Resenha do capítulo 6 - Padrões de Projeto

Ao ler o capítulo sobre Padrões de Projeto, comecei a refletir sobre como essas soluções estruturais, originalmente desenvolvidas por Christopher Alexander para a arquitetura de cidades, podem ser adaptadas para o desenvolvimento de software. O que mais me impressionou foi saber que esse é um trabalho de 1995, que começou quando os autores adaptaram esses conceitos para a construção de sistemas, criando padrões de projeto para ajudar os desenvolvedores a resolverem problemas recorrentes. O mais interessante é que, além de oferecerem soluções práticas, esses padrões criam uma ponte entre o pessoal de T.i, o que facilita a comunicação entre os desenvolvedores. Isso é muito importante por experiência própria é muito ruim desenvolver algo em um time onde não se tem clareza na troca de ideias e nem compreensão e um acordo em relação aos problemas e soluções

O capítulo explicita que existem 23 padrões de projeto, divididos em três grandes categorias: criacionais, estruturais e comportamentais. O ponto que mais me chamou a atenção foi como cada um desses padrões contribui para a criação de sistemas mais flexíveis, adaptáveis e fáceis de manter. Em vez de criar soluções únicas e fechadas, os padrões de projetos nos permitem antecipar mudanças e evoluções no sistema. Essa flexibilidade é muito importante, especialmente ao pensarmos em um sistema que está começando do zero e evolui rapidamente, como em um projeto de Trabalho interdisciplinar.

Um exemplo que me fez refletir bastante foi o padrão Fábrica. Este padrão foi projetado para separar o processo de criação de objetos do código cliente, permitindo que a criação de novos tipos de objetos seja mais flexível sem precisar mexer no código cliente. No caso apresentado no capítulo, o sistema criava objetos TCPChannel diretamente, o que limitava a flexibilidade. A solução foi criar uma fábrica, a ChannelFactory, que seria responsável pela criação desses objetos. Isso permite que, se futuramente fosse necessário usar outro tipo de canal de comunicação, o código poderia ser alterado, sem modificar diretamente o código cliente.

Outro conceito abordado que eu conhecia superficialmente foi o padrão Singleton, já tinha estudado na matéria de programação modular. Esse padrão busca garantir que uma classe tenha apenas uma instância durante a execução do programa. O exemplo utilizado no capítulo foi o de uma classe Logger, responsável por registrar eventos no sistema. Se múltiplas instâncias fossem criadas para essa classe, poderiam ocorrer problemas como a criação de arquivos de login duplicados. Transformar essa classe em um Singleton resolve esse problema, pois garante que a mesma instância seja usada em toda a aplicação. No entanto, o uso deste padrão pode gerar um acoplamento forte entre as classes e dificultar testes automatizados.

Além dos padrões já mencionados, o padrão Adaptador se destacou como uma solução interessante para integrar sistemas com interfaces diferentes, por exemplo, precisamos controlar projetores de marcas distintas, com interfaces específicas, o padrão Adaptador permite criar uma interface comum que oculta essas diferenças, simplificando a comunicação entre o sistema e os projetores. Esse conceito é útil para integrar sistemas de origens diversas, sem a necessidade de refazer grande parte do código. Para ficar mais fácil isso, imagine o Adaptador como uma espécie de adaptador de portas HDMI, onde, através dele, é possível conectar monitores de marcas diferentes, garantindo que eles funcionem de maneira compatível, sem complicações.

O capítulo também fala sobre outros padrões como Fachada, Decorador, Strategy, Observador, Template Method e Visitor, cada um com suas aplicações específicas para resolver problemas. Por exemplo, o padrão Strategy permite que o comportamento de um objeto seja alterado em tempo de execução sem modificar sua classe original, como ao trocar o algoritmo de ordenação de uma lista. Já o padrão Template Method oferece uma estrutura de algoritmo em uma classe base, permitindo que as subclasses personalizem certas etapas. Esses padrões têm um papel fundamental na modularidade e flexibilidade do código, pois tornam possível alterar o comportamento do sistema sem a necessidade de grandes refatorações.

Esses padrões me fazem refletir sobre o meu trabalho com o CRUD de caminhões, onde percebo que a aplicação de padrões poderia ter evitado várias limitações. Se eu tivesse aplicado, por exemplo, o Adaptador para lidar com diferentes tipos de operações no CRUD, o sistema seria mais fácil de modificar e expandir no futuro, sem que fosse necessário refatorar o código todo. Isso mostra que a modularidade que esses padrões oferecem são importantes demais, especialmente quando estamos desenvolvendo sistemas que precisam evoluir com o tempo.

Ao mesmo tempo, o capítulo alerta para o uso excessivo de padrões de projeto. Não se deve usar um padrão apenas porque ele está na moda ou porque parece uma boa prática. A escolha de aplicar um padrão deve ser feita de acordo com as necessidades reais do sistema. Por exemplo, se o sistema não exige flexibilidade para mudanças futuras ou troca de comportamentos, o uso de padrões como o Strategy ou o Factory pode ser um exagero, apenas adicionando complexidade desnecessária.

No final das contas, o capítulo me ensinou que padrões de design são poderosos, mas como qualquer ferramenta, devem ser usados com a mão na consciência. A aplicação desses padrões pode realmente melhorar a qualidade do código, tornando-o mais fácil de manter, adaptar e expandir. Mas o mais importante é saber quando e como aplicar esses padrões de forma estratégica, para que eles tragam benefícios sem complicar desnecessariamente o desenvolvimento do sistema.