Documentação de Refatoração: Geração de Relatórios de Funcionários

Este documento detalha o processo de identificação de *code smells* e a aplicação de técnicas de refatoração no código-fonte do Grupo 06, visando melhorar sua legibilidade, manutenibilidade e extensibilidade.

ETAPA 1: Identificação de Smells

Analisando o código original, foram identificados os seguintes *code smells*:

Código Original (Grupo 06):

```
class Grupo6_ReportBuilder {
     public static String montarRelatorio(String nome, String departamento,
String cargo, int ano, int mes, int dia) {
         String cabecalho = "Relatório de " + nome + " - " + departamento + "
(" + cargo
                 + ")\n";
        String data = "Data: " + dia + "/" + mes + "/" + ano + "\n";
        String corpo = "Atividades:\n";
        corpo += "- Reuniões com equipe\n";
        corpo += "- Entregas do sprint\n";
        corpo += "- Pendências\n";
        String resumo = "Resumo: OK\n";
        String relatorio = cabecalho + data + corpo + resumo;
                                    if     (departamento     !=     null     &&
departamento.toUpperCase().contains("TI")) {
            String extra = "Recursos de TI utilizados: Git, CI/CD\n";
            relatorio = relatorio + extra;
        }
        return relatorio;
    }

    public static void main(String[] args) {
          System.out.println(montarRelatorio("Carla", "TI", "Analista", 2025,
9, 18));
    }
}
```

1. Long Method (Método Longo)

Onde: O método montarRelatorio concentra todas as responsabilidades da criação do relatório.

Problema: Ele é responsável por montar o cabeçalho, formatar a data, definir um corpo de atividades fixo e adicionar um rodapé condicional. Isso viola o Princípio da Responsabilidade Única (SRP), tornando o método de baixa coesão.

Dificuldades Geradas:

- **Manutenção:** Alterar uma parte do relatório (ex: formato da data) exige navegar por um método que faz muitas outras coisas.
- **Reuso:** É impossível reutilizar apenas a lógica de criação do cabeçalho ou do rodapé em outro contexto.
- **Legibilidade:** É preciso ler o método inteiro para entender como o relatório final é construído, em vez de ver uma sequência de passos claros.

2. Primitive Obsession (Obsessão por Tipos Primitivos) e Data Clumps (Aglomeração de Dados)

Onde: Na assinatura do método montarRelatorio(String nome, String departamento, String cargo, int ano, int mes, int dia).

Problema: Conceitos complexos do domínio são representados por tipos primitivos.

- Um Funcionário é representado por três Strings (nome, departamento, cargo).
- Uma Data é representada por três ints (ano, mes, dia).
- Esses grupos de dados (Data Clumps) aparecem sempre juntos e não possuem comportamento ou validação próprios.

Dificuldades Geradas:

- **Falta de Segurança de Tipo:** O uso de String para departamento permite erros de digitação ("TI", "ti", "tecnologia"), quebrando a lógica condicional. Não há garantia de que os valores de dia e mes sejam válidos.
- **Baixa Expressividade:** A assinatura do método não deixa claro quais parâmetros pertencem a qual entidade.
- **Código Duplicado:** Qualquer outro método que precise lidar com um funcionário ou uma data teria que receber novamente essa longa lista de primitivos.

3. Inflexibility (Inflexibilidade / Dados Embutidos)

Onde: O corpo do relatório (Atividades) é fixo dentro do método montarRelatorio.

Problema: Os dados das atividades estão "chumbados" (hardcoded) no código.

Dificuldades Geradas:

- **Inutilidade Prática:** O método só consegue gerar um único relatório, com as mesmas três atividades, para qualquer funcionário. Para gerar um relatório com atividades diferentes, seria necessário alterar o código-fonte do método.
- **Falta de Reusabilidade:** A classe não serve como um construtor de relatórios genérico, mas apenas para um caso de uso específico e estático.

ETAPA 2: Refatoração e Alternativas

Para solucionar os smells identificados, foram aplicadas as seguintes refatorações, resultando no código final que você já desenvolveu.

Refatorações Aplicadas:

1. **Extract Class (Extrair Classe):** A lógica de construção do relatório foi movida da classe principal para uma nova classe dedicada, RelatorioBuilder, seguindo o SRP.
2. **Introduce Parameter Object (Introduzir Objeto como Parâmetro):**
    - Os parâmetros nome, departamento e cargo foram agrupados na classe Funcionario.
    - Os parâmetros ano, mes e dia foram substituídos pelo objeto java.time.LocalDate.
3. **Replace Type Code with Enum (Substituir Código de Tipo por Enum):** A String departamento foi substituída pelo enum Departamento, garantindo a validação dos valores e adicionando comportamento (como getNomeExibicao()).
4. **Extract Method (Extrair Método):** O corpo do método montarRelatorio foi dividido em métodos privados e coesos: construirCabecalho, construirData, construirCorpo e construirRodape.
5. **Generalize Method Signature (Generalizar Assinatura):** A lista de atividades, que era fixa, passou a ser recebida como um parâmetro (List<String> atividades), tornando o builder flexível.

Alternativa de Refatoração Não Utilizada

Smell Abordado: Long Method e Primitive Obsession.

Alternativa: Implementação completa do **Design Pattern "Builder"**.

Como Seria: Em vez de um único método montarRelatorio que recebe todos os dados, a classe RelatorioBuilder teria métodos encadeados para construir o relatório passo a passo.

```
RelatorioBuilder builder = new RelatorioBuilder();
String relatorio = builder.paraFuncionario(carla)
                          .naData(dataRelatorio)
                                .comAtividades(atividadesCarla)
                          .build();
```

Por que não foi escolhida?
A estrutura do relatório é fixa e linear (cabeçalho, data, corpo, rodapé). Não há partes opcionais ou que possam ser montadas em ordens diferentes. A abordagem de "orquestração" adotada (um método público que chama vários privados em sequência) é mais simples, direta e atende perfeitamente à necessidade. O padrão Builder completo seria um exagero (over-engineering), adicionando uma complexidade desnecessária para um problema que não exige a flexibilidade de construção que o padrão oferece.

ETAPA 3: Diário de Refatoração

A seguir, o registro detalhado de cada mudança realizada.

Mudança 1: Extrair a lógica de negócio e agrupar dados

Smells Identificados: Primitive Obsession, Data Clumps.

Refatorações Aplicadas: Introduce Parameter Object (Funcionario, LocalDate), Replace Type Code with Enum (Departamento).

Código antes:

```
     public static String montarRelatorio(String nome, String departamento,
String cargo, int ano, int mes, int dia)
```

Código depois:

```
class Funcionario { /* ... */ }
enum Departamento { /* ... */ }
```

e

```
      public String montarRelatorio(Funcionario funcionario, LocalDate data,
List<String> atividades)
```

Impacto:

Legibilidade: A nova assinatura é muito mais expressiva e clara.

Segurança: O uso de enum e LocalDate previne dados inválidos (ex: um mês "13" ou um departamento "RHs").

Coesão: Os dados relacionados a um funcionário agora estão encapsulados em sua própria classe, podendo futuramente conter regras de negócio específicas.

Reuso: As classes Funcionario e Departamento podem ser reutilizadas em outras partes do sistema.

Mudança 2: Decompor o método de montagem do relatório

Smell Identificado: Long Method.

Refatoração Aplicada: Extract Method.

Antes (Corpo do método monolítico):

Código antes:

```
String cabecalho = "Relatório de " + nome + " - " + departamento + " (" +
cargo + ")\n";
String data = "Data: " + dia + "/" + mes + "/" + ano + "\n";
String corpo = "Atividades:\n";
// ...
if (departamento != null && departamento.toUpperCase().contains("TI")) {
    // ...
}
return relatorio;
```

Código depois:

```
public String montarRelatorio(Funcionario funcionario, LocalDate data,
List<String> atividades) {
    StringBuilder relatorio = new StringBuilder();

    construirCabecalho(relatorio, funcionario);
```

```
    construirData(relatorio, data);
    construirCorpo(relatorio, atividades);
    construirRodape(relatorio, funcionario);

    return relatorio.toString();
}

private void construirCabecalho(StringBuilder sb, Funcionario f) { /* ... */
}
private void construirData(StringBuilder sb, LocalDate data) { /* ... */ }
private void construirCorpo(StringBuilder sb, List<String> atividades) { /*
... */ }
private void construirRodape(StringBuilder sb, Funcionario f) { /* ... */ }
```

Impacto:

Legibilidade: O método público agora descreve o "quê" (os passos para montar um relatório), enquanto os métodos privados detalham o "como" cada passo é executado.

Manutenção: É muito mais fácil e seguro modificar uma parte específica do relatório (ex: o rodapé) sem arriscar quebrar outra.

Clareza: O fluxo de construção do relatório tornou-se explícito e fácil de seguir. O uso de StringBuilder também é mais eficiente para a manipulação de strings.

Mudança 3: Generalizar o conteúdo do relatório

Smell Identificado: Inflexibility (Dados Embutidos).

Refatoração Aplicada: Generalizar a assinatura do método para aceitar dados dinâmicos.

Antes (Atividades fixas no código):

Código antes:

```
String corpo = "Atividades:\n";


corpo += "- Reuniões com equipe\n";
```

```
corpo += "- Entregas do sprint\n";


corpo += "- Pendências\n";
```

Código depois:

```
private void construirCorpo(StringBuilder sb, List<String> atividades) {

    sb.append("Atividades:\n");

    if (atividades == null || atividades.isEmpty()) {

        sb.append("- Nenhuma atividade registrada.\n");

    } else {

        for (String atividade : atividades) {

            sb.append("- ").append(atividade).append("\n");

        }

    }

}
```

Impacto:

Flexibilidade e Reuso: A classe RelatorioBuilder agora é uma ferramenta genérica e reutilizável, capaz de gerar relatórios com qualquer lista de atividades para qualquer funcionário.

Manutenção: Os dados (atividades) foram desacoplados da lógica de apresentação, permitindo que os dados mudem sem que o código do builder precise ser alterado.

Testabilidade: É possível testar o builder com diferentes cenários (lista vazia, lista com muitos itens, etc.) de forma simples e direta.