**DESING PATTERNS**

Marcelo Alves Barbosa

Sistemas de Informação

Programação Orientada a Objetos II  
Professor: Yassushi Miguel Ávila Okada

Uberlândia - MG  
2025

1.**Singleton**

O padrão Singleton é uma maneira de garantir que uma classe tenha apenas uma única instância, além de fornecer um ponto de acesso global a essa instância.

Existem dois problemas principais a serem considerados: primeiro, é preciso assegurar que só exista uma instância da classe, o que é especialmente útil quando se está controlando o acesso a um recurso compartilhado, como uma base de dados. Em segundo lugar, é necessário oferecer um acesso global a essa instância, algo que se assemelha ao uso de variáveis globais, mas de uma forma mais controlada.

A solução para isso envolve algumas etapas: primeiro, o construtor da classe deve ser privado para evitar que alguém crie instâncias diretamente usando o operador new. Em seguida, é preciso criar um método estático, como o getInstance(), que, nas primeiras chamadas, cria a instância e, nas chamadas subsequentes, retorna sempre o objeto já existente

O padrão Singleton é ideal quando uma classe precisa ter apenas uma instância acessível a todos os clientes, como no caso de um objeto de banco de dados que é compartilhado entre diferentes partes do programa.

Entre os prós e contras, podemos destacar que ele garante uma instância única, oferece acesso global e permite a inicialização sob demanda. Por outro lado, ele pode violar o princípio de responsabilidade única, pode esconder designs ruins, requer atenção em ambientes multithread e pode dificultar os testes unitários.

Em relação a outros padrões, uma classe fachada, por exemplo, pode frequentemente ser transformada em Singleton, já que geralmente só precisamos de um único objeto fachada. O padrão Flyweight pode parecer semelhante ao Singleton se você reduzir todos os estados compartilhados a um único objeto, mas existem diferenças fundamentais, como a questão da mutabilidade.

**2. Facade**

O padrão Facade tem como propósito central fornecer uma interface unificada e simplificada para um subsistema complexo, seja ele uma biblioteca, um framework ou um conjunto de classes interdependentes. A motivação para sua aplicação surge quando o código cliente necessita interagir com um subsistema que expõe múltiplas classes, exigindo uma ordem específica de inicialização e tratamento de dependências. Tal cenário resulta em um forte acoplamento do cliente aos detalhes de implementação do subsistema, o que eleva a complexidade e dificulta a manutenção do código.

A solução proposta pelo padrão é a introdução de uma classe "fachada", que atua como um ponto de entrada único, encapsulando a complexidade do subsistema. Esta classe delega as chamadas do cliente para os objetos apropriados dentro do subsistema, expondo apenas a funcionalidade estritamente necessária ao cliente.

Sua aplicabilidade é notável em duas situações principais: quando se deseja fornecer uma interface limitada, porém simplificada, para um subsistema complexo; ou ao estruturar um sistema em camadas, onde o Facade define pontos de entrada claros para cada camada. O principal benefício é o isolamento do código cliente da complexidade interna do subsistema, promovendo o baixo acoplamento. Contudo, há um risco associado: a classe fachada pode evoluir para um "objeto deus" (God Object), tornando-se acoplada a um número excessivo de classes do sistema e centralizando responsabilidades indevidamente.

No que tange às suas relações, o Facade se distingue do Adapter, pois o primeiro define uma nova interface simplificada, ao passo que o segundo visa adaptar uma interface existente para torná-la compatível. A depender do contexto, uma classe Facade pode ser implementada como um Singleton.

**3. Strategy**

O padrão Strategy é um padrão comportamental que permite a definição de uma família de algoritmos, encapsulando cada um deles em classes separadas e permitindo que seus objetos sejam intercambiáveis. O problema que este padrão soluciona manifesta-se quando uma classe de "contexto" precisa implementar múltiplas variantes de um mesmo algoritmo. Na ausência do padrão, isso frequentemente resulta em código com extensas estruturas condicionais (como if-else ou switch-case), o que aumenta a complexidade, dificulta a manutenção e viola o Princípio Aberto/Fechado (Open/Closed Principle - OCP).

A solução consiste em extrair esses algoritmos variantes para classes "estratégia" concretas, que aderem a uma interface "Estratégia" comum. O objeto de "contexto" passa a manter uma referência a essa interface, delegando a execução do algoritmo ao objeto estratégia ativo. Isso confere ao cliente o poder de selecionar e, inclusive, substituir a estratégia em tempo de execução.

O padrão Strategy é particularmente aplicável quando se necessita alternar algoritmos dinamicamente (em runtime), ou quando existem diversos algoritmos similares que diferem apenas na implementação. Entre suas vantagens, destacam-se a capacidade de trocar algoritmos dinamicamente e o isolamento da implementação dos algoritmos, favorecendo o OCP. Como desvantagem, em cenários com poucas variantes e raras modificações, sua implementação pode introduzir uma complexidade desnecessária. Além disso, o cliente precisa ter conhecimento prévio das estratégias disponíveis.

O Strategy difere do Template Method, pois este utiliza herança para alterar partes de um algoritmo, enquanto o Strategy favorece a composição. Ele também possui similaridades com o padrão State; ambos alteram o comportamento de um objeto via delegação, mas o State adicionalmente gerencia o estado interno e as transições de estado.

**4. Observer**

O padrão Observer define um mecanismo de subscrição (subscription) que possibilita a múltiplos objetos, denominados "observadores" (observers), serem notificados automaticamente sobre quaisquer mudanças de estado que ocorram em outro objeto, o "sujeito" (subject) ou "publicador" (publisher). Este padrão aborda de forma eficiente o problema da sincronização de estado entre objetos dependentes.

Na ausência deste padrão, os objetos dependentes teriam que recorrer ao polling (verificação constante e ativa do estado do sujeito) ou o sujeito teria que enviar notificações indiscriminadas a todos os clientes, sendo ambas as abordagens ineficientes. A solução estabelece que o sujeito mantenha uma lista dinâmica de seus observadores. Quando uma mudança de estado relevante ocorre, o sujeito itera sobre esta lista e invoca um método de atualização (notificação) em cada observador, sem, contudo, acoplar-se às classes concretas dos observadores.

A aplicabilidade do Observer é ideal quando mudanças no estado de um objeto devem ser propagadas para outros, especialmente quando o conjunto de objetos dependentes é dinâmico ou desconhecido a priori. O padrão adere ao Princípio Aberto/Fechado, permitindo que novos assinantes sejam adicionados sem qualquer modificação no publicador. Como desvantagem, a ordem de notificação dos observadores é frequentemente indefinida. Em sistemas com um número massivo de assinantes, o processo de notificação pode gerar gargalos de performance ou complexidade no gerenciamento das atualizações.

Em comparação com outros padrões, o Observer difere do Mediator, que centraliza a comunicação de forma explícita entre componentes; o Observer, por sua vez, estabelece um canal de notificação dinâmico e mais descentralizado.

**REFÊRENCIA**

REFCTORING.GURU. *Padrão de Projeto Singleton*. Disponível em: [https://refactoring.guru/pt-br/design-patterns/singleton](https://refactoring.guru/pt-br/design-patterns/singleton?utm_source=chatgpt.com). Acesso em: 30 out. 2025.

REFCTORING.GURU. *Padrão de Projeto Facade*. Disponível em: [https://refactoring.guru/pt-br/design-patterns/facade](https://refactoring.guru/pt-br/design-patterns/facade?utm_source=chatgpt.com). Acesso em: 30 out. 2025.

REFCTORING.GURU. *Padrão de Projeto Strategy*. Disponível em: [https://refactoring.guru/pt-br/design-patterns/strategy](https://refactoring.guru/pt-br/design-patterns/strategy?utm_source=chatgpt.com). Acesso em: 30 out. 2025.

REFCTORING.GURU. *Padrão de Projeto Observer*. Disponível em: [https://refactoring.guru/pt-br/design-patterns/observer](https://refactoring.guru/pt-br/design-patterns/observer?utm_source=chatgpt.com). Acesso em: 30 out. 2025.