JUSTIFICATIVA DAS ESCOLHAS DE DESIGN PATTERNS

Documento de Decisões de Arquitetura

Disciplina: Padrões de Projeto
Aluno: Nicolas
Data: 09 de Novembro de 2025

QUESTÃO 1: POR QUE STRATEGY PATTERN?

Contexto do Problema

Uma empresa financeira precisa calcular métricas de risco usando diferentes algoritmos (Value at Risk, Expected Shortfall, Stress Testing) que podem mudar dinamicamente durante a execução do sistema.

Por Que Strategy?

1. Intercambialidade de Algoritmos em Runtime

- O Strategy permite trocar o algoritmo de cálculo sem recompilar ou alterar o código cliente
- O sistema pode decidir qual algoritmo usar baseado em regras de negócio, condições de mercado ou preferências do usuário
- É possível mudar de VaR para Stress Testing com uma única chamada de método

2. Encapsulamento de Variações de Comportamento

- Cada algoritmo de risco tem sua própria lógica complexa e específica
- Sem Strategy, teríamos um único método gigante com vários if/else ou switch/case para escolher o algoritmo
- Com Strategy, cada algoritmo fica em sua própria classe, isolado e focado

3. Compartilhamento de Contexto Complexo

- Os algoritmos precisam de múltiplos parâmetros financeiros (portfolio, volatilidade, histórico, taxa de juros, mercado)
- O Strategy permite passar um objeto de contexto único contendo todos esses parâmetros

- Evita métodos com 5, 10 ou mais parâmetros

4. Open/Closed Principle

- Novos algoritmos de risco podem ser adicionados sem modificar código existente
- Basta criar uma nova classe implementando a interface do Strategy
- O cliente não precisa ser alterado para usar novos algoritmos

5. Testabilidade

- Cada algoritmo pode ser testado isoladamente sem depender dos outros
- Bugs em um algoritmo não afetam os demais
- Facilita testes unitários com diferentes estratégias

Alternativas Descartadas

Por que NÃO usar Template Method?

- Template Method requer herança, o que é mais rígido
- Com Strategy, podemos mudar o algoritmo em runtime; com Template Method, o algoritmo é fixo na criação do objeto

Por que NÃO usar simples herança?

- Herança cria acoplamento forte entre classes
- Não permite trocar algoritmo depois que o objeto é criado
- Viola o princípio "favorecer composição sobre herança"

QUESTÃO 2: POR QUE ADAPTER PATTERN?

Contexto do Problema

Integração com um sistema bancário legado que possui interface complexa e incompatível. O legado usa HashMap com tipos Object e codificação específica de moedas, enquanto o sistema moderno usa tipos fortemente tipados e strings.

Por Que Adapter?

1. Incompatibilidade de Interfaces

- A interface moderna usa métodos tipados: autorizar(String cartao, double valor, String moeda)
- O legado usa métodos genéricos: processarTransacao(HashMap<String, Object> parametros)
- São interfaces fundamentalmente incompatíveis que não podem se comunicar diretamente

2. Impossibilidade de Modificar o Legado

- O sistema legado está em produção e não pode ser alterado (risco muito alto)
- Pode ser de terceiros ou sem acesso ao código-fonte
- Alterar o legado impactaria outros sistemas que já o usam

3. Conversão de Dados e Formatos

- Moedas: "USD" → 1, "EUR" → 2, "BRL" → 3
- Respostas: STATUS_PROC numérico → boolean de sucesso
- Campos obrigatórios: adicionar COD_BANDEIRA que não existe na interface moderna

4. Bidirecionaldiade

- Não é apenas chamar o legado, mas também interpretar suas respostas
- Converte requisições modernas em formato legado
- Converte respostas legado em formato moderno
- Atua como tradutor bidirecional

5. Centralização da Lógica de Integração

- Toda a lógica complexa de conversão fica em um único lugar (o Adapter)
- Se o legado mudar códigos de moeda, só o Adapter precisa ser atualizado
- Facilita manutenção e debugging

6. Transparência para o Cliente

- O código cliente usa apenas a interface moderna
- Não sabe (nem precisa saber) que por trás existe um sistema legado complexo
- Isso permite eventualmente substituir o legado sem afetar o cliente

Alternativas Descartadas

Por que NÃO usar Facade?

- Facade simplifica interfaces complexas, mas não resolve incompatibilidade de tipos
- Não faz conversão de dados (String → Integer, etc.)

- Facade é para simplificar, Adapter é para compatibilizar

Por que NÃO modificar o legado?

- Risco operacional muito alto
- Pode quebrar outros sistemas que dependem dele
- Pode não ter acesso ao código-fonte

QUESTÃO 3: POR QUE STATE PATTERN?

Contexto do Problema

Sistema de controle para usina nuclear com 5 estados (DESLIGADA, OPERACAO*NORMAL, ALERTA*AMARELO, ALERTA_VERMELHO, EMERGENCIA) e transições complexas baseadas em múltiplas condições de segurança.

Por Que State?

1. Comportamento Dependente de Estado

- O sistema age completamente diferente dependendo do estado atual
- Em OPERACAO_NORMAL: monitora temperatura normalmente
- Em ALERTA_VERMELHO: ativa protocolos de emergência, mobiliza equipes
- Em EMERGENCIA: inicia evacuação, contenção de radiação
- Cada estado tem seu próprio conjunto de comportamentos

2. Transições Complexas e Condicionais

- OPERACAO*NORMAL* → *ALERTA*AMARELO: só se temperatura > 300°C
- ALERTA*AMARELO* → *ALERTA*VERMELHO: temperatura > 400°C **por mais de 30 segundos**
- ALERTA_VERMELHO → EMERGENCIA: só se sistema de resfriamento **falhar**
- Sem State, essas regras ficariam espalhadas em vários lugares

3. Validação de Transições (Segurança Crítica)

- Algumas transições são proibidas por segurança
- EMERGENCIA só pode ser ativada após passar por ALERTA_VERMELHO (não pode pular estados)
- Transições circulares devem ser bloqueadas
- Cada estado valida se pode transicionar para outro estado específico

4. Eliminação de Condicionais Gigantes

- Sem State Pattern, teríamos algo como:
- if (estado == DESLIGADA) { ... } else if (estado == OPERACAO_NORMAL) { ... } else if (estado == ALERTA_AMARELO) { ... }
- Com 5 estados e múltiplas operações, isso se torna ingerenciável
- State Pattern elimina esses condicionais, cada estado é uma classe

5. Facilidade de Adicionar Estados

- Adicionar um novo estado (ex: MANUTENCAO_EMERGENCIAL) é simples
- Basta criar uma nova classe de estado
- Não precisa modificar os estados existentes

6. Modo Manutenção Especial

- Modo manutenção precisa sobrescrever validações normais
- Precisa "lembrar" o estado anterior para restaurá-lo depois
- State Pattern permite criar estados especiais com comportamentos únicos

7. Rastreabilidade e Auditoria

- Métodos entrar() e sair() em cada estado permitem logging detalhado
- Importante para sistemas críticos como usinas nucleares
- Facilita investigação de incidentes

Alternativas Descartadas

Por que NÃO usar enum com switch?

- Enums com switch centralizam lógica em um único lugar
- Adicionar novo estado requer modificar o switch (viola Open/Closed)
- Dificulta manutenção conforme estados crescem

Por que NÃO usar variáveis booleanas?

- Com 5 estados, precisaríamos muitas flags booleanas
- Combinações inválidas seriam possíveis (ex: estar em NORMAL e EMERGENCIA simultaneamente)
- Muito propenso a erros

QUESTÃO 4: POR QUE CHAIN OF RESPONSIBILITY?

Contexto do Problema

Sistema de validação de NF-e com 5 validadores especializados que devem processar em sequência, com suporte a validações condicionais, circuit breaker, rollback e timeouts individuais.

Por Que Chain of Responsibility?

1. Processamento Sequencial com Múltiplos Handlers

- Documento precisa passar por 5 validações diferentes em ordem específica
- Cada validador é especializado em um aspecto (XML, Certificado, Impostos, Banco, SEFAZ)
- A ordem importa: não adianta validar regras fiscais se XML está malformado

2. Desacoplamento entre Validadores

- Cada validador não sabe quem é o próximo na cadeia
- Validador de XML não precisa conhecer o Validador de Certificado
- Isso permite reordenar, adicionar ou remover validadores facilmente
- Facilita testes unitários isolados

3. Validações Condicionais

- Validador de Regras Fiscais (3) só executa se XML e Certificado passaram
- Validador SEFAZ (5) só executa se todos anteriores passaram
- Sem Chain, precisaríamos verificar condições manualmente em cada passo
- Com Chain, cada validador decide se passa adiante ou interrompe

4. Circuit Breaker Pattern Integrado

- Após 3 falhas consecutivas, não faz sentido continuar
- O circuit breaker "abre" e interrompe a cadeia automaticamente
- Economiza recursos e falha rápido (fail fast)
- Evita tentar validar com SEFAZ se já houve 3 erros anteriores

5. Capacidade de Rollback

- Validador de Banco de Dados insere registro
- Se validadores posteriores (SEFAZ) falharem, o BD precisa fazer rollback
- Cada validador pode implementar sua própria lógica de rollback
- Mantém consistência dos dados

6. Timeout Individual por Validador

- Validação de XML: 5 segundos (rápida, local)
- Validação SEFAZ: 10 segundos (lenta, chamada externa)
- Cada validador pode ter timeout apropriado ao seu contexto
- Previne que um validador lento trave todo o sistema

7. Responsabilidade Distribuída

- Cada validador é responsável por um aspecto específico
- Single Responsibility Principle aplicado
- Facilita manutenção (bug no validador de impostos não afeta validador de XML)
- Facilita substituição (trocar validador de certificado por outro)

8. Pipeline de Processamento

- O documento é "enriquecido" conforme passa pela cadeia
- Validador XML adiciona conteúdo XML parseado
- Validador Fiscal adiciona cálculo de impostos
- Validador BD adiciona ID do banco
- Ao final, documento está completo e validado

Alternativas Descartadas

Por que NÃO usar uma classe Validador única?

- Violaria Single Responsibility (uma classe fazendo 5 coisas)
- Difícil de testar (teria que testar tudo junto)
- Difícil de manter (mudança em uma validação afeta todas)

Por que NÃO usar métodos sequenciais?

- Acoplaria validadores entre si
- Difícil adicionar nova validação no meio da sequência
- Difícil implementar lógica condicional e rollback

Por que NÃO usar Observer Pattern?

- Observer é para notificações assíncronas
- Aqui precisamos de processamento sequencial síncrono
- Observer não garante ordem de execução

PRINCÍPIOS APLICADOS EM TODAS AS ESCOLHAS

1. Open/Closed Principle

- **Strategy**: Adicionar novos algoritmos sem modificar código existente
- **Adapter**: Mudar legado sem afetar código moderno
- **State**: Adicionar novos estados sem modificar estados existentes
- **Chain**: Adicionar novos validadores sem modificar validadores existentes

2. Single Responsibility Principle

- **Strategy**: Cada algoritmo em sua própria classe
- **Adapter**: Conversão de dados centralizada em um único lugar
- **State**: Cada estado com sua própria lógica
- **Chain**: Cada validador responsável por um aspecto

3. Dependency Inversion Principle

- Todos os padrões dependem de abstrações (interfaces), não de implementações concretas
- Permite substituir implementações sem afetar clientes

4. Liskov Substitution Principle

- **Strategy**: Qualquer estratégia pode substituir outra
- **State**: Qualquer estado pode substituir outro
- **Chain**: Qualquer validador pode substituir outro na cadeia

5. Interface Segregation Principle

- Interfaces pequenas e focadas
- Clientes não dependem de métodos que não usam

CONCLUSÃO

Cada padrão foi escolhido porque resolve exatamente o problema específico apresentado:

Strategy □ Para quando você tem **variações de algoritmo** que precisam ser **intercambiáveis**

Adapter → Para quando você precisa **integrar sistemas incompatíveis** sem **modificar código existente**

State → Para quando o **comportamento muda baseado em estado interno** com **transições complexas**

Chain of Responsibility → Para quando você tem **múltiplos processadores** em **sequência** com **lógica condicional**

Não foram escolhas arbitrárias, mas sim análise técnica dos requisitos e seleção do padrão que melhor se adequa a cada cenário, seguindo princípios de engenharia de software e boas práticas da indústria.

Elaborado por: Nicolas
Data: 09 de Novembro de 2025
Ferramenta: Inner AI - Claude Sonnet 4.5