadeMaxima() * 0.1) {
            if ((c.getQuantidadeCombustivel() + quantidade)<= c.capacidadeMaxima){
                c.quantidadeCombustivel += quantidade;
            } else {
                System.out.println("Capacidade máxima do tanque ultrapassada.");
            }
        }
    }
}
```

4) (1,0) Altere a questão para que as validações considerem o princípio Fail Fast e lance exceções quando as validações falharem.

5) (2,0) A classe Transacao gerencia transações financeiras, incluindo o salvamento de informações em arquivo e o cálculo de taxas. Atualmente, o cálculo de taxas é baseado em quatro diferentes tipos de transação: Depósito, Retirada, Transferência e Pagamento de Serviços. Cada tipo tem sua taxa específica. Esta classe se tornou complexa e difícil de manter, violando o Princípio da Responsabilidade Única (SRP). Refatore a classe Transacao, separando as responsabilidades de salvar as informações da transação, do cálculo de taxas em outras classes.

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

public class Transacao {
    private double valor;
    private Date data;
    private String tipo; // Tipos: DEPOSITO, RETIRADA, TRANSFERENCIA...

    public Transacao(double valor, Date data, String tipo) {
    }

    public void salvarTransacao() {
        try (FileWriter writer = new FileWriter("log_transacoes.txt", true)) {
            writer.write("Data: " + data + ", Valor: " + valor +
                         ", Tipo: " + tipo + "\n");
        } catch (IOException e) {
            System.out.println("Erro ao salvar a transação: " + e.getMessage());
        }
    }

    public double calcularTaxas() {
        switch (tipo.toUpperCase()) {
            case "DEPOSITO":
                return valor * 0.01;
            case "RETIRADA":
                return valor * 0.02;
            case "TRANSFERENCIA":
                return valor * 0.015;
             default:
                return 0;
        }
    }
}
```

6) (2,0) Considere as classes abaixo:

```
public class Produto {
    private String id;
    private String descricao;
    private double valor;
    private double taxa;

     public Produto(String id, String descricao, double valor,
                    double taxa) {
        this.id = id;
        this.descricao = descricao;
        this.valor = valor;
        this.taxa = taxa;
    }
    //métodos de leitura...
}
public class Servico {
    private String id;
    private String descricao;
    private double valor;
    private int horas;

     public Servico(String id, String descricao, double valor,
                    int horas) {
        this.id = id;
        this.descricao = descricao;
        this.valor = valor;
        this.horas = horas;
    }
    //métodos de leitura...
}


public class Doacao implements Totalizavel {
    private String id;
    private String descricao;
    private double valor;
    private double bonus;

     public Doacao(String id, String descricao, double valor,
                   double bonus) {
        this.id = id;
        this.descricao = descricao;
        this.valor = valor;
        this.bonus = bonus;
    }
    //métodos de leitura...

}
```

```
public class Totalizacao {

    public double totalizarServicos(Servico[] servicos) {
        double total = 0;
        for (Servico servico : servicos) {
            total += servico.getValor() * servico.getHoras();
        }
        return total;
    }

    public double totalizarDoacoes(Doacao[] doacoes) {
        double total = 0;
        for (Doacao doacao : doacoes) {
            total += doacao.getValor() + doacao.getBonus();
        }
        return total;
    }

    public double totalizarProdutos(Produto[] produtos) {
        double total = 0;
        for (Produto produto : produtos) {
            total += produto.getValor() * (1 + produto.getTaxa());
        }
        return total;
    }
}
```

a) Refatore as classes criando uma classe base chamada Item com os atributos id, descricao e valor, bem como os métodos de leitura e escrita. Refatore os construtores As classes Produto, Servico e Doacao devem ser refatoradas para herdarem de item e ter seus métodos e construtores ajustados para evitar repetições de código;

b) Como a classe Totalizacao fere o princípio OCP, proponha uma interface comum às classes Produto, Serviço e doação de forma que a classe Totalizacao fique com o seguinte funcionamento:

```
public class Totalizacoes {
    public double totalizar(Totalizavel[] itens) {
        double total = 0;
        for (Totalizavel item : itens) {
            total += item.calcularTotal();
        }
        return total;
    }
}
```