RESUMO DE PADRÕES DE PEOJETOS

O que são Padrões de Projeto?

Padrões de Projeto (*Design Patterns*) são soluções gerais, reutilizáveis para problemas comuns que surgem no design de software. Eles são como "plantas" ou "receitas" para resolver problemas recorrentes em diferentes contextos dentro do desenvolvimento de software. Não são designs acabados que podem ser diretamente convertidos em código, mas sim modelos ou descrições de como resolver um problema que pode ser usado em muitas situações diferentes. O conceito foi popularizado pelo livro "*Design Patterns: Elements of Reusable Object-Oriented* Software" (também conhecido como "*Gang of Four*" ou *GoF*) de *Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides*.

Eles buscam promover a reusabilidade, flexibilidade, manutenibilidade e compreensão do código, permitindo que os desenvolvedores se comuniquem sobre soluções de design em um nível mais abstrato e comum.

Tipos de Padrões de Projeto e Utilidade de Cada Um

Os padrões de projeto são geralmente categorizados em três tipos principais, baseados em sua finalidade:

1. **Padrões Criacionais (Creational Patterns):**

   Padrões criacionais abstraem e ocultam a forma como os objetos são criados, proporcionando flexibilidade na escolha e criação de objetos. Eles tornam o sistema independentemente de como seus objetos são criados, compostos e representados.

   Utilidade:

   - Delega a criação de objetos: Permitem que o sistema crie objetos sem saber a classe concreta que está sendo instanciada.
   - Promove a flexibilidade: Facilita a alteração da forma como os objetos são criados sem afetar o código cliente que os utiliza.
   - Controla a instanciação: Podem garantir que apenas uma instância de uma classe exista (*Singleton*) ou que um conjunto de objetos relacionados seja criado de forma consistente (*Abstract Factory*).

   Exemplos Notáveis:

   - **Singleton:** Garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela. Útil para gerenciadores de log, configurações ou pool de conexões de banco de dados.

- **Factory Method:** Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe instanciar. Usado quando uma classe não pode prever a classe de objetos que precisa criar.
- **Abstract Factory:** Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Útil para criar componentes de interface gráfica multi-plataforma.
- **Builder:** Separa a construção de um objeto complexo de sua representação, de modo que o mesmo processo de construção possa criar diferentes representações. Usado para criar objetos com muitas partes ou configurações diferentes.
- **Prototype:** Especifica os tipos de objetos a serem criados usando uma instância prototípica e cria novos objetos copiando este protótipo. Útil quando a criação de um objeto é dispendiosa ou complexa.

2. **Padrões Estruturais (Structural Patterns):**

Padrões estruturais tratam da composição de classes e objetos, ou seja, como eles são montados para formar estruturas maiores e mais complexas. Eles focam em como diferentes classes e objetos podem ser combinados para criar novas funcionalidades, mantendo a flexibilidade e eficiência.

Utilidade:

- **Simplificam a estrutura:** Permitem que um sistema funcione em conjunto, mesmo que as classes tenham interfaces diferentes.
- **Promovem a flexibilidade na composição:** Facilitam a adição ou remoção de funcionalidades sem alterar a estrutura central.
- **Gerenciam relacionamentos:** Ajudam a organizar as interconexões entre diferentes partes do software.

Exemplos Notáveis:

- **Adapter:** Converte a interface de uma classe para outra interface que o cliente espera. Útil para fazer classes incompatíveis trabalharem juntas.
- **Decorator:** Anexa responsabilidades adicionais a um objeto dinamicamente. Proporciona uma alternativa flexível à herança para estender a funcionalidade.
- **Composite:** Compõe objetos em estruturas de árvore para representar hierarquias parte-todo. Permite que os clientes tratem objetos individuais e composições de objetos uniformemente.
- **Facade:** Fornece uma interface unificada para um conjunto de interfaces em um subsistema. Define uma interface de alto nível que torna o subsistema mais fácil de usar.

- **Proxy:** Fornece um substituto ou placeholder para outro objeto para controlar o acesso a ele. Útil para lazy loading, controle de acesso ou logging.
- **Bridge:** Desacopla uma abstração de sua implementação, de modo que as duas possam variar independentemente.
- **Flyweight:** Permite que muitos objetos de granularidade fina sejam usados de forma eficiente, compartilhando partes de seu estado.

3. **Padrões Comportamentais (Behavioral Patterns):**

Padrões comportamentais lidam com os algoritmos e a atribuição de responsabilidades entre objetos. Eles descrevem como os objetos interagem e se comunicam entre si, garantindo a flexibilidade na comunicação e na distribuição de tarefas.

Utilidade:

- **Promovem a comunicação flexível:** Permitem que os objetos interajam de maneiras diferentes sem que suas classes sejam firmemente acopladas.
- **Encapsulam algoritmos:** Tornam os algoritmos e as responsabilidades facilmente alteráveis e reutilizáveis.
- **Definem responsabilidades:** Ajudam a distribuir o comportamento entre as classes de forma lógica e coesa.

Exemplos Notáveis:

- **Observer:** Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. Útil para sistemas de eventos e interfaces de usuário.
- **Strategy:** Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.
- **Template Method:** Define o esqueleto de um algoritmo em uma operação, adiando alguns passos para subclasses. Permite que subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
- **Command:** Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar solicitações ou logar operações e suportar operações que podem ser desfeitas.
- **Iterator:** Fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.
- **State:** Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá mudar sua classe.

- **Chain of Responsibility:** Evita acoplar o remetente de uma solicitação ao seu receptor, dando a mais de um objeto a chance de lidar com a solicitação. Encadear os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a manipule.

Referências

Livros e Artigos:

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Freeman, E., Freeman, E., Sierra, K., & Bates, B. (2004). *Head First Design Patterns*. O'Reilly Media.

GEEKSFORGEEKS. *Design Patterns in C++*. Disponível em: https://www.geeksforgeeks.org/design-patterns-in-cpp/. Acesso em: 3 jun. 2025.

REFACTORING.GURU. *Design Patterns*. Disponível em: https://refactoring.guru/design- patterns. Acesso em: 3 jun. 2025.