Documentação Padrão - Revisor de Código com IA

1. Clean Code

Nomenclatura

- Use nomes descritivos: `calculateTotalPrice()` não `calc()`
- Evite abreviações: `userRepository` não `usrRepo`
- Funções são verbos: `getData()`, `validateInput()`
- Classes são substantivos: `User`, `PaymentProcessor`
- Booleanos começam com `is`, `has`, `can`: `isValid`, `hasPermission`

Funções

- Uma responsabilidade por função
- Máximo 3-4 parâmetros
- Retorne cedo para reduzir aninhamento
- Não cause efeitos colaterais inesperados

Código

- Remova código comentado
- Elimine código morto (nunca executado)
- Mantenha formatação consistente
- Evite comentários óbvios

2. SOLID

Single Responsibility

- Uma classe = uma razão para mudar
- Função faz apenas o que o nome promete

Open/Closed

- Extensível sem modificar código existente
- Use interfaces/abstrações

Liskov Substitution

- Subclasses substituem classes base sem quebrar funcionalidade

Interface Segregation

- Interfaces pequenas e específicas
- Não force implementação de métodos não utilizados

Dependency Inversion

- Dependa de abstrações, não implementações concretas
- Use injeção de dependência

3. YAGNI (You Ain't Gonna Need It)

- Implemente apenas o que é necessário agora
- Não adicione funcionalidades "por precaução"
- Evite abstrações prematuras
- Mantenha soluções simples

4. Bugs Comuns

Null/Undefined

- Sempre verificar se objeto existe antes de usar
- Inicializar variáveis adequadamente
- Validar parâmetros de função

Loops

- Verificar condições de parada
- Evitar loops infinitos
- Cuidado com índices fora do limite (off-by-one)

Condicionais

- Verificar todos os casos possíveis
- Cuidado com comparações (`==` vs `===`)
- Validar condições complexas

Recursos

- Fechar arquivos, conexões, streams
- Liberar memória quando necessário
- Limpar timers/listeners

Concorrência

- Race conditions

- Deadlocks
- Sincronização inadequada

5. Code Smells

Função/Método

- **Função longa**: >20-30 linhas
- **Lista de parâmetros longa**: >3-4 parâmetros
- **Função complexa**: Muitos `if/else` aninhados
- **Função que faz tudo**: Múltiplas responsabilidades

Classe/Objeto

- **Classe grande**: >200-300 linhas
- **Classe com poucos métodos**: Pode ser apenas dados
- **Classe com muitas responsabilidades**: Viola SRP
- **Herança muito profunda**: Dificulta manutenção

Código

- **Código duplicado**: Mesmo código em vários lugares
- **Magic numbers**: Números sem contexto
- **Variáveis globais**: Acoplamento forte
- **Long chains**: `obj.a.b.c.d.e`

Nomes

- **Nomes genéricos**: `data`, `info`, `temp`, `obj`
- **Nomes enganosos**: Nome não reflete funcionalidade
- **Inconsistência**: Padrões diferentes no mesmo projeto

Comentários

- **Comentários obsoletos**: Não refletem o código atual
- **Comentários explicando código ruim**: Refatore ao invés
- **Código comentado**: Remova, use controle de versão

6. Tratamento de Erros

Identificar

- Nunca ignore exceções silenciosamente

- Não use `try/catch` vazio
- Não mascare erros
- Sempre valide entrada de dados

Corrigir

- Use tipos específicos de exceção
- Forneça mensagens descritivas
- Libere recursos em caso de erro
- Implemente fallbacks quando possível

7. Testes

Problemas

- Testes que sempre passam
- Testes dependentes entre si
- Testes sem assertions
- Testes com lógica complexa

Boas Práticas

- Nomes descritivos do que está testando
- Um conceito por teste
- Arrange-Act-Assert
- Testes rápidos e independentes

8. Segurança

Vulnerabilidades

- SQL Injection
- XSS (Cross-Site Scripting)
- Credenciais hardcoded
- Dados sensíveis em logs
- Validação inadequada de entrada

Prevenir

- Sempre validar/sanitizar entrada
- Usar prepared statements
- Escapar output

- Não expor informações sensíveis

9. Performance

Problemas

- Loops desnecessários
- Consultas dentro de loops (N+1)
- Vazamentos de memória
- Recursos não liberados
- Algoritmos ineficientes

Melhorar

- Use estruturas de dados apropriadas
- Cache quando faz sentido
- Evite operações custosas desnecessárias
- Lazy loading para recursos grandes

10. Checklist Rápido

Bugs

- [ ] Verificações de null/undefined
- [ ] Condições de loop corretas
- [ ] Recursos liberados adequadamente
- [ ] Tratamento de exceções presente
- [ ] Validação de entrada implementada

Code Smells

- [ ] Funções pequenas (<30 linhas)
- [ ] Nomes descritivos e consistentes
- [ ] Sem código duplicado
- [ ] Sem magic numbers
- [ ] Sem código morto/comentado

Clean Code

- [ ] Uma responsabilidade por função/classe
- [ ] Código autoexplicativo
- [ ] Formatação consistente
- [ ] Sem efeitos colaterais inesperados

SOLID

- [ ] Classes com responsabilidade única
- [ ] Dependências invertidas quando apropriado
- [ ] Interfaces específicas

YAGNI

- [ ] Todo código é necessário
- [ ] Sem abstrações prematuras
- [ ] Solução mais simples possível

Segurança

- [ ] Entrada validada/sanitizada
- [ ] Sem credenciais expostas
- [ ] Tratamento seguro de dados sensíveis

Performance

- [ ] Sem loops desnecessários
- [ ] Algoritmos eficientes
- [ ] Recursos gerenciados adequadamente

11. Red Flags Críticos

Bugs Sérios

- Divisão por zero não verificada
- Array/lista acessado sem verificar tamanho
- Recursos não fechados (memory leak)
- Race conditions em código concorrente
- Validação de segurança ausente

Code Smells Graves

- Função com >50 linhas
- Classe com >500 linhas
- > 5 parâmetros em função
- Aninhamento >4 níveis
- Código duplicado >3 vezes

Anti-patterns

- God Class (classe que faz tudo)

- God Method (método que faz tudo)
- Spaghetti Code (fluxo confuso)
- Magic Strings/Numbers sem constantes
- Copy-Paste Programming

12. Ações Imediatas

Para Bugs

1. Identifique o problema específico
2. Sugira correção concreta
3. Mencione casos de teste necessários

Para Code Smells

1. Aponte o smell específico
2. Sugira refatoração
3. Explique benefício da mudança

Para Violações de Princípios

1. Cite o princípio violado
2. Mostre como corrigir
3. Dê exemplo se necessário