**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS**

MATHEUS VINICIUS MOTA RODRIGUES
814410

# ANÁLISE DE BAD SMELLS

BELO HORIZONTE
2025

## Análise de Smells

Durante a análise do código original ReportGenerator.js, foram identificados três *Bad Smells* principais:

**Método Longo (Long Method)**

O método generateReport() tinha mais de 70 linhas, englobando toda a lógica de formatação (HTML e CSV), filtragem de itens e cálculo de total.
Esse *smell* dificulta:

- Leitura e compreensão do fluxo lógico;
- Testabilidade (não há como isolar partes da lógica);
- Manutenção (alterações em uma parte podem quebrar outras).

report

```
if (reportType === 'CSV') {
      report += 'ID,NOME,VALOR,USUARIO\n';
    } else if (reportType === 'HTML') {
      report += '<html><body>\n';
      report += '<h1>Relatório</h1>\n';
      report += `<h2>Usuário: ${user.name}</h2>\n`;
      report += '<table>\n';
      report += '<tr><th>ID</th><th>Nome</th><th>Valor</th></tr>\n';
    }
```

A formatação HTML/CSV estava misturada à lógica de negócio (filtro por usuário e cálculo de total).

**Condicionais Complexas e Duplicadas (Complex Conditional)**

A presença de condicionais aninhadas (if/else if) para lidar com combinações de user.role e reportType elevava a complexidade cognitiva. Além disso, havia duplicação de código para formatação de linhas CSV e HTML em diferentes blocos.

Esse *smell* é problemático porque:

- Reduz a clareza do comportamento esperado;
- Aumenta o risco de erro ao adicionar novos tipos de usuário ou formato de relatório;
- Dificulta o uso de ferramentas estáticas (como ESLint) que medem *cognitive complexity*.

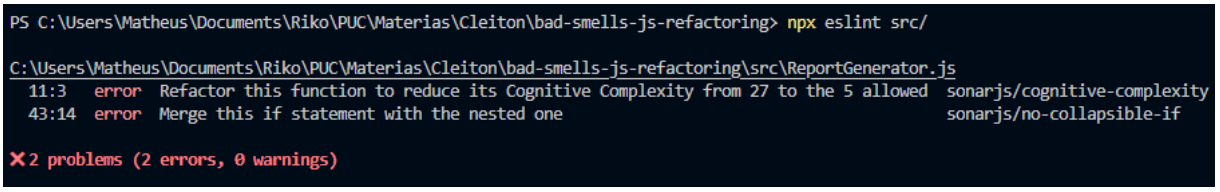
## Duplicação de Código (Duplicated Code)

A geração de linhas tanto para HTML quanto CSV se repetia em múltiplos blocos condicionais. Além disso, a lógica de somar total e aplicar estilo de prioridade aparecia mais de uma vez. Como consequência, manter o código duplicado exige atualizações em múltiplos pontos. Um pequeno ajuste no formato de saída obrigaria a alterar quatro trechos diferentes.

## Relatório da Ferramenta (ESLint + SonarJS)

O uso da ferramenta ESLint com o plugin eslint-plugin-sonarjs foi essencial para quantificar e detectar esses problemas.

### Print (Antes da Refatoração)




O SonarJS destacou funções muito complexas que apresentavam complexidade acima de 20, muito além do ideal (<15) e condicionais aninhadas demais. Essas métricas reforçaram o diagnóstico manual, mostrando numericamente o impacto da complexidade e indicando pontos exatos para refatoração.

## Processo de Refatoração

O método generateReport() foi o principal foco da refatoração. A abordagem utilizada foi combinar a abordagem Extract Method com Replace Conditional with Polymorphism.

### Antes

report

```
if (user.role === 'ADMIN') {
  if (item.value > 1000) item.priority = true;
  if (reportType === 'CSV') {
    report += `${item.id},${item.name},${item.value},${user.name}\n`;
  } else if (reportType === 'HTML') {
    const style = item.priority ? ' style="font-weight:bold;"' : '';
    report += `<tr${style}><td>${item.id}</td><td>${item.name}</td><td>${item.value}</td>
</tr>\n`;
  }
} else if (user.role === 'USER') {
  if (item.value <= 500) { ... }
}
```

**Depois**

```
report

const formatter = this.createFormatter(reportType);
const strategy = this.createStrategy(user);

for (const item of items) {
  if (!strategy.shouldInclude(item)) continue;
  strategy.postProcessItem(item);
  report += formatter.formatItem(item, user);
  total += item.value;
}
```

**Técnicas aplicadas**

- Extract Method: quebrou o método em unidades menores (createFormatter, createStrategy, formatItem, etc.).
- Replace Conditional with Polymorphism: substituiu múltiplos if/else por classes específicas (CsvFormatter, HtmlFormatter, AdminStrategy, CommonUserStrategy).
- DRY (Don't Repeat Yourself): eliminou duplicações de formatação e soma de totais.

## Conclusão

A refatoração do código trouxe melhorias significativas em termos de clareza, organização e manutenibilidade. O método `generateReport`, originalmente extenso e repleto de condicionais aninhadas, foi dividido em partes menores e mais coesas, tornando o código mais legível e fácil de evoluir. A substituição de estruturas condicionais complexas por polimorfismo reduziu a duplicação e a complexidade ciclomática, além de facilitar a inclusão de novos tipos de relatórios ou papéis de usuários sem necessidade de alterar o código central. Durante esse processo, os testes automatizados desempenharam um papel essencial como rede de segurança, garantindo que o comportamento funcional permanecesse o mesmo após as mudanças estruturais. Em suma, a eliminação de *Bad Smells* e o uso de testes contribuíram diretamente para um software mais robusto, compreensível e sustentável a longo prazo.