TP3 - Grupo 26 (link repositório)

Bruno Campos Ribeiro - 211039288
Gabriel de Souza Fonseca Ribeiro - 190087439
Igor e Silva Penha - 211029352
Lucas Gobbi Bergholz - 211029441

Questão 1: Para cada um dos princípios de bom projeto de código mencionados acima, apresente sua definição e relacione-o com os maus-cheiros de código apresentados por Fowler em sua obra.

R ->

- **Simplicidade:** Manter o código o mais simples possível, sem complexidades desnecessárias, para facilitar a compreensão e a manutenção.
    - **Long Method:** Se relaciona com o mau cheiro de método longo, ou seja, métodos longos demais evitam a simplicidade;
    - **Large Class:** Classes longas demais pecam na questão de simplicidade, já que seu tamanho reflete justamente em sua complexidade estar mais alta que o necessário;
    - **Lazy Class:** A "classe preguiçosa" se caracteriza por uma classe que faz pouca coisa, ou foi adicionada para resolver um possível problema que acabou por não acontecer. Dessa forma, a retirada dessas classes melhora a simplicidade do código.

- **Elegância:** Escrever código que seja claro e estéticamente agradável, com soluções diretas e eficazes.
    - **Long Method:** Se relaciona com o mau cheiro de método longo, ou seja, métodos longos demais evitam a Elegância;
    - **Large Class:** Classes longas demais pecam na questão de Elegância, já que seu tamanho reflete justamente em sua complexidade estar mais alta que o necessário;
    - **Lazy Class:** A retirada das classes preguiçosas melhora consideravelmente a claridade do código, por diminuir a quantidade de classes envolvidas no programa;
    - **Shotgun Surgery:** O Shotgun Surgery se caracteriza por, quando for feita alguma mudança, ser necessário mudar diversos pequenos aspectos de outros lugares do código, ou seja, o código está muito entrelaçado em si mesmo. Diminuir essas ligações desnecessárias do código aumenta sua elegância e compreensão.

- **Modularidade:** Dividir o sistema em módulos independentes, cada um com uma responsabilidade clara.
    - **Large Class:** Classes longas demais são classes que necessitam de maior modularidade, ou seja, dividir esta classe em mais de uma;
    - **Long Method:** Métodos longos demais são métodos que necessitam se dividir em mais de um, aumentando sua modularidade;

    - **Duplicated Code:** O Duplicated Code trata exatamente dessa característica, que é o mau cheiro por um trecho de código repetitivo no programa, o qual pode ser modularizado de alguma forma;
    - **Middle Man:** O Middle Man acontece quando duas classes que comumente se comunicam entre si, acabam usando uma outra classe para intermediar isso. Isso pode ser um sinal de uma modularização feita de maneira equivocada.

- **Boas interfaces:** Definir interfaces claras e simples, que ofereçam métodos bem definidos para interação com outros componentes.
    - **Parallel Inheritance Hierarchies:** Ocorre quando, ao fazer uma subclasse de uma classe, ser necessário fazer uma subclasse de outra classe. Isso ocorre pelas classes pais estarem correlacionadas de maneiras perigosas para o código. Boas interfaces previnem isso.
    - **Middle Man:** Interfaces feitas corretamente evitam o uso de classes intermediárias sendo criadas de maneira desnecessária.

- **Extensibilidade:** Projetar o código de forma que novas funcionalidades possam ser adicionadas com facilidade, sem necessidade de grandes alterações na base existente.
    - **Speculative Generality:** Ocorre quando classes e métodos são criados em prol de resolver problemas potenciais, não ainda presentes no programa, dificultando a manutenção do código por algo que nem aconteceu ainda. Esse mal cheiro prejudica a extensibilidade do código.

- **Evitar duplicação:** Minimizar a duplicação de código para reduzir o risco de inconsistências e facilitar a manutenção.
    - **Duplicated Code:** O Duplicated Code trata exatamente dessa característica, que é o mal cheiro por um trecho de código repetitivo no programa, o qual pode ser modularizado de alguma forma.

- **Portabilidade:** Escrever código que possa ser executado em diferentes ambientes ou plataformas com o mínimo de ajustes.
    - **Speculative Generality:** Ocorre quando classes e métodos são criados em prol de resolver problemas potenciais, não ainda presentes no programa, dificultando a manutenção do código por algo que nem aconteceu ainda. Esse mal cheiro prejudica a portabilidade do código, pois o problema de uma plataforma pode não ser o mesmo da outra.

- **Código deve ser idiomático e bem documentado:** Usar as melhores práticas e convenções do idioma de programação, além de fornecer documentação clara para facilitar a compreensão e uso do código.
    - **Comments:** O mau cheiro por falta de comentários pode ser resolvido por boa documentação, visto que comentar o código é uma maneira de documentação.

Questão 2: Identifique quais são os maus-cheiros que persistem no trabalho prático 2 do grupo, indicando quais os princípios de bom projeto ainda estão sendo violados e indique quais as operações de refatoração são aplicáveis. Atenção: não é necessário aplicar as operações de refatoração, apenas indicar os princípios violados e operações possíveis de serem aplicadas.

R -> As classes citadas nas respostas abaixo podem ser encontradas na pasta Model do projeto (https://github.com/BrunoRiibeiro/TPPE/blob/master/src/Model)

- **Middle Man:** As Classes *VendaCalculator*, *Venda* e *ValorVenda* possuem o clássico problema de Middle Man, que é uma classe intermediando duas outras que se comunicam muito entre si. Após a refatoração do TP2 pelo grupo, a classe Venda ficou responsável apenas por chamar *ValorVenda*, a qual manipula os dados de Venda pelos métodos de *VendaCalculator*, ou seja, *ValorVenda* age como Middle Man.
    - **Princípios violados:** Modularidade e Boas Interfaces;
    - **Correção:** Uma correção para esse mau cheiro seria remover a classe *ValorVenda*, passando sua função *calcularValor* para a classe Venda ou *VendaCalculator*.

Classe ValorVenda

```
public class ValorVenda {
   private Venda venda;
   private VendaCalculator vendaCalculator;
   private float valor;
   public ValorVenda(Venda venda) {
       this.venda = venda;
       this.vendaCalculator = new VendaCalculator(venda);
   }
   public float calcularValor() {
       valor = vendaCalculator.calcularValorItens();
       float desconto = vendaCalculator.calcularDesconto();
       float frete = vendaCalculator.calcularFrete();
       float icms = vendaCalculator.calcularIcms();
       float impostoMunicipal = vendaCalculator.calcularImpostoMunicipal();
       valor -= desconto;
       valor += frete;
```

```
        valor += icms + impostoMunicipal;
        return valor;
    }
}
```

Classe Venda

```
public class Venda {
    private LocalDate data;
    private Cliente cliente;
    private Produto[] itens;
    private String metodoPagamento;
    private float icms;
    private float impostoMunicipal;
    private float frete;
    private float desconto;
    public Venda(LocalDate data, Cliente cliente, Produto[] itens, String
metodoPagamento) {
        this.data = data;
        this.cliente = cliente;
        this.itens = itens;
        this.metodoPagamento = metodoPagamento;
    }
    public LocalDate getData() { return data; }
    public Cliente getCliente() { return cliente; }
    public Produto[] getItens() { return itens; }
    public String getMetodoPagamento() { return metodoPagamento; }
    public float getICMS() { return icms; }
    public float getImpostoMunicipal() { return impostoMunicipal; }
    public float getFrete() { return frete; }
    public float getDesconto() { return desconto; }
    public float getValor() {
        return new ValorVenda(this).calcularValor();
    }
}
```

Classe VendaCalculator

```
public class VendaCalculator {
    private Venda venda;
    public VendaCalculator(Venda venda) {
        this.venda = venda;
    }
    public float calcularValorItens() {
        float valorItens = 0.0f;
        for (Produto item : venda.getItens())
```

```
            valorItens += item.getValor();
        return valorItens;
    }
    public float calcularDesconto() {
        float valorItens = calcularValorItens();
        float desconto = 0.0f;
        if (venda.getCliente().getTipo() == TipoCliente.ESPECIAL) {
            desconto = valorItens * 0.10f;
            if (venda.getMetodoPagamento().startsWith("429613"))
                desconto += valorItens * 0.10f;
        }
        return desconto;
    }
    public float calcularImpostoMunicipal() {
        float valorItens = calcularValorItens();
        if (venda.getCliente().getEndereco().getEstado() == Estado.DF)
            return 0.0f;
        return valorItens * 0.12f;
    }
    public float calcularIcms() {
        float valorItens = calcularValorItens();
        if (venda.getCliente().getEndereco().getEstado() == Estado.DF)
            return valorItens * 0.18f;
        else
            return valorItens * 0.12f;
    }
    public float calcularFrete() {
        Estado estado = venda.getCliente().getEndereco().getEstado();
        boolean isCapital = venda.getCliente().getEndereco().getCapital();
        Regiao regiao = estado.getRegiao();
        float frete = 0.0f;
        switch (regiao) {
            case CENTRO_OESTE:
                frete = isCapital ? 10.0f : 13.0f;
                break;
            case NORDESTE:
                frete = isCapital ? 15.0f : 18.0f;
                break;
            case NORTE:
                frete = isCapital ? 20.0f : 25.0f;
                break;
            case SUDESTE:
                frete = isCapital ? 7.0f : 10.0f;
                break;
            case SUL:
                frete = isCapital ? 10.0f : 13.0f;
                break;
            default:
                frete = isCapital ? 5.0f : 0.0f;
                break;
        }
```

```
        if (venda.getCliente().getTipo() == TipoCliente.ESPECIAL)
            frete *= 0.70f;
        else if (venda.getCliente().getTipo() == TipoCliente.PRIME)
            frete = 0.0f;
        return frete;
    }
    public float calcularCashback() {
        float valorItens = calcularValorItens();
        if (venda.getCliente().getTipo() == TipoCliente.PRIME) {
            float cashbackRate = venda.getMetodoPagamento().startsWith("429613")
? 0.05f : 0.03f;
            float cashback = valorItens * cashbackRate;
            venda.getCliente().setCashback(venda.getCliente().getCashback() +
cashback);
            return cashback;
        }
        return 0.0f;
    }
}
```

- **Comments:** O código inteiro não tem comentários, o que faz ele ser muito difícil de compreender para alguém que não desenvolveu ele.
    - **Princípios violados:** Código deve ser idiomático e bem documentado;
    - **Correção:** A correção desse mau cheiro seria algum dos 4 desenvolvedores do projeto comentar todo o código desenvolvido, adicionando nas classes o que cada método faz e a função da classe no projeto.

- **Data Class:** Um dos maiores mau cheiros do código é o de Data Class, ou seja, classes com a única responsabilidade de guardar dados. Data Class são caracterizadas por ter apenas atributos e *getters* e *setters*. No código, as classes *Cliente*, *Produto*, *Endereco* possuem esse mau cheiro.
    - **Princípios violados:** Simplicidade e Elegância;
    - **Correção:** Para a classe *Endereco*, uma solução seria transformá-la em atributo de *Cliente*. Já a de produto, poderia ser transformada em atributos de *Venda*, porém, *Venda* passaria a ter o problema de Long Class.

Classe Endereco

```
public class Endereco {
      public Endereco(Estado estado, Boolean isCapital) {
            this.estado = estado;
            this.isCapital = isCapital;
      }

      private Estado estado;
      private Boolean isCapital;
```

```
        public Estado getEstado() {
                return estado;
        }
        public Boolean getCapital() {
                return isCapital;
        }
}
```

Classe Produto

```
public class Produto {
   //metodo construtor com codigo, descricao, valor, unidade(peça, unidade,
metro, cm3, etc)
   public Produto(int codigo, String descricao, Float valor, String unidade) {
       this.codigo = codigo;
       this.descricao = descricao;
       this.valor = valor;
       this.unidade = unidade;
   }
   //atributos
   private int codigo;
   private String descricao;
   private Float valor;
   private String unidade;
   //metodos
   public int getCodigo() {
       return codigo;
   }
   public String getDescricao() {
       return descricao;
   }
   public Float getValor() {
       return valor;
   }
   public String getUnidade() {
       return unidade;
   }

}
```

Classe Cliente

```
public class Cliente {
```

```
    public Cliente(int id, TipoCliente tipo, Endereco endereco, Float cashback) {
        this.id = id;
        this.tipo = tipo;
        this.endereco = endereco;
        this.cashback = cashback;
    }
    //atributos
    private int id;
    private TipoCliente tipo;
    private Endereco endereco;
    private Float cashback;
    //metodos
    public int getId() {
        return id;
    }
    public TipoCliente getTipo() {
        return tipo;
    }
    public Endereco getEndereco() {
        return endereco;
    }
    public Float getCashback() {
        return cashback;
    }
        public void setCashback(float f) {
              // TODO Auto-generated method stub

        }

    public static Cliente criarCliente(int id, TipoCliente tipo, Endereco
endereco, Float cashback) {
        return new Cliente(id, tipo, endereco, cashback);
    }

}
```

- **Lazy Class:** A Lazy Class pode ser encontrada após o processo de refatoração de um classe, pois, antes de refatorá-la, a classe possuía grande propósito, mas após a refatoração ela se torna desnecessária. Nesse projeto, a classe *Venda* passa por isso. No TP1, ela era o coração do projeto, onde a maioria das ações aconteciam. Após sua refatoração em três classes (*VendaCalculator*, *Venda* e *ValorVenda*), ela se tornou quase desnecessária.
    - **Princípios violados:** Simplicidade e Elegância;
    - **Correção:** Uma correção para isso seria transformar essas três classes em apenas duas, pois uma das três classes sempre será uma classe preguiçosa.