Aula 03 - Princípios SOLID

Disciplina: Manutenção de Software

Prof. Me. João Paulo Biazotto

UniCesumar

Agenda

- Revisão de DDD
- Princípios SOLID
- Princípio DRY
- Lei de Demeter

Domain-driven design (DDD)

- Abordagem para desenvolvimento de software que visa alinhar o código com o domínio do negócio, priorizando a compreensão e a resolução dos problemas desse domínio.

Domain-driven design (DDD)

1. Compreender o domínio do negócio
2. Identificar os conceitos fundamentais (modelagem do domínio)
3. Modelagem do domínio
4. Definição de bounded contexts (contextos delimitados)
5. Design da arquitetura
6. Implementação do código

Por que falar de SOLID?

- Sistemas crescem: mais regras, mais integrações, mais “gambiarras”
- O problema não é mudar o software — é mudar sem quebrar tudo
- SOLID = conjunto de princípios para:
  - reduzir acoplamento
  - aumentar coesão
  - facilitar evolução e testes
- Objetivo da aula: **entender o que é, por que existe e quando aplicar**

Por que falar de SOLID?

- Temos um fluxo simples: calcular total + aplicar desconto + cobrar + enviar e-mail
- No início funciona... até começarem as mudanças:
  - novos tipos de desconto (cupom, fidelidade, Black Friday)
  - novas formas de pagamento (Pix, cartão, boleto)
  - novas notificações (e-mail, SMS, push)
  - logs/auditoria obrigatória
- Pergunta
  - Como evoluir sem transformar tudo em um “monstro”?

Por que falar de SOLID?

- CheckoutService.finalizarPedido(pedido) faz tudo:
  - valida estoque
  - calcula total
  - aplica desconto
  - processa pagamento
  - envia e-mail
  - registra log/auditoria

Qual o problema?

- classe gigante, difícil de entender
- testes difíceis (precisa de gateway de pagamento + SMTP)
- mudanças em “desconto” exigem mexer no checkout inteiro

Requisitos novos viram cascata de if/else

- Mudança 1: adicionar Pix e “comprovante”
- Mudança 2: desconto por categoria + desconto progressivo
- Mudança 3: notificar por SMS quando for Pix

Requisitos novos viram cascata de if/else

- if (pagamento == CARTAO) ... else if (pagamento == PIX) ...
- Cada mudança aumenta:
  - complexidade
  - risco de bug
  - retrabalho de testes

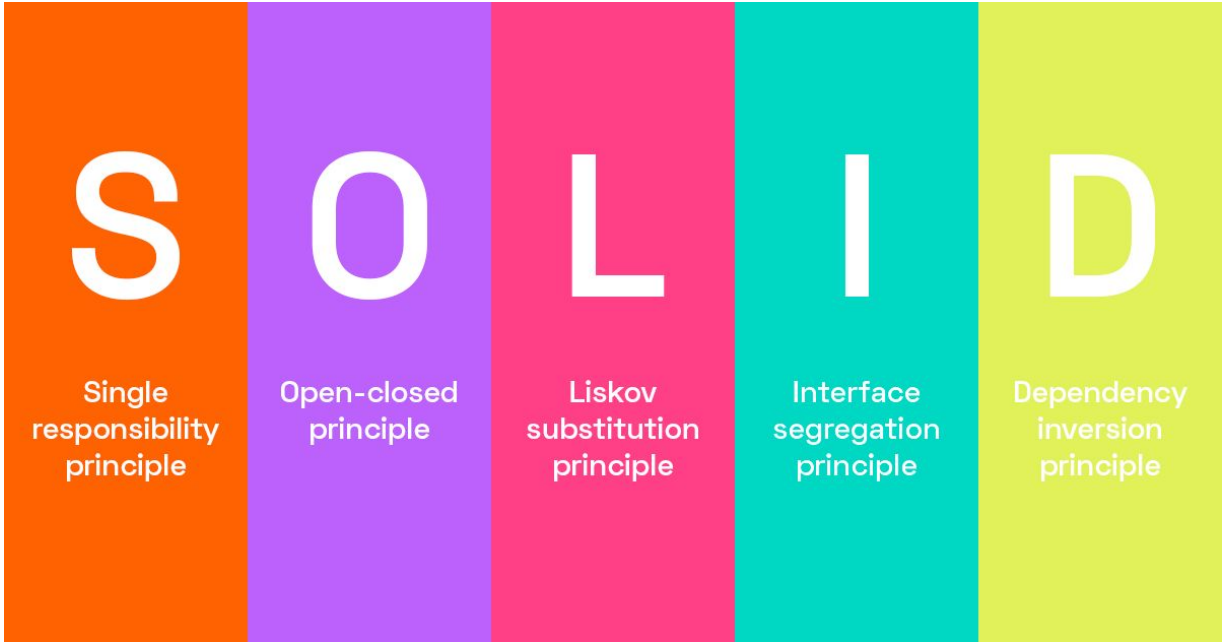
O que está por trás da bagunça?

- Causas principais
  - muitas responsabilidades no mesmo lugar (uma classe faz “tudo”)
  - dependência direta de detalhes (gateway, email, sms dentro do checkout)
  - extensão por edição (toda nova regra obriga alterar código existente)
  - interfaces inchadas (serviço obrigado a conhecer métodos que nem usa)
- No fim das contas
  - O problema não é OO, mas sim o design sem princípios para mudança constante.

Como solucionar o problema?

- Separar “o que muda junto” em componentes:
  - CalculadoraDeTotal
  - PoliticaDeDesconto (várias implementações)
  - ProcessadorDePagamento (Pix, Cartão, Boleto)
  - Notificador (Email, SMS, Push)
  - Fazer o Checkout depender de abstrações, não de detalhes

O que é SOLID?


S
Single responsibility principle
O
Open-closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle

Ideia central de cada princípio

- SRP: uma classe/módulo tem um motivo principal para mudar
- OCP: evoluir por extensão, não por alteração constante do código existente
- LSP: subtipos devem ser substituíveis sem surpresas (contratos respeitados)
- ISP: interfaces pequenas e focadas; evitar “interfaces gordas”
- DIP: depender de abstrações, não de implementações (inversão de dependência)

DÚVIDAS?