

Code Smells + Princípios SOLID

Boas Práticas de Programação - BPP 2025.2

Prof. Fernando Marques Filho

12 de Setembro de 2025

Universidade Federal do Rio Grande do Norte

Agenda

Introdução a Code Smells

Ferramentas de Detecção

Princípios SOLID

Aplicação Prática

Conclusão

Introdução a Code Smells

O que são Code Smells?

Sinais de problemas no código

Indicam possíveis violações de boas práticas

Não são bugs, mas podem levar a eles

Definição

Code smells são características superficiais no código-fonte que podem indicar problemas mais profundos no design, estrutura ou implementação.

Exemplos Comuns

- **Long Method** - Métodos muito longos
- **Duplicate Code** - Código duplicado
- **Large Class** - Classes muito grandes
- **Poor Naming** - Nomenclatura inadequada

Origem e Importância

Termo criado por: Martin Fowler (1999)

Inspiração: Kent Beck

Livro: "Refactoring: Improving the Design of Existing Code"

Por que são importantes?

- Facilitam identificação de problemas
- Guiam decisões de refatoração
- Melhoram manutenibilidade
- Reduzem débito técnico



Code Smell

Sinal de que algo pode
estar errado no design,
mas não é um bug, mas
pode levar a problemas

Recurso Oficial:

<https://luzkan.github.io/smells/>

Categorias Principais

- **Bloaters** - Código inchado
- **Object-Orientation Abusers** - Abuso de OO
- **Change Preventers** - Impedimentos de mudança
- **Dispensables** - Código desnecessário
- **Couplers** - Acoplamento excessivo

Projeto BPP 2025.2

Consulte também os exemplos no repositório:

[https:](https://github.com/fmarquesfilho/bpp-2025-2)

[//github.com/fmarquesfilho/bpp-2025-2](https://github.com/fmarquesfilho/bpp-2025-2)

Exemplo: Long Method

❌ Violação

```
1 def process_user_registration(name, email, password, age, address,
2     phone):
3     # Validacao (15 linhas)
4     if not name or len(name) < 2:
5         raise ValueError("Nome invalido")
6     if not email or "@" not in email:
7         raise ValueError("Email invalido")
8     if not password or len(password) < 8:
9         raise ValueError("Senha muito fraca")
10    if age < 18 or age > 120:
11        raise ValueError("Idade invalida")
12    if not phone or len(phone) < 10:
13        raise ValueError("Telefone invalido")
14
15    # Formatacao (10 linhas)
16    formatted_name = name.strip().title()
```

Exemplo: Long Method Refatorado

✓ Solução

```
1 def process_user_registration(user_data):
2     """Processa registro de usuario aplicando validacao, formatacao e
3     persistencia."""
4     validate_user_input(user_data)
5     formatted_data = format_user_data(user_data)
6     save_user(formatted_data)
7     send_welcome_email(formatted_data.email)
8     return True
9
10 def validate_user_input(user_data):
11     """Valida todos os dados de entrada do usuario."""
12     if not is_valid_name(user_data.name):
13         raise ValueError("Nome invalido")
14     if not is_valid_email(user_data.email):
15         raise ValueError("Email invalido")
16     if not is_valid_password(user_data.password):
```


Ferramentas de Detecção

pylint

Análise estática completa

```
pip install pylint
pylint meu_projeto/
```

flake8

Estilo e complexidade

```
pip install flake8
flake8 --max-complexity=10
```

radon

Métricas de complexidade

```
pip install radon
radon cc . -a
```

vulture

Código não utilizado

```
pip install vulture
vulture .
```

Exemplo: Saída do pylint

Resultado da Análise

```
user_service.py:45:0: R0903: Too few public methods (1/2) (too-few-public-methods)
user_service.py:50:4: R0913: Too many arguments (6/5) (too-many-arguments)
user_service.py:50:4: R0915: Too many statements (55/50) (too-many-statements)
auth.py:23:4: W0612: Unused variable 'temp' (unused-variable)
utils.py:10:0: C0103: Function name "doStuff" doesn't conform to snake_case...
```

Code Smells Detectados

- Too many arguments → Long Parameter List
- Too many statements → Long Method
- Unused variable → Dead Code
- Bad naming → Poor Naming

Princípios SOLID

Princípios SOLID

S	SRP	Single Responsibility Principle
O	OCP	Open/Closed Principle
L	LSP	Liskov Substitution Principle
I	ISP	Interface Segregation Principle
D	DIP	Dependency Inversion Principle

Objetivo

Criar software mais flexível, compreensível e sustentável através de princípios de design orientado a objetos.

SRP - Single Responsibility Principle

❌ Violação

```
1 class UserManager:
2     def create_user(self, user_data):
3         # Valida dados
4         # Salva no banco
5         # Envia email
6         # Gera log
7         pass
8
9     def generate_report(self):
10        # Busca dados
11        # Formata relatorio
12        # Salva PDF
13        # Envia por email
14        pass
```

SRP - Single Responsibility Principle

✓ Solução

```
1 class UserValidator:
2     def validate(self, user_data): pass
3
4 class UserRepository:
5     def save(self, user_data): pass
6
7 class EmailService:
8     def send_welcome_email(self, email): pass
9
10 class Logger:
11     def log_user_creation(self, user_data): pass
12
13 # Cada classe tem uma unica responsabilidade
```

❌ Violação

```
1 class PaymentProcessor:
2     def process_payment(self, payment_type, amount):
3         if payment_type == "credit_card":
4             # Processa cartao
5         elif payment_type == "paypal":
6             # Processa paypal
7         elif payment_type == "pix":
8             # Processa pix
9         # Novo metodo -> modificar classe
```

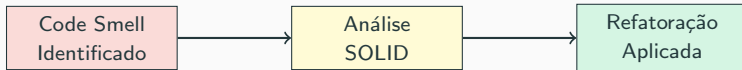

OCP - Open/Closed Principle

✓ Solução

```
1 from abc import ABC, abstractmethod
2
3 class PaymentMethod(ABC):
4     @abstractmethod
5     def process(self, amount): pass
6
7 class CreditCardPayment(PaymentMethod):
8     def process(self, amount): pass
9
10 class PayPalPayment(PaymentMethod):
11     def process(self, amount): pass
12
13 class PaymentProcessor:
14     def process_payment(self, payment_method, amount):
15         payment_method.process(amount)
```

Aplicação Prática

Processo de Identificação e Refatoração



Processo

1. **Identificar** code smells
2. **Analisar** princípio SOLID violado
3. **Aplicar** refatoração apropriada
4. **Validar** melhoria na qualidade

Dica

Priorize refatorações baseado no impacto e frequência de mudanças na área do código.

Conclusão

Próximos Passos

1. **Prática:** Aplicar análise estática no seu projeto
2. **Identificar:** Code smells no código atual
3. **Refatorar:** Aplicar princípios SOLID
4. **Documentar:** Registrar as melhorias

Ferramentas Recomendadas

- **pylint:** Análise completa
- **radon:** Métricas de complexidade
- **vulture:** Dead code detection
- **flake8:** Estilo e boas práticas

Referências

- Fowler, Martin. **Refactoring: Improving the Design of Existing Code**
- Martin, Robert C. **Clean Code: A Handbook of Agile Software Craftsmanship**
- Catálogo de Code Smells: <https://luzkan.github.io/smells/>
- SOLID Principles: <https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898>
- Repositório do curso: <https://github.com/fmarquesfilho/bpp-2025-2>

Dúvidas?