Aula 3 – Princípios de Design de Software (SOLID)

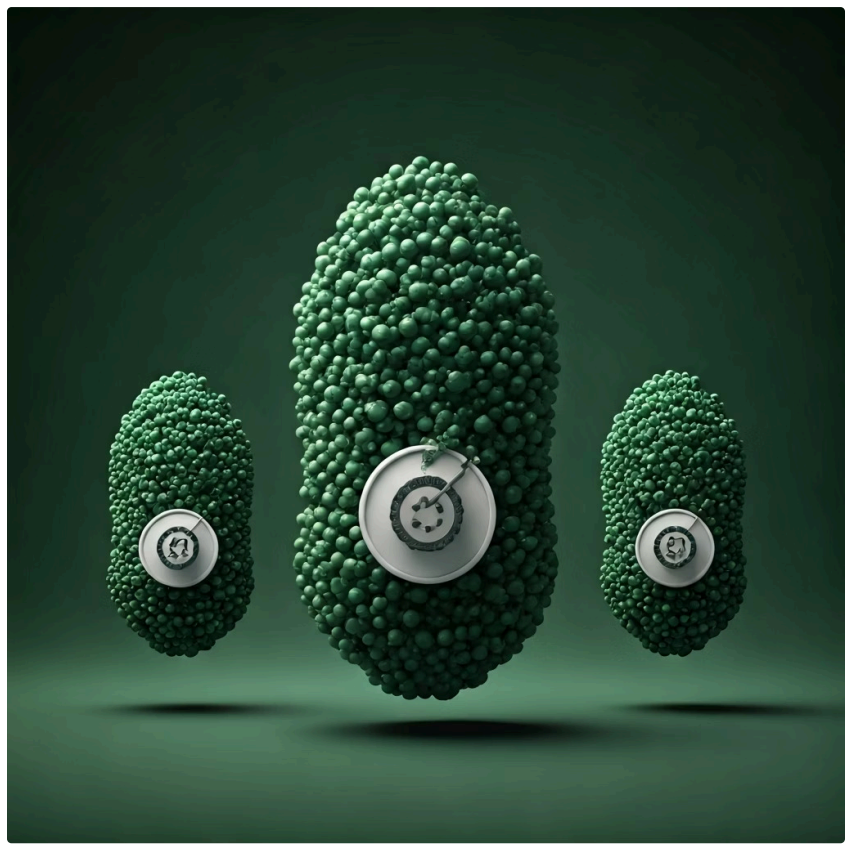
No universo do desenvolvimento de software, a criação de aplicações robustas, escaláveis e de fácil manutenção é um desafio constante. Assim como um arquiteto precisa de princípios sólidos para erguer um edifício que resista ao tempo e às intempéries, nós, desenvolvedores, precisamos de diretrizes para construir sistemas que se adaptem às mudanças e cresçam sem se tornarem um pesadelo de manutenção. É fácil começar um projeto com um código limpo, mas a complexidade tende a se acumular rapidamente, transformando o que era promissor em um emaranhado difícil de entender e modificar.

Imagine que você está construindo uma casa. Se cada parede tiver que suportar o telhado, a fiação elétrica e o encanamento ao mesmo tempo, qualquer pequena alteração se torna um risco para toda a estrutura. No software, isso se traduz em classes e módulos que acumulam muitas responsabilidades, tornando-os frágeis e rígidos. É nesse cenário que os Princípios SOLID entram em cena, oferecendo um conjunto de cinco diretrizes que, quando aplicadas, promovem um design de software mais flexível, compreensível e, acima de tudo, sustentável.

Ao longo desta aula, exploraremos cada um dos princípios SOLID – Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) e Dependency Inversion Principle (DIP). Você aprenderá a identificar os problemas que cada princípio visa resolver e como aplicá-los para criar um código mais coeso e menos acoplado. Nosso objetivo é que, ao final, você seja capaz de analisar o design de suas aplicações e tomar decisões que levem a sistemas mais resilientes, prontos para as demandas de arquiteturas modernas como microserviços e serverless.

A Base da Responsabilidade Única (SRP)

Você já se deparou com aquela classe "faz-tudo" em um projeto? Aquela que, além de gerenciar dados de usuário, também envia e-mails de notificação, registra logs de atividade e talvez até processe pagamentos? Classes assim são como um canivete suíço: parecem úteis no início, mas para uma tarefa específica, uma ferramenta especializada sempre será mais eficiente e segura. O problema é que, quando essa classe precisa de uma pequena alteração em uma de suas muitas funções, você corre o risco de introduzir bugs em outras partes que não têm relação alguma com a mudança original.



Essa situação descreve perfeitamente o problema que o Single Responsibility Principle (SRP), ou Princípio da Responsabilidade Única, busca resolver. Em sua essência, o SRP afirma que **uma classe deve ter apenas uma razão para mudar**. Isso significa que cada classe ou módulo deve ser responsável por apenas uma parte da funcionalidade do sistema. Se você consegue pensar em mais de uma razão para modificar uma classe, é um forte indício de que ela está violando o SRP e, portanto, possui responsabilidades demais.

Exemplo Prático: Sistema de E-commerce

Imagine uma classe **Pedido** que é responsável por:

1. Calcular o valor total do pedido
2. Salvar o pedido no banco de dados
3. Enviar uma notificação ao cliente

Esta classe **viola o SRP**. Se o cálculo do valor mudar, a classe muda. Se a forma de salvar no banco mudar, a classe muda. Se o método de notificação mudar, a classe muda.

CalculadoraDePedido

Responsável apenas por calcular valores

RepositorioDePedido

Responsável apenas por persistência

NotificadorDePedido

Responsável apenas por notificações

A aplicação do SRP é fundamental para a criação de sistemas com baixo acoplamento e alta coesão, características desejáveis em arquiteturas modernas como microserviços, onde cada serviço é, idealmente, uma unidade com responsabilidade única.

Estendendo sem Modificar (OCP)

Pense na última vez que você precisou adicionar uma nova funcionalidade a um sistema existente. Foi um processo suave, onde você apenas adicionou código novo, ou você teve que mergulhar em classes já consolidadas, alterando sua lógica interna e torcendo para não quebrar nada? Muitas vezes, a segunda opção é a realidade, e cada nova alteração se torna um campo minado, gerando regressões e aumentando o custo de manutenção. Esse é o cenário que o Open/Closed Principle (OCP), ou Princípio Aberto/Fechado, visa evitar.

Aberto para Extensão

Você deve ser capaz de adicionar novas funcionalidades ao sistema

Fechado para Modificação

Sem precisar alterar o código existente que já funciona e foi testado

O OCP, formulado por Bertrand Meyer, afirma que entidades de software (classes, módulos, funções, etc.) devem ser **abertas para extensão, mas fechadas para modificação**. O que isso significa na prática? Significa que você deve ser capaz de adicionar novas funcionalidades ao seu sistema sem precisar alterar o código existente que já funciona e foi testado. Em vez de modificar o que já está lá, você estende o comportamento através de novas implementações ou configurações.

> **Analogia:** Uma tomada universal permite conectar diferentes aparelhos (extensões) sem modificar a tomada em si (fechada para modificação). A tomada oferece uma interface padrão, e os aparelhos se adaptam a ela.

Exemplo: Sistema de Cálculo de Frete

❌ Sem OCP

Classe `CalculadoraDeFrete` com série de if/else ou switch para cada transportadora.

Para adicionar nova transportadora: **modificar a classe existente**

✔ Com OCP

Interface `ITransportadora` com método `calcularFrete()`.

Para adicionar nova transportadora: **criar nova classe que implementa a interface**

Considere um sistema de cálculo de frete que suporta diferentes transportadoras. Sem o OCP, você teria uma classe CalculadoraDeFrete com uma série de if/else ou switch para cada transportadora. Para adicionar uma nova transportadora, você teria que modificar essa classe. Com o OCP, você define uma interface ITransportadora com um método calcularFrete(). Cada transportadora (Correios, FedEx, JadLog) implementaria essa interface. A CalculadoraDeFrete então receberia uma ITransportadora e chamaria seu método calcularFrete(), sem se importar com a implementação específica. Adicionar uma nova transportadora significa apenas criar uma nova classe que implementa ITransportadora, sem alterar a CalculadoraDeFrete existente.

Substituição sem Surpresas (LSP)

A herança é uma ferramenta poderosa na programação orientada a objetos, permitindo a reutilização de código e a modelagem de relações "é um tipo de". No entanto, seu uso inadequado pode levar a comportamentos inesperados e bugs difíceis de rastrear. Já imaginou substituir um componente por uma versão "melhorada" e, de repente, partes do seu sistema começarem a falhar sem motivo aparente? Esse é o tipo de problema que o Liskov Substitution Principle (LSP), ou Princípio da Substituição de Liskov, busca prevenir.

Definição do LSP

O LSP, formulado por Barbara Liskov, afirma que **objetos de um programa devem ser substituíveis por instâncias de seus subtipos sem alterar a correção do programa**. Em outras palavras, se você tem uma classe A e uma classe B que herda de A, você deveria ser capaz de usar um objeto de B em qualquer lugar onde um objeto de A é esperado, e o programa ainda deve funcionar corretamente, sem quebras ou comportamentos inesperados.

O Problema Clássico: Retângulo vs Quadrado

Retângulo

Métodos: `setLargura()` e `setAltura()`

Quadrado herda Retângulo

Sobrescreve métodos para manter largura = altura

Violação do LSP

Cliente espera área 50, mas recebe 25 ou 100

Uma analogia clássica para o LSP é a relação entre um Retangulo e um Quadrado. Um quadrado *é um tipo de* retângulo? Matematicamente, sim. Mas no contexto de programação, se você tem uma classe Retangulo com métodos setLargura(int largura) e setAltura(int altura), e uma classe Quadrado que herda de Retangulo e sobrescreve esses métodos para garantir que largura e altura sejam sempre iguais, você viola o LSP. Se um cliente espera um Retangulo e define setLargura(10) e setAltura(5), ele espera um retângulo de área 50. Se, sem saber, ele recebe um Quadrado, a área pode ser 25 (se setLargura também alterar altura) ou 100 (se setAltura também alterar largura), quebrando a expectativa.

Contratos que os Subtipos Devem Manter

- Não lançar novas exceções não esperadas pelo tipo base
- Não exigir pré-condições mais fortes
- Não fornecer pós-condições mais fracas

Para aderir ao LSP, os subtipos devem manter os contratos do tipo base. Isso inclui não lançar novas exceções não esperadas pelo tipo base, não exigir pré-condições mais fortes e não fornecer pós-condições mais fracas. O LSP nos força a pensar cuidadosamente sobre a hierarquia de classes e a garantir que o polimorfismo seja seguro e previsível. Em arquiteturas distribuídas, onde diferentes serviços podem implementar a mesma interface, o LSP garante que a substituição de uma implementação por outra não cause falhas inesperadas no sistema como um todo.

Interfaces Enxutas e Focadas (ISP)

Você já se viu implementando uma interface que continha dezenas de métodos, mas sua classe só precisava de três ou quatro deles? O que você fez com os métodos restantes? Provavelmente os deixou vazios ou lançou exceções de "não implementado". Essa situação é um sintoma de uma "interface gorda", e ela representa um problema significativo de design, pois força classes a dependerem de funcionalidades que não utilizam, aumentando o acoplamento e diminuindo a flexibilidade. É exatamente isso que o Interface Segregation Principle (ISP), ou Princípio da Segregação de Interfaces, busca combater.

> **ISP:** Clientes não devem ser forçados a depender de interfaces que não usam.

O ISP afirma que clientes não devem ser forçados a depender de interfaces que não usam. Em vez de ter uma única interface grande e abrangente, é melhor ter várias interfaces menores e mais específicas, cada uma atendendo a um grupo particular de clientes ou a uma responsabilidade coesa. Pense em um controle remoto universal com centenas de botões para todos os tipos de aparelhos, versus um controle remoto simples e otimizado para sua TV. O controle simples é mais fácil de usar e não te sobrecarrega com funcionalidades irrelevantes.

Exemplo: Interface Trabalhador

❌ Interface "Gorda" (Viola ISP)

Interface `Trabalhador` com métodos:

- `trabalhar()`
- `comer()`
- `dormir()`
- `gerenciarProjetos()`
- `programar()`

Problema: Programador precisa implementar `gerenciarProjetos()` (irrelevante). Gerente precisa implementar `programar()` (desnecessário).

✓ Solução com ISP: Interfaces Segregadas

IComedor
`comer()`

IDormidor
`dormir()`

ITrabalhador
`trabalhar()`

IProgramador
`programar()`

IGerenteDeProjetos
`gerenciarProjetos()`

Vamos a um exemplo. Considere uma interface Trabalhador com métodos como trabalhar(), comer(), dormir(), gerenciarProjetos(), programar(). Uma classe Programador precisaria implementar trabalhar(), comer() e dormir(), mas gerenciarProjetos() seria irrelevante. Uma classe Gerente precisaria de trabalhar(), comer(), dormir() e gerenciarProjetos(), mas programar() seria desnecessário. Isso força ambas as classes a implementar métodos que não lhes pertencem.

A solução do ISP é segregar essa interface em interfaces menores e mais coesas: IComedor com comer(), IDormidor com dormir(), ITrabalhador com trabalhar(), IProgramador com programar(), IGerenteDeProjetos com gerenciarProjetos(). Agora, a classe Programador pode implementar IComedor, IDormidor, ITrabalhador e IProgramador, enquanto a classe Gerente implementa IComedor, IDormidor, ITrabalhador e IGerenteDeProjetos. Cada classe implementa apenas o que realmente precisa, reduzindo o acoplamento e tornando o design mais limpo e flexível. Em sistemas de microserviços, onde cada serviço expõe uma API específica, o ISP é crucial para garantir que os clientes (outros serviços ou front-ends) consumam apenas os contratos de que realmente precisam.

Invertendo o Controle (DIP - Parte 1)

Introdução e Problema

Até agora, falamos sobre como organizar classes, estender funcionalidades e criar interfaces mais limpas. Agora, vamos mergulhar em um princípio que muda fundamentalmente a forma como os módulos do seu sistema se relacionam: o Dependency Inversion Principle (DIP), ou Princípio da Inversão de Dependência. Este princípio é frequentemente considerado um dos mais poderosos e, ao mesmo tempo, um dos mais desafiadores de se compreender e aplicar corretamente, mas seus benefícios para a flexibilidade e testabilidade do código são imensos.

O Problema Tradicional

01	02	03
Módulos de Alto Nível	**Dependem Diretamente**	**Módulos de Baixo Nível**
Contêm lógica de negócios principal e orquestram comportamento	Criam e usam instâncias de módulos de baixo nível	Lidam com detalhes de implementação (banco de dados, rede, etc.)

Tradicionalmente, em muitos designs de software, módulos de alto nível (aqueles que contêm a lógica de negócios principal e orquestram o comportamento) dependem diretamente de módulos de baixo nível (aqueles que lidam com detalhes de implementação, como acesso a banco de dados, comunicação de rede, etc.). Por exemplo, um serviço que gerencia usuários (UserService) pode criar e usar diretamente uma instância de um repositório de banco de dados (MySQLUserRepository). Essa dependência direta cria um acoplamento forte: se você decidir mudar de MySQL para PostgreSQL, ou mesmo para um serviço de cache, você terá que modificar o UserService, que é um módulo de alto nível.

Consequências do Acoplamento Forte

- **Dificuldade de Manutenção:** Mudanças em detalhes afetam lógica de negócio
- **Dificuldade de Evolução:** Sistema rígido e resistente a mudanças
- **Dificuldade de Testes:** Necessário banco de dados real para testar

O problema com essa abordagem é que o módulo de alto nível, que deveria ser mais estável e menos propenso a mudanças, torna-se refém dos detalhes de implementação do módulo de baixo nível. Isso dificulta a manutenção, a evolução e, crucialmente, os testes. Como você testaria o UserService sem ter um banco de dados MySQL real configurado? Você teria que mockar ou simular o banco de dados, mas a dependência ainda está lá no código.

> **DIP propõe:** Em vez de módulos de alto nível dependerem de módulos de baixo nível, **ambos devem depender de abstrações**. As abstrações não devem depender dos detalhes; os detalhes devem depender das abstrações.

O DIP propõe uma inversão dessa dependência. Em vez de módulos de alto nível dependerem de módulos de baixo nível, ambos devem depender de abstrações. Além disso, as abstrações não devem depender dos detalhes; os detalhes devem depender das abstrações. Isso significa que a lógica de negócios (módulo de alto nível) não deve se preocupar com *como* os dados são persistidos, apenas que eles *podem ser* persistidos através de uma interface. E o módulo de persistência (módulo de baixo nível) implementa essa interface.

Invertendo o Controle (DIP - Parte 2)

Solução e Benefícios

Continuando nossa exploração do Dependency Inversion Principle (DIP), a solução para o acoplamento forte que descrevemos na página anterior reside na introdução de abstrações. Em vez de o UserService depender diretamente de MySQLUserRepository, criamos uma interface, por exemplo, IUserRepository. Agora, tanto o UserService quanto o MySQLUserRepository dependerão dessa interface. O UserService dependerá da interface para *usar* um repositório, e o MySQLUserRepository dependerá da interface para *implementá-la*.

Como Funciona a Inversão

A mágica acontece porque a dependência foi invertida. O módulo de alto nível (UserService) não conhece mais os detalhes concretos do módulo de baixo nível (MySQLUserRepository). Ele só conhece a abstração (IUserRepository). Quem é responsável por "injetar" a implementação concreta (MySQLUserRepository) no UserService é um mecanismo externo, geralmente um contêiner de Inversão de Controle (IoC) ou Injeção de Dependência (DI). Isso significa que o UserService não cria suas dependências; ele as recebe.

Exemplo de Código

```
// Abstração
interface IUserRepository {
    void salvar(Usuario usuario);
    Usuario buscarPorId(int id);
}

// Módulo de Baixo Nível (implementação concreta)
class MySQLUserRepository implements IUserRepository {
    // Lógica de acesso ao MySQL
    void salvar(Usuario usuario) { /* ... */ }
    Usuario buscarPorId(int id) { /* ... */ }
}

// Módulo de Alto Nível (depende da abstração)
class UserService {
    private final IUserRepository userRepository;

    // Injeção de Dependência via construtor
    public UserService(IUserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void criarUsuario(Usuario usuario) {
        // Lógica de negócio
        userRepository.salvar(usuario);
    }
}
```

Benefícios do DIP

Flexibilidade

Troque implementações (MySQL → MongoDB) sem alterar UserService

Testabilidade

Teste isoladamente com MockUserRepository, sem banco real

Reusabilidade

Abstrações podem ser reutilizadas em diferentes contextos

Conceito	Âmbito/Aplicação	Exemplo
Acoplamento Forte	Módulos de alto nível dependem de concretos	UserService instanciando new MySQLUserRepository()
Acoplamento Fraco	Módulos de alto nível dependem de abstrações	UserService recebendo IUserRepository via construtor

Os benefícios do DIP são enormes: **Flexibilidade** – Você pode trocar a implementação do repositório (ex: de MySQL para MongoDB, ou para um serviço externo) sem alterar o UserService. **Testabilidade** – É muito mais fácil testar o UserService isoladamente, injetando um MockUserRepository que simula o comportamento do banco de dados, sem a necessidade de um ambiente de banco de dados real. **Reusabilidade** – As abstrações podem ser reutilizadas em diferentes contextos.

SOLID na Prática e Conexão com Tendências

Compreender os princípios SOLID individualmente é um passo crucial, mas o verdadeiro poder reside em como eles se complementam e se aplicam no contexto de arquiteturas de software modernas. O desenvolvimento web atual é dominado por tendências como Microserviços, Arquitetura Serverless, APIs flexíveis como GraphQL e comunicação de alta performance com gRPC. Longe de serem conceitos acadêmicos, os princípios SOLID são a base para construir sistemas que prosperam nesse ambiente dinâmico e distribuído.

SOLID e Microserviços

SRP

Cada microserviço tem responsabilidade única e bem definida

ISP

APIs enxutas e focadas, clientes consomem apenas o necessário

OCP

Adicione funcionalidades sem modificar serviços em produção

LSP

Substitua versões de serviços sem quebrar clientes

Pense nos Microserviços: a ideia central é quebrar uma aplicação monolítica em pequenos serviços independentes, cada um com uma responsabilidade bem definida. Isso ressoa diretamente com o **Single Responsibility Principle (SRP)**. Cada microserviço, idealmente, deve ter uma única razão para mudar, encapsulando uma funcionalidade de negócio específica. Da mesma forma, o **Interface Segregation Principle (ISP)** é vital para microserviços, garantindo que as APIs expostas por cada serviço sejam enxutas e focadas, e que os clientes (outros serviços ou front-ends) consumam apenas os contratos de que realmente precisam, evitando interfaces "gordas" que forçam dependências desnecessárias.

O **Open/Closed Principle (OCP)** é fundamental para a evolução de APIs e serviços. Em um ecossistema de microserviços, você quer ser capaz de adicionar novas funcionalidades ou versões de APIs (como novas consultas GraphQL ou métodos gRPC) sem precisar modificar os serviços existentes que já estão em produção. Isso é alcançado através de extensões, como a adição de novos *resolvers* em GraphQL ou a criação de novos serviços que implementam uma interface comum. O **Liskov Substitution Principle (LSP)**, por sua vez, garante que, ao evoluir ou substituir um serviço por uma nova versão, os clientes que dependem da interface original não sejam quebrados, mantendo a compatibilidade e a previsibilidade do comportamento.

DIP: A Espinha Dorsal de Sistemas Distribuídos

O **Dependency Inversion Principle (DIP)** é a espinha dorsal para a orquestração e comunicação em arquiteturas distribuídas. Em vez de um serviço depender diretamente da implementação concreta de outro serviço, ambos dependem de contratos (interfaces ou schemas de API).

Resultado: Você pode trocar implementações de serviços, testá-los isoladamente e construir sistemas mais resilientes onde a falha de um componente não derruba todo o sistema.

Finalmente, o **Dependency Inversion Principle (DIP)** é a espinha dorsal para a orquestração e comunicação em arquiteturas distribuídas. Em vez de um serviço depender diretamente da implementação concreta de outro serviço, ambos dependem de contratos (interfaces ou schemas de API). Isso permite que você troque implementações de serviços, teste-os isoladamente e construa sistemas mais resilientes onde a falha de um componente não derruba todo o sistema. SOLID não é apenas sobre classes, é sobre a arquitetura de todo o sistema, preparando-o para a escalabilidade e a agilidade que as tendências de 2025 exigem.

Desafios e Boas Práticas na Aplicação de SOLID

Embora os princípios SOLID ofereçam um caminho claro para um design de software de alta qualidade, sua aplicação não é isenta de desafios. É fácil cair na armadilha da "over-engenharia", onde a busca por aderência perfeita a cada princípio resulta em uma complexidade desnecessária, com muitas classes e interfaces para funcionalidades simples. O objetivo não é aplicar todos os princípios cegamente em cada linha de código, mas sim usá-los como um guia para tomar decisões de design conscientes, especialmente em áreas críticas do sistema.

Um dos maiores desafios é identificar a "responsabilidade única" no SRP ou a "abstração correta" para o OCP e DIP. Isso exige experiência e um bom entendimento do domínio do problema. Muitas vezes, a melhor abordagem é começar com um design mais simples e refatorar o código para aplicar os princípios SOLID à medida que a complexidade cresce e as necessidades do sistema se tornam mais claras. SOLID é um processo contínuo de melhoria, não um estado final a ser alcançado de uma vez.

Boas Práticas para Aplicar SOLID

1 Comece Simples, Refatore Depois

Não tente aplicar SOLID em cada detalhe desde o início. Comece com um design funcional e, à medida que o código evolui e a complexidade aumenta, identifique as áreas onde a aplicação dos princípios traria mais benefícios. A refatoração é sua aliada.

2 Use o Bom Senso

SOLID são diretrizes, não dogmas. Haverá situações em que a aplicação estrita de um princípio pode levar a um código mais complexo do que o necessário. Priorize a clareza e a manutenibilidade.

3 Foque nas Abstrações

A chave para OCP, LSP, ISP e DIP é o uso eficaz de interfaces e classes abstratas. Pense em contratos, não em implementações concretas, ao projetar as interações entre seus módulos.

4 Testes como Guia

Um código que segue SOLID é inerentemente mais testável. Se você está tendo dificuldade para testar uma classe isoladamente, isso pode ser um sinal de que ela viola o SRP ou o DIP. Use os testes unitários como um feedback sobre a qualidade do seu design.

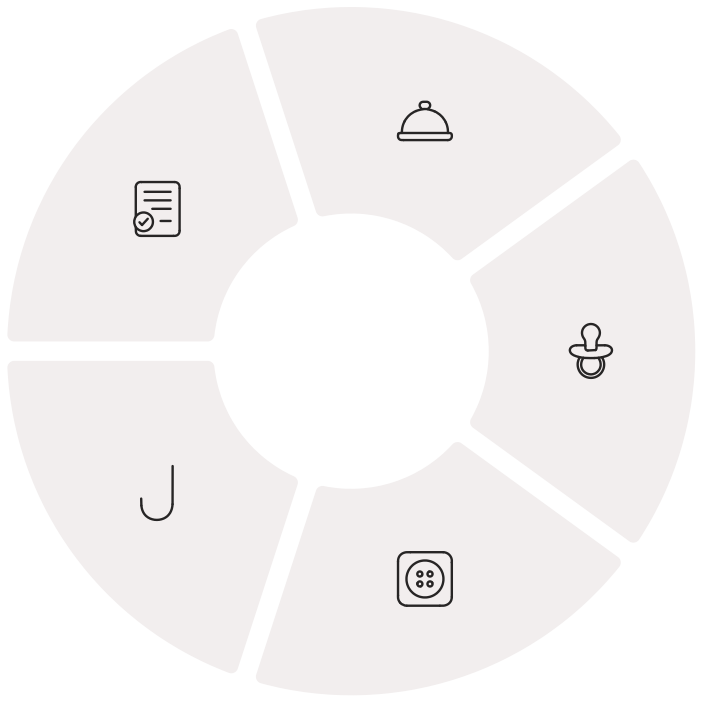
5 Revisão de Código

Discutir o design com colegas durante revisões de código ajuda a identificar violações de SOLID e a encontrar soluções colaborativas. Duas cabeças pensam melhor que uma.

Conceito	Benefícios	Desafios
Aplicação SOLID	Manutenibilidade, Flexibilidade, Testabilidade, Escalabilidade	Over-engenharia, Dificuldade em identificar responsabilidades, Custo inicial
Design Evolutivo	Adaptação a mudanças, Redução de riscos, Melhoria contínua do código	Exige disciplina de refatoração, Pode parecer "mais lento" no início

Consolidação e Próximos Passos

Chegamos ao final da nossa jornada pelos Princípios SOLID. Vimos que o **SRP** nos ensina a dar a cada classe uma única razão para mudar, promovendo coesão e clareza. O **OCP** nos capacita a estender funcionalidades sem tocar no código existente, garantindo a estabilidade. O **LSP** nos alerta sobre os perigos da herança mal aplicada, assegurando que subtipos se comportem como seus tipos base. O **ISP** nos guia para criar interfaces enxutas e focadas, evitando dependências desnecessárias. E o **DIP** inverte o fluxo de controle, fazendo com que módulos de alto e baixo nível dependam de abstrações, resultando em sistemas altamente flexíveis e testáveis.



SRP

Uma razão para mudar

OCP

Aberto para extensão, fechado para modificação

LSP

Subtipos substituíveis

DIP

Dependa de abstrações

ISP

Interfaces enxutas

> **Em Prática**
>
> A aplicação dos princípios SOLID não é um destino, mas uma jornada contínua de aprimoramento do design. Comece identificando as classes que fazem demais e separe suas responsabilidades. Ao adicionar novas funcionalidades, pense em como estender o sistema em vez de modificá-lo. Use interfaces para definir contratos claros e inverta as dependências para facilitar testes e futuras mudanças.

Autoavaliação

1. Qual princípio SOLID afirma que uma classe deve ter apenas uma razão para mudar?
   - a) Open/Closed Principle (OCP)
   - b) Liskov Substitution Principle (LSP)
   - c) Single Responsibility Principle (SRP)
   - d) Dependency Inversion Principle (DIP)
2. Um sistema de processamento de pagamentos que permite adicionar novos métodos de pagamento (cartão, boleto, PIX) sem alterar o código existente da classe ProcessadorDePagamentos está aplicando qual princípio SOLID?
   - a) Interface Segregation Principle (ISP)
   - b) Open/Closed Principle (OCP)
   - c) Single Responsibility Principle (SRP)
   - d) Liskov Substitution Principle (LSP)
3. Se uma classe PatoBorracha herda de Pato e lança uma exceção ao tentar voar(), qual princípio SOLID está sendo violado?
   - a) Dependency Inversion Principle (DIP)
   - b) Interface Segregation Principle (ISP)
   - c) Single Responsibility Principle (SRP)
   - d) Liskov Substitution Principle (LSP)
4. O uso de interfaces para que um módulo de alto nível (ServicoDeEmail) dependa de uma abstração (IEnviadorDeEmail) em vez de uma implementação concreta (SmtpEnviadorDeEmail) é uma aplicação direta de qual princípio SOLID?
   - a) Single Responsibility Principle (SRP)
   - b) Open/Closed Principle (OCP)
   - c) Interface Segregation Principle (ISP)
   - d) Dependency Inversion Principle (DIP)
5. Explique como o Princípio da Segregação de Interfaces (ISP) contribui para a construção de APIs mais eficientes em um ambiente de microserviços.

Gabarito

Questão 1

c) Single Responsibility Principle (SRP)

Questão 2

b) Open/Closed Principle (OCP)

Questão 3

d) Liskov Substitution Principle (LSP)

Questão 4

d) Dependency Inversion Principle (DIP)

Próxima Aula e Recursos Adicionais

Próxima Aula

Aula 4 – Do Monólito aos Sistemas Distribuídos

Exploraremos as arquiteturas que se beneficiam enormemente dos princípios SOLID, entendendo como eles pavimentam o caminho para a transição de monólitos para microserviços e arquiteturas serverless, e como a comunicação eficiente via GraphQL e gRPC se encaixa nesse cenário.

Recursos Adicionais

📚 Livro "Clean Architecture"

Autor: Robert C. Martin

Para aprofundar nos princípios de design e arquitetura de software.

📝 Artigos de Martin Fowler

Tema: Design Patterns

Para exemplos práticos e discussões sobre padrões que aplicam SOLID.

🔧 Documentação de Frameworks

Exemplos: Spring, .NET Core

Para ver o DIP em ação através de Injeção de Dependência.

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.