Princípios SOLID em POO

Os princípios SOLID são cinco diretrizes que ajudam a escrever códigos orientados a objetos mais limpos, flexíveis, reutilizáveis e fáceis de manter. Cada letra representa um princípio:

---

1. S – Single Responsibility Principle (Princípio da Responsabilidade Única)

> **"Uma classe deve ter apenas um motivo para mudar."**

- **Cada classe deve ter apenas uma responsabilidade.**
- **Exemplo: Uma classe `RelatorioFinanceiro` que gera relatórios não deve também salvá-los no banco de dados.**

---

2. O – Open/Closed Principle (Princípio Aberto/Fechado)

> **"Entidades devem estar abertas para extensão, mas fechadas para modificação."**

- **Você deve poder adicionar novos comportamentos sem precisar alterar o código existente.**
- **Exemplo: Criar uma interface `Pagamento` e diferentes classes como `CartaoCredito`, `Boleto`, `Pix` sem mudar o código que processa o pagamento.**

---

3. L – Liskov Substitution Principle (Princípio da Substituição de Liskov)

> **"Objetos de uma superclasse devem poder ser substituídos por objetos de suas subclasses sem quebrar o sistema."**

- **Uma subclasse deve preservar o comportamento esperado da superclasse.**
- **Exemplo: Se `Quadrado` herda de `Retangulo`, ambos devem funcionar corretamente onde um `Retangulo` for esperado, sem comportamento estranho.**

---

4. I – Interface Segregation Principle (Princípio da Segregação de Interface)

> **"Muitas interfaces específicas são melhores do que uma interface única e geral."**

- **Evite interfaces grandes e genéricas. Divida em interfaces menores e mais específicas.**
- **Exemplo: Em vez de uma interface `Funcionario` com métodos `programar()`, `vender()`, `atenderCliente()`, crie interfaces específicas como `Programador`, `Vendedor`.**

---

5. D – Dependency Inversion Principle (Princípio da Inversão de Dependência)

> **"Dependa de abstrações, não de implementações."**

- **Classes devem depender de interfaces ou classes abstratas, e não de classes concretas.**
- **Exemplo: Um `RelatorioService` não deve depender diretamente de `MySQLDatabase`, mas sim de uma interface `BancoDeDados`.**

💡 **Benefícios ao aplicar SOLID**

- **Facilita manutenção e testes**
- **Reduz o acoplamento entre componentes**
- **Aumenta a reutilização e legibilidade**
- **Deixa o sistema mais preparado para mudanças futuras**

TRABALHO IV

POO - TADS RIO GRANDE 2025-1

Estudantes com a incial do nome

- **de A até K devem fazer o TRABALHO 1**
- **de L até V devem fazer o TRABALHO 2**

DATA DE APRESENTAÇÃO 18/07/2025 até 22/07/2025

Importante: Durante a apresentação do trabalho, será solicitado que vocês demonstrem a aplicação dos princípios SOLID e a extensibilidade do sistema, inserindo novas regras de negócio via código (sim, você vai codar comigo), **sem a necessidade de alterar as interfaces ou a estrutura existente** do sistema. Isso validará que os padrões solicitados foram efetivamente atendidos.

Trabalho 1 – Sistema de Pedidos com Regras Dinâmicas

Tema: Criação de um sistema para registrar pedidos com múltiplos tipos de produto, com regras de desconto e tributação específicas.

Exigências:

- Cada tipo de produto (digital, físico, assinatura) deve seguir regras de imposto diferentes.
- As regras de desconto podem variar conforme o mês/data do pedido (ex: campanhas de Black Friday).
- Sistema deve poder ser **extendido** com novas regras de forma simples.

Classes e Interfaces Essenciais:

1. **`Pedido`**
    - **Propósito:** Representa um pedido realizado pelo cliente, contendo uma lista de itens e o controle geral.
    - **Atributos:**
        - `id` (string/UUID): Identificador único do pedido.

        - `dataPedido` (DateTime): Data e hora em que o pedido foi realizado (essencial para regras de desconto baseadas em data/mês).
        - `cliente` (`Cliente`): Objeto que representa o cliente que fez o pedido.
        - `itens` (List<`ItemPedido`>): Lista dos produtos incluídos no pedido.
        - `valorTotalBruto` (decimal): Soma dos valores dos itens antes de descontos e impostos.
        - `valorTotalDescontos` (decimal): Total de descontos aplicados.
        - `valorTotalImpostos` (decimal): Total de impostos calculados.
        - `valorTotalLiquido` (decimal): Valor final do pedido após descontos e impostos.
    - **Métodos:**
        - `AdicionarItem(ItemPedido item)`
        - `CalcularTotais()`: Coordena o cálculo de descontos e impostos para todos os itens.
2. **`ItemPedido`**
    - **Propósito:** Representa uma linha de um pedido, associando um produto a uma quantidade e seu valor no contexto do pedido.
    - **Atributos:**
        - `produto` (`IProduto`): O produto que está sendo pedido (interface para flexibilidade).
        - `quantidade` (int): Quantidade do produto.
        - `valorUnitario` (decimal): Valor unitário do produto no momento da compra.
        - `valorTotalItem` (decimal): `valorUnitario * quantidade`.
        - `descontoAplicado` (decimal): Desconto específico para este item.
        - `impostoAplicado` (decimal): Imposto específico para este item.
        - `valorLiquidoItem` (decimal): `valorTotalItem - descontoAplicado + impostoAplicado`.
3. **`Cliente`**
    - **Propósito:** Armazena os dados do cliente.
    - **Atributos:**
        - `id` (string/UUID): Identificador único do cliente.
        - `nome` (string): Nome completo do cliente.
        - `email` (string): E-mail do cliente.
        - `dataNascimento` (DateTime): Para a regra de desconto de aniversário (se implementada).
        - `endereco` (string, opcional): Para regras de frete ou impostos baseados em localização.
4. **`IProduto` (Interface)**
    - **Propósito:** Define o contrato básico para todos os tipos de produto. Garante o princípio L (Substituição de Liskov).
    - **Atributos (somente get):**
        - `Nome` (string)
        - `ValorBase` (decimal)
        - `Tipo` (`EnumTipoProduto`)
    - **Comportamento/Métodos:**

        - `decimal GetValorComDesconto(IDescontoStrategy descontoStrategy, Pedido pedido, ItemPedido item)`: Calcula o valor do produto após aplicar uma estratégia de desconto.
        - `decimal GetValorComImposto(IImpostoStrategy impostoStrategy)`: Calcula o valor do imposto para o produto, dada uma estratégia.
5. **Classes de Produtos (Implementações de `IProduto`)**
    - **`ProdutoDigital`:**
        - `Atributos específicos`: `urlDownload` (string), `tamanhoMB` (decimal).
    - **`ProdutoFisico`:**
        - `Atributos específicos`: `pesoKG` (decimal), `dimensoes` (string), `estoque` (int).
    - **`Assinatura`:**
        - `Atributos específicos`: `periodoMeses` (int), `recorrencia` (EnumTipoRecorrencia).
6. **`IDescontoStrategy` (Interface)**
    - **Propósito:** Define o contrato para diferentes estratégias de cálculo de desconto. Garante o princípio O (Aberto/Fechado) e I (Segregação de Interfaces).
    - **Comportamento/Métodos:**
        - `decimal CalcularDesconto(Pedido pedido, ItemPedido item, IProduto produto)`: Calcula o valor do desconto para um item específico de um pedido.
7. **Classes de Estratégias de Desconto (Implementações de `IDescontoStrategy`)**
    - **`DescontoMesAniversarioStrategy`:**
        - `Atributos específicos`: `percentualDesconto` (decimal).
        - **Comportamento:** Verifica se o mês do pedido (`pedido.DataPedido.Month`) coincide com o mês de aniversário do cliente (`pedido.Cliente.DataNascimento.Month`). Pode ter uma condição adicional de valor mínimo para o produto (ex: "para produtos digitais vendidos no mês de aniversário do cliente e acima de R$300").
    - **`DescontoBlackFridayStrategy`:**
        - `Atributos específicos`: `percentualDesconto` (decimal).
        - **Comportamento:** Verifica se a `dataPedido` está dentro do período de Black Friday.
    - **`DescontoPorVolumeStrategy`:**
        - `Atributos específicos`: `quantidadeMinima` (int), `percentualDesconto` (decimal).
        - **Comportamento:** Verifica se `item.Quantidade` é maior ou igual à `quantidadeMinima`.
8. **`IImpostoStrategy` (Interface)**
    - **Propósito:** Define o contrato para diferentes estratégias de cálculo de imposto. Garante o princípio O (Aberto/Fechado) e I (Segregação de Interfaces).
    - **Comportamento/Métodos:**
        - `decimal CalcularImposto(IProduto produto, decimal valorBase)`: Calcula o valor do imposto para um produto.

9. **Classes de Estratégias de Imposto (Implementações de `IImpostoStrategy`)**
    - **`ImpostoProdutoDigitalStrategy`:**
        - `Atributos específicos`: `aliquota` (decimal).
        - **Comportamento:** Aplica uma alíquota específica para produtos digitais.
    - **`ImpostoProdutoFisicoStrategy`:**
        - `Atributos específicos`: `aliquotaNacional` (decimal), `aliquotaEstadual` (decimal, opcional, se precisar de regra por localização).
        - **Comportamento:** Aplica alíquotas diferentes para produtos físicos.
    - **`ImpostoAssinaturaStrategy`:**
        - `Atributos específicos`: `aliquota` (decimal).
        - **Comportamento:** Aplica uma alíquota específica para assinaturas.
    - **`IsencaoImpostoStrategy`:**
        - **Comportamento:** Retorna 0 para o imposto (usada na regra de aniversário para isenção).
10. **`CalculadoraPedido`**
    - **Propósito:** Orquestra os cálculos de descontos e impostos para um pedido completo. Aplica o princípio D (Inversão de Dependência) e S (Responsabilidade Única).
    - **Atributos:**
        - `descontoStrategy` (`IDescontoStrategy`): Estratégia de desconto principal ou padrão.
        - `impostoStrategyFactory` (`IImpostoStrategyFactory`): Uma fábrica para obter a estratégia de imposto correta para cada tipo de produto.
    - **Métodos:**
        - `ProcessarPedido(Pedido pedido)`: Itera sobre os itens do pedido, aplicando as estratégias de desconto e imposto adequadas a cada um, e atualiza os totais do pedido.

Enums:

- **`EnumTipoProduto`:**
    - `Digital`
    - `Fisico`
    - `Assinatura`
- **`EnumTipoRecorrencia`** (para assinaturas):
    - `Mensal`
    - `Trimestral`
    - `Anual`

Aplicações de SOLID:

- **S:** Separação clara entre cálculo de preço, desconto, impostos.
- **O:** Novos tipos de produto ou imposto devem ser incluídos sem alterar os existentes.
- **L:** Tipos de produto devem ser substituíveis nos cálculos.
- **I:** Interfaces específicas para comportamento de desconto, imposto, produto.
- **D:** Injete estratégias de cálculo via interfaces.

Trabalho 2 – Sistema de Aprovação de Créditos

Tema: Simular um sistema de análise de crédito para empréstimos, com regras que mudam conforme perfil do solicitante.

Exigências:

- Solicitações passam por múltiplos "avaliadores": CPF limpo, renda, score, tempo de emprego, etc.
- Deve ser possível adicionar novos avaliadores sem quebrar os antigos.
- A decisão final é composta por um pipeline de validações.

Com base na descrição do Trabalho 2 – Sistema de Aprovação de Créditos, aqui estão sugestões de atributos e classes necessárias, aplicando os princípios SOLID:

Classes e Interfaces Essenciais:

1. **`SolicitacaoCredito` (ou `PedidoCredito`)**
    - **Propósito:** Representa a solicitação de crédito feita por um cliente, contendo todos os dados necessários para a análise.
    - **Atributos:**
        - `id` (string/UUID): Identificador único da solicitação.
        - `cliente` (`Cliente`): Objeto que representa o cliente.
        - `valorEmprestimoDesejado` (decimal): O valor que o cliente deseja.
        - `prazoPagamentoMeses` (int): Prazo em meses para o pagamento.
        - `statusAprovacao` (`EnumStatusAprovacao`): Pendente, Aprovado, Reprovado.
        - `motivoReprovacao` (string, opcional): Em caso de reprovação, o motivo.
        - `dataSolicitacao` (DateTime): Data e hora da solicitação.
        - `pontuacaoFinalCredito` (int, opcional): Score final consolidado, se aplicável.
2. **`Cliente`**
    - **Propósito:** Armazena os dados do solicitante do crédito.
    - **Atributos:**
        - `cpf` (string): CPF do cliente (identificador único).
        - `nome` (string): Nome completo do cliente.
        - `rendaMensal` (decimal): Renda declarada do cliente.
        - `scoreSerasa` (int): Pontuação de crédito (ex: Serasa, Boa Vista).
        - `temNomeLimpo` (bool): Indica se o CPF está limpo (sem restrições).
        - `tempoEmpregoMeses` (int): Tempo de emprego atual em meses.
        - `historicoDividas` (List<`DividaAnterior`>, opcional): Histórico de dívidas passadas.
        - `idade` (int): Idade do cliente.

3. **`IAvaliadorCredito` (Interface)**
    - **Propósito:** Define o contrato para qualquer avaliador de crédito. Garante o princípio O (Aberto/Fechado) e L (Substituição de Liskov).
    - **Métodos:**
        - `bool Avaliar(SolicitacaoCredito solicitacao)`: Realiza a avaliação e retorna `true` para aprovado, `false` para reprovado por essa regra específica.
        - `string GetNomeRegra()`: Retorna o nome da regra de avaliação (ex: "Validador CPF Limpo").
        - `string GetMensagemReprovacao()`: Retorna a mensagem específica em caso de reprovação por essa regra.
4. **Classes de Avaliadores (Implementações de `IAvaliadorCredito`) - Você precisa implementar pelo menos 3. Outros podem ser solicitados durante a apresentação.**
    - Cada uma implementa uma regra específica.
    - **`ValidadorCpfLimpo`:**
        - Avalia: `solicitacao.Cliente.TemNomeLimpo`
    - **`ValidadorRendaMinima`:**
        - Atributos: `rendaMinimaExigida` (decimal).
        - Avalia: `solicitacao.Cliente.RendaMensal` vs. `rendaMinimaExigida` e `solicitacao.ValorEmprestimoDesejado`.
    - **`ValidadorScoreCredito`:**
        - Atributos: `scoreMinimoExigido` (int).
        - Avalia: `solicitacao.Cliente.ScoreSerasa` vs. `scoreMinimoExigido`.
    - **`ValidadorTempoEmprego`:**
        - Atributos: `tempoEmpregoMinimoMeses` (int).
        - Avalia: `solicitacao.Cliente.TempoEmpregoMeses` vs. `tempoEmpregoMinimoMeses`.
    - **`ValidadorHistoricoDividas` (exemplo mais complexo):**
        - Avalia: A presença ou o status de `solicitacao.Cliente.HistoricoDividas`. Pode ter atributos como `limiteDividasPendentes` ou `permitirDividasAteValor`.
    - **`ValidadorIdadeMinima`:**
        - Atributos: `idadeMinima` (int).
        - Avalia: `solicitacao.Cliente.Idade` vs. `idadeMinima`.
5. **`ProcessadorAnaliseCredito` (ou `PipelineAprovacao`)**
    - **Propósito:** Gerencia a execução dos avaliadores em sequência (pipeline). Aplica o princípio D (Inversão de Dependência) e S (Responsabilidade Única).
    - **Atributos:**
        - `avaliadores` (List<`IAvaliadorCredito`>): Uma lista dos avaliadores a serem executados.
    - **Métodos:**
        - `Analisar(SolicitacaoCredito solicitacao)`: Itera sobre a lista de avaliadores. Se qualquer um reprovar, a solicitação é reprovada e o processo pode ser interrompido.
        - `AdicionarAvaliador(IAvaliadorCredito avaliador)`: Permite adicionar novos avaliadores dinamicamente.

Enums:

- **`EnumStatusAprovacao`:**
    - `Pendente`
    - `Aprovado`
    - `Reprovado`

Estruturas de Dados Auxiliares (se necessário):

- **`DividaAnterior`:**
    - `valor` (decimal)
    - `status` (string, ex: "Paga", "Em Atraso", "Negociada")
    - `dataVencimento` (DateTime)

Como as regras seriam adicionadas dinamicamente na apresentação:

Você teria sua classe `ProcessadorAnaliseCredito` com alguns validadores já configurados. Para a demonstração, você instanciará uma nova classe que implementa `IAvaliadorCredito` (ex: `ValidadorNovaRegraEspecifica`) e a adicionará à lista de avaliadores do `ProcessadorAnaliseCredito` no momento, sem precisar recompilar ou alterar as classes existentes. Isso demonstraria a extensibilidade (Princípio Aberto/Fechado).

Aplicações de SOLID:

- **S:** Cada validador tem sua responsabilidade.
- **O:** Novo validador pode ser adicionado sem modificar os existentes.
- **L:** Todos os validadores devem poder ser usados no processo de análise.
- **I:** Interface comum para validadores.
- **D:** Composição de validadores via injeção.