17 - Smells and Heuristics

2025-10-04 11:09 am

Resumo das razões para código ruim:

Comentários:

1. **Informação inapropriada**
    - Informação que deve existir em outros lugares, como controle de versão
2. **Comentário obsoleto**
3. **Comentário redundante**
    - Não reescrever o que o código já diz por si só
4. **Comentário mal-escrito**
5. **Código comentado**

Ambiente:

1. **Build requer mais de um passo**
    - Deve ser são simples como dar um único comando
2. **Teste requer mais de um passo**
    - Também rodar testes com um único comando, de preferência direto na IDE
3. **Argumentos demais**
4. **Ter output arguments**
    - Contraintuitivo, fonte de bugs. Objeto alterado deve ser o dono do método chamado.
5. **Ter flag arguments**
    - Viola SRP, melhor separar em funções diferentes
6. **Dead Functions**
    - Código nunca chamado deve ser descartado

Geral:

1. **Múltiplas linguagens num único arquivo de código**
    - Não colocar SQL, Inglês, HTML e JavaDoc tudo junto, quanto menos melhor.
    - Ideal é uma única linguagem, mas às vezes requer mais algumas.
2. **Comportamento óbvio não é implementado**
    - *Principle of Least Surprise*
    - Funcionalidades devem ser óbvias e intuitivas.
3. **Comportamento incorreto nas *boundaries*:**
    - Inspecionar cada *edge case* e idiossincrasia.
    - Escrever testes para toda situação atípica
    - Não confiar na intuição.
4. **Segurança Burlada (overriden)**:
    - Sempre respeitar warnings, testes falhando e barreiras de segurança.
5. **Duplicação:**
    - Um dois pontos mais importantes
    - **DRY** - Don't repeat yourself
    - Duplicação é uma oportunidade de abstração.

    - Isso inclui duplicação de conceitos e responsabilidades mesmo quando o código não é igual.
6. **Código no Nível Errado de Abstração:**
    - Constantes e variáveis que são detalhe de implementação não devem existir na classe base
    - Implementações devem estar nas etapas menos abstratas do código.
7. **Classe Base dependendo das suas classes derivadas**:
    - Base classe deve ser independente de seus derivados de nível mais baixo.
    - Há exceções. E.g. código na classe base que seleciona entre as classes derivadas.
    - Devem existir em arquivos diferentes.
8. **Informação Demais:**
    - Interfaces devem ser pequenas e rasas.
    - Deve ter poucas funções.
    - Acoplamento assim fica baixo, que é bom.
    - Evitar multidão de funções utilitárias e dados privados.
9. **Dead Code**:
    - Código não executado.
    - E.g. condição que é impossível de ocorrer.
10. **Separação Vertical:**
    - Variáveis e funções devem ser definidas próximas de onde elas são usadas
11. **Inconsistência:**
    - Organizar o código sempre da mesma forma
12. **Bagunça (Clutter):**
    - Remover código sem uso ou propósito.
13. **Acoplamento Artificial:**
    - Só acoplar coisas que dependem um do outro
    - e.g. Não acoplar *enums* ou funções estáticas gerais a classes
14. **Feature Envy:**
    - Método de uma classe deve lidar apenas consigo própria e não variáveis de outros objetos.
15. **Argumentos seletores:**
    - Análogo a flag arguments, melhor dividir em várias funções.
16. **Intenção obscura:**
    - Usar nomes claros e evitar notações muito obtusas ou magic numbers
17. **Responsabilidade mal colocada:**
    - Juntar códigos que ficam intuitivamente juntos a um leitor qualquer.
18. ***Static* Inapropriado:
    - Só usar quando método realmente não depender de nada do objeto, apenas argumentos.
    - Em geral é melhor método ser não-estático
19. **Use Variáveis Explicativas:**
    - Quebrar cálculos em etapas intermediárias óbvias.
20. **Nomes devem dizer o que é feito**
21. **Entenda o Algoritmo:**
    - Não basta passar testes e parecer funcionar.
    - Devemos **saber** que está funcionando.
    - Requer refatoração com frequência.
22. **Faça Dependências Lógicas serem Físicas:**
    - Se um método depende de uma informação, isso deve estar explícito dentro daquela classe e não em outra.
23. **Prefira Polimorfismo a if/else ou switch/case**
24. **Siga Convenções Padrões:**
    - Para o time inteiro, qualquer que seja.
25. **Substitua Magic Number por Constantes Nomeadas:**

    - Podemos deixar o número mesmo para constantes muito óbvias, como o 2 em `PI * RADIUS * 2`
26. **Seja Preciso:**
    - Evitar ambiguidades e imprecisões, código deve lidar com todas as suas possibilidades, mesmo improváveis.
27. **Estrutura é mais importante que Convenção**
    - Forçar código ter uma certa estrutura é melhor do que escolher bons nomes
    - Mais rígido e previsível.
28. **Encapsule Condições:**
    - Evitar sequência longa de and e or pouco explícitos para o if/else
29. **Evite Condições Negativas**
30. **Funções devem fazer uma única coisa**
31. **Não esconda acoplamento temporal:**
    - Importante forçar uma certa ordem de execução de métodos quando esta ordem importa.
    - Colocar como argumentos aquilo que deve vir antes, mesmo que fique mais complexo.
32. **Não Seja Arbitrário:**
    - Estrutura do código deve fazer sentido e ser clara por que razão é assim.
33. **Encapsule *Boundary Conditions:***
    - Deixar tudo num lugar só, e.g. +1, -1.
34. **Funções devem descer um único nível de Abstração:**
    - Um dois pontos mais importantes e mais difíceis.
35. **Mantenha Dados Configuráveis em Alto Nível:**
    - Código de baixo nível deve receber do alto nível as constantes e variáveis necessárias.
36. **Evite Navegação Transitiva:**
    - Módulos não devem saber muito sobre outros módulos, devem ser independentes tanto quanto possível.
    - Se A depende de B e B depende de C, A não deve ter nada a ver com C.
    - *Lei de Demeter*

Java:

1. **Evite longas listas de imports, use o wildcard**
2. **Não herde constantes**
3. **Use Enums no lugar de Constantes quando apropriado**

Nomes:

1. **Escolha nomes descritivos:**
    - Requer cuidado e reescrita contínuos
2. **Escolha nomes apropriados ao nível de abstração:**
    - Não escolha nomes que comunicam implementação.
    - E.g. misturar *Modem* com *phoneNumber*, níveis diferentes
3. **Use nomenclatura padrão quando possível:**
    - Usar o nome *Decorator* para um código no padrão Decorator.
4. **Não use nomes ambíguos:**
    - Ser bem claro mesmo que o nome seja longo
5. **Use Nomes Longos para escopos maiores**
6. **Evite Encodings**:
    - e.g. prefixos ou notação húngara
7. **Nomes devem descrever side-effects**

Testes:

1. **Testes Insuficientes**
2. **Use uma ferramenta de cobertura**
3. **Não pule testes triviais**
4. **Um teste ignorado pode gerar ambiguidade de comportamento**
5. **Teste *Boundary Conditions***
6. **Concentre mais testes onde há mais bugs**
    - Bugs costumam andar em bando
7. **Padrões de falha de testes podem revelar razões de problema no código**
8. **Padrões de Cobertura de testes podem revelar problemas**
9. **Testes devem ser rápidos**