1. Princípios Fundamentais do Código Limpo

1.1 Legibilidade

- **Regra:** O código deve ser fácil de ler e entender para outros desenvolvedores.
- **Como aplicar:** Use nomes descritivos para variáveis, métodos e classes.
    - Exemplos:
        - Em vez de `var x = 10;`, use `var maxUserAttempts = 10;`.
        - Métodos como `calculateInterest()` são mais claros do que `calcInt()`.

1.2 Simplicidade

- **Regra:** Escreva apenas o que é necessário para resolver o problema.
- **Como aplicar:**
    - Evite lógica desnecessária ou funcionalidades que ainda não foram solicitadas (princípio YAGNI – *You Aren't Gonna Need It*).
    - Refatore código redundante para reduzir complexidade.

1.3 Coesão

- **Regra:** Cada classe ou módulo deve ter uma única responsabilidade clara (Princípio de Responsabilidade Única - SRP).
- **Como aplicar:**
    - Divida grandes classes em classes menores e específicas.
    - Use nomes para representar a responsabilidade de forma explícita, como `InvoiceService` em vez de `Helper`.

1.4 Acoplamento Baixo

- **Regra:** Mantenha dependências entre módulos mínimas e bem definidas.
- **Como aplicar:**
    - Use injeção de dependência e abstrações (interfaces).
    - Evite dependências circulares e módulos que "sabem demais".

1.5 Testabilidade

- **Regra:** Código limpo é fácil de testar.
- **Como aplicar:**
    - Escreva código modular com responsabilidades claras.

- Use mocks e stubs para isolar dependências ao testar.

2. Estrutura do Código

2.1 Organização

- **Regra:** Estruture seu código com um padrão lógico.
- **Como aplicar:**
  - Use pastas para separar responsabilidades (ex.: `controllers/`, `services/`, `repositories/`).
  - Siga convenções de nomenclatura da linguagem.

2.2 Tamanho das Funções

- **Regra:** Mantenha funções curtas (idealmente menos de 20 linhas).
- **Como aplicar:**
  - Cada função deve realizar apenas uma tarefa específica.
  - Extraia trechos complexos para funções auxiliares.

2.3 Comentários

- **Regra:** Comente apenas quando necessário, mas escreva código que se explique sozinho.
- **Como aplicar:**
  - Substitua comentários com nomes melhores ou refatoração.
  - Exemplo:

```
php
Copiar código
// Calcula o total de um pedido
public function calculateOrderTotal() { ... }
```

Pode ser substituído por:

```
php
Copiar código
public function calculateOrderTotal() { ... }
```

3. Aplicação dos Conceitos

3.1 Refatoração

- **Quando aplicar:** Sempre que o código parecer confuso, repetitivo ou difícil de testar.
- **Ferramentas:** Use ferramentas de análise estática e cobertura de testes.

3.2 Abstração e Encapsulamento

- **Quando aplicar:** Para isolar detalhes internos e expor apenas o necessário.
- **Como aplicar:**
  - Use interfaces e classes abstratas.
  - Esconda detalhes de implementação com modificadores de acesso (`private`, `protected`).

3.3 Tratamento de Erros

- **Quando aplicar:** Em operações suscetíveis a falhas.
- **Como aplicar:**
  - Use exceções para cenários excepcionais.
  - Evite o uso de exceções para controle de fluxo normal.
  - Sempre forneça mensagens de erro significativas.

4. Princípios de Design

4.1 Princípios SOLID

- **S** - **Single Responsibility Principle (SRP):** Uma classe deve ter apenas um motivo para mudar.
- **O** - **Open/Closed Principle (OCP):** Classes devem estar abertas para extensão, mas fechadas para modificação.
- **L** - **Liskov Substitution Principle (LSP):** Subtipos devem ser substituíveis por seus tipos base.
- **I** - **Interface Segregation Principle (ISP):** Não force classes a implementar interfaces que elas não usam.
- **D** - **Dependency Inversion Principle (DIP):** Dependa de abstrações, não de implementações.

4.2 Design Patterns

- **Quando aplicar:** Para resolver problemas recorrentes de forma eficiente.
- **Padrões comuns:**
    - **Factory:** Criar objetos sem expor a lógica de criação.
    - **Singleton:** Garantir uma única instância de uma classe.
    - **Observer:** Atualizar partes do sistema quando uma mudança ocorre.

5. Ferramentas e Boas Práticas

5.1 Ferramentas

- **Linters:** Para identificar problemas de estilo e bugs (ex.: PHP_CodeSniffer, ESLint).
- **Formatadores:** Automatize o formato de código (ex.: Prettier).
- **Análise estática:** Identifique vulnerabilidades (ex.: SonarQube).

5.2 Revisão de Código

- Sempre faça revisões de código em equipe para garantir qualidade.
- Discuta melhorias em problemas identificados.

5.3 Automação de Testes

- Utilize frameworks de teste unitário, como PHPUnit, para validar funcionalidades.

6. Exemplos Práticos

Antes: Código confuso e acoplado

```
php
Copiar código
function process($data) {
    if ($data['type'] == 'email') {
        // lógica para email
    } elseif ($data['type'] == 'sms') {
        // lógica para sms
```

```
    }
    // mais condições...
}
```

Depois: Código limpo e coeso

```
php
Copiar código
interface Notification {
    public function send($data): void;
}

class EmailNotification implements Notification {
    public function send($data): void {
        // lógica para email
    }
}

class SmsNotification implements Notification {
    public function send($data): void {
        // lógica para SMS
    }
}

class NotificationFactory {
    public static function create($type): Notification {
        return match ($type) {
            'email' => new EmailNotification(),
            'sms' => new SmsNotification(),
            default => throw new InvalidArgumentException('Tipo
inválido.'),
        };
    }
}

// Uso
$notification = NotificationFactory::create($data['type']);
$notification->send($data);
```