Padrões de Projeto

O capítulo 6 do livro *Engenharia de Software Moderna* introduz e detalha os **padrões de projeto**, que são soluções reutilizáveis para problemas recorrentes na engenharia de software. Esses padrões não são trechos de código específicos, mas sim abordagens generalizadas para estruturar sistemas de software de maneira eficiente, flexível e extensível.

A grande vantagem dos padrões de projeto é que eles criam um **vocabulário comum** entre desenvolvedores. Em vez de descrever longamente uma solução para um problema de design, basta dizer "use um padrão Fábrica" ou "esse módulo está seguindo um padrão Observador", facilitando a comunicação e a compreensão do código.

No entanto, o capítulo enfatiza que os padrões de projeto **não devem ser usados indiscriminadamente**. Eles são úteis para tornar o código mais flexível e reutilizável, mas um uso excessivo pode levar a sistemas mais complexos do que o necessário. O autor menciona o conceito de *design for change*, ou seja, projetar sistemas prevendo futuras mudanças para evitar a necessidade de grandes refatorações no futuro.

Padrões Criacionais

Os padrões criacionais ajudam a gerenciar a criação de objetos de maneira eficiente e flexível.

1. Fábrica (*Factory*)

- **Contexto**: Um sistema distribuído baseado em TCP/IP onde diferentes funções criam objetos TCPChannel para comunicação remota.
- **Problema**: O sistema precisa suportar tanto TCP quanto UDP. Criar instâncias diretamente com new (TCPChannel c = new TCPChannel();) impede flexibilidade.
- **Solução**: Introduzir uma **classe fábrica**, que decide qual tipo de canal instanciar, abstraindo a decisão dos módulos clientes.

2. Singleton

- **Contexto**: Um sistema pode ter componentes que exigem uma única instância, como gerenciadores de logs ou conexões com banco de dados.
- **Problema**: Garantir que uma classe tenha apenas uma instância e que essa instância seja acessível globalmente.
- **Solução**: Criar um método estático que retorna sempre a mesma instância.

Padrões Estruturais

Os padrões estruturais lidam com a forma como classes e objetos se relacionam, permitindo composições flexíveis.

3. Proxy

- **Contexto**: Um objeto precisa ser acessado de maneira controlada, como em chamadas remotas ou sistemas que exigem cache.
- **Problema**: Controle de acesso ou otimização do uso de recursos.
- **Solução**: Criar um objeto intermediário (*proxy*) que gerencia o acesso ao objeto real.

4. Adaptador (*Adapter*)

- **Contexto**: Duas classes com interfaces incompatíveis precisam se comunicar.
- **Problema**: Como fazer com que um sistema legado funcione com uma nova API sem modificar o código original?
- **Solução**: Criar uma classe que converte a interface de um objeto para a interface esperada pelo cliente.

5. Fachada (*Facade*)

- **Contexto**: Um sistema tem múltiplos módulos complexos e pouco intuitivos para os usuários.
- **Problema**: Como simplificar a interação com um sistema sem expor sua complexidade interna?
- **Solução**: Criar uma única classe (*fachada*) que fornece uma interface simplificada para acessar funcionalidades internas.

6. Decorador (*Decorator*)

- **Contexto**: Deseja-se adicionar funcionalidades a um objeto sem modificar sua estrutura básica.
- **Problema**: Como tornar um sistema extensível sem recorrer a herança, que pode tornar o código rígido?
- **Solução**: Criar classes que **envolvem** objetos existentes e adicionam novos comportamentos.

Padrões Comportamentais

Os padrões comportamentais definem como objetos interagem e compartilham responsabilidades.

7. Strategy

- **Contexto**: Um sistema precisa suportar múltiplos algoritmos para realizar uma mesma tarefa.
- **Problema**: Como permitir que diferentes algoritmos sejam utilizados sem modificar a lógica principal do código?
- **Solução**: Criar uma interface para estratégias, permitindo a escolha dinâmica da implementação desejada.

8. Observador (*Observer*)

- **Contexto**: Um objeto (*sujeito*) deve notificar múltiplos *observadores* sempre que seu estado mudar.

- **Problema**: Como evitar dependências rígidas entre classes quando um objeto precisa informar outros sobre mudanças?
- **Solução**: Implementar um mecanismo de inscrição/assinatura onde objetos interessados podem se registrar para receber atualizações.

9. Template Method

- **Contexto**: Um conjunto de classes tem um fluxo de execução similar, mas com pequenas variações.
- **Problema**: Como garantir que certas etapas do algoritmo sejam fixas, enquanto outras podem ser personalizadas?
- **Solução**: Definir um **método base abstrato** e permitir que subclasses sobrescrevam apenas partes específicas do processo.

10. Visitor

- **Contexto**: Um sistema contém uma estrutura de objetos complexa, mas novas operações precisam ser adicionadas frequentemente.
- **Problema**: Como adicionar novas operações sem modificar as classes dos objetos afetados?
- **Solução**: Criar um objeto visitante separado, que encapsula a lógica das novas operações e interage com os elementos necessários.

Outros Padrões

Além dos dez padrões principais, o capítulo menciona brevemente outros padrões úteis, como:

- **Builder** – Para construção progressiva de objetos complexos.
- **Chain of Responsibility** – Para encaminhamento dinâmico de requisições entre objetos.
- **Iterator** – Para percorrer coleções sem expor sua estrutura interna.
- **State** – Para alterar o comportamento de um objeto conforme seu estado interno.

Quando Não Usar Padrões de Projeto

O capítulo finaliza alertando contra a **"paternite"**, o uso excessivo de padrões de projeto sem necessidade real. Embora esses padrões tornem o código mais flexível, sua aplicação inadequada pode levar a uma complexidade desnecessária. O autor recomenda que os desenvolvedores priorizem soluções simples antes de recorrer a padrões mais sofisticados. Com isso, temos que o capítulo 6 do livro "*Engenharia de Software Moderna"* apresenta uma visão clara e prática sobre os padrões de projeto, explicando seu propósito, aplicação e exemplos de código. O livro destaca que padrões de projeto não são apenas soluções técnicas, mas também um **meio de comunicação entre desenvolvedores**, ajudando a construir software robusto, flexível e de fácil manutenção. Este capítulo é fundamental para qualquer engenheiro de software, pois ensina não apenas **como usar padrões de projeto**, mas **quando e por que utilizá-los**, incentivando o design orientado a mudanças.