Atividade 2: Meios de Pagamento

CONCEITUAÇÃO:

Ritual Comum de Processamento de Pagamentos

O processamento segue um fluxo fixo de três etapas principais:

1. **Validar** - Verificar dados essenciais e viabilidade do pagamento
2. **Autorizar/Capturar** - Garantir fundos/reserva do valor e efetivar a transação
3. **Confirmar** - Finalizar o processo e gerar comprovante

Este ritual é **invariável** para todos os meios de pagamento, garantindo consistência no processamento.

Variações entre Meios de Pagamento

Meio	Validação Específica	Autorização/Captura	Comprovante
Cartão	Dados do cartão, CVV, limite	Autorização + captura junto à operadora	Comprovante com código de autorização
PIX	Chave PIX, limite temporal	Geração de QR Code + confirmação via banco	Comprovante PIX com QR e ID da transação
Boleto	Data de vencimento, beneficiário	Geração de linha digitável + aguardar compensação	Boleto com instruções e linha digitável

Diferenças cruciais:

- **Cartão**: Requer comunicação em tempo real com operadora para autorização
- **PIX**: Geração de payload dinâmico e confirmação assíncrona
- **Boleto**: Processamento assíncrono com compensação bancária

---

Justificativa para Herança dos Meios de Pagamento

A **herança** é apropriada porque:

1. **Ritual Fixo**: O fluxo `Validar → Autorizar/Capturar → Confirmar` é comum a todos os meios
2. **Especialização Controlada**: Cada meio implementa o **"como"** de cada etapa, mas não altera a sequência
3. **Template Method Natural**:
   - Classe base `Pagamento` define o esqueleto do processo
   - Subclasses (`PagamentoCartao`, `PagamentoPIX`, `PagamentoBoleto`) especializam cada etapa
4. **Manutenção Coesa**: Alterações específicas de cada meio ficam isoladas em suas classes

Exemplo de especialização:

```
// Base
public abstract class Pagamento
{
    public void Processar() // Template Method
    {
        Validar();
        AutorizarCapturar();
        Confirmar();
    }

    protected abstract void Validar();
    protected abstract void AutorizarCapturar();
    protected abstract void Confirmar();
}
```

Justificativa para Composição das Políticas

A **composição** (via delegates) é ideal para políticas porque:

1. **Independência**: Antifraude e câmbio são **ortogonais** ao meio de pagamento

- Um pagamento por PIX pode precisar de câmbio
- Um pagamento por cartão pode precisar de antifraude
- Ambos podem precisar das duas políticas

2. **Combinabilidade Flexível**:

```
var pagamento = new PagamentoCartao();
pagamento.Antifraude = politicaAntifraudeAvançada;
pagamento.Cambio = conversorUSD_BRL;
// Ou nenhuma, ou ambas, ou qualquer combinação
```

3. **Baixo Acoplamento**:
   - Políticas podem evoluir independentemente
   - Novas políticas (ex.: parcelamento, cashback) adicionadas sem modificar hierarquia existente
   - Fácil teste com políticas mock
4. **Open/Closed Principle**:
   - Novas políticas criadas sem alterar código dos meios de pagamento
   - Meios de pagamento novos herdam automaticamente capacidade de usar políticas existentes

---

Conclusão Arquitetural

- **Herança** organiza as **variações essenciais** entre meios de pagamento (cartão/PIX/boleto)
- **Composição** gerencia **funcionalidades transversais** (antifraude, câmbio) que cortam verticalmente todos os meios
- **Separação de Concerns**: O "o quê" (ritual) é fixo na hierarquia, o "como" (implementações) é especializado, e os "extras" (políticas) são compostos

DESIGN:

Contrato da Classe Base `Pagamento`

```
public abstract class Pagamento
{
    // Template Method - Ritual fixo NÃO virtual
    public void Processar()
    {
        Validar();
        AutorizarOuCapturar();
        Confirmar();
    }

    // Ganchos para especialização (protected virtual)
    protected virtual void Validar()
    {
        // Validações mínimas comuns
        // - Valor positivo
        // - Dados do pagador presentes
        // - Meio de pagamento válido
    }

    protected virtual void AutorizarOuCapturar()
    {
        // Implementação base vazia - será sobrescrita
    }

    protected virtual void Confirmar()
    {
        // Implementação base vazia - será sobrescrita
    }

    // Aplicação de políticas plugáveis
    protected bool AplicarAntifraude(decimal valor)
    {
        return _antifraude?.Invoke(valor) ?? true; // Default: aprova se não hou
ver política
    }

    protected decimal AplicarCambio(decimal valor)
    {
```

```
        return _cambio?.Invoke(valor) ?? valor; // Default: mantém valor origina
l
    }

    // Delegates para políticas
    public Func<decimal, bool> Antifraude { set => _antifraude = value; }
    public Func<decimal, decimal> Cambio { set => _cambio = value; }

    private Func<decimal, bool>? _antifraude;
    private Func<decimal, decimal>? _cambio;
}
```

Regras de LSP (Liskov Substitution Principle)

1. Substituibilidade Total

- **Regra**: Qualquer cliente que usa `Pagamento` via `Processar()` deve funcionar identicamente com `PagamentoCartao`, `PagamentoPIX`, `PagamentoBoleto`
- **Garantia**:
  - Cliente trabalha exclusivamente com tipo base `Pagamento`
  - Sem necessidade de `is`, `as` ou `downcast`
  - Exemplo:

```
Pagamento pagamento = new PagamentoCartao(); // Ou PIX, Boleto
pagamento.Processar(); // Comportamento consistente
```

2. Invariantes de Validação Preservadas

- **Regra**: Validações mínimas da base não podem ser enfraquecidas
- **Implementação**:
  - Derivadas podem **adicionar** validações específicas, mas não **remover** as básicas
  - Padrão obrigatório: chamada a `base.Validar()`

```
protected override void Validar()
{
    base.Validar(); // Preserva validações mínimas
    ValidarCVV();   // Validações específicas do cartão
    ValidarLimite();
}
```

3. Contrato de Processamento Consistente

- **Regra**: `Processar()` sempre completa o ciclo e produz resultado coerente
- **Garantias**:
  - Sempre executa as 3 etapas na ordem definida
  - Estado final do pagamento é consistente (autorizado/confirmado ou falha clara)
  - Não introduz exceções inesperadas em relação ao contrato base
  - Comportamentos assíncronos (ex.: boleto) são encapsulados internamente

---

Eixos Plugáveis (Delegates)

Política	Delegate	Assinatura	Papel	Exemplo de Implementação
Antifraude	`Func<decimal, bool>`	`decimal → bool`	Analisa risco do pagamento. Retorna `true` se seguro, `false` se suspeito.	`valor ⇒ valor < 5000.00m`
Cambio	`Func<decimal, decimal>`	`decimal → decimal`	Converte valor entre moedas. Recebe valor original, retorna valor convertido.	`valor ⇒ valor * 5.40m` (USD→BRL)

Outras Políticas Opcionais:
| **Parcelamento** | `Func<decimal, decimal[]>` | `decimal → decimal[]` | Divide valor em

parcelas. Retorna array com valores das parcelas. | `valor => new[] {valor/3, valor/3, valor/3}` |
| **Cashback** | `Func<decimal, decimal>` | `decimal -> decimal` | Calcula benefício de cashback. Retorna valor do cashback. | `valor => valor * 0.05m` (5% de volta) |

Fluxo de Execução com Políticas

```
public class PagamentoCartao : Pagamento
{
    protected override void Validar()
    {
        base.Validar(); // Validações básicas

        // Validações específicas do cartão
        ValidarNumeroCartao();
        ValidarDataValidade();

        // Aplica política de antifraude
        if (!AplicarAntifraude(Valor))
            throw new FraudeDetectadaException();
    }

    protected override void AutorizarOuCapturar()
    {
        decimal valorConvertido = AplicarCambio(Valor);

        // Comunicação com operadora de cartão
        var autorizacao = _gatewayCartao.Autorizar(valorConvertido);
        _gatewayCartao.Capturar(autorizacao);
    }

    protected override void Confirmar()
    {
        // Gera comprovante com código de autorização
        _comprovante = new ComprovanteCartao(_dadosTransacao);
    }
}
```

Exemplo de Uso das Políticas

```
// Configuração flexível de políticas
var pagamento = new PagamentoCartao();

// Política de antifraude personalizada
pagamento.Antifraude = valor ⇒ {
    if (valor > 10000m) return false; // Bloqueia valores muito altos
    if (valor < 1m) return false;     // Bloqueia valores muito baixos
    return true;                      // Aprova os demais
};

// Política de câmbio USD para BRL
pagamento.Cambio = valor ⇒ valor * 5.40m;

// Processa com políticas aplicadas
pagamento.Processar(); // Agnóstico às políticas específicas
```

Vantagens do Design

1. **Extensibilidade**: Novos meios de pagamento herdam o ritual automaticamente
2. **Flexibilidade**: Políticas podem ser combinadas dinamicamente
3. **Manutenibilidade**: Alterações em políticas não afetam meios de pagamento
4. **Testabilidade**: Fácil mock de políticas para testes unitários
5. **Coesão**: Cada classe tem responsabilidade bem definida