

Universidade de Brasília - UnB

Técnicas de Programação em Plataformas Emergentes

Trabalho Prático 3 - Projeto de software

Luciano Ricardo da Silva Junior - 221007653

João Pedro Rodrigues Morbeck - 202063300

Pedro Henrique da Silva Melo - 211039662

1. Introdução

O desenvolvimento de software requer um planejamento bem estruturado para garantir a qualidade do código e a manutenção do sistema ao longo do tempo. Um bom projeto de software deve seguir princípios fundamentais que promovam clareza, modularidade e eficiência. No entanto, problemas recorrentes, conhecidos como maus-cheiros de código, podem comprometer esses princípios e dificultar a evolução do sistema.

Este trabalho tem como objetivo analisar os princípios de um bom projeto de código e relacioná-los com os maus-cheiros descritos por Martin Fowler. Além disso, serão identificadas as deficiências presentes no Trabalho Prático, destacando quais princípios estão sendo violados e sugerindo operações de refatoração adequadas para melhorar a qualidade do código.

2. Princípios de um bom projeto de código

Os princípios de bom projeto de código são:

- simplicidade;
- elegância;
- modularidade;
- boas interfaces;
- extensibilidade;
- evitar duplicação;
- portabilidade;
- código deve ser idiomático e bem documentado.

2.1. Simplicidade

Um código simples evita complexidade desnecessária, tornando a compreensão e manutenção mais fáceis. Isso significa que o código deve ser direto, intuitivo e conter apenas o necessário para cumprir sua função. Os maus-cheiros relacionados são:

- **Long Method (Método Longo):** Métodos longos tornam o código difícil de entender e manter.
- **Large Class (Classe Grande):** Classes grandes acumulam muitas responsabilidades, dificultando a reutilização e o entendimento.
- **Duplicated Code (Código Duplicado):** Código repetido aumenta a complexidade e a chance de inconsistências.
- **Primitive Obsession (Obsessão por Primitivos):** O uso excessivo de tipos primitivos em vez de objetos pode levar a código menos expressivo e mais complexo.
- **Speculative Generality (Generalização Especulativa):** Adicionar funcionalidades desnecessárias aumenta a complexidade sem benefício.

2.2. Elegância

Um código elegante é simples, claro e expressa a solução de forma natural. Ele evita excessos e torna a lógica fácil de entender. Os maus-cheiros relacionados são:

- **Long Method (Método Longo):** Métodos longos tornam o código difícil de entender.
- **Large Class (Classe Grande):** Classes muito grandes acumulam responsabilidades desnecessárias.
- **Message Chains (Cadeias de Mensagens):** Cadeias longas de chamadas dificultam a leitura e manutenção.
- **Feature Envy (Inveja de Funcionalidade):** Métodos que acessam muitos dados de outras classes indicam falta de coesão e elegância.
- **Switch Statements (Declarações Switch):** Uso excessivo de switch pode indicar falta de polimorfismo e tornar o código menos elegante.

2.3. Modularidade

Separar o código em módulos bem definidos facilita a reutilização e manutenção. Cada módulo deve ter uma responsabilidade específica e interagir de forma mínima com os outros. Os maus-cheiros relacionados são:

- **Divergent Change (Mudança Divergente):** Quando um módulo precisa ser alterado por vários motivos, ele perde coesão.
- **Shotgun Surgery (Cirurgia de Espingarda):** Pequenas mudanças exigem modificações em muitos lugares, indicando acoplamento excessivo.
- **Parallel Inheritance Hierarchies (Hierarquias de Herança Paralelas):** Classes com dependências paralelas dificultam a modularização.
- **Inappropriate Intimacy (Intimidade Inadequada):** Classes que acessam detalhes internos de outras classes reduzem a modularidade.
- **Data Clumps (Agrupamentos de Dados):** Conjuntos de dados sempre usados juntos deveriam ser encapsulados em um objeto.

2.4. Boas interfaces

Interfaces bem projetadas tornam o código mais intuitivo e fácil de usar. Uma boa interface esconde detalhes desnecessários e expõe apenas o essencial. Os maus-cheiros relacionados são:

- **Long Parameter List (Lista Longa de Parâmetros):** Funções com muitos parâmetros dificultam a usabilidade.
- **Alternative Classes with Different Interfaces (Classes Alternativas com Interfaces Diferentes):** Classes que fazem a mesma coisa, mas com interfaces diferentes, geram inconsistências.
- **Middle Man (Homem do Meio):** Classes que apenas encaminham chamadas sem agregar valor.

- **Primitive Obsession (Obsessão por Primitivos):** Uso excessivo de tipos primitivos em vez de objetos apropriados.

2.5. Extensibilidade

Um código bem estruturado permite adicionar novas funcionalidades sem alterar o que já funciona. Isso evita retrabalho e reduz o risco de erros ao evoluir o software. Os maus-cheiros relacionados são:

- **Speculative Generality (Generalização Especulativa):** Criar código genérico sem necessidade adiciona complexidade sem benefício.
- **Refused Bequest (Herança Recusada):** Subclasses que herdam métodos desnecessários indicam um design ruim.
- **Incomplete Library Class (Classe de Biblioteca Incompleta):** Dependência em bibliotecas que não oferecem as funcionalidades esperadas.
- **Switch Statements (Declarações Switch):** Uso excessivo de switch pode indicar falta de preparação para extensão.
- **Parallel Inheritance Hierarchies (Hierarquias de Herança Paralelas):** Hierarquias paralelas dificultam a extensibilidade.

2.6. Evitar duplicação

Código duplicado aumenta a complexidade e dificulta a manutenção. Um bom código reutiliza lógica sempre que possível. Os maus-cheiros relacionados são:

- **Duplicated Code (Código Duplicado):** Repetição de código dificulta a manutenção e aumenta a chance de inconsistências.
- **Data Clumps (Agrupamentos de Dados):** Conjuntos de dados sempre usados juntos deveriam ser encapsulados em um objeto.
- **Parallel Inheritance Hierarchies (Hierarquias de Herança Paralelas):** Hierarquias paralelas frequentemente indicam duplicação de código.

2.7. Portabilidade

Um código portátil funciona em diferentes ambientes com pouca ou nenhuma modificação. Isso evita dependências desnecessárias e facilita a adaptação para novas plataformas. Os maus-cheiros relacionados são:

- **Temporary Field (Campo Temporário):** Campos temporários em classes podem indicar dependência de um ambiente específico.
- **Inappropriate Intimacy (Intimidade Inadequada):** Classes que acessam diretamente detalhes internos de outras classes reduzem a portabilidade.
- **Lazy Class (Classe Preguiçosa):** Classes com pouca funcionalidade podem indicar um design mal pensado para múltiplos ambientes.
- **Incomplete Library Class (Classe de Biblioteca Incompleta):** Dependência em bibliotecas incompletas pode prejudicar a portabilidade.

- **Speculative Generality (Generalização Especulativa):** Adicionar funcionalidades desnecessárias pode criar dependências desalinhadas com a portabilidade.

2.8. Código deve ser idiomático e bem documentado

Escrever código seguindo as melhores práticas da linguagem melhora sua clareza e manutenção. A documentação deve ser objetiva e explicar o que é essencial. Os maus-cheiros relacionados são:

- **Data Class (Classe de Dados):** Classes que apenas armazenam dados sem comportamento podem indicar falta de orientação a objetos.
- **Comments (Comentários):** Comentários excessivos podem indicar que o código não é autoexplicativo.
- **Speculative Generality (Generalização Especulativa):** Generalizações prematuras podem resultar em código não idiomático.
- **Primitive Obsession (Obsessão por Primitivos):** Uso excessivo de tipos primitivos em vez de objetos pode tornar o código menos idiomático.

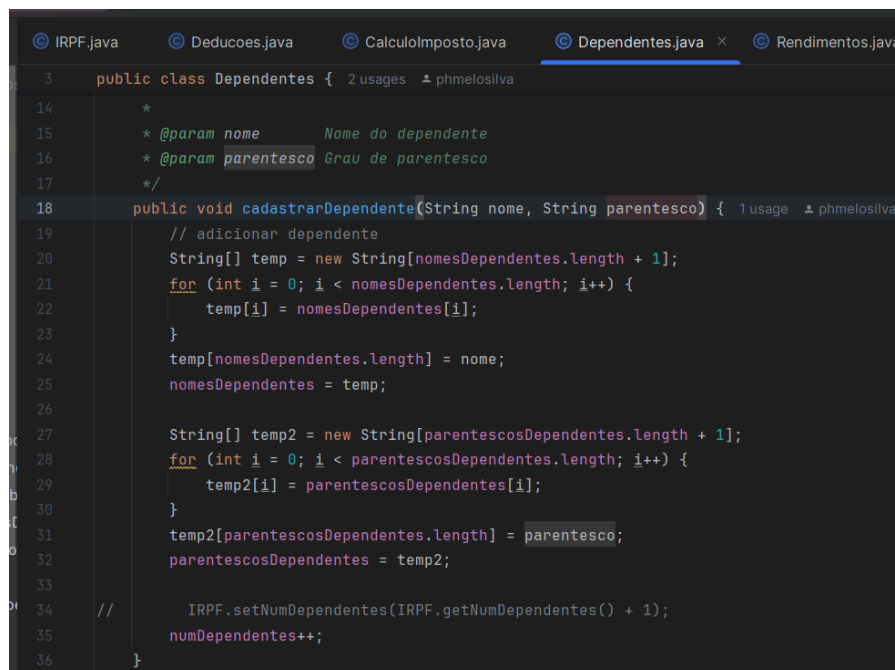
3. Identificação dos maus-cheiros

De acordo com os princípios de um bom projeto de código, e com a referência do livro “Refactoring: Improving the design of Existing Code” pode-se identificar diversos tipos de mau cheiro (“bad smell”) presentes em nosso trabalho prático.

3.1. Classe Dependentes - Duplicate Code

Na classe Dependentes, pode-se observar a presença de código duplicado, referente ao método “Dependentes.cadastrarDependente()”. Nele, observa-se que há dois arrays temporários, um para o nome dos dependentes, e outro para os parentescos desses dependentes. O problema com isso é que caso o array de dependentes for redimensionado, isso vai causar atualizações em dois lugares diferentes, o que arrisca inconsistências.

Além disso, copiar e colar o código aumenta a chance de perder uma alteração crítica (por exemplo, um erro de digitação em temp2 vs. temp). Por fim, o código repetido sobrecarrega o método e obscurece a lógica de negócios real (adicionando um dependente).

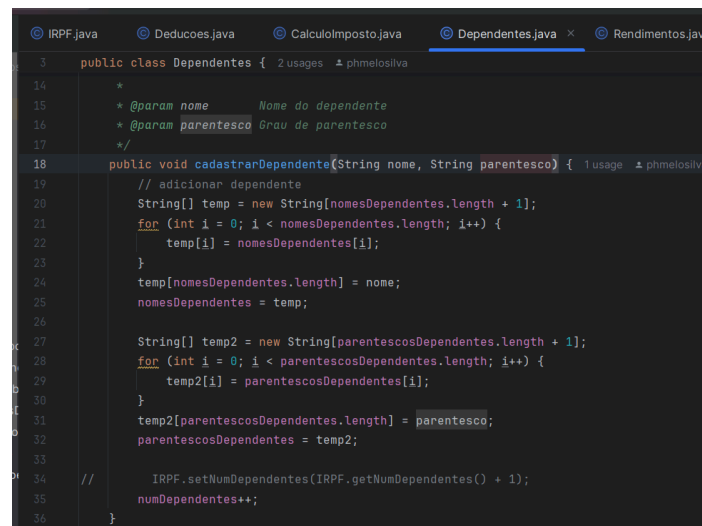


```
3 public class Dependentes { 2 usages 1 phmelosilva
14 *
15 * @param nome Nome do dependente
16 * @param parentesco Grau de parentesco
17 */
18 public void cadastrarDependente(String nome, String parentesco) { 1 usage 1 phmelosilva
19 // adicionar dependente
20 String[] temp = new String[nomesDependentes.length + 1];
21 for (int i = 0; i < nomesDependentes.length; i++) {
22     temp[i] = nomesDependentes[i];
23 }
24 temp[nomesDependentes.length] = nome;
25 nomesDependentes = temp;
26
27 String[] temp2 = new String[parentescosDependentes.length + 1];
28 for (int i = 0; i < parentescosDependentes.length; i++) {
29     temp2[i] = parentescosDependentes[i];
30 }
31 temp2[parentescosDependentes.length] = parentesco;
32 parentescosDependentes = temp2;
33
34 // IRPF.setNumDependentes(IRPF.getNumDependentes() + 1);
35 numDependentes++;
36 }
```

Para poder refatorar e lidar com esse tipo de mau cheiro, pode-se utilizar a operação “**Extract Method**”. Isso pode ser feito com a extração da lógica de redimensionamento de array em um único método auxiliar, ou com o uso de um ArrayList de objetos, que guardaria tanto o nome, quanto o parentesco do dependente.

3.2. Classe Dependentes - Long Method

Na classe Dependentes, no mesmo local onde ocorre duplicação de código, temos a presença de um método longo devido ao problema de duplicação, outro mau cheiro ocasionado por outro problema presente em um mesmo método. Operações de refatoração como o uso de “**ArrayList**” ou extração da lógica de manipulação de array como visto em 3.2, podem ser usados para este caso.

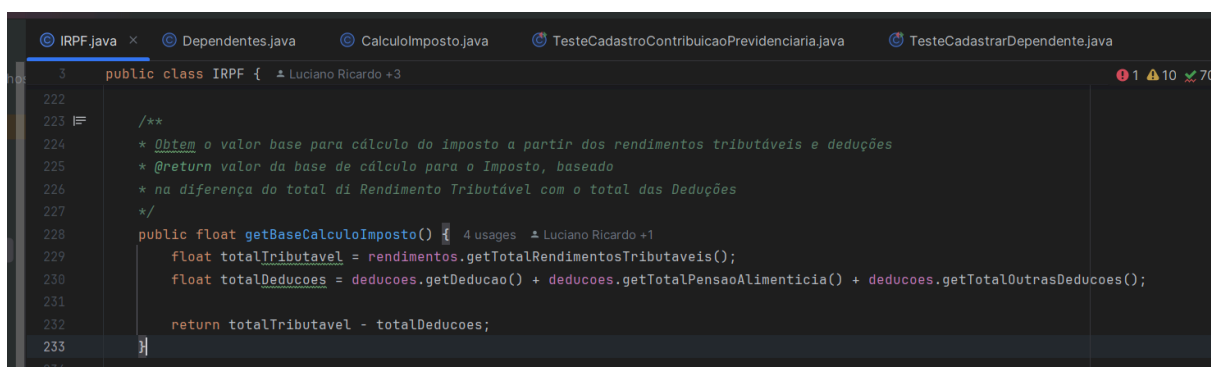


```
3 public class Dependentes { 2 usages 1 phmelosilva
14
15     * @param nome      Nome do dependente
16     * @param parentesco Grau de parentesco
17     */
18 public void cadastrarDependente(String nome, String parentesco) { 1 usage 1 phmelosilva
19     // adicionar dependente
20     String[] temp = new String[nomesDependentes.length + 1];
21     for (int i = 0; i < nomesDependentes.length; i++) {
22         temp[i] = nomesDependentes[i];
23     }
24     temp[nomesDependentes.length] = nome;
25     nomesDependentes = temp;
26
27     String[] temp2 = new String[parentescosDependentes.length + 1];
28     for (int i = 0; i < parentescosDependentes.length; i++) {
29         temp2[i] = parentescosDependentes[i];
30     }
31     temp2[parentescosDependentes.length] = parentesco;
32     parentescosDependentes = temp2;
33
34     // IRPF.setNumDependentes(IRPF.getNumDependentes() + 1);
35     numDependentes++;
36 }
```

3.3. Classe IRPF - Message Chains

Na classe IRPF, temos o método de cálculo da base do imposto, que acaba utilizando de uma longa cadeia de chamadas do objeto deducoes, na linha 230 do código abaixo. Com isso, uma operação de refatoração simples a ser implementada é expor o cálculo dessa cadeia em “getTotalDeducoes()” dentro da própria classe Deducoes.

Dessa forma, o código se tornará menos acoplado e mais coeso para este caso.



```
3 public class IRPF { 1 Luciano Ricardo +3
222
223 /**
224  * @return o valor base para cálculo do imposto a partir dos rendimentos tributáveis e deduções
225  * @return valor da base de cálculo para o Imposto, baseado
226  * na diferença do total de Rendimento Tributável com o total das Deduções
227  */
228 public float getBaseCalculoImposto() 4 usages 1 Luciano Ricardo +1
229     float totalTributavel = rendimentos.getTotalRendimentosTributaveis();
230     float totalDeducoes = deducoes.getDeducacao() + deducoes.getTotalPensaoAlimenticia() + deducoes.getTotalOutrasDeducoes();
231
232     return totalTributavel - totalDeducoes;
233 }
```

3.4. Classe Dependentes - Primitive Obsession

Na classe Dependentes, os dois arrays paralelos "nomeDependentes" e "parentescoDependentes" são usados para armazenar dados relacionados (nome e parentesco), ao invés de um objeto próprio para isso.

Dessa forma, a refatoração "**Extract Class**" continua sendo a melhor opção. Pois é a operação que consegue lidar com vários outros maus cheiros que apresentam problemas em relação a essa mesma estrutura desses dois arrays identificados.

Mas, fica a menção ao uso de técnicas como "**Replace Data Value with Object**" ou "**Replace Type Code with Class**" que lidam bem nos casos deste tipo de mau cheiro. Portanto, como há a presença de um grupo de campos que precisam ir juntos, nome, parentesco e qualquer outro campo que possa vir a ser adicionado, então o uso da refatoração "**Extract Class**" se encaixa perfeitamente para este caso.

```
2
3 public class Dependentes { 2 usages  👤 phmelosilva
4     public String[] nomesDependentes; 10 usages
5     String[] parentescosDependentes; 7 usages
```

3.5. Classe Dependentes - Data Clumps

Na classe Dependentes, temos arrays e um contador que estão fortemente acoplados, mas armazenados separadamente, correndo o risco de inconsistência. Veja que a classe utiliza de 2 arrays separados e um contador "numDependentes". Nisso temos que "nomesDependentes" e "parentescosDependentes" estão intrinsecamente ligados, e um contador, onde ambos array e contador precisam ser atualizados sempre que um dependente é adicionado ou removido, o que gera um risco de inconsistência. Além disso, há o uso de arrays/inteiro em vez de um objeto adequado para representar um dependente.

```
2
3 public class Dependentes { 2 usages  👤 phmelosilva
4     public String[] nomesDependentes; 10 usages
5     String[] parentescosDependentes; 7 usages
6     public int numDependentes; 3 usages
```


Com isso, veja o impacto que esse tipo de mau cheiro causa no método “getParentesco” na classe IRPF. Temos problemas como dependência direta de índices de array e se os arrays estiverem fora de sincronia, este método retornará dados incorretos ou travará.

```
146      /**
147       * Método que retorna o grau de parentesco para um dado dependente, caso ele
148       * conste na lista de dependentes
149       * @param dependente nome do dependente
150       * @return grau de parentesco, nulo caso não exista o dependente
151       */
152      public String getParentesco(String dependente) { 6 usages  Luciano Ricardo +1
153          for (int i = 0; i < dependentes.nomesDependentes.length; i++) {
154              if (dependentes.nomesDependentes[i].equalsIgnoreCase(dependente))
155                  return dependentes.parentescosDependentes[i];
156          }
157          return null;
158      }
```

Para lidar com este mau cheiro, deve-se utilizar de uma refatoração que encapsule os dados agrupados em uma única classe, além de buscar substituir os arrays paralelos com uma lista de dependentes, que já vai estar armazenando tanto o nome, quanto o parentesco do dependente. Ou seja, basicamente utilizar a operação de refatoração “**Extract Class**” e o usar “**Introduce Parameter Object**” é o suficiente para resolver esse grande problema.

3.6. Classe IRPF- Long Parameter List

Na Classe IRPF, existe o método getImpostoPorFaixa que é responsável por calcular o valor do imposto a se pagar de acordo com cada faixa, este método possui quatro parâmetros e todos são utilizados para o cálculo. O excesso de parâmetros é um mau cheiro que torna o método difícil de se entender e que tende a piorar com evoluções do software.

Como podemos ver, os métodos baseCalculo, minimo, maximo e aliquota são utilizados no cálculo, porém estes mesmos métodos são variáveis estáticas na classe CalculoImposto, que por sua vez utiliza o método diversas vezes usando como parâmetros estas variáveis estáticas, uma solução é utilizar a operação de refatoração “**Preserve Whole Object**” onde o agrupamento de dados é passado como uma classe e não como uma lista de parâmetros.

```
public float getImpostoPorFaixa(float baseCalculo, float minimo, float maximo, float aliquota) {
    if (baseCalculo < minimo) {
        return 0.0f;
    }
    if (baseCalculo >= maximo) {
        baseCalculo = maximo;
    }
    return (float) (Math.floor((baseCalculo - minimo) * aliquota * 100.0) / 100.0);
}
```

4. Referências

Martin Fowler. Refactoring: Improving the design of Existing Code. Addison-Wesley Professional, 1999.

Pete Goodliffe. Code Craft: The practice of Writing Excellent Code. No Starch Press, 2006.