Introdução ao SOLID

O que esperar do curso?

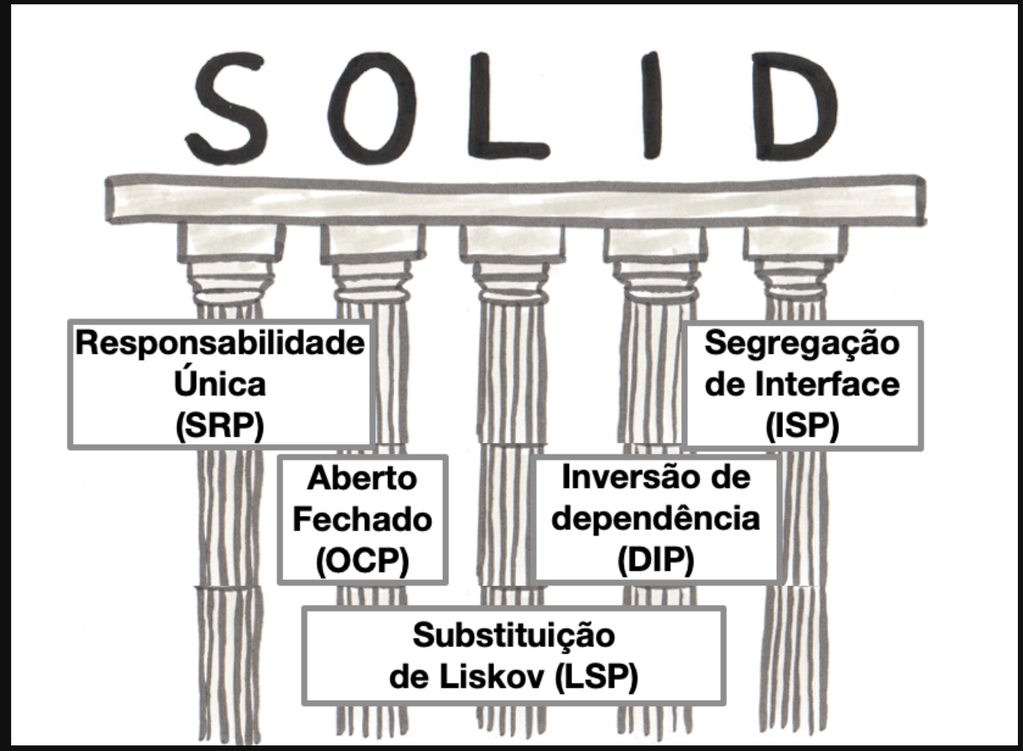
- **Visão geral:**
  - O que é SOLID?
  - Estudo de cada princípio com aplicações práticas
- **Método de ensino:**
  - Suporte téorico através dos slides
  - Repositório git com todos os exemplos de código

O que é SOLID?

- **Definição:** é um acrônimo que representa cinco princípios fundamentais de design de software, que buscam criar sistemas compreensíveis, flexíveis e manuteníveis (sustentáveis).
- Estes princípios foram organizados e publicado por Robert C. Martin (Uncle Bob), estes princípios são de extrema relevância em programação orientada à objetos (poo)

Por que preciso estudar isto?

- Código mais **manutenível**
- ~~Facilidade?~~ Flexibilidade para **estender** e **evoluir** o sistema
- Maior **reusabilidade** de código
- Código mais **testável**
- Redução de **acoplamento** aferente/eferente indevido entre componentes



SRP

Single Responsability Principle

Princípio da Responsabilidade Única

SRP

- **Conceito:** Uma classe (um componente), deve ter uma, e apenas uma, razão para mudar.

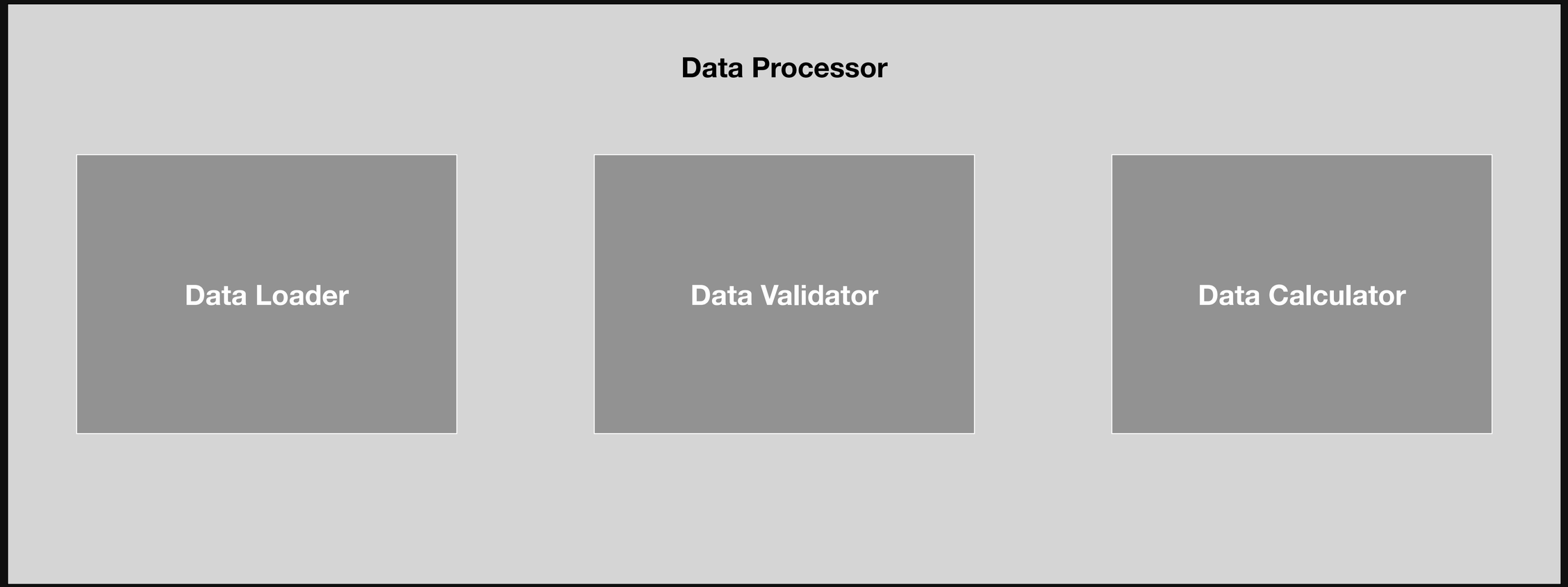
SRP - Exemplo 1

- **Exemplo teórico:** Imagine uma classe ***User*** que além da representação de um usuário também é responsável por salvar informações no banco de dados e enviar um e-mail.
- **Violação:** Explícita! Dificuldade de manutenção e alto grau de acomplamento

SRP - Exemplo 2

- **Exemplo teórico:** Imagine uma classe ***DataProcessor*** que possui 3 métodos: ***Load, Validate*** *e* ***CalculateStatistics.*** Todos estes métodos fazem parte do grupo de funcionalidades de “Processamento de dados”.
  - Load: Carrega as informações de um arquivo
  - Validate: Valida se os dados estão no formato correto e atendem as regras de negócio
  - CalculateStatistics: Calcula estatísticas com base nos dados importados
- **Violação:** Implícita! Dificuldade de manutenção e alto grau de acomplamento

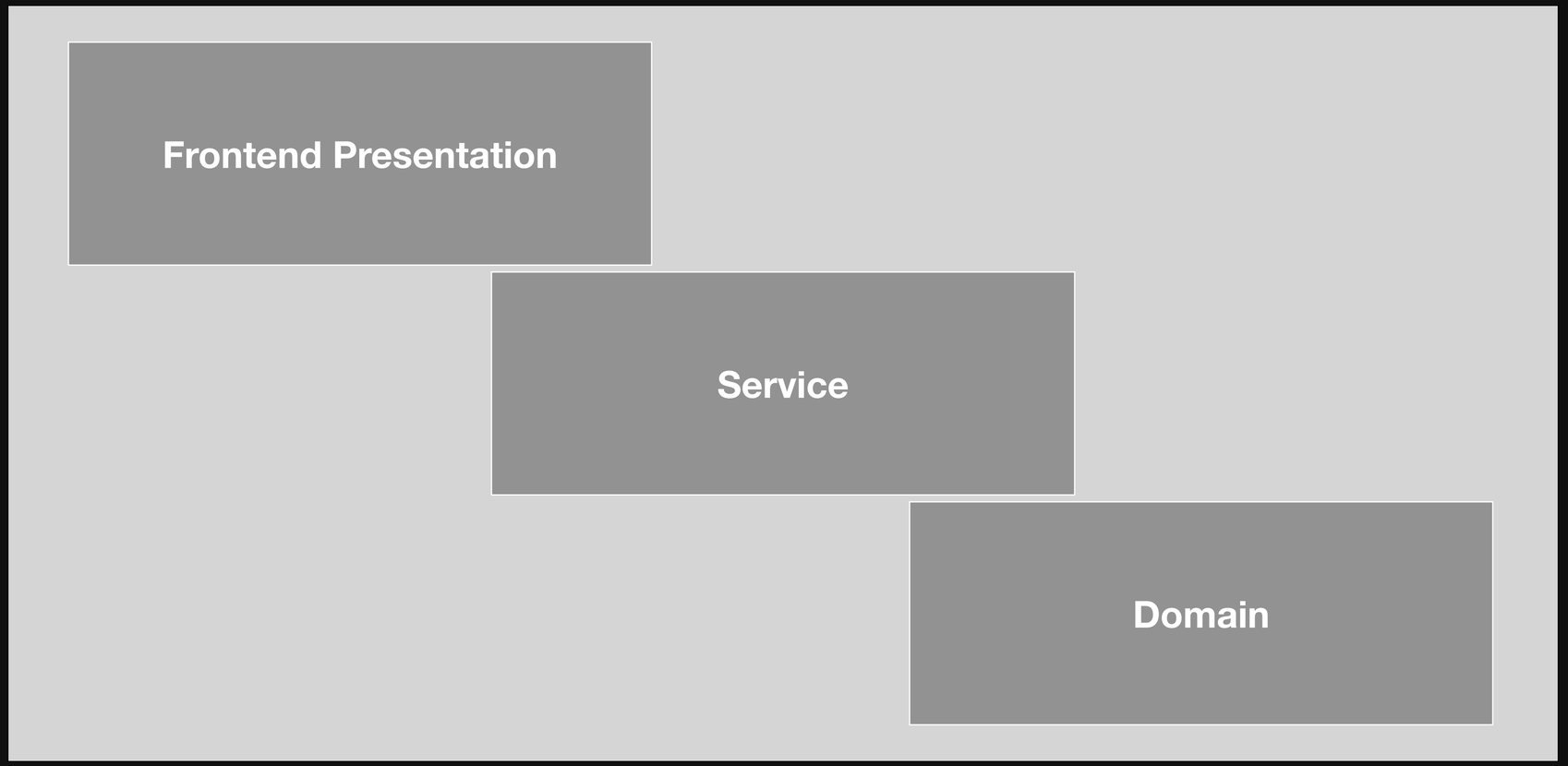
SRP - Exemplo 2



SRP – Exemplo 3

- **Exemplo teórico:** Imagine uma classe ***GetPurchaseOrderService*** que tem a função de buscar dados de um Pedido de compra.
  - Contexto: A aplicação já está funcionando! Recebemos a solicitação de acrescentarmos o nome do cliente para ser exibido no frontend.
- **Violação:** Implícita! Dificuldade de manutenção e alto grau de acomplamento

SRP – Exemplo 3



SRP - Benefícios

- **Manutenção:** Facilidade em atualizar ou modificar funcionalidades específicas.
- **Testabilidade:** Aumento na facilidade de testar cada responsabilidade de forma isolada.

OCP

Open/Closed Principle

Princípio Aberto-Fechado

OCP

- **Conceito:** As entidades de software (classes, módulos, funções) devem estar abertas para extensão, mas fechadas para modificação.

OCP - Exemplo 1

- **Exemplo teórico:** Imagine um sistema RH que calcula o salário de funcionários. Temos as seguintes classes:
  - **Employee**: Representa um funcionário
  - **EmployeeSalaryCalculator**: Faz o cálculo de salário baseado no tipo de funcionário
- **Violação:** Explícita! A cada novo tipo de remuneração, precisamos modificar o código da calculadora de salário

OCP - Exemplo 2

- **Exemplo teórico:** Imagine a geração de relatório de custos por departamento. O relatório pode ser impresso ou gerado em arquivo CSV (Comma Separated Values).
- **Violação:** Explícita! A cada novo tipo de saída, precisamos modificar o código da geração do relatório

OCP - Exemplo 3

- **Exemplo teórico:** Imagine um sistema que possui notificações e o usuário pode decidir em quais canais deseja receber a notificação.
- **Violação:** Explícita! A cada novo canal de notificação, precisamos modificar o código.

OCP - Benefícios

- **Extensibilidade:** Permite adicionar novas funcionalidades de forma rápida e segura.
- **Redução de Bugs:** Minimize o risco de introduzir erros ao não modificar o código existente.

LSP

Liskov (Barbara) Substitution Principle

Princípio de Substituição de Liskov

LSP

- **Conceito:** Se **S** é um subtipo de **T**, então objetos do tipo **T** podem ser substituídos por objetos do tipo **S** sem alterar as propriedades desejáveis do programa.

LSP

- **Conceito:** Objetos de uma superclasse devem poder ser substituídos por objetos de uma subclasse sem alterar o comportamento esperado do programa.

LSP - Exemplo 1

- **Exemplo clássico:** Imagine um software de desenho (Paint). Temos as seguintes classes:
  - **Rectangle**: Representa um retângulo
  - **Square**: Herda da classe Rectangle
- **Violação:** Explícita! A subclasse não respeita o comportamento definido pela superclasse

LSP - Exemplo 2

- **Exemplo:** Imagine um software que lide com diversos tipos de documentos. Temos as seguintes classes:
  - **MyDocument**: Representação abstrata de um documento
  - **Contract**: Herda da classe MyDocument
  - **Report**: Herda da classe MyDocument
- **Violação:** Explícita! A subclasse não respeita o comportamento definido pela superclasse

LSP - Exemplo 3

- **Exemplo:** Nossa aplicação precisa fazer requisições HTTP usando diferentes libs. Temos as seguintes classes:
  - **FetchHttpClient**: Client de lib open source
  - **CustomHttpClient**: Construção própria
- **Violação:** Explícita! A subclasse não respeita o comportamento definido pela interface

Identificar violação LSP

1. **Testes de substituição:**
    - Verifique se objetos da subclasse podem ser usados no lugar da superclasse sem alterar o comportamento esperado.
2. **Verifique sobrescritas de métodos:**
    - Se uma subclasse sobrescreve métodos da superclasse de forma que altera o comportamento esperado, isso pode indicar uma violação do LSP.
3. **Analise pré-condições e pós-condições:**
    - As pré-condições (requisitos para executar um método) não devem ser mais restritivas na subclasse.
    - As pós-condições (resultados garantidos após a execução) não devem ser mais fracas na subclasse.

Identificar violação LSP

4. **Verifique invariantes de classe:**

    - As propriedades que sempre devem ser verdadeiras para uma classe (invariantes), não devem ser quebradas pela subclasse.

5. **Use ferramentas de análise estática:**

    - Ferramentas como linters e analisadores de código podem ajudar a identificar possíveis violações do LSP.

LSP – Benefícios

- **Consistência:** Garante que a substituição de classes não altera o comportamento do sistema.
- **Reutilização:** Facilita o uso de componentes em diferentes contextos sem ajustes adicionais.

ISP

Interface Segregation Principle

Princípio de Segregação de Interface

ISP

- **Conceito:** Os clientes não devem ser forçados a depender de interfaces que não usam.

ISP

- **Traduzindo:** Interfaces menores, mais coesas e específicas. Classes implementam somente o que realmente é necessário.

ISP - Exemplo 1

- **Exemplo clássico:** Software de produção que controla etapas do processo dos trabalhadores. Temos as seguintes classes:
  - **WorkerEmployee**: Interface "inchada"
  - **HumanWorker**: Classe que representa um colaborador humano
  - **RobotWorker**: Classe que representa um colaborador robô
- **Violação:** Explícita! Obriga que todos os colaboradores implemente a interface, mesmo que não faça sentido!

ISP - Exemplo 2

- **Exemplo:** Imagine um software que lide com diversos tipos de documentos. Temos as seguintes classes:
  - **OurDocument**: Interface “inchada" define as ações de um documento
  - **SimpleTextDocument**: Documento simples
  - **CollaborativeDocument**: Simula um documento editado de forma colaborativa
- **Violação:** Explícita! Obriga que todos os documentos implemente a interface, mesmo que um documento não possua aquela característica!

ISP - Exemplo 3

- **Exemplo:** Nossa aplicação precisa integrar-se com diferentes tipos de gateway de pagamento:
  - **PaymentGateway**: Interface "inchada"
  - **SimplePaymentGateway**: Gateway simples! Não possui todas as funções
  - **AdvancedPaymentGateway**: Gateway robusto, completo
- **Violação:** Explícita! A interface “inchada” obriga à implementação de funcionalidades que o gateway não possui!

ISP - Benefícios

- **Flexibilidade:** Permite que classes implementem apenas o que realmente precisam.
- **Granularidade:** Agrupamento de funcionalidades em interfaces específicas
- **Simplicidade:** Reduz a complexidade de implementação, erros de utilização e melhora a legibilidade do código.

DIP

Dependency Inversion Principle

Princípio da Inversão de Dependência

DIP

Conceitos:

1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
2. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

DIP

Conceitos:

1. Módulos de **alto nível** não devem depender de módulos de **baixo nível**. Ambos devem depender de abstrações.
2. Abstrações não devem depender de **detalhes**. **Detalhes** devem depender de abstrações.

DIP

- **Alto nível:** a lógica de negócio (serviços, casos de uso).
- **Baixo nível:** infraestrutura/tecnologia (bancos, frameworks, protocolos de comunicação, SDKs de terceiros, drivers).
- **Abstrações:** interfaces/contratos (em TypeScript, interface ou tipos estruturais).
- **Detalhes:** Implementação concreta de uma abstração.

DIP

- **Traduzindo:** As funcionalidades e regras de negócios do software não devem depender das decisões de implementação e/ou tecnologias utilizadas.

DIP - Exemplo 1

- **Exemplo clássico:** Um registro de usuário em uma aplicação web. Cenário muito comum!
  - **UserService**: Representa nossa regra de negócio! Registrar um um novo usuário e notificar via e-mail.
  - **EmailService**: Classe com implementação concreta para envio de um e-mail.
- **Violação:** Explícita! Acoplamento muito forte! UserService está 100% dependente de EmailService e conhece detalhes da implementação.

DIP - Exemplo 2

- **Exemplo:** Imagine um software vendas! Gestão de pedidos.
  - **OrderService**: Camada de alto nível; Possui regras de negócios! Casos de uso
  - **MySQLDatabase**: Classe de comunicação com o banco de dados.
- **Violação:** Explícita! Apesar de OrderService ser uma camada de alto nível, ainda estamos dependendo muito da tecnologia de armazenamento escolhida.

DIP - Exemplo 3

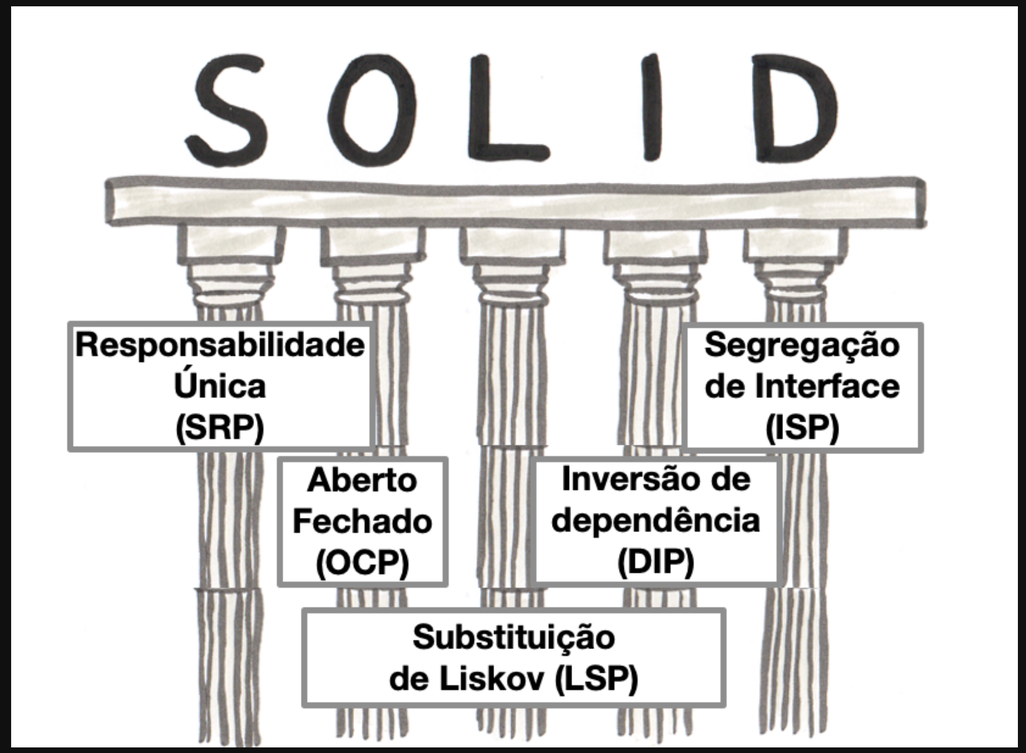
- **Exemplo:** Sistema de processamento de pagamentos com diversas dependências!
    - Dependências: Gateway de pagamento, serviço de e-mail, notificação por SMS, banco de dados, log da aplicação e métricas
- **Violação:** Explícita! Nossa aplicação conhece e depende 100% da implementação concreta de cada serviço. Difícil de automatizar testes e difícil de substituir tecnologias.

DIP - Benefícios

- **Baixo acoplamento:** Não dependemos mais de implementações concretas, detalhes específicos de uma decisão tecnológica e/ou componentes de camadas distintas.
- **Maior testabilidade:** facilidade em automatizar testes substituindo dependências concretas por mocks/stubs.
- **Melhor separação de camadas:** Conseguimos substituir totalmente a tecnologia (baixo nível) sem impactar o nível de negócio (alto nível).

S.O.L.I.D

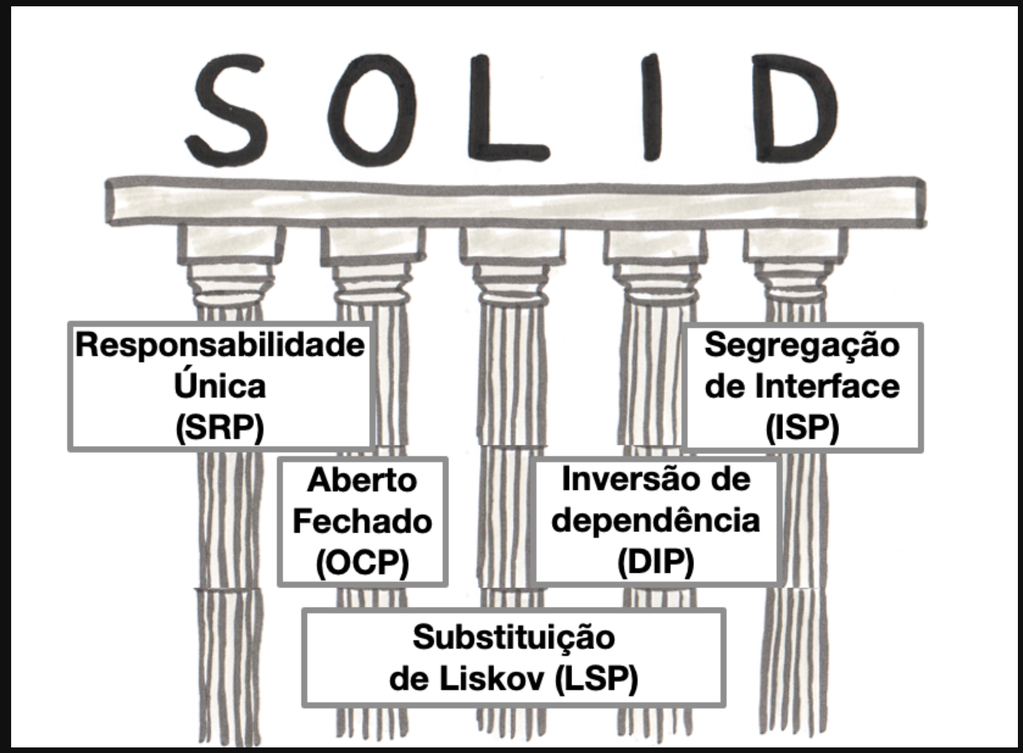
- **S - Single Responsability Principle:** Sugere que uma classe deve fazer apenas uma coisa e fazê-la bem. Isso ajuda a manter o código mais organizado e facilita a manutenção.
- **O - Open-Closed Principle:** Sugere que você deve conseguir estender o comportamento de uma classe sem modificar o código já existente. Geralmente, isso é alcançado através do uso de abstrações e polimorfismo.



S.O.L.I.D

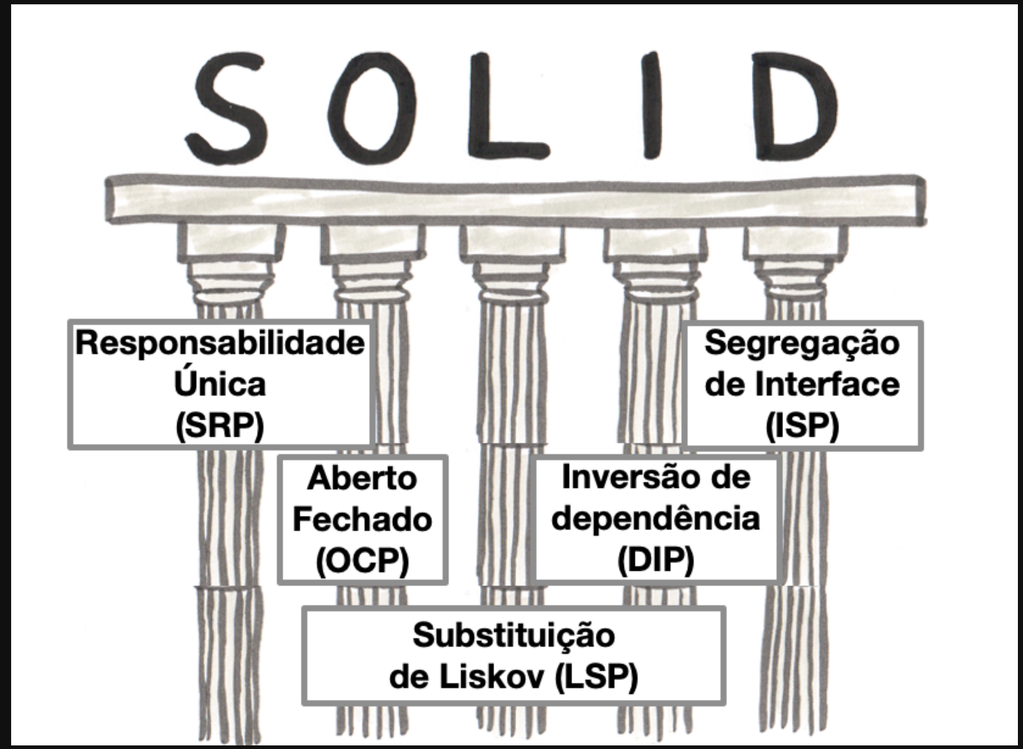
- **L – Liskov Substitution Principle:** Sugere que as subclasses devem poder substituir suas classes base sem alterar o comportamento esperado ou gerar efeitos colaterais no software.

- **I – Interface Segregation Principle:** Sugere que você não deve forçar os clientes a depender de interfaces que eles não usam. É melhor ter várias interfaces menores e mais específicas do que uma grande interface geral.



S.O.L.I.D

- **D – Dependency Inversion Principle:** Afirma que módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações.



S.O.L.I.D - Resumo

- **S:** Cada classe deve ter uma única responsabilidade, ou seja, deve haver apenas uma razão para uma classe mudar.
- **O:** Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação.
- **L:** Objetos de uma superclasse devem ser substituíveis por objetos de suas subclasses sem afetar a corretude do software.
- **I:** Muitas interfaces específicas são melhores do que uma interface geral.
- **D:** Dependa de abstrações, não de implementações concretas.