Padrões GoF (Gang of Four) e Aplicabilidade no CRUD de Pedidos

---

1. O que é GoF?

Os padrões de projeto GoF foram definidos pelos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides no livro "Design Patterns: Elements of Reusable Object-Oriented Software". Esses padrões são soluções comprovadas para problemas comuns encontrados no desenvolvimento de software orientado a objetos. Eles são divididos em três categorias principais:

- **Criacionais**: Focam na criação de objetos.
- **Estruturais**: Tratam da composição de classes e objetos.
- **Comportamentais**: Definem a interação e responsabilidade entre objetos.

---

2. Os 23 Padrões de Projeto GoF

2.1 Padrões Criacionais

1. **Factory Method**: Permite a criação de objetos sem especificar a classe exata que será instanciada.
2. **Abstract Factory**: Cria grupos de objetos relacionados sem especificar suas classes concretas.
3. **Builder**: Constrói objetos complexos passo a passo.
4. **Prototype**: Cria novos objetos clonando um modelo existente.
5. **Singleton**: Garante que apenas uma instância de uma classe seja criada.

2.2 Padrões Estruturais

6. **Adapter**: Permite a comunicação entre interfaces incompatíveis.
7. **Bridge**: Separa uma abstração de sua implementação.
8. **Composite**: Permite trabalhar com estruturas hierárquicas de objetos como se fossem individuais.
9. **Decorator**: Adiciona funcionalidades dinamicamente a objetos.
10. **Facade**: Fornece uma interface simples para um sistema complexo.
11. **Flyweight**: Otimiza o uso da memória compartilhando objetos comuns.
12. **Proxy**: Atua como substituto ou intermediário para outro objeto.

2.3 Padrões Comportamentais

13. **Chain of Responsibility**: Permite que várias classes processem uma requisição sem especificar explicitamente o manipulador.

14. **Command**: Encapsula uma solicitação como um objeto, permitindo fila de solicitações e reversibilidade de ações.
15. **Interpreter**: Define uma gramática para interpretar expressões.
16. **Iterator**: Fornece uma maneira sequencial de acessar elementos de uma coleção sem expor sua representação interna.
17. **Mediator**: Define um objeto central para controlar a comunicação entre vários objetos.
18. **Memento**: Permite salvar e restaurar estados de objetos sem violar seu encapsulamento.
19. **Observer**: Permite que objetos sejam notificados automaticamente quando outro objeto é modificado.
20. **State**: Permite que um objeto altere seu comportamento quando seu estado muda.
21. **Strategy**: Permite selecionar um algoritmo em tempo de execução.
22. **Template Method**: Define um esqueleto de algoritmo e permite que subclasses implementem partes dele.
23. **Visitor**: Permite adicionar novas operações a uma estrutura de classes sem modificar essas classes.

---

3. Aplicabilidade ao CRUD de Pedidos

Para o projeto de CRUD de pedidos, selecionamos 10 padrões que se encaixam melhor:

1. **Factory Method**:
    a. Sempre que o usuário fizer um novo pedido, podemos criar instâncias de diferentes tipos de pedido sem precisar especificar diretamente qual classe usar.
    b. Exemplo: Um método `PedidoFactory.criarPedido(tipo)` poderia criar pedidos de diferentes categorias (ex.: normal, agendado, urgente).
2. **Builder**:
    a. Para permitir que o usuário monte um pedido personalizado, adicionando pratos, escolhendo quantidade e definindo o método de pagamento.
    b. Exemplo: Um `PedidoBuilder` poderia adicionar itens um a um e construir um objeto final com todas as informações antes da confirmação.
3. **Singleton**:
    a. Garante que exista apenas uma conexão com o banco de dados SQLite, evitando múltiplas instâncias desnecessárias.
    b. Exemplo: Uma classe `BancoDeDados` com um método `getInstance()` que retorna sempre a mesma conexão ativa.
4. **Adapter**:
    a. Facilita a integração com APIs externas, como serviços de pagamento ou de localização para entrega.
    b. Exemplo: Um `PagamentoAdapter` poderia padronizar chamadas para diferentes serviços, como PayPal, Stripe ou um saldo virtual no app.
5. **Facade**:

    a. Encapsula a lógica do CRUD de pedidos em uma interface simplificada, evitando que a interface do usuário precise lidar com detalhes internos.
    b. Exemplo: Uma classe `PedidoFacade` com métodos como `criarPedido()`, `listarPedidos()`, `cancelarPedido()`.
6. **Observer**:
    a. Mantém a interface do usuário sempre atualizada conforme o status do pedido muda.
    b. Exemplo: Quando um pedido é confirmado, um evento é disparado para atualizar a tela do usuário.
7. **Command**:
    a. Permite que ações como "cancelar pedido" ou "refazer pedido" sejam tratadas como objetos independentes, facilitando o controle e até o desfazer de ações.
    b. Exemplo: Um `CancelarPedidoCommand` pode ser armazenado em um histórico e revertido se necessário.
8. **Strategy**:
    a. Permite trocar dinamicamente o método de pagamento sem modificar a lógica principal do pedido.
    b. Exemplo: Uma interface `PagamentoStrategy` com implementações como `PagamentoCartao`, `PagamentoPix`, `PagamentoSaldo`.
9. **State**:
    a. Gerencia os diferentes estados de um pedido (pendente, pago, em preparo, entregue).
    b. Exemplo: Um objeto `Pedido` poderia mudar de comportamento conforme seu estado muda.
10. **Template Method**:

- Define um fluxo fixo para a criação e finalização do pedido, mas permite pequenas variações.
- Exemplo: Um método `finalizarPedido()` que segue sempre os mesmos passos, mas pode ser personalizado para diferentes tipos de pedidos.

11.

Esses padrões garantirão que o sistema seja modular, flexível e fácil de manter.