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
- **God Class** - Classe que faz tudo

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

Não é um bug, mas pode levar a problemas

Catálogo de Code Smells

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:

Clique aqui para abrir

Exemplo: Long Method

✖ Violação

```python
def process_user_registration(name, email, password, age):
    # Validacao (15+ linhas no total)
    if not name or len(name) < 2:
        raise ValueError("Nome invalido")
    if not email or "@" not in email:
        raise ValueError("Email invalido")
    if not password or len(password) < 8:
        raise ValueError("Senha muito fraca")
    if age < 18 or age > 120:
        raise ValueError("Idade invalida")

    # ... mais codigo aqui
    return True
```

Clique aqui para ver exemplo completo.

Exemplo: Long Method Refatorado

✔ Solução

```python
def process_user_registration(user_data):
    """Processa registro aplicando validacao, formatacao e persistencia.
    """
    validate_user_input(user_data)
    formatted_data = format_user_data(user_data)
    save_user(formatted_data)
    send_welcome_email(formatted_data.email)
    return True

def validate_user_input(user_data):
    """Valida todos os dados de entrada do usuario."""
    if not is_valid_name(user_data.name):
        raise ValueError("Nome invalido")
    # ... outras validacoes
```

Clique aqui para ver exemplo completo.

Ferramentas de Detecção

Ferramentas Python para Análise

pylint
Análise estática completa

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

radon
Métricas de complexidade

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

flake8
Estilo e complexidade

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

vulture
Código não utilizado

```
pip install vulture
vulture .
```

Exemplo Prático - Aula de Hoje

</> Vamos analisar código real usando essas ferramentas!

Códigos em:

`https://github.com/fmarquesfilho/bpp-2025-2/tree/main/exemplos/analise-pratica`

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 - Classe com múltiplas responsabilidades

```python
class UserManager:
    def create_user(self, user_data):
        # Valida dados + Salva no banco + Envia email + Gera log
        pass

    def generate_report(self):
        # Busca dados + Formata + Salva PDF + Envia email
        pass
```

Exemplos Completos

Ver violações e soluções SOLID em:

https://github.com/fmarquesfilho/bpp-2025-2/tree/main/exemplos/solid

SRP - Single Responsibility Principle

✔ Solução - Responsabilidades separadas

```python
class UserValidator:
    def validate(self, user_data): pass

class UserRepository:
    def save(self, user_data): pass

class EmailService:
    def send_welcome_email(self, email): pass

class Logger:
    def log_user_creation(self, user_data): pass

# Cada classe tem uma única responsabilidade
```

OCP - Open/Closed Principle

✗ Violação - Modificação para extensão

```python
class PaymentProcessor:
    def process_payment(self, payment_type, amount):
        if payment_type == "credit_card":
            # Processa cartao
        elif payment_type == "paypal":
            # Processa paypal
        elif payment_type == "pix":
            # Processa pix
        # Novo método -> modificar classe existente
```

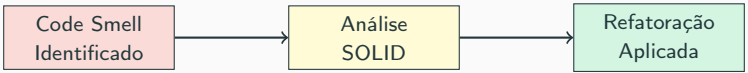
OCP - Open/Closed Principle

✔ Solução - Extensão sem modificação

```python
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def process(self, amount): pass

class CreditCardPayment(PaymentMethod):
    def process(self, amount): pass

class PaymentProcessor:
    def process_payment(self, payment_method, amount):
        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.

Exercícios Práticos - Aula de Hoje

Tarefa 1: Instalação das Ferramentas

1. Instalar as ferramentas: `pip install pylint flake8 radon vulture`
2. Verificar instalação: `pylint -version`
3. Clonar repositório: `git clone https://github.com/fmarquesfilho/bpp-2025-2`

Tarefa 2: Análise com pylint

Analisar o arquivo `exemplos/analise-pratica/problema1.py` e identificar:

- Quais code smells foram detectados?
- Qual a pontuação do código?
- Quais princípios SOLID estão sendo violados?

Tarefa 3: Métricas de Complexidade

Usar radon para medir complexidade: `radon cc exemplos/analise-pratica/ -a`

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?