Programação Orientada a Objetos

Prof.ª Ma. Jessica Oliveira

Católica
Universidade Católica de Brasília

Aula 10 – 12/05/2025

Padrões de projeto: introdução e aplicação prática.

Católica
Universidade Católica de Brasília

Introdução aos *Design Patterns*.

- Os padrões de projeto **surgiram da necessidade de encontrar soluções recorrentes e eficazes para problemas comuns no desenvolvimento de *software* orientado a objetos**.
- A ideia foi inspirada originalmente nos trabalhos do arquiteto Christopher Alexander, que, ao estudar construções arquitetônicas, observou que **certos problemas surgem repetidamente e que há formas estruturadas e eficazes de solucioná-los**.
- Posteriormente, esse conceito foi adaptado para a Engenharia de *Software* por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos como a "*Gang of Four*" (GoF) .

Introdução aos *Design Patterns*.

- Segundo os autores, **um padrão de projeto descreve uma solução reutilizável para um problema recorrente** no *design* de sistemas orientados a objetos.
- Eles **não são algoritmos ou *frameworks***, mas descrições generalizadas que abordam como organizar classes e objetos para resolver problemas específicos de forma elegante, eficiente e sustentável.
- Um padrão **não define uma implementação rígida**, mas fornece uma estrutura flexível que pode ser adaptada a diferentes contextos e linguagens.

Introdução aos *Design Patterns*.

- Além disso, os padrões promovem boas práticas de design, como:
  - Baixo acoplamento e alta coesão;
  - Reutilização de código;
  - Facilidade de manutenção e evolução do sistema;
  - Comunicação entre desenvolvedores, por meio de uma linguagem comum.
- Para Freeman *et al*. (2004), os padrões também são importantes como um meio de ensinar boas práticas de projeto, além de permitir que soluções de alto nível possam ser replicadas e discutidas com mais clareza em equipes de desenvolvimento.

Classificação dos Padrões.

Criacionais.

- Esses padrões tratam da forma como os objetos são instanciados, promovendo flexibilidade e desacoplamento no processo de criação.
- São úteis para situações em que a lógica de criação de objetos é complexa, exige abstração ou depende do contexto de execução.
- Entre os principais padrões criacionais, destacam-se:

Criacionais.

- ***Singleton*:** garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela.
- ***Factory Method*:** define uma interface para criar objetos, permitindo que subclasses decidam qual classe instanciar.
- ***Abstract Factory*:** fornece uma interface para criar famílias de objetos relacionados sem especificar suas classes concretas.
- ***Builder*:** separa a construção de um objeto complexo da sua representação, permitindo que o mesmo processo de construção crie representações diferentes.
- ***Prototype*:** permite a criação de novos objetos a partir da cópia de um protótipo existente, ao invés da instanciação direta.

Estruturais.

- Os padrões estruturais dizem respeito à composição de classes e objetos, visando facilitar a construção de estruturas maiores e mais eficientes.
- São úteis quando há necessidade de reorganizar ou adaptar as interfaces entre componentes de software, promovendo reutilização e flexibilidade.
- Os principais padrões estruturais incluem:

Estruturais.

- ***Adapter*:** permite que classes com interfaces incompatíveis trabalhem juntas, através da conversão de uma interface em outra.
- ***Bridge*:** separa uma abstração da sua implementação, permitindo que ambas variem independentemente.
- ***Composite*:** permite tratar objetos individuais e composições de objetos de forma uniforme.
- ***Decorator*:** adiciona funcionalidades a objetos dinamicamente, sem alterar suas estruturas originais.
- ***Facade*:** fornece uma interface simplificada para um subsistema complexo.
- ***Flyweight*:** reduz o número de objetos criados, compartilhando instâncias semelhantes.
- ***Proxy*:** fornece um substituto ou representante para outro objeto, controlando o acesso a ele.

Comportamentais.

- Os padrões comportamentais se concentram na forma como os objetos interagem e comunicam entre si, promovendo a flexibilidade no fluxo de controle e a descentralização das responsabilidades.
- Estes padrões ajudam a definir protocolos de comunicação e coordenação de ações entre diferentes objetos.
- Entre os principais estão:

Comportamentais.

- ***Observer***: define uma dependência de um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados automaticamente.
- ***Strategy***: define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis.
- ***Command***: encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações.
- ***Chain of Responsibility***: permite que vários objetos tenham a chance de tratar uma solicitação, sem que o remetente precise saber qual objeto vai tratá-la.
- ***State***: permite que um objeto altere seu comportamento quando seu estado interno muda.
- ***Template Method***: define o esqueleto de um algoritmo em uma operação, permitindo que subclasses redefinam certos passos sem alterar a estrutura do algoritmo.

O Padrão *Singleton*.

Introdução.

- É um dos padrões criacionais mais conhecidos e simples.
- Ele visa garantir que uma classe tenha apenas **uma única instância** durante o ciclo de vida de uma aplicação, além de fornecer um **ponto global de acesso** a essa instância.

Características principais.

- **Construtor privado:** impede que a classe seja instanciada diretamente por outras classes.
- **Atributo estático:** armazena a instância única da classe.
- **Método de acesso público (geralmente estático):** verifica se a instância já existe e, se não, cria-a. Depois, retorna a instância existente.

Aplicabilidades típicas.

- Sistemas de **configuração centralizada**, onde é necessário garantir consistência nos parâmetros globais.
- **Gerenciadores de logs**, que devem registrar eventos de forma uniforme.
- **Conexões com banco de dados** (em contextos simples e sem concorrência).
- Controle de **acesso a recursos compartilhados**, como dispositivos físicos.

Cuidados e limitações.

- Em contextos *multithread*, o *Singleton* pode apresentar problemas se não for devidamente sincronizado.
- O uso excessivo pode indicar mau design, aproximando-se de um anti-padrão (por simular variáveis globais).
- Pode dificultar a realização de testes unitários, devido à persistência de estado.

Identificação de contextos adequados para aplicação de padrões.

Quando utilizar um padrão?

- Quando há repetição de lógica de construção de objetos (padrões criacionais).
- Quando há necessidade de desacoplar abstrações e implementações (padrões estruturais).
- Quando se deseja facilitar a comunicação e coordenação entre objetos (padrões comportamentais).
- Quando um problema recorrente foi identificado, e um padrão oferece uma solução testada e validada pela comunidade.

Quando evitar?

- Quando o padrão não resolve um problema real.
- Quando a aplicação do padrão torna o código mais difícil de entender para outros desenvolvedores.
- Quando alternativas mais simples são suficientes para atender aos requisitos do sistema.

Indicadores de que um padrão pode ser útil.

- Código duplicado em diferentes partes do sistema.
- Baixa coesão entre funcionalidades de uma classe.
- Forte acoplamento entre objetos que precisam variar de forma independente.
- Crescimento excessivo de condicionais para alternar comportamentos.

Vamos para a prática?

- Atualmente, cada objeto **`Publicacao`** possui sua própria instância de **`Avaliacao`**, o que faz com que as notas atribuídas fiquem isoladas (ou seja, dois livros distintos, mesmo que sejam edições do mesmo título, terão médias independentes).
- Sua missão é:
  - Refatorar a classe **`Avaliacao`** para aplicar o padrão **`Singleton`** de forma que:
    - Haja uma instância única de **`Avaliacao`** para cada tipo de publicação: uma para **`Livro`**, uma para **`Revista`** e uma para **`Ebook`**.
    - Essa instância única deve ser utilizada por todas as publicações daquele tipo.
    - As notas atribuídas a diferentes instâncias da mesma classe sejam acumuladas na mesma instância de **`Avaliacao`**.
  - Além disso:
    - Remova a dependência direta de **`Avaliacao`** como atributo nas subclasses.
    - Crie mecanismos adequados para recuperação da instância Singleton dentro da hierarquia.
    - Garanta que o comportamento da aplicação principal (classe **`Principal`**) não seja alterado na forma de uso.

Microprojeto 10

🎯 **Objetivo:** aplicar padrões de projeto e refatorar código para melhor manutenção.

Católica
Universidade Católica de Brasília

Para o pós-aula...

- Implementar *Singleton* para gerenciamento de instâncias;
- Refatorar código aplicando boas práticas;
- Corrigir más práticas de desenvolvimento identificadas na análise.

Dúvidas?

jessica.oliveira@p.ucb.br

Referências Bibliográficas.

- GAMMA, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995). ***Design Patterns: Elements of Reusable Object-Oriented Software***. Addison-Wesley.
- FREEMAN, E.; Robson, E.; Bates, B.; Sierra, K. (2004). ***Head First Design Patterns***. O'Reilly Media.
- LARMAN, C. (2007). **Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientados a Objetos e ao Processo Unificado**. 3. ed. Bookman.
- BUSCHMANN, F. et al. (1996). ***Pattern-Oriented Software Architecture: A System of Patterns***. Wiley.
- ALEXANDER, C. (1977). ***A Pattern Language: Towns, Buildings, Construction***. Oxford University Press.
- ORACLE. (2023). ***Java Design Patterns***. Disponível em: https://docs.oracle.com/javase/tutorial/