

# Análise de Antipadrões no Projeto

---

"Dia & Noite: O Mundo dos Animais" - Activity Provider

Arquitetura e Padrões de Software

Universidade Aberta

**Autor:**

Fábio Amado (2501444)

## **1. Introdução**

Este documento apresenta uma análise crítica dos antipadrões identificados no projeto "Dia & Noite: O Mundo dos Animais", um web service educativo desenvolvido para a plataforma Inven!RA. O projeto implementa três padrões de design (Factory Method, Decorator e Strategy), mas também apresenta algumas ocorrências de antipadrões que podem ser melhoradas.

Os antipadrões são soluções recorrentes a problemas comuns que inicialmente parecem benéficas, mas que eventualmente produzem consequências negativas mais significativas do que os benefícios.

## **2. Metodologia de Análise**

A análise foi conduzida através de:

- Revisão sistemática do código-fonte de todos os módulos
- Identificação de padrões recorrentes que violam princípios SOLID
- Comparação com catálogo de antipadrões conhecidos
- Análise de impacto na manutenibilidade e extensibilidade

### 3. Antipadrão Identificado #1: Blob / God Object

#### 3.1. Descrição do Antipadrão

O antipadrão "Blob" (também conhecido como "God Object") ocorre quando uma única classe concentra a maioria das funcionalidades do sistema, enquanto outras classes têm responsabilidades mínimas. Esta classe acumula muitas responsabilidades não relacionadas, violando o princípio da Responsabilidade Única (Single Responsibility Principle - SRP).

#### Características do Blob:

- Classe excessivamente grande com muitos métodos
- Múltiplas responsabilidades não relacionadas
- Alta complexidade ciclomática
- Dificuldade em testar isoladamente
- Forte acoplamento com outras partes do sistema

#### 3.2. Situação Inicial: Classe SessionAnalytics

A classe SessionAnalytics no arquivo session\_module/session\_analytics.py apresenta características claras do antipadrão Blob:

**Arquivo:** session\_module/session\_analytics.py

**Classe:** SessionAnalytics

**Linhas:** 25-464

Estrutura da classe:

```
class SessionAnalytics:  
    """Sistema de Analytics de Sessão"""\n\n    def __init__(self):  
        # Gerencia 3 estruturas de dados diferentes  
        self.sessions: Dict[str, Dict] = {}  
        self.user_sessions: Dict[str, List[str]] = {}  
        self.user_stats: Dict[str, Dict] = {}  
        self.score_calculator = ScoreCalculator(...)  
  
        # RESPONSABILIDADE 1: Gestão de Sessões  
    def start_session(self, user_id: str, ...) -> str: ...  
    def end_session(self, session_id: str) -> Dict: ...  
  
        # RESPONSABILIDADE 2: Tracking de Desafios  
    def log_challenge_start(self, session_id: str, ...) -> None: ...  
    def log_challenge_complete(self, session_id: str, ...) -> Dict: ...  
  
        # RESPONSABILIDADE 3: Tracking de Interações  
    def log_interaction(self, session_id: str, ...) -> None: ...  
  
        # RESPONSABILIDADE 4: Cálculo de Estatísticas  
    def get_session_summary(self, session_id: str) -> Dict: ...  
    def get_user_sessions_report(self, user_id: str) -> Dict: ...
```

```
# RESPONSABILIDADE 5: Gestão de Dias Consecutivos
def _update_consecutive_days(self, user_id: str) -> None: ...

# RESPONSABILIDADE 6: Exportação de Dados
def export_analytics(self, user_id: str) -> Dict: ...
```

### 3.3. Por que é um Antipadrão?

A classe SessionAnalytics viola múltiplos princípios de design:

Princípio Violado	Como é Violado
<b>Single Responsibility Principle (SRP)</b>	A classe tem pelo menos 6 responsabilidades distintas não relacionadas
<b>Open/Closed Principle</b>	Adicionar nova funcionalidade requer modificar esta classe monolítica
<b>Interface Segregation</b>	Clientes que só precisam de uma funcionalidade são forçados a depender de toda a classe
<b>Testabilidade</b>	Difícil testar isoladamente cada responsabilidade. Testes exigem mock de múltiplos componentes
<b>Manutenibilidade</b>	Mudanças em uma funcionalidade arriscam quebrar outras. 464 linhas difíceis de navegar
<b>Coesão</b>	Métodos operam em estruturas de dados diferentes (sessions, user_sessions, user_stats) sem forte relação lógica

### 3.4. Evidências Concretas

#### 1. Múltiplas Estruturas de Dados:

A classe gerencia 3 dicionários diferentes sem clara separação de responsabilidades:

```
self.sessions: Dict[str, Dict] = {}          # Dados de sessões ativas
self.user_sessions: Dict[str, List[str]] = {}  # Mapeamento user ->
                                                sessões
self.user_stats: Dict[str, Dict] = {}          # Estatísticas agregadas
```

#### 2. Métodos com Propósitos Diferentes:

start\_session() vs export\_analytics() vs \_update\_consecutive\_days() fazem coisas completamente diferentes

#### 3. Alto Acoplamento:

A classe depende de Challenge, ScoreCalculator, múltiplas estratégias, datetime, e coordena toda a lógica de negócio

#### 4. Métrica de Complexidade:

- 464 linhas de código
- 13 métodos públicos
- 3 estruturas de dados principais
- Responsável por 6 áreas funcionais distintas



### 3.5. Consequências Negativas

O antipadrão Blob na classe SessionAnalytics resulta em:

#### Dificuldade de Manutenção:

- Desenvolvedores precisam entender 464 linhas para fazer qualquer mudança
- Alto risco de efeitos colaterais ao modificar código

#### Problemas de Teste:

- Impossível testar funcionalidades isoladamente
- Necessário mock de múltiplas dependências
- Testes tornam-se frágeis e acoplados

#### Dificuldade de Extensão:

- Adicionar nova métrica requer modificar classe monolítica
- Risco de quebrar funcionalidades existentes

#### Violação de SRP:

- Classe tem múltiplas razões para mudar
- Mudanças em estatísticas afetam tracking de sessões
- Mudanças em exportação afetam cálculos internos

## 3.6. Refatoração Proposta

Para eliminar o antipadrão Blob, propõe-se decompor a classe SessionAnalytics em classes especializadas, cada uma com uma responsabilidade única:

```
# Classe 1: Gerenciamento de Sessões
class SessionManager:
    """Responsável apenas por criar/encerrar sessões"""
    def start_session(self, user_id: str) -> str: ...
    def end_session(self, session_id: str) -> SessionSummary: ...
    def get_session(self, session_id: str) -> Session: ...

# Classe 2: Tracking de Eventos
class EventTracker:
    """Responsável por registrar eventos e interações"""
    def track_challenge_start(self, session: Session, challenge: Challenge): ...
    def track_challenge_complete(self, session: Session, result: ChallengeResult): ...
    def track_interaction(self, session: Session, event: Event): ...

# Classe 3: Cálculo de Estatísticas
class StatisticsCalculator:
    """Responsável por agregar e calcular estatísticas"""
    def calculate_session_stats(self, session: Session) -> SessionStats: ...
    def calculate_user_stats(self, user_id: str) -> UserStats: ...
    def calculate_average_time(self, challenges: List[Challenge]) -> float: ...

# Classe 4: Gestão de Streaks
class StreakManager:
```

```

"""Responsável por rastrear dias consecutivos"""
def update_consecutive_days(self, user_id: str) -> int: ...
def get_streak(self, user_id: str) -> int: ...
def reset_streak(self, user_id: str): ...

# Classe 5: Exportador de Analytics
class AnalyticsExporter:
    """Responsável por formatar dados para exportação"""
    def export_to_invenira(self, user_stats: UserStats) -> Dict: ...
    def export_to_json(self, session: Session) -> str: ...
    def export_to_csv(self, sessions: List[Session]) -> str: ...

# Facade para coordenar (se necessário)
class SessionAnalyticsFacade:
    """Coordena as classes especializadas"""
    def __init__(self):
        self.session_manager = SessionManager()
        self.event_tracker = EventTracker()
        self.stats_calculator = StatisticsCalculator()
        self.streak_manager = StreakManager()
        self.exporter = AnalyticsExporter()

    def start_session(self, user_id: str) -> str:
        session_id = self.session_manager.start_session(user_id)
        self.streak_manager.update_consecutive_days(user_id)
        return session_id

```

### **Benefícios da Refatoração:**

- [+] Single Responsibility: Cada classe tem uma responsabilidade clara
- [+] Testabilidade: Classes podem ser testadas isoladamente
- [+] Manutenibilidade: Mudanças localizadas em classes específicas
- [+] Extensibilidade: Fácil adicionar novo exportador ou calculadora
- [+] Coesão: Métodos relacionados agrupados logicamente
- [+] Baixo Acoplamento: Classes dependem apenas do necessário

## 4. Antipadrão Identificado #2: Cut-and-Paste Programming

### 4.1. Descrição do Antipadrão

Cut-and-Paste Programming é o antipadrão onde código é copiado e colado com pequenas modificações em vez de criar abstrações reutilizáveis. Resulta em duplicação de código que dificulta manutenção e aumenta probabilidade de bugs.

### 4.2. Situação Inicial: Validação de Endpoints

No arquivo session\_endpoints.py, a lógica de validação de entrada é repetida em múltiplos endpoints com estrutura quase idêntica:

Arquivo: session\_module/session\_endpoints.py

```
# ENDPOINT 1: start_session (linhas 42-48)
@app.route("/api/session/start", methods=['POST'])
def start_session():
    data = request.get_json()

    if not data or 'user_id' not in data:
        return jsonify({
            'success': False,
            'error': 'user_id é obrigatório'
        }), 400

# ENDPOINT 2: session_challenge (linhas 88-95)
@app.route("/api/session/challenge", methods=['POST'])
def session_challenge():
    data = request.get_json()

    required = ['session_id', 'animal_id']
    if not all(field in data for field in required):
        return jsonify({
            'success': False,
            'error': f'Campos obrigatórios: {required}'
        }), 400

# ENDPOINT 3: complete_challenge (linhas 154-161)
@app.route("/api/session/complete-challenge", methods=['POST'])
def complete_challenge():
    data = request.get_json()

    required = ['session_id', 'challenge_id', 'is_correct']
    if not all(field in data for field in required):
        return jsonify({
            'success': False,
            'error': f'Campos obrigatórios: {required}'
        }), 400

# ENDPOINT 4: log_interaction (linhas 206-211)
@app.route("/api/session/interaction", methods=['POST'])
def log_interaction():
    data = request.get_json()

    if not data or 'session_id' not in data or 'event_type' not in data:
```

```
        return jsonify({
            'success': False,
            'error': 'session_id e event_type são obrigatórios'
        }), 400

# ENDPOINT 5: end_session (linhas 245-251)
@app.route("/api/session/end", methods=['POST'])
def end_session():
    data = request.get_json()

    if not data or 'session_id' not in data:
        return jsonify({
            'success': False,
            'error': 'session_id é obrigatório'
        }), 400
```

### **4.3. Por que é um Antipadrão?**

A duplicação de código de validação apresenta vários problemas:

#### **1. Violação do Princípio DRY (Don't Repeat Yourself):**

- O mesmo padrão de validação é repetido 7+ vezes
- Estrutura try-except idêntica em todos os endpoints

#### **2. Dificuldade de Manutenção:**

- Alterar formato de erro requer mudança em 7 locais
- Adicionar logging requer modificar todos os endpoints

#### **3. Inconsistência:**

- Endpoint 1 valida de forma diferente do Endpoint 2
- Mensagens de erro não são uniformes

#### **4. Propensão a Bugs:**

- Corrigir bug em um local não propaga para outros
- Desenvolvedor pode esquecer de atualizar todos os locais

### **4.4. Evidências de Duplicação**

Análise quantitativa da duplicação:

Padrão Duplicado	Ocorrências	Linhas Totais
<code>data = request.get_json()</code>	7 endpoints	~7 linhas
<code>if not data or 'field' not in data:</code>	5 endpoints	~25 linhas
<code>return jsonify({'success': False, 'error': ...})</code>	7 endpoints	~35 linhas
<code>try-except Exception as e</code>	7 endpoints	~70 linhas

Total estimado: ~137 linhas de código duplicado (37% do arquivo)

## 4.5. Refatoração Proposta

Eliminar duplicação através de decorators e funções auxiliares:

```
# Solução 1: Decorator para validação
from functools import wraps

def validate_request(*required_fields):
    """Decorator para validar campos obrigatórios"""
    def decorator(f):
        @wraps(f)
        def wrapper(*args, **kwargs):
            data = request.get_json()

            # Validação centralizada
            if not data:
                return jsonify({
                    'success': False,
                    'error': 'Corpo da requisição vazio'
                }), 400

            # Validar campos obrigatórios
            missing = [field for field in required_fields
                       if field not in data]

            if missing:
                return jsonify({
                    'success': False,
                    'error': f'Campos obrigatórios faltando: {missing}'
                }), 400

            # Passar para função original
            return f(data, *args, **kwargs)

        return wrapper
    return decorator

# Solução 2: Error handler centralizado
@app.errorhandler(Exception)
def handle_error(error):
    """Handler global de erros"""
    app.logger.error(f'Erro: {str(error)}')
    return jsonify({
        'success': False,
        'error': str(error)
    }), 500

# USO: Endpoints simplificados
@app.route("/api/session/start", methods=['POST'])
@validate_request('user_id')  # Validação automática!
def start_session(data):
    """Agora recebe data já validado"""
    session_id = session_analytics.start_session(
        data['user_id'],
        data.get('session_id')
    )

    return jsonify({
        'success': True,
        'session_id': session_id,
        'message': 'Sessão iniciada com sucesso'
    })
```

```

    })

@app.route("/api/session/challenge", methods=['POST'])
@validate_request('session_id', 'animal_id') # Lista de campos!
def session_challenge(data):
    """Validação automática de múltiplos campos"""
    challenge_type = data.get('challenge_type', 'random')

    if challenge_type == 'random':
        challenge = ChallengeFactory.create_random_challenge(
            data['animal_id']
        )
    else:
        challenge = ChallengeFactory.create_challenge(
            challenge_type,
            data['animal_id']
        )

    session_analytics.log_challenge_start(
        data['session_id'],
        challenge
    )

    return jsonify({
        'success': True,
        'challenge': challenge.to_dict()
    })

@app.route("/api/session/complete-challenge", methods=['POST'])
@validate_request('session_id', 'challenge_id', 'is_correct')
def complete_challenge(data):
    """Código limpo focado na lógica de negócio"""
    session_analytics.log_challenge_complete(
        data['session_id'],
        data['challenge_id'],
        data['is_correct']
    )

    return jsonify({
        'success': True,
        'message': 'Desafio concluído'
    })

```

### **Benefícios da Refatoração:**

- [+] DRY: Validação centralizada em um único local
- [+] Consistência: Todas as validações seguem mesmo padrão
- [+] Manutenibilidade: Mudança em validação afeta todos os endpoints
- [+] Legibilidade: Endpoints focam na lógica de negócio
- [+] Testabilidade: Validação pode ser testada isoladamente
- [+] Redução de Código: ~137 linhas reduzidas para ~40 linhas

## 5. Antipadrão Identificado #3: Input Kludge

### 5.1. Descrição do Antipadrão

Input Kludge é o antipadrão onde a validação e sanitização de entrada são inadequadas ou inconsistentes. Dados não são validados apropriadamente quanto a tipo, formato, ou range, resultando em comportamento imprevisível e potenciais vulnerabilidades de segurança.

### 5.2. Situação Inicial: Validação Superficial

Os endpoints validam apenas a presença de campos, mas não validam tipos, formatos, ou valores válidos:

```
# Validação ATUAL (inadequada)
@app.route("/api/session/challenge", methods=['POST'])
def session_challenge():
    data = request.get_json()

    # Apenas verifica se campos existem
    required = ['session_id', 'animal_id']
    if not all(field in data for field in required):
        return jsonify({'success': False, 'error': '...'}), 400

    # NÃO VALIDA:
    # - animal_id é inteiro?
    # - animal_id existe na base de dados?
    # - session_id está no formato correto?
    # - session_id corresponde a sessão ativa?
    # - challenge_type é válido ('audio', 'visual', etc)?

    # Usa dados sem validação adicional
    challenge = ChallengeFactory.create_challenge(
        data.get('challenge_type', 'random'), # Pode ser qualquer
        string!
        data['animal_id'] # Pode ser string, float, None, etc!
    )
```

### 5.3. Problemas Identificados

A validação inadequada resulta em múltiplos problemas:

Campo	Validação Atual	Problema
animal_id	Apenas presença	Aceita string, float, None. Sem verificação se existe
session_id	Apenas presença	Não valida formato ou se sessão está ativa
challenge_type	Nenhuma	Aceita qualquer string. Não valida se tipo existe
difficulty	Nenhuma	Pode ser negativo, string, ou > 5
is_correct	Apenas presença	Não valida se é booleano (pode ser string)
event_type	Apenas presença	Não valida contra lista de eventos válidos

## 5.4. Exemplos de Inputs Problemáticos

Exemplos de requests que passariam na validação atual mas causariam erros:

```
# Exemplo 1: animal_id inválido (string em vez de int)
{
    "session_id": "valid_session",
    "animal_id": "gato", # String! Deveria ser int
    "challenge_type": "audio"
}
# Resultado: Erro não tratado ao tentar criar desafio

# Exemplo 2: animal_id não existe
{
    "session_id": "valid_session",
    "animal_id": 999, # ID inexistente
    "challenge_type": "visual"
}
# Resultado: ValueError não capturado

# Exemplo 3: challenge_type inválido
{
    "session_id": "valid_session",
    "animal_id": 1,
    "challenge_type": "xyz" # Tipo inexistente
}
# Resultado: ValueError "Tipo de desafio inválido: xyz"

# Exemplo 4: difficulty fora do range
{
    "session_id": "valid_session",
    "animal_id": 1,
    "challenge_type": "audio",
    "difficulty": 100 # Deveria ser 1-5
}
# Resultado: Comportamento indefinido

# Exemplo 5: session_id de sessão inativa
{
    "session_id": "expired_session",
    "animal_id": 1
}
# Resultado: Dados registados em sessão encerrada

# Exemplo 6: is_correct como string
{
    "session_id": "valid_session",
    "challenge_id": "audio_1_1234",
    "is_correct": "true" # String! Deveria ser boolean
}
# Resultado: Sempre True (string não vazia)
```

## 5.5. Refatoração Proposta

Implementar validação robusta usando schemas de validação:

```
from typing import Any, Dict, List, Optional
from dataclasses import dataclass
from enum import Enum

# 1. Definir tipos válidos
```

```

class ChallengeType(Enum):
    AUDIO = 'audio'
    VISUAL = 'visual'
    HABITAT = 'habitat'
    CLASSIFICATION = 'classification'
    RANDOM = 'random'

class EventType(Enum):
    CLICK_HINT = 'click_hint'
    CLICK_AUDIO = 'click_audio'
    HOVER_OPTION = 'hover_option'
    # ... outros tipos

# 2. Schemas de validação
@dataclass
class SessionStartRequest:
    user_id: str
    session_id: Optional[str] = None

    def validate(self):
        if not self.user_id or not isinstance(self.user_id, str):
            raise ValueError("user_id deve ser string não vazia")

        if len(self.user_id) > 100:
            raise ValueError("user_id muito longo (max 100 caracteres)")

@dataclass
class ChallengeRequest:
    session_id: str
    animal_id: int
    challenge_type: str = 'random'
    difficulty: int = 1

    def validate(self):
        # Validar session_id
        if not isinstance(self.session_id, str):
            raise ValueError("session_id deve ser string")

        # Validar animal_id
        if not isinstance(self.animal_id, int):
            raise ValueError("animal_id deve ser inteiro")

        if self.animal_id < 1:
            raise ValueError("animal_id deve ser positivo")

        # Validar se animal existe
        from data.animals_data import ANIMALS_DB
        if not any(a['id'] == self.animal_id for a in ANIMALS_DB):
            raise ValueError(f"Animal {self.animal_id} não existe")

        # Validar challenge_type
        try:
            ChallengeType(self.challenge_type)
        except ValueError:
            valid_types = [t.value for t in ChallengeType]
            raise ValueError(
                f"challenge_type '{self.challenge_type}' inválido. "
                f"Tipos válidos: {valid_types}"
            )

    # Validar difficulty

```

```

        if not isinstance(self.difficulty, int):
            raise ValueError("difficulty deve ser inteiro")

        if not 1 <= self.difficulty <= 5:
            raise ValueError("difficulty deve estar entre 1 e 5")

        # Validar se sessão existe e está ativa
        if self.session_id not in session_analytics.sessions:
            raise ValueError(f"Sessão {self.session_id} não existe")

        if not session_analytics.sessions[self.session_id]['active']:
            raise ValueError(f"Sessão {self.session_id} já está
encerrada")

@dataclass
class ChallengeCompleteRequest:
    session_id: str
    challenge_id: str
    is_correct: bool

    def validate(self):
        # Validar tipos
        if not isinstance(self.session_id, str):
            raise ValueError("session_id deve ser string")

        if not isinstance(self.challenge_id, str):
            raise ValueError("challenge_id deve ser string")

        if not isinstance(self.is_correct, bool):
            raise ValueError("is_correct deve ser booleano
(true/false)")

        # Validar se sessão existe
        if self.session_id not in session_analytics.sessions:
            raise ValueError(f"Sessão {self.session_id} não existe")

        # Validar se desafio foi iniciado nesta sessão
        session = session_analytics.sessions[self.session_id]
        challenge_started = any(
            i['type'] == 'challenge_start' and
            i['challenge_id'] == self.challenge_id
            for i in session['interactions']
        )

        if not challenge_started:
            raise ValueError(
                f"Desafio {self.challenge_id} não foi iniciado nesta
sessão"
            )

    # 3. Helper para criar schema a partir de request
    def parse_and_validate(schema_class, data: Dict[str, Any]):
        """Cria e valida schema"""
        try:
            schema = schema_class(**data)
            schema.validate()
            return schema
        except TypeError as e:
            raise ValueError(f"Campos inválidos: {str(e)}")

    # 4. Usar nos endpoints

```

```

@app.route("/api/session/challenge", methods=['POST'])
def session_challenge():
    data = request.get_json()

    if not data:
        return jsonify({'success': False, 'error': 'Corpo vazio'}), 400

    try:
        # Validação robusta!
        req = parse_and_validate(ChallengeRequest, data)

        # Agora dados estão validados e tipados
        if req.challenge_type == 'random':
            challenge = ChallengeFactory.create_random_challenge(
                req.animal_id,
                req.difficulty
            )
        else:
            challenge = ChallengeFactory.create_challenge(
                req.challenge_type,
                req.animal_id,
                req.difficulty
            )

        session_analytics.log_challenge_start(req.session_id, challenge)

        return jsonify({
            'success': True,
            'challenge': challenge.to_dict()
        })
    except ValueError as e:
        return jsonify({
            'success': False,
            'error': str(e)
        }), 400

```

### **Benefícios da Refatoração:**

- [+] Segurança: Validação rigorosa previne inputs maliciosos
- [+] Robustez: Erros são capturados antes de causar problemas
- [+] Documentação: Schemas servem como documentação viva
- [+] Debugging: Erros claros indicam exatamente o problema
- [+] Manutenibilidade: Validação centralizada em schemas
- [+] Consistência: Mesmas regras aplicadas uniformemente

## 6. Conclusão e Recomendações

Este documento identificou três antipadrões principais no projeto "Dia & Noite: O Mundo dos Animais":

1. **Blob / God Object** - Classe SessionAnalytics com múltiplas responsabilidades
2. **Cut-and-Paste Programming** - Duplicação de validação em endpoints
3. **Input Kludge** - Validação inadequada de entradas

Embora o projeto implemente corretamente três padrões de design (Factory Method, Decorator e Strategy), a presença destes antipadrões impacta negativamente a qualidade do código em termos de manutenibilidade, testabilidade e robustez.

### 6.1. Priorização das Refatorações

Recomenda-se implementar as refatorações na seguinte ordem:

#### Prioridade ALTA - Input Kludge:

- Impacto em segurança e robustez
- Relativamente simples de implementar
- Benefícios imediatos

#### Prioridade MÉDIA - Cut-and-Paste Programming:

- Melhora manutenibilidade
- Reduz código duplicado
- Pode ser feito incrementalmente

#### Prioridade BAIXA - Blob/God Object:

- Refatoração mais complexa
- Requer reestruturação significativa
- Mas traz maior melhoria arquitetural a longo prazo

### 6.2. Lições Aprendidas

1. Mesmo com uso correto de padrões de design, antipadrões podem emergir se princípios SOLID não forem seguidos rigorosamente.
2. Refatoração contínua é essencial. O código tende a degradar-se sem manutenção ativa.
3. Revisões de código devem identificar duplicação e responsabilidades excessivas antes que se tornem problemas maiores.
4. Validação de entrada deve ser tratada como requisito de segurança, não como detalhe de implementação.