Relatório de Análise de Código

Boas Práticas de Código vs Maus-Cheiros

A seguir uma relação entre as boas práticas de código, essenciais para elaboração de bons projetos, e os maus-cheiros que podem ser resolvidos e/ou evitados a partir da aplicação delas.

1. Simplicidade

Um projeto apresenta simplicidade quando o seu código é fácil de entender e manter, sem complexidades desnecessárias, com métodos curtos e objetivos, o que pode evitar os seguintes maus-cheiros:

- *Complexidade desnecessária* (*Needless Complexity*)
- *Método longo* (*Long Method*)
- *Inveja de recursos* (*Feature Envy*)

2. Elegância

A elegância é visível em projetos com código claro e direto, características que ajudam a evitar redundâncias e estruturas complicadas, representadas pelos maus-cheiros:

- *Código duplicado* (*Duplicated Code*)
- *Switches longos* (*Long Method* ou *Switch Statements*)

3. Modularidade

A modularidade consiste em dividir o código em módulos independentes com responsabilidades específicas. Essa organização do código em partes gerenciáveis ajuda a evitar os seguintes maus-cheiros:

- *Classes grandes* (*Large Class*)
- *Cargas de dados* (*Data Clumps*)
- *Divergência de código* (*Divergent Change*)

4. Boas Interfaces

Boas interfaces ajudam a padronizar e modularizar o código. Elas reduzem o acoplamento e facilitam a manutenção, resolvendo os seguintes maus-cheiros:

- *Interface inchada* (*Large Interface*)
- *Dependência excessiva* (*Inappropriate Intimacy*)

5. Extensibilidade

Um código é extensível quando pode ser facilmente adaptado a novas funcionalidades sem grandes modificações. Essa adaptação é possível quando o código não possui os seguintes maus-cheiros:

- *Métodos gananciosos* (*Greedy Method*)
- *Código rígido* (*Rigid Code* ou *Shotgun Surgery*)

6. Evitar Duplicação

A duplicação de código ocorre quando o mesmo trecho se repete ao longo do projeto, e isso pode ser evitado aplicando o princípio DRY (Don’t Repeat Yourself)[2]. A duplicação de código geralmente está associada aos seguintes maus-cheiros:.

- *Código duplicado* (*Duplicated Code*)
- *Métodos e classes redundantes* (*Redundant Methods/Classes*)
- *Código rígido* (*Rigid Code* ou *Shotgun Surgery*)

7. Portabilidade

Um projeto portável pode ser executado em diferentes ambientes e plataformas sem grandes modificações. Essa prática evita a ocorrência do maus-cheiros:

- *Código dependente de plataforma* (*Platform-Dependent Code*)

8. Código Idiomático e Bem Documentado

É um código legível e compreensível, que, por exemplo, possui nomes claros para as funções e classes, e bem documentado, possuindo instruções de como executar e quais padrões são utilizados na codificação. Essas características ajudam a evitar:

- *Comentários obsoletos* (*Obsolete Comments*)
- *Código incompreensível* (*Incomprehensible Code*)

Análise do Trabalho Prático 2

Após o estudo sobre boas práticas de código, e as suas relações com os maus-cheiros em código, foi realizada a análise do código entregue no TP2, após a refatoração. Em uma análise inicial foram identificados alguns problemas, elencados a seguir, com suas possíveis soluções.

```
// br.unb.Main
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    boolean sair = false;

    while (!sair) {
        System.out.println("Menu:");
        System.out.printf("%s. Cadastrar Cliente",
        COD_CADASTRO_CLIENTE);
        System.out.printf("%s. Cadastrar Produto",
        COD_CADASTRO_PRODUTO);
        System.out.printf("%s. Cadastrar Venda",
        COD_CADASTRO_VENDA);

        System.out.printf("%s. Sair", COD_SAIR);
        System.out.print("Escolha uma opção: ");

        String opcao = scanner.nextLine();

        switch (opcao) {
            case COD_CADASTRO_CLIENTE:
                cadastrarCliente(scanner);
                break;
            case COD_CADASTRO_PRODUTO:
                cadastrarProduto(scanner);
                break;
            case COD_CADASTRO_VENDA:
                cadastrarVenda(scanner);
                break;
            case COD_SAIR:
                sair = true;
                break;
            default:
                System.out.println("Opção inválida. Tente
        novamente.");
        }
    }

    System.out.println("Saindo do programa...");
}
```

- Mau-cheiro: Método longo

- Operação de refatoração: Extrair Método
  - Como: a parte de exibir o menu e coletar a resposta do usuário poderia ser extraída para outro método.

```java
// br.unb.Main

private static void cadastrarCliente(Scanner scanner) {
    System.out.print("Nome: ");
    String nome = scanner.nextLine();
    System.out.print("Região: ");
    String regiao = scanner.nextLine();
    System.out.print("Estado: ");
    String estado = scanner.nextLine();
    System.out.print("Categoria: ");
    String categoria = scanner.nextLine();
    System.out.print("Email: ");
    String email = scanner.nextLine();

    Cliente cliente = Cadastro.criaCliente(nome, regiao, estado,
        categoria, email);
    System.out.println("Cliente cadastrado: " + cliente);
}
private static void cadastrarProduto(Scanner scanner) {
    System.out.print("Descrição: ");
    String descricao = scanner.nextLine();
    System.out.print("Valor de Venda: ");
    String valorDeVenda = scanner.nextLine();
    System.out.print("Unidade: ");
    String unidade = scanner.nextLine();
    System.out.print("Código: ");
    String codigo = scanner.nextLine();

    Produto produto = criaProduto(descricao, valorDeVenda, unidade,
        codigo);
    System.out.println("Produto cadastrado: " + produto);
}
```

- Mau-Cheiro: Código duplicado
- Operação de refatoração: Extrair Método
  - Como: Um método para exibir uma mensagem para o usuário e fazer a leitura da entrada inserida pode ser extraído.

```java
// br.unb.model.categorias.Endereco
public static RegiaoDoPais getRegiaoDoPais(String uf) {
```

```
    HashMap<RegiaoDoPais, List<String>> regioes = new HashMap<>();
    regioes.put(RegiaoDoPais.DF, List.of("DF"));
    regioes.put(RegiaoDoPais.CENTRO_OESTE, List.of("GO", "MT",
        "MS"));
    regioes.put(RegiaoDoPais.NORDESTE, List.of("AL", "BA", "CE",
        "MA", "PB", "PE", "PI", "RN", "SE"));
    regioes.put(RegiaoDoPais.NORTE, List.of("AC", "AP", "AM", "PA",
        "RO", "RR", "TO"));
    regioes.put(RegiaoDoPais.SUDESTE, List.of("ES", "MG", "RJ",
        "SP"));
    regioes.put(RegiaoDoPais.SUL, List.of("PR", "RS", "SC"));

    for (RegiaoDoPais regiao : regioes.keySet()) {
        if (regioes.get(regiao).contains(uf))
            return regiao;
    }
    return null;
}

public static boolean isUfValida(String uf) {
    if (uf.trim().length() != 2) {
        throw new IllegalArgumentException("Insira a sigla do
        estado.");
    }
    List<String> ufs = List.of("AC", "AL", "AP", "AM", "BA", "CE",
        "DF", "ES", "GO", "MA", "MT", "MS", "MG", "PA", "PB", "PR",
        "PE", "PI", "RJ", "RN", "RS", "RO", "RR", "SC", "SP", "SE",
        "TO");
    return ufs.contains(uf.toUpperCase().trim());
}
```

- Mau-cheiro: Obsessão por tipos primitivos
- Operação de refatoração: Substituir tipos primitivos com objetos
  - Como: os estados poderiam ser representados por objetos do tipo `enum`.

```
// br.unb.util.OperacoesFinanceiras.java
switch (categoriaDeCliente) {
    case PRIME:
        return 0;
    case ESPECIAL:
        return 0.7F * calculaFrete(estado, regiao);
    case PADRAO:
        return calculaFrete(estado, regiao);
    default:
        return -1F;
}
```

- Mau-Cheiro: Switches longos

- Operação de refatoração: Substituir condicional por polimorfismo
  - Como: Seria possível criar uma classe-pai de cliente, e suas filhas: ClientePrime, ClienteEspecial e ClientePadrao, cada uma com seus cálculos particulares.

Referências

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
[2] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
[3] Andrew Hunt, David Thomas. *The Pragmatic Programmer: Your Journey to Mastery*. Addison-Wesley Professional, 1999.
[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.