# RELATÓRIO DE REESTRUTURAÇÃO DO CÓDIGO COM BASE NOS PRINCÍPIOS SOLID

## Visão Geral

Este relatório apresenta as refatorações realizadas em quatro exemplos distintos, cada um violando um dos princípios SOLID (exceto o D). Foram corrigidos exemplos relacionados ao SRP (S), OCP (O), LSP (L) e ISP (I), de acordo com os conceitos ensinados em sala de aula. Abaixo, detalhamos o problema encontrado em cada exemplo, a solução aplicada e o resultado final.

## 1. SSOLID - Single Responsibility Principle

❌ Situação Original:  
A classe `ProcessadorEncomendas` realizava múltiplas tarefas:  
- Entrada de dados (Scanner)  
- Cálculo do valor do frete  
- Escrita em arquivo (persistência)

🔥 Problema:  
Violação do Princípio da Responsabilidade Única. A classe tinha mais de uma razão para mudar, o que fere a coesão e dificulta manutenção.

✅ Refatoração Aplicada:  
- Criada a classe `Encomenda` para encapsular os dados e lógica de cálculo.  
- Criada a classe `ArquivoEncomendaDAO` para responsabilidade de persistência.  
- A classe `ProcessadorEncomendas` ficou responsável apenas por coordenar o processo.

🎯 Resultado:  
Cada classe agora tem uma responsabilidade única e bem definida, facilitando testes, manutenção e extensão.

## 2. OSOLID - Open/Closed Principle

❌ Situação Original:  
A classe `SistemaPagamento` utilizava condicionais (`if/else`) para cada tipo de pagamento ("CARTAO", "PIX", "BOLETO").

🔥 Problema:  
Toda vez que um novo método de pagamento era adicionado, a classe precisava ser modificada, violando o OCP.

✅ Refatoração Aplicada:  
- Criada a interface `PagamentoStrategy`.  
- Criadas as classes `PagamentoCartao`, `PagamentoPix` e `PagamentoBoleto`, implementando `PagamentoStrategy`.  
- `SistemaPagamento` agora depende de abstração e não mais de condicionais.

🎯 Resultado:  
O sistema agora está aberto para extensão, mas fechado para modificação. Novos métodos de pagamento podem ser adicionados sem alterar o código existente.

## 3. LSOLID - Liskov Substitution Principle

❌ Situação Original:  
Exemplo 1:  
A classe `Quadrado` herdava de `Retangulo` e sobrescrevia os métodos `setAltura` e `setLargura` de forma a alterar o comportamento esperado, tornando substituições perigosas.

Exemplo 2:  
`ContaPoupanca` herdava de `ContaBancaria`, mas sobrescrevia o método `sacar()` lançando exceção — comportamento incompatível com a superclasse.

🔥 Problema:  
Em ambos os casos, subclasses violavam o comportamento esperado das superclasses, impossibilitando o uso polimórfico sem efeitos colaterais.

✅ Refatoração Aplicada:  
- Criada uma interface `FormaGeometrica` e removida a herança direta entre `Retangulo` e `Quadrado`.  
- Criada uma interface `Conta` e duas implementações: `ContaComum` (com saque) e `ContaPoupanca` (sem saque, mas com rendimento).

🎯 Resultado:  
Agora, substituições entre tipos são seguras e consistentes, respeitando o LSP.

## 4. ISOLID - Interface Segregation Principle

❌ Situação Original:  
A interface `Veiculo` tinha métodos `dirigir()`, `voar()` e `navegar()`, e a classe `Carro` era forçada a implementar métodos que não utilizava, lançando exceções.

🔥 Problema:  
Violação do ISP — classes eram forçadas a depender de métodos que não usavam.

✅ Refatoração Aplicada:  
- Interface `Veiculo` foi segregada em três interfaces: `Terrestre`, `Aereo` e `Aquatico`.  
- A classe `Carro` agora implementa apenas `Terrestre`.

🎯 Resultado:  
As classes agora dependem apenas dos métodos que realmente usam, promovendo coesão e independência.

## Conclusão

Cada exemplo foi reestruturado para refletir fielmente o princípio SOLID correspondente. As novas implementações aumentam a legibilidade, manutenibilidade e extensibilidade do sistema, tornando-o mais robusto e profissional.