

Utilização de padrões de projeto clássicos no contexto de desenvolvimento de software

Aplicação prática em cenários reais de desenvolvimento

1. Padrões clássicos no contexto real de desenvolvimento

Conhecer a teoria dos padrões de projeto clássicos, catalogados originalmente pela Gang of Four (GoF), é apenas o primeiro passo. O verdadeiro valor desses padrões aparece quando são aplicados em situações reais de desenvolvimento: ao refatorar um sistema legado que cresceu de forma desorganizada, ao integrar bibliotecas de terceiros, ao lidar com múltiplas variações de um mesmo processo de negócio ou ao construir frameworks que outras equipes irão utilizar.

Este texto apresenta padrões clássicos adicionais, complementares aos mais comumente citados, sempre associados a cenários concretos do dia a dia de desenvolvimento de software.

2. Padrões clássicos aplicados a cenários reais

2.1 Builder: construção de objetos complexos passo a passo

O padrão Builder separa a construção de um objeto complexo de sua representação final, permitindo criar diferentes configurações do mesmo tipo de objeto por meio de um processo passo a passo, em vez de construtores com dezenas de parâmetros.

Contexto de uso: em um sistema de geração de relatórios, um objeto `RelatorioFinanceiro` pode ter cabeçalho, filtros, gráficos e rodapé opcionais. Em vez de um construtor com dez parâmetros (muitos deles opcionais), a equipe usa um `RelatorioBuilder` que permite encadear chamadas como `.comCabecalho().comGrafico().comRodape().construir()`, tornando o código muito mais legível.

2.2 Facade: simplificando o acesso a subsistemas complexos

O padrão Facade cria uma interface simplificada para um conjunto de classes ou um subsistema complexo, escondendo detalhes internos e facilitando o uso por parte de outras partes do sistema.

Contexto de uso: ao integrar um sistema de emissão de notas fiscais eletrônicas, que exige assinatura digital, validação de schema XML e comunicação com o webservice da Receita Federal, a equipe cria uma classe `FacadeNotaFiscal` que expõe apenas o método `emitir(pedido)`, escondendo toda a complexidade técnica dos módulos internos.

2.3 Composite: tratando objetos individuais e composições de forma uniforme

O padrão Composite permite tratar objetos individuais e grupos de objetos (compostos por outros objetos, possivelmente recursivamente) de maneira uniforme, por meio de uma interface comum.

Contexto de uso: em um sistema de gestão de arquivos, tanto um Arquivo individual quanto uma Pasta (que pode conter outros arquivos e pastas) implementam a mesma interface *ElementoDoSistema*, com um método *calcularTamanho()*. Isso permite calcular o tamanho total de uma pasta somando recursivamente o tamanho de tudo que ela contém, sem tratar arquivos e pastas de forma diferente no código cliente.

2.4 Proxy: controlando o acesso a um objeto

O padrão Proxy cria um objeto substituto que controla o acesso a outro objeto, podendo adicionar comportamentos como controle de permissão, cache ou carregamento tardio (lazy loading), sem que o código cliente perceba a diferença.

Contexto de uso: em um sistema com imagens de alta resolução em um catálogo de produtos, um ProxyImagem é usado para adiar o carregamento pesado da imagem real até o momento em que ela realmente precisa ser exibida na tela, melhorando o desempenho percebido pelo usuário.

2.5 Template Method: definindo o esqueleto de um algoritmo

O padrão Template Method define, em uma classe base, o esqueleto de um algoritmo com etapas fixas, deixando que subclasses especializem apenas os passos que variam, sem alterar a estrutura geral do processo.

Contexto de uso: em um sistema de processamento de pedidos, a classe abstrata *ProcessadorDePedido* define o fluxo geral (validar, calcular total, processar pagamento, confirmar), enquanto subclasses como *ProcessadorPedidoNacional* e *ProcessadorPedidoInternacional* sobrescrevem apenas o passo de cálculo de impostos, que muda conforme o destino da entrega.

2.6 Chain of Responsibility: encadeando manipuladores de uma requisição

O padrão Chain of Responsibility organiza uma sequência de objetos manipuladores, permitindo que uma requisição passe por essa cadeia até que algum deles seja capaz de tratá-la, sem que o solicitante precise saber qual objeto específico fará o tratamento.

Contexto de uso: em um sistema de aprovação de despesas corporativas, uma solicitação passa por uma cadeia de aprovadores — Supervisor, Gerente e Diretor — cada um autorizado a aprovar valores até um determinado limite; se o valor ultrapassar o limite do Supervisor, a solicitação é automaticamente encaminhada ao próximo nível da cadeia.

3. Resumo dos padrões e seus cenários de uso

Padrão	Cenário típico de uso
Builder	Construção passo a passo de objetos com muitas opções
Facade	Simplificar o acesso a integrações ou subsistemas complexos
Composite	Tratar itens individuais e coleções de forma uniforme
Proxy	Controlar acesso, cache ou carregamento tardio de recursos

Template Method	Definir um fluxo fixo com etapas específicas variáveis
Chain of Responsibility	Encadear múltiplos possíveis responsáveis por tratar uma solicitação

4. Padrões clássicos presentes em ferramentas do dia a dia

Muitos desenvolvedores utilizam padrões de projeto clássicos sem perceber, pois eles já estão embutidos em frameworks e bibliotecas amplamente usadas:

- Bibliotecas de construção de interfaces (como builders de layout em frameworks mobile) frequentemente aplicam o padrão Builder para configurar componentes visuais de forma encadeada.
- Frameworks de injeção de dependência aplicam princípios próximos ao Facade, escondendo a complexidade da criação e conexão de diversos objetos do sistema.
- Sistemas de arquivos e interfaces de árvore (como menus e estruturas de pastas em aplicativos) aplicam o padrão Composite para representar hierarquias de forma uniforme.
- Middlewares em frameworks web (que processam uma requisição HTTP por várias camadas antes de chegar à lógica da aplicação) seguem a mesma ideia do Chain of Responsibility.

5. Padrões clássicos como ferramenta de refatoração

Além de serem usados no projeto de novos sistemas, os padrões clássicos são frequentemente aplicados durante a refatoração de código legado, quando a equipe percebe sinais de problemas estruturais, conhecidos como "maus cheiros de código" (code smells).

Exemplo prático: ao encontrar um método com uma longa cadeia de estruturas condicionais (*if/else*) para tratar diferentes tipos de desconto em um sistema de vendas, a equipe refatora o código aplicando o padrão Strategy, criando uma classe separada para cada tipo de desconto e eliminando a necessidade de alterar esse método sempre que uma nova regra de desconto for criada.

6. Aplicação prática no dia a dia dos projetos

A utilização consciente de padrões clássicos no contexto real de desenvolvimento traz benefícios concretos para diferentes situações:

- **Times que mantêm sistemas legados** utilizam padrões como Facade e Adapter para isolar partes antigas e problemáticas do código, permitindo modernizar o sistema aos poucos, sem reescrevê-lo do zero.
- **Equipes que constroem bibliotecas ou frameworks internos** aplicam Builder e Template Method para oferecer APIs mais claras e flexíveis para outras equipes que irão consumir esse código.
- **Times de plataforma e infraestrutura** utilizam Proxy e Chain of Responsibility ao lidar com questões transversais, como autenticação, cache e balanceamento de carga.

- **Desenvolvedores em processos de revisão de código (code review)** reconhecem oportunidades de aplicar padrões clássicos ao identificar código duplicado, condicionais excessivos ou classes com responsabilidades demais.

7. Considerações finais

Padrões de projeto clássicos deixam de ser apenas conceitos teóricos quando aplicados a situações reais de desenvolvimento: integração de sistemas externos, manutenção de código legado, construção de frameworks reutilizáveis e refatoração de código problemático. Reconhecer o contexto certo para aplicar cada padrão — em vez de simplesmente memorizar sua estrutura — é o que realmente diferencia desenvolvedores capazes de criar soluções elegantes, flexíveis e sustentáveis ao longo do tempo.