CAPÍTULO 6 - PADRÕES DE PROJETO

Os Padrões de Projeto foram inspirados por Cristopher Alexander, que criou padrões para a construção de cidades e prédios. Em 1995, quatro autores adaptaram essa ideia para o desenvolvimento de software, criando soluções para problemas comuns que acontecem em projetos de software. Esses padrões ajudam a criar sistemas mais flexíveis e fáceis de modificar, além de facilitar a comunicação entre os desenvolvedores, pois eles usam o mesmo vocabulário. Eles podem ser usados tanto ao criar novos sistemas quanto ao entender sistemas de terceiros.

Os padrões são divididos em três categorias:

- Criacionais: Como criar objetos de maneira flexível (ex.: Factory Method, Singleton).
- Estruturais: Como organizar classes e objetos (ex.: Adapter, Decorator).
- Comportamentais: Como gerenciar a interação entre objetos (ex.: Strategy, Observer).

Fábrica

Contexto: Em um sistema distribuído baseado em TCP/IP, três funções criam objetos do tipo TCPChannel para comunicação remota.

Problema: Se precisar usar UDP em vez de TCP, o sistema não está preparado para isso. A criação de objetos dos tipos TCPChannel ou UDPChannel precisa ser parametrizada, mas o operador "new" não permite passar a classe como parâmetro.

Solução: O padrão Fábrica resolve esse problema criando um método estático que cria e retorna objetos de uma classe específica, ocultando o tipo desses objetos atrás de uma interface. As funções não precisam saber qual tipo de Channel estão criando, e o código fica mais flexível. Se precisar mudar o tipo de canal, basta alterar o método na classe Fábrica.

Singleton

Contexto: Considerando uma classe Logger usada para registrar operações, cada método no sistema cria sua própria instância de Logger, como mostrado nos exemplos de código.

Problema: O problema surge porque, com várias instâncias de Logger, pode ocorrer uma proliferação de objetos e, em situações como o registro de eventos em arquivos, o novo objeto pode sobrescrever dados de objetos anteriores. O objetivo é garantir que exista apenas uma instância de Logger para todo o sistema.

Solução: A solução é transformar a classe Logger em um Singleton, ou seja, garantir que ela tenha apenas uma instância durante a execução do programa. Para isso, o construtor da classe é privado, impedindo a criação de novas instâncias fora da classe. A instância única é armazenada em um atributo estático e é acessada por meio de um método estático getInstance(). Com isso, todos os métodos podem acessar a mesma instância de Logger, garantindo consistência.

Embora o padrão Singleton seja útil para modelar recursos exclusivos, como arquivos de log, ele é criticado por criar acoplamento forte entre classes e por dificultar testes automáticos, já que o estado global de um Singleton pode afetar o comportamento dos métodos.

Proxy

Contexto: Suponha uma classe BookSearch, que tem um método para pesquisar por um livro dado o seu ISBN.

Problema: O serviço de pesquisa de livros se torna popular e precisa ser otimizado com um sistema de cache. Antes de realizar uma nova pesquisa, o sistema deve verificar se o livro já está no cache, mas queremos evitar modificar a classe BookSearch para implementar o cache, pois ela deve se concentrar apenas na pesquisa. Além disso, o cache será implementado por uma equipe diferente, utilizando uma biblioteca externa.

Solução: O padrão de projeto Proxy propõe o uso de um objeto intermediário, chamado proxy, entre o objeto base e seus clientes. O proxy mediará o acesso à classe base, agregando funcionalidades como o cache, sem que o objeto base tenha que saber disso. O proxy implementa a mesma interface da classe base, permitindo que os clientes usem o proxy como se estivessem acessando o objeto base diretamente.

O proxy também pode ser usado para outras funcionalidades não-funcionais, como:

- Comunicação com um cliente remoto (proxies chamados stubs).
- Alocação de memória sob demanda para objetos grandes, como imagens em alta resolução.
- Controle de acesso, garantindo que apenas clientes autenticados possam acessar certos métodos do objeto base.

Adaptador

Contexto: Suponha um sistema que precisa controlar projetores multimídia. Ele deve instanciar objetos de classes fornecidas pelos fabricantes, como o ProjetorSamsung e ProjetorLG.

Problema: O sistema de controle de projetores precisa de uma interface única para ligar os projetores, mas as classes de cada projetor têm métodos diferentes e não implementam a interface comum Projetor. Além disso, não temos acesso ao código dessas classes para modificá-las.

Solução: O padrão de projeto Adaptador é usado para converter a interface de uma classe para outra interface esperada pelos seus clientes. Nesse caso, ele converte a interface Projetor (usada no sistema de controle) para as interfaces específicas dos projetores Samsung e LG.

Fachada

Contexto: Suponha que implementamos um interpretador para uma linguagem X, permitindo a execução de programas X a partir de Java. Para executar programas X, são necessários vários passos, como usar um scanner, parser, gerar um AST e finalmente executar o código.

Problema: A linguagem X está ganhando popularidade, mas os desenvolvedores reclamam da complexidade do código atual, pois precisam entender as classes internas do interpretador para executar programas. Eles buscam uma interface mais simples para interagir com o interpretador.

Solução: O padrão de projeto Fachada resolve esse problema ao fornecer uma interface simplificada. A Fachada é uma classe que encapsula toda a complexidade do sistema e expõe apenas um método simples para o usuário.

Decorador

Contexto: Em um sistema de comunicação remota, temos canais de comunicação como TCP e UDP. Esses canais implementam uma interface comum que permite enviar e receber mensagens.

Problema: Os clientes precisam adicionar funcionalidades extras aos canais, como buffers, compactação das mensagens ou log das mensagens. Uma abordagem com herança para adicionar essas funcionalidades resultaria em muitas subclasses, criando uma explosão de combinações de funcionalidades.

Solução: O padrão Decorador oferece uma solução flexível usando composição em vez de herança. Com esse padrão, funcionalidades extras podem ser adicionadas dinamicamente a um objeto.

Strategy

Contexto: Suponha que estamos implementando um pacote de estruturas de dados, com uma classe de lista que utiliza o algoritmo Quicksort para ordenar seus elementos.

Problema: Os clientes querem a opção de definir diferentes algoritmos de ordenação. Atualmente, a lista usa apenas o Quicksort, o que impede a flexibilidade e não segue o princípio Aberto/Fechado.

Solução: O padrão Strategy resolve o problema ao permitir a escolha do algoritmo de ordenação sem modificar o código da classe MyList. O algoritmo de ordenação é parametrizado, ou seja, ele é definido por uma estratégia que pode ser alterada dinamicamente.

Observador

Contexto: Imagine um sistema de estação meteorológica que possui objetos como Temperatura, representando os dados do modelo, e Termometro, que exibe visualmente essas temperaturas.

Problema: Não queremos acoplar diretamente as classes de modelo (como Temperatura) com as classes de interface (como Termometro), pois as interfaces podem mudar com frequência, e queremos reusar o mecanismo de notificação entre diferentes pares de classe.

Solução: O padrão Observador resolve esse problema ao estabelecer uma relação de um-para-muitos entre os objetos. Quando o estado de um sujeito (como Temperatura) muda, todos os observadores registrados (como Termometros) são notificados automaticamente. O sujeito (Temperatura) mantém uma lista de observadores e os notifica através de um método, enquanto os observadores (como TermometroCelsius) implementam uma interface para receber essas atualizações.

Template Method

Contexto: Um sistema de folha de pagamento possui uma classe base Funcionario com subclasses como FuncionarioPublico e FuncionarioCLT.

Problema: É necessário padronizar o cálculo do salário na classe base, mas permitindo que as subclasses implementem os detalhes específicos.

Solução: O padrão Template Method define o esqueleto do algoritmo na classe base, deixando alguns passos como métodos abstratos para que as subclasses os implementem. Assim, o algoritmo geral permanece inalterado, mas cada tipo de funcionário pode personalizar seu próprio cálculo de salário.

Visitor

Contexto: Em um sistema de estacionamento, veículos como Carro, Onibus e Motocicleta são armazenados juntos.

Problema: É preciso realizar operações (como imprimir informações ou salvar dados) em todos os veículos sem acoplar essas operações às classes dos veículos.

Solução: O padrão Visitor permite adicionar operações externas à hierarquia de veículos. Cada veículo implementa um método que aceita um visitante, que possui métodos específicos para cada tipo de veículo. Isso simula o despacho duplo, possibilitando que a operação correta seja executada conforme o tipo real do veículo, sem alterar as classes originais.

Iterador: Padrão que facilita percorrer estruturas de dados sem conhecer seu tipo, usando métodos como hasNext() e next().

Builder: Padrão que ajuda a criar objetos com muitos atributos, evitando construtores complicados e oferecendo um processo claro para definir atributos.

Padrões de projeto devem ser usados com cautela, pois podem adicionar complexidade desnecessária. Antes de adotá-los, é importante avaliar se realmente resolvem o problema ou se uma solução mais simples seria melhor.