Guia de Melhores Práticas de Implementação

Padrões de Engenharia e Qualidade de Código

Setor: Software Construction

Responsável Técnico: Gabriel de Freitas Villela
Contexto: Transição BIRD → Fábrica de Software

Dezembro de 2025

Sumário

1 Propósito e Filosofia **4**

2 Princípios Fundamentais (Clean Code) **4**
2.1 KISS (Keep It Simple, Stupid) . . . 4
2.1.1 O que é Simplicidade? . . . 4
2.1.2 Sinais de Alerta (Code Smells) . . . 4
2.1.3 Técnica Prática: Guard Clauses . . . 5
2.2 DRY (Don't Repeat Yourself) . . . 5
2.2.1 O Problema da Duplicação . . . 5
2.2.2 A "Regra de Três" (Rule of Three) . . . 6
2.2.3 Falsa Duplicação (Cuidado) . . . 6
2.2.4 Exemplo Prático: Centralização de Lógica . . . 6
2.3 SOLID Principles . . . 6
2.3.1 S - Single Responsibility Principle (SRP) . . . 6
2.3.2 O - Open/Closed Principle (OCP) . . . 7
2.3.3 L - Liskov Substitution Principle (LSP) . . . 7
2.3.4 I - Interface Segregation Principle (ISP) . . . 8
2.3.5 D - Dependency Inversion Principle (DIP) . . . 8

3 Convenções de Estilo e Nomenclatura **9**
3.1 Idioma do Código: Inglês . . . 9
3.2 Sintaxe: Tabela de Referência por Linguagem . . . 9
3.3 Semântica de Nomenclatura (Regras Universais) . . . 9
3.3.1 Funções são Ações (Verbos) . . . 9
3.3.2 Classes são Entidades (Substantivos) . . . 10
3.3.3 Variáveis Booleanas (Perguntas) . . . 10
3.4 Segurança de Tipos (Type Safety) . . . 10
3.5 Exemplo Prático: Refatoração e Clareza . . . 10

4 Ferramentas de Automação (Qualidade Contínua) **11**
4.1 Pilar 1: Formatter Automatizado . . . 11
4.2 Pilar 2: Analisador Estático (Linter) . . . 11
4.3 Pilar 3: Type Checker . . . 12

5 Documentação e Legibilidade **12**
5.1 Regra de Ouro . . . 12
5.2 Padrões de Docstrings (API) . . . 13
5.2.1 Estrutura Obrigatória . . . 13
5.2.2 Exemplo Prático (Python - Google Style) . . . 13
5.3 Comentários Internos (O "Porquê") . . . 14
5.4 Tags de Manutenção (Anotações) . . . 14

6 Tratamento de Erros e Observabilidade (Logs) **14**
6.1 "A Morte do `print`" . . . 14
6.2 Logs Estruturados (JSON) . . . 15
6.3 Níveis de Log (Padronização) . . . 15

6.4 Segurança no Log (Sanitização) . . . 15
6.5 Tratamento de Exceções (Exception Handling) . . . 15
6.5.1 Regra 1: Não engula exceções (Silent Failure) . . . 16
6.5.2 Regra 2: Envelopamento (Pattern de Camadas) . . . 16
6.5.3 Regra 3: Correlation ID (Rastreabilidade) . . . 16

7 Segurança na Implementação (AppSec) 16
7.1 Gerenciamento de Segredos (Credenciais) . . . 17
7.2 Blindagem contra Injeção (SQL Injection) . . . 17
7.3 Validação e Sanitização de Entrada . . . 17
7.4 Vazamento de Informação (Error Handling) . . . 17
7.5 5. Dependências Vulneráveis (Supply Chain) . . . 18

8 Integração e Fluxo de Trabalho 18
8.1 Fluxo de Entrada (Antes de Codificar) . . . 18
8.2 Fluxo de Apoio (Durante a Codificação) . . . 18
8.3 Fluxo de Saída (Entrega) . . . 19

9 Checklist de Code Review (Pull Request) 19
9.1 Padrões e Legibilidade . . . 19
9.2 Arquitetura e Design (SOLID/KISS) . . . 20
9.3 Segurança e Performance (Crítico) . . . 20
9.4 Operação e Observabilidade . . . 20
9.5 Testes . . . 20

10 Anexo Técnico: Setup do Ambiente de Desenvolvimento 20
10.1 Perfil A: Stack Python (Projetos de Backend / Scripts) . . . 20
10.1.1 Instalação . . . 21
10.1.2 Configuração (.pre-commit-config.yaml) . . . 21
10.2 Perfil B: Stack C# / .NET . . . 21
10.2.1 Instalação das Ferramentas . . . 21
10.2.2 Automação (Husky.Net ou Script) . . . 21
10.3 Perfil C: Stack Java . . . 22
10.3.1 Configuração no pom.xml (Maven) . . . 22
10.4 Integração com IDE (VS Code) . . . 22

11 Referências e Leitura Recomendada 23
11.1 Literatura Fundamental . . . 23
11.2 Guias de Estilo e Normas . . . 23
11.3 Segurança . . . 23

1 Propósito e Filosofia

Este documento, elaborado pela área responsável por **Implementação**, serve como a "Constituição Técnica" do time. O objetivo não é engessar a criatividade, mas garantir que o software produzido seja:

- **Legível:** O código é lido muito mais vezes do que é escrito.
- **Manutenível:** Fácil de corrigir e evoluir.
- **Testável:** Preparado para as validações de QA.

Regra de Ouro

"Sempre deixe o código mais limpo do que você o encontrou." (Boy Scout Rule)

2 Princípios Fundamentais (Clean Code)

Todo código desenvolvido na fábrica deve aderir aos seguintes princípios:

2.1 KISS (Keep It Simple, Stupid)

A complexidade é o inimigo da segurança e da manutenção. Evite super-engenharia. Se uma função faz "coisas demais", ela deve ser quebrada. O objetivo da fábrica não é produzir código "inteligente" que ninguém entende, mas sim código óbvio que funciona.

2.1.1 O que é Simplicidade?

Simplicidade não significa simplismo. Significa resolver o problema sem adicionar camadas desnecessárias de abstração ou "complexidade acidental".

- Se você precisa de um diagrama complexo para explicar uma única função de 20 linhas, ela viola o KISS.
- Se você está implementando uma estrutura genérica para "caso a gente precise no futuro", pare. (Ver princípio YAGNI - *You Ain't Gonna Need It*).

2.1.2 Sinais de Alerta (Code Smells)

O revisor deve rejeitar o código se encontrar:

- **Ninhada Profunda (Deep Nesting):** Muitos 'if' dentro de 'for' dentro de 'if'. Isso aumenta a carga cognitiva.
- **Funções Gigantes:** Funções com mais de 20-30 linhas geralmente fazem coisas demais.
- **Nomes Genéricos:** Variáveis chamadas 'data', 'info' ou 'manager' geralmente escondem complexidade mal definida.

2.1.3 Técnica Prática: Guard Clauses

Para aplicar o KISS e evitar a "seta de código" (código que cresce para a direita devido à indentação), utilize *Guard Clauses* (retorno antecipado).

```python
# VIOLACAO DO KISS (Complexo e aninhado)
def process_payment(order):
    if order:
        if order.status == 'OPEN':
            if order.balance > 0:
                order.pay()
                return True
            else:
                return False
        else:
            return False
    else:
        return False

# APLICANDO KISS (Simples e plano)
def process_payment(order):
    # Validacoes iniciais (Guard Clauses)
    if not order:
        return False
    if order.status != 'OPEN':
        return False
    if order.balance <= 0:
        return False

    # Execucao principal limpa
    order.pay()
    return True
```

Listing 1: Aplicando KISS com Guard Clauses

2.2 DRY (Don't Repeat Yourself)

O princípio DRY preconiza que "cada parte do conhecimento deve ter uma representação única, não ambígua e definitiva dentro do sistema". Não se trata apenas de economizar digitação, mas de garantir consistência.

2.2.1 O Problema da Duplicação

A duplicação é a maior causa de bugs de regressão (quando algo que funcionava para de funcionar).

- **Manutenção Pesadelo:** Se a regra de validação de CPF muda, e você tem essa validação espalhada em 3 telas diferentes, a chance de esquecer de atualizar uma delas é altíssima.
- **Inconsistência:** O usuário percebe o sistema como "quebrado" quando a API recusa um dado que o Front-end aceitou (lógicas duplicadas e divergentes).

2.2.2 A "Regra de Três" (Rule of Three)

Evite abstração prematura. Às vezes, criar uma função genérica cedo demais aumenta a complexidade (violando o KISS). Utilize a seguinte heurística:

1. **Primeira vez:** Escreva o código.
2. **Segunda vez:** Copie e cole (se necessário), mas fique alerta.
3. **Terceira vez: Pare.** Refatore para uma função, classe ou componente reutilizável.

2.2.3 Falsa Duplicação (Cuidado)

Nem tudo que parece igual é duplicado. Se dois trechos de código fazem a mesma coisa, mas por **motivos de negócio diferentes** (ex: validação de cadastro de cliente vs. validação de cadastro de fornecedor), eles podem evoluir de formas diferentes. Unificá-los forçadamente cria um acoplamento ruim.

2.2.4 Exemplo Prático: Centralização de Lógica

```
# VIOLACAO DO DRY (Logica repetida)
# File A (Report)
final_price = product.value * 1.15 # Taxa de 15% hardcoded
print(f"Total: {final_price}")

# File B (Checkout)
total_to_pay = cart.sum * 1.15 # A mesma taxa repetida
print(f"Total: {total_to_pay}")

# ------------------------------------------------------------

# APLICANDO DRY
# File: constants.py
SERVICE_TAX_RATE = 1.15

def calculate_price_with_tax(base_value):
    return base_value * SERVICE_TAX_RATE

# Uso no sistema
final_price = calculate_price_with_tax(product.value)
total_to_pay = calculate_price_with_tax(cart.sum)
```

Listing 2: Aplicando DRY (Single Source of Truth)

2.3 SOLID Principles

O acrônimo SOLID representa cinco princípios de design de classes orientados a objetos. O objetivo não é seguir regras cegamente, mas criar software que tolere mudanças.

2.3.1 S - Single Responsibility Principle (SRP)

"Uma classe deve ter um, e apenas um, motivo para mudar."

Se você tem uma classe chamada `PedidoManager` que: 1) Calcula o total, 2) Salva no banco e 3) Envia e-mail de confirmação, ela está errada. Se a regra de e-mail mudar, você corre o risco de quebrar o cálculo do pedido.

```python
# VIOLACAO (Classe "Deus" que faz tudo)
class Order:
    def calculate_total(self): ...
    def save_to_database(self): ... # Mistura persistencia
    def send_email_confirmation(self): ... # Mistura notificacao

# CORRETO (Cada um com sua responsabilidade)
class Order:
    def calculate_total(self): ... # Regra de negocio

class OrderRepository:
    def save(self, order): ... # Banco de dados

class EmailService:
    def send_confirmation(self, order): ... # Notificacao
```

Listing 3: Aplicando SRP

2.3.2 O - Open/Closed Principle (OCP)

"Entidades de software devem estar abertas para extensão, mas fechadas para modificação."

Você deve ser capaz de adicionar novas funcionalidades sem alterar o código fonte existente. Isso evita introduzir bugs em funcionalidades que já estão estáveis.

```python
# VIOLACAO (Muitos IFs)
class Discount:
    def calculate(self, type, value):
        if type == "VIP": return value * 0.8
        elif type == "BLACK_FRIDAY": return value * 0.5

# CORRETO (Uso de Interface/Heranca)
class DiscountRule(ABC):
    @abstractmethod
    def calculate(self, value): pass

class VipDiscount(DiscountRule):
    def calculate(self, value): return value * 0.8

class BlackFridayDiscount(DiscountRule):
    def calculate(self, value): return value * 0.5
```

Listing 4: Aplicando OCP com Polimorfismo

2.3.3 L - Liskov Substitution Principle (LSP)

"Subclasses devem ser substituíveis por suas classes base."

Se a classe `B` herda de `A`, o sistema deve funcionar usando `B` no lugar de `A` sem quebrar. O exemplo clássico é: um Pinguim é uma Ave, mas se a classe `Ave` tem um método `voar()`, o Pinguim não pode herdar dela (ou lançará um erro inesperado).

```python
# VIOLACAO
class Bird:
    def fly(self): ...

class Penguin(Bird):
    def fly(self):
```

```
        raise Exception("Penguins can't fly!") # Quebra o contrato!

# CORRETO
class Bird: ... # Classe base geral

class FlyingBird(Bird):
    def fly(self): ...

class Penguin(Bird): ... # Nao herda de FlyingBird
```

Listing 5: Respeitando a Substituicao de Liskov

2.3.4 I - Interface Segregation Principle (ISP)

"Muitas interfaces específicas são melhores do que uma interface única geral."

Não force uma classe a implementar métodos que ela não usa. Isso cria dependências fantasmas.

```
# VIOLACAO (Interface gorda)
class SmartDevice(ABC):
    def print(self): pass
    def scan(self): pass
    def fax(self): pass

class SimplePrinter(SmartDevice):
    def print(self): print("Printing...")
    def scan(self): pass # Forcado a implementar inutilmente
    def fax(self): pass  # Forcado a implementar inutilmente

# CORRETO
class Printer(ABC):
    def print(self): pass

class Scanner(ABC):
    def scan(self): pass

class SimplePrinter(Printer): ...
```

Listing 6: Segregacao de Interfaces

2.3.5 D - Dependency Inversion Principle (DIP)

"Dependa de abstrações, não de implementações."

Este é o ponto mais crucial para a **Qualidade** e **Testes**. Classes de alto nível (Regra de Negócio) não devem instanciar classes de baixo nível (Conexão MySQL) diretamente dentro delas. Elas devem receber a dependência "injetada".

```
# VIOLACAO (Alto acoplamento)
class ReportService:
    def __init__(self):
        # Preso ao MySQL para sempre. Dificil de testar.
        self.db = MySQLConnection()

# CORRETO (Injecao de Dependencia)
class ReportService:
    # Aceita QUALQUER coisa que siga o contrato "DatabaseInterface"
    def __init__(self, db: DatabaseInterface):
```

```
        self.db = db

# Production:
service = ReportService(MySQLConnection())
# Tests (Mock):
service = ReportService(MockDatabase())
```

Listing 7: Inversao de Dependencia

3 Convenções de Estilo e Nomenclatura

Embora a Fábrica de Software trabalhe com múltiplas tecnologias, a **legibilidade** é um princípio universal. Um código bem escrito deve ser autoexplicativo, independente se é Python, C# ou Java.

A responsabilidade de configurar as ferramentas de validação é da área de **Padrões**, mas a execução diária é dever de quem implementa.

3.1 Idioma do Código: Inglês

Para alinhar a Fábrica com padrões globais e facilitar a integração open-source, o idioma oficial do código (variáveis, funções, classes) será o Inglês.

Exceção (Domínio Específico): Termos de negócio estritamente brasileiros ou siglas da Algar devem ser mantidos no original para evitar perda de sentido (ex: `cpf`, `pix`, `bairro`).

3.2 Sintaxe: Tabela de Referência por Linguagem

Como cada linguagem tem sua "gramática" própria, respeite o padrão nativo da tecnologia:

Linguagem	Variáveis	Funções/Métodos	Classes
Python	`snake_case` *user_id*	`snake_case` *get_user()*	`PascalCase` *UserHandler*
Java / TS	`camelCase` *userId*	`camelCase` *getUser()*	`PascalCase` *UserHandler*
C#	`camelCase` *userId*	`PascalCase` *GetUser()*	`PascalCase` *UserHandler*

3.3 Semântica de Nomenclatura (Regras Universais)

Independente da linguagem, o **significado** do nome deve seguir estas regras:

3.3.1 Funções são Ações (Verbos)

O nome da função deve dizer o que ela faz. Se você precisa ler o código da função para entender o nome, refatore.

- Ruim: `pdf_report()` (Parece um objeto).
- Bom: `generate_pdf_report()` (Python) ou `GeneratePdfReport()` (C#).
- **Prefixos comuns:** `get`, `set`, `is`, `has`, `calc`, `validate`.

3.3.2 Classes são Entidades (Substantivos)

Classes representam o "sujeito" da ação.

- Ruim: `ManageUser` (Verbo).
- Bom: `UserManager` ou `UserRepository` (Substantivo).

3.3.3 Variáveis Booleanas (Perguntas)

Variáveis que guardam `True/False` devem soar como perguntas de sim ou não.

- Ruim: `open`, `valid`, `admin`.
- Bom: `is_open`, `is_valid`, `has_admin_permission`.

3.4 Segurança de Tipos (Type Safety)

Erros de tipo são a maior causa de bugs em produção.

- **Em C#/Java:** A tipagem é obrigatória pelo compilador. Use tipos explícitos em vez de `var` sempre que a leitura ficar ambígua.
- **Em Python:** O uso de *Type Hints* é **obrigatório** nas assinaturas de métodos públicos.

```python
from typing import List, Dict

# RUIM (O que e 'data'? O que retorna?)
def process(data):
    return data['val'] * 2

# BOM (Contrato claro)
def process_transaction(transaction_data: Dict[str, float]) -> float:
    """
    Receives transaction data and returns the final value.
    """
    return transaction_data.get('value', 0.0) * 2
```

Listing 8: Exemplo de Tipagem (Python Reference)

3.5 Exemplo Prático: Refatoração e Clareza

O exemplo abaixo está em Python, mas o conceito de **"Evitar Números Mágicos"** aplica-se a C#, Java e qualquer outra linguagem.

```python
# RUIM (Mistura de idiomas e numeros magicos)
# O que e 86400? Por que estamos multiplicando?
def converter_dias(lista):
    res = []
    for x in lista:
        res.append(x * 86400)
    return res

# ------------------------------------------------------------
```

```
10
11 # BOM (Ingles Tecnico, Constantes e Clareza)
12 SECONDS_IN_A_DAY = 86400
13
14 def convert_days_to_seconds(days_list: List[int]) -> List[int]:
15     seconds_list = []
16     for day in days_list:
17         seconds = day * SECONDS_IN_A_DAY
18         seconds_list.append(seconds)
19     return seconds_list
```

Listing 9: De Código Obscuro para Clean Code

4 Ferramentas de Automação (Qualidade Contínua)

Para garantir que a equipe produza código com padrão industrial e não artesanal, o uso de ferramentas de análise estática é **mandatório**.

O objetivo não é burocratizar, mas sim **automatizar o esforço operacional desnecessário**. O Code Review deve focar em lógica de negócio e arquitetura, e não em discussões sobre espaços, vírgulas ou indentação.

Nossa estratégia de automação se baseia em três pilares fundamentais, que devem ser aplicados em qualquer linguagem utilizada no projeto:

4.1 Pilar 1: Formatter Automatizado

Cada linguagem tem uma ferramenta que reescreve o código automaticamente para seguir o guia de estilo oficial.

- **O que faz:** Remove espaços extras, ajusta quebras de linha e padroniza a indentação ao salvar o arquivo.
- **Por que usar:** Elimina 100% das discussões subjetivas sobre estética. O código de um estagiário e de um sênior tornam-se visualmente idênticos.
- **Ferramentas Oficiais:**
  - **Python:** `Black` (Rigoroso, sem configuração).
  - **C#:** `dotnet format` (Nativo do SDK .NET).
  - **Java:** `Google Java Format` (Padrão de mercado).

4.2 Pilar 2: Analisador Estático (Linter)

Enquanto o formatador cuida da estética, o Linter cuida da "saúde" do código.

- **O que faz:** Analisa o código estaticamente em busca de:
  - Variáveis declaradas mas não usadas.
  - Funções complexas demais (violação do KISS).
  - Bugs lógicos óbvios (ex: `if (x == x)`).

- **Por que usar:** Impede que "code smells" (cheiro de código ruim) se acumulem, garantindo que a dívida técnica seja paga antes do commit.
- **Ferramentas Oficiais:**
  - **Python:** `Pylint` ou `Flake8`.
  - **C# / Java:** `SonarLint` (Plugin poderoso que roda direto na IDE).

4.3 Pilar 3: Type Checker

Erros de tipo são os bugs mais comuns e evitáveis em engenharia de software.

- **O que faz:** Garante que se uma função pede um `Número`, ela não receba um `Texto`.
- **Por que usar:** Em linguagens compiladas (C#/Java), isso é nativo, mas warnings não devem ser ignorados. Em Python, evita quebras em tempo de execução (Runtime Errors).
- **Ferramentas Oficiais:**
  - **Python:** `Mypy` (Verifica a consistência dos Type Hints).
  - **C# / Java:** O próprio Compilador (Configurado com *Treat Warnings as Errors*).

Regra de Ouro (Atenção)

Código que não passa nessas ferramentas **não deve ser aceito** no repositório. O **responsável por "Padrões"** deve configurar o pipeline (CI/CD ou Pre-commit) para rejeitar automaticamente qualquer entrega fora do padrão.

5 Documentação e Legibilidade

Código é lido muito mais vezes do que é escrito. A documentação não serve para explicar o que o código faz (o código já diz isso), mas sim para explicar 'como usar' (interface) e 'por que foi feito assim' (decisões).

5.1 Regra de Ouro

- **Código ruim não deve ser documentado, deve ser refatorado.** Não escreva comentários para explicar variáveis com nomes ruins como `x` ou `val`. Renomeie-as.
- **APIs Públicas:** Toda função, classe ou método que pode ser acessado por outro módulo **deve** ter documentação formal (Docstring).

5.2 Padrões de Docstrings (API)

Docstrings são a documentação que acompanha o código e permite a geração automática de manuais (via Sphinx, Swagger, Javadoc). A Fábrica adota os seguintes padrões de mercado:

Linguagem	Padrão Adotado	Ferramenta de Geração
Python	Google Style Docstrings	Sphinx / MkDocs
C#	XML Documentation	DocFX / Swagger
Java	Javadoc	Javadoc / Maven Site

5.2.1 Estrutura Obrigatória

Uma boa documentação de função deve responder a quatro perguntas, nesta ordem:

1. **Resumo:** O que isso faz? (Verbo no imperativo: "Calcula", "Busca", "Envia").
2. **Args (Parâmetros):** O que eu preciso passar? Qual o tipo? Existem restrições?
3. **Returns (Retorno):** O que sai de lá?
4. **Raises (Exceções):** O que pode dar errado? (Essencial para quem vai fazer o `try/catch`).

5.2.2 Exemplo Prático (Python - Google Style)

```python
# RUIM (Docstring preguicosa)
def calculate_churn(users):
    """Calcula o churn."""
    ...

# BOM (Padrao Google Style)
def calculate_churn_rate(active_users: int, lost_users: int) -> float:
    """
    Calculates the monthly churn rate based on user data.

    Args:
        active_users (int): Total number of users at the start of the
    period.
        lost_users (int): Number of users who cancelled the service.

    Returns:
        float: The churn rate as a percentage (0.0 to 100.0).

    Raises:
        ValueError: If active_users is zero or negative.
    """
    if active_users <= 0:
        raise ValueError("Active users must be greater than zero.")

    return (lost_users / active_users) * 100.0
```

Listing 10: Documentacao de API Profissional

5.3 Comentários Internos (O "Porquê")

Enquanto a Docstring é para quem *usa* a função, o comentário é para quem *mantém* a função. Use comentários para registrar dívidas técnicas e decisões de negócio não óbvias.

- **NÃO COMENTE O ÓBVIO:**

```
    i = i + 1  # Incrementa i (INUTIL - O codigo ja diz isso)

```

- **COMENTE A DECISÃO:**

```
    # Usamos uma query bruta (SQL) aqui em vez do ORM porque
    # a performance do ORM estava causando timeout em relatorios > 1GB.
    # Ver ticket JIRA-123.
    results = db.execute_raw_sql(...)

```

5.4 Tags de Manutenção (Anotações)

Em um ambiente colaborativo, use tags padronizadas para sinalizar pendências no código. A maioria das IDEs mapeia isso automaticamente.

- `TODO`: Algo que precisa ser feito, mas não bloqueia a entrega atual.
- `FIXME`: Um código que funciona, mas é "gambiarra" e precisa de correção urgente.
- `DEPRECATED`: Funcionalidade antiga que será removida na próxima versão.
- `NOTE`: Um aviso importante sobre o comportamento do bloco.

```
def validate_cpf(cpf: str) -> bool:
    # TODO: Implementar validacao completa com digito verificador.
    # Atualmente valida apenas o tamanho para nao travar o MVP.
    return len(cpf) == 11
```

Listing 11: Uso de Tags

6 Tratamento de Erros e Observabilidade (Logs)

Esta disciplina é a ponte entre o Desenvolvimento e a Operação. Um sistema sem logs adequados é uma "caixa preta" cara de manter.

Não logamos apenas para "debugar", logamos para 'monitorar a saúde' do negócio.

6.1 "A Morte do `print`"

O uso de `print()` (Python) ou `System.out.println` (Java) é "proibido" em código de produção.

- **Por quê?** Prints não possuem 'timestamp', não possuem nível de severidade (ERROR vs INFO) e, em muitas linguagens, bloqueiam a thread principal (I/O blocking), degradando a performance.
- **Solução:** Use sempre a instância de Logger configurada pelo framework (Log4j, Serilog, Python Logging).

6.2 Logs Estruturados (JSON)

Em vez de frases soltas, nossos logs devem ser objetos estruturados. Isso permite que ferramentas (ELK Stack, Datadog, CloudWatch) indexem os campos.

```
# RUIM (Texto Plano - Dificil de filtrar)
logger.info(f"Usuario {user_id} comprou o item {item_id}")

# BOM (Estruturado - Facil de criar dashboards)
# O log sai como um JSON: {"event": "purchase", "user_id": 123, "item": 99}
logger.info("Purchase completed", extra={
    "event": "purchase_success",
    "user_id": user_id,
    "item_id": item_id,
    "amount": 50.00
})
```

Listing 12: Texto vs Logs Estruturados

6.3 Níveis de Log (Padronização)

O uso incorreto dos níveis gera alertas falsos ou silêncio perigoso.

Nível	Quando usar?
DEBUG	Informações granulares para desenvolvimento. **Desligado em Produção**. (Ex: Payload completo de uma requisição).
INFO	Eventos de negócio bem sucedidos. (Ex: "Pedido criado", "Job de sincronização finalizado").
WARNING	Algo inesperado aconteceu, mas o sistema se recuperou. Não requer acordar ninguém de madrugada. (Ex: "Tentativa de login falhou", "API demorou mas respondeu").
ERROR	Uma operação falhou. O usuário percebeu o erro. Requer investigação futura. (Ex: "Falha ao salvar no banco", "NullPointerException").
CRITICAL	O sistema (ou uma parte vital dele) parou. Requer atuação imediata da Operação. (Ex: "Banco de dados fora do ar").

6.4 Segurança no Log (Sanitização)

Risco Crítico (LGPD)

Nunca, sob hipótese alguma, logue Dados Pessoais Sensíveis (PII), Senhas, Tokens ou Chaves de API.

- Ruim: `logger.info(f"User login: {password}")`
- Bom: `logger.info(f"User login attempt for: {username}")`

6.5 Tratamento de Exceções (Exception Handling)

Tratar erros não é apenas evitar que o programa feche ("crash"), é garantir que o sistema falhe de forma segura e informativa.

6.5.1 Regra 1: Não engula exceções (Silent Failure)

O `catch` vazio é o maior inimigo da manutenção. Se você capturou um erro, você tem três opções:

1. **Logar e lançar:** Registra e deixa o erro subir.
2. **Recuperar:** Aplica uma lógica de correção (ex: tenta de novo).
3. **Envelopar:** Transforma uma exceção técnica em uma exceção de negócio.

6.5.2 Regra 2: Envelopamento (Pattern de Camadas)

Não exponha erros de banco de dados (SQL Injection risk) para o usuário final/frontend.

```
try:
    user = db.find_user(user_id)
except DatabaseConnectionError as original_error:
    # 1. Logamos o erro tecnico (para o responsavel pela area de
   Operacao ver no servidor)
    logger.error("DB connection failed", exc_info=original_error)

    # 2. Lancamos um erro limpo de negocio (para o Frontend receber)
    # O usuario recebe "Servico indisponivel", nao "Error 500 at line
   40..."
    raise ServiceUnavailableError("User service is temporarily down.")
```

Listing 13: Envelopamento de Excecao (Python)

6.5.3 Regra 3: Correlation ID (Rastreabilidade)

Em sistemas distribuídos (como o Open Gateway), um erro pode ocorrer em um serviço profundo. Todo log deve conter um `correlation_id` (gerado na entrada da requisição) que é repassado para todas as funções internas.

```
def process_payment(order_id, correlation_id):
    try:
        payment_gateway.charge(order_id)
    except Exception as e:
        # O responsavel por Operacao consegue pesquisar pelo ID e ver
   todo o rastro
        logger.error("Payment failed", extra={
            "correlation_id": correlation_id,
            "order_id": order_id,
            "error": str(e)
        })
        raise
```

Listing 14: Exemplo com Correlation ID

7 Segurança na Implementação (AppSec)

Segurança não é responsabilidade exclusiva da área de "Segurança do Software". A vulnerabilidade nasce no momento em que o código é digitado. Adotamos a filosofia *Shift Left*: pensar em segurança desde a primeira linha de código.

7.1 Gerenciamento de Segredos (Credenciais)

Crime Capital

NUNCA, sob hipótese alguma, comite senhas, tokens, chaves de API ou strings de conexão no Git. O histórico do Git é eterno.

- **Problema:** `API_KEY = "12345"` no código.
- **Solução:** Use Variáveis de Ambiente (`.env`).
- **Ferramenta:** Em Python, use `python-dotenv`. Em C#, use `appsettings.json` (com User Secrets) ou Key Vault.

7.2 Blindagem contra Injeção (SQL Injection)

A falha mais antiga e comum. Ocorre quando você concatena strings para formar uma query de banco de dados.

Regra: Jamais concatene input de usuário diretamente em comandos SQL ou de Sistema Operacional. Use *Parameterized Queries* (Prepared Statements).

```
# VULNERAVEL (Concatenacao de String)
# Se o usuario enviar: " ' OR '1'='1 "
# Ele apaga ou le todo o seu banco.
query = f"SELECT * FROM users WHERE name = '{user_input}'"
cursor.execute(query)

# SEGURO (Query Parametrizada)
# O banco trata o input estritamente como dado, nao como comando.
query = "SELECT * FROM users WHERE name = %s"
cursor.execute(query, (user_input,))
```

Listing 15: SQL Injection: O Jeito Errado vs Certo

7.3 Validação e Sanitização de Entrada

Adote o princípio de **Zero Trust**. Todo dado que vem de fora (Frontend, API externa, Arquivo) é potencialmente malicioso.

- **Validação de Tipo:** Se o campo é idade, aceite apenas inteiros. Recuse strings.
- **Allow-list (Lista Branca):** Em vez de tentar bloquear caracteres ruins (o que é difícil), aceite apenas os bons.
  - *Exemplo:* Para um campo "UF", aceite apenas `[A-Z]{2}`. Qualquer outra coisa é rejeitada.

7.4 Vazamento de Informação (Error Handling)

Erros detalhados são úteis para o desenvolvedor, mas são mapas do tesouro para atacantes.

- **Stack Trace:** Nunca mostre o "caminho das pedras" (ex: `Line 40 in /var/www/auth.py: ConnectionRefused`). Isso revela sua estrutura de pastas e tecnologia.

- **Mensagens Genéricas:**
  - Ruim: "A senha para o usuário 'admin' está incorreta." (Revela que o usuário 'admin' existe).
  - Bom: "Usuário ou senha inválidos."

7.5 5. Dependências Vulneráveis (Supply Chain)

Bibliotecas modernas facilitam a vida, mas podem conter falhas. Não use versões antigas.

- O responsável pela área de Segurança do Software pode rodar scanners, mas o desenvolvedor deve estar atento aos alertas do GitHub/GitLab (Dependabot) e atualizar os pacotes (`pip`, `npm`, `nuget`) regularmente.

8 Integração e Fluxo de Trabalho

A área de Implementação atua como o motor da fábrica, transformando definições em produto real. Para isso, atua no centro de um fluxo de comunicação constante:

8.1 Fluxo de Entrada (Antes de Codificar)

Nesta etapa, o objetivo é garantir que o problema foi bem compreendido antes de gastar horas programando.

- **Engenharia de Requisitos:** O código deve resolver o problema de negócio descrito no ERS.
  - *Atenção:* Não confie cegamente apenas nos diagramas técnicos. Se o diagrama parecer contradizer a regra de negócio do ERS, consulte o responsável pela área imediatamente. A regra de negócio sempre tem precedência sobre o desenho técnico.
- **Projeto e Modelagem:**
  - **Viabilidade:** Se a arquitetura proposta ou o diagrama de classes for inviável de implementar no prazo estipulado, é dever do Implementador levantar a mão ("Pushback").
  - **Fidelidade:** O código deve refletir os diagramas. Se você precisou mudar a estrutura da classe durante o código, o diagrama precisa ser atualizado. *Código e Documentação devem andar juntos.*

8.2 Fluxo de Apoio (Durante a Codificação)

Você não está codando sozinho. Use os especialistas para blindar seu código.

- **Segurança:** Adote a postura de *Shift Left*. Não espere o código estar pronto para perguntar se ele é seguro.
  - *Exemplo:* Perguntando ao responsável por Segurança do Código - "Vou usar essa lib para gerar PDF, ela tem alguma vulnerabilidade conhecida?"

- **Padrões:** Se o Linter ou o Pipeline estiverem travando seu commit injustamente, acione o responsável para ajustar as regras de automação. Não tente burlar as regras locais.

8.3 Fluxo de Saída (Entrega)

A implementação só termina quando o próximo da fila consegue trabalhar.

- **QA e Entrega:**
  - **Smoke Test:** Nunca entregue código que "nem builda". Antes de passar para QA, rode o caminho feliz (happy path) na sua máquina.
  - **Testes Unitários:** O código deve ir para QA com a cobertura mínima de testes unitários definida no projeto. QA foca em testes integrados e de sistema, não deveria perder tempo pegando erro de sintaxe.
- **Operação:**
  - **"Na minha máquina funciona":** Essa frase é proibida. Garanta que todas as dependências novas estejam no `requirements.txt` ou `Dockerfile`.
  - **Variáveis de Ambiente:** Se você criou uma nova chave ou configuração, avise o responsável da Operação para que ele possa configurá-la no ambiente de Homologação/Produção.

9 Checklist de Code Review (Pull Request)

O Code Review é a última linha de defesa antes de um bug ou vulnerabilidade chegar à produção. O revisor não deve aprovar o PR se qualquer um dos itens abaixo não for atendido.

9.1 Padrões e Legibilidade

Idioma: O código (variáveis, funções) está 100% em Inglês? (Exceto termos de domínio local).

Clean Code: Nomes de variáveis e funções revelam claramente a intenção?

Documentação: Funções públicas possuem *Docstrings* no padrão definido (Args, Returns, Raises)?

Sujeira: Código comentado, prints de debug e imports não usados foram removidos?

Automação: O código passou no pipeline de Linter, Formatter e Type Checker sem erros?

9.2 Arquitetura e Design (SOLID/KISS)

KISS: Existem funções complexas demais que poderiam ser quebradas? (Ninhada de `if/else`).

DRY: Existe lógica de negócio duplicada que deveria virar uma função auxiliar?

Responsabilidade: A classe/função faz apenas uma coisa? (Princípio SRP).

Fidelidade: A implementação reflete os diagramas e arquitetura desenhados pelo time de Projeto?

9.3 Segurança e Performance (Crítico)

Segredos: GARANTIA de que não há senhas, tokens ou chaves *hardcoded*?

Injeção: Queries SQL estão parametrizadas (sem concatenação de string)?

Validação: Inputs externos são validados e sanitizados antes do processamento?

Loops: Existe algum loop (`for/while`) perigoso que pode travar com grandes volumes de dados?

9.4 Operação e Observabilidade

Logs: Os logs estão estruturados (JSON)? O nível (INFO/ERROR) está correto?

LGPD: Garantia de que nenhum dado sensível (PII) ou senha está sendo logado?

Tratamento de Erro: As exceções são tratadas ou envelopadas corretamente (sem `try/catch` vazios)?

9.5 Testes

Cobertura: Existem testes unitários cobrindo o Happy Path - "Caminho Feliz" - e as principais falhas?

Independência: Os testes rodam isolados (Mock) sem depender de banco de dados real?

10 Anexo Técnico: Setup do Ambiente de Desenvolvimento

Para garantir a padronização, utilizamos automação de *git hooks*. Abaixo estão as instruções de configuração separadas por stack tecnológica.

10.1 Perfil A: Stack Python (Projetos de Backend / Scripts)

Este perfil utiliza o framework nativo `pre-commit` e é o padrão para projetos de ciência de dados e APIs em Python.

10.1.1 Instalação

O arquivo `requirements-dev.txt` deve conter: `black`, `mypy`, `pylint`, `pre-commit`.

```
# No terminal (ambiente virtual ativo):
pip install -r requirements-dev.txt
pre-commit install
```

Listing 16: Setup Python

10.1.2 Configuração (.pre-commit-config.yaml)

```
repos:
  - repo: https://github.com/psf/black
    rev: 23.9.1
    hooks:
      - id: black
        language_version: python3

  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.5.1
    hooks:
      - id: mypy
        additional_dependencies: [types-requests]

  - repo: local
    hooks:
      - id: pylint
        name: pylint
        entry: pylint
        language: system
        types: [python]
        args: ["-rn", "-sn"]
```

Listing 17: Configuração Padrão Python

10.2 Perfil B: Stack C# / .NET

Para projetos .NET, utilizamos a ferramenta oficial `dotnet format` combinada com hooks locais.

10.2.1 Instalação das Ferramentas

```
# Instala o formatador globalmente ou localmente no projeto
dotnet tool install -g dotnet-format
```

Listing 18: Setup C

10.2.2 Automação (Husky.Net ou Script)

Recomendamos o uso do pacote **Husky.Net** para gerenciar os commits.

```
dotnet new tool-manifest
dotnet tool install Husky
dotnet husky install
```

Adicione a tarefa no arquivo `task-runner.json` gerado pelo Husky:

```
{
   "tasks": [
      {
         "name": "dotnet-format",
         "command": "dotnet",
         "args": ["format", "--verify-no-changes"],
         "group": "pre-commit"
      }
   ]
}
```

Listing 19: Tarefa do Husky para C

10.3 Perfil C: Stack Java

Para Java, a validação é feita via plugins do Maven/Gradle.

10.3.1 Configuração no pom.xml (Maven)

Adicione o plugin **Spotless** (Formatação) e **Checkstyle** (Lint) no `pom.xml`:

```
<plugin>
  <groupId>com.diffplug.spotless</groupId>
  <artifactId>spotless-maven-plugin</artifactId>
  <version>2.40.0</version>
  <configuration>
    <java>
      <googleJavaFormat />
    </java>
  </configuration>
</plugin>
```

Listing 20: Exemplo Spotless Maven

10.4 Integração com IDE (VS Code)

Para feedback visual em tempo real, instale as extensões conforme sua linguagem:

- **Python:**
  - Extensão: *Black Formatter* (Microsoft)
  - Extensão: *Mypy Type Checker*
- **C# / .NET:**
  - Extensão: *C# Dev Kit*
  - Extensão: *SonarLint*
- **Java:**
  - Extensão: *Extension Pack for Java*
  - Extensão: *Checkstyle for Java*

11 Referências e Leitura Recomendada

As práticas descritas neste manual baseiam-se em literatura técnica consagrada e padrões de mercado. A leitura das obras abaixo é encorajada para o aprimoramento técnico da equipe.

11.1 Literatura Fundamental

- **Clean Code: A Handbook of Agile Software Craftsmanship** – Robert C. Martin. (Base para os princípios SOLID e Nomenclatura).
- **The Pragmatic Programmer** – Andrew Hunt & David Thomas. (Base para os princípios DRY e ETC).
- **Refactoring** – Martin Fowler. (Técnicas para melhorar código legado sem alterar comportamento).

11.2 Guias de Estilo e Normas

- **PEP 8** – Style Guide for Python Code. Disponível em: `https://peps.python.org/pep-0008/`
- **Google Java Style Guide**. Disponível em: `https://google.github.io/styleguide/javaguide.html`
- **C# Coding Conventions (Microsoft)**. Disponível em: `https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions`

11.3 Segurança

- **OWASP Top 10** – Padrão global de conscientização sobre segurança de aplicações web.
- **SWEBOK v4** – Guide to the Software Engineering Body of Knowledge (IEEE Computer Society).