Capítulo 5: Princípios de Projeto

Neste capítulo, o autor apresenta os princípios SOLID, um conjunto de diretrizes fundamentais para o design de software orientado a objetos. Estes princípios têm como objetivo garantir que os sistemas sejam mais modulares, fáceis de entender e manter. Com a aplicação desses princípios, o desenvolvimento de software se torna mais eficiente e a evolução do sistema se torna mais simples e sem grandes dificuldades.

- Princípio da Responsabilidade Única (SRP): Esse princípio afirma que cada classe deve ter apenas uma responsabilidade, ou seja, um único motivo para ser alterada. Ele facilita a manutenção do código, pois qualquer modificação que seja necessária será mais localizada. No mercado, um exemplo prático pode ser um sistema de gerenciamento de pedidos em um e-commerce. A classe responsável por um pedido deve lidar apenas com as informações do pedido (como itens, quantidade, e valores), enquanto o processamento de pagamento e o envio de pedidos podem ser delegados a outras classes, como ProcessadorPagamento e GerenciadorEnvio. A adoção desse princípio facilita a escalabilidade do sistema e sua integração com outros serviços, algo fundamental quando lidamos com microsserviços.

- Princípio Aberto/Fechado (OCP): Segundo esse princípio, uma classe deve ser aberta para extensão mas fechada para modificação. Isso significa que, para adicionar novas funcionalidades a uma classe, deve-se preferir a herança ou composição em vez de alterar o código existente. Um exemplo no mercado seria um sistema de relatórios, onde podemos criar uma interface Relatorio e, a partir dela, implementar várias classes concretas como RelatorioVendas ou RelatorioClientes. Ao adicionar um novo tipo de relatório, como RelatorioFinanceiro, basta criar uma nova implementação sem alterar o código original. Essa abordagem facilita a expansão do sistema sem comprometer a estabilidade do código.

- Princípio da Substituição de Liskov (LSP): Esse princípio sugere que subtipos devem ser substituíveis por seus tipos base sem afetar o funcionamento do sistema. Ou seja, ao usar herança, as subclasses não podem alterar o comportamento esperado da classe base. Um exemplo simples é a relação entre Quadrado e Retângulo. Se um quadrado herda de retângulo e redefine os métodos de ajuste de altura e largura, isso pode gerar comportamentos inesperados, pois quem espera um retângulo pode não obter a funcionalidade esperada ao substituir por um quadrado. No mercado, garantir que as classes e seus subtipos sejam compatíveis é essencial para evitar falhas em sistemas integrados.

- Princípio da Segregação de Interface (ISP): Este princípio afirma que uma classe não deve ser forçada a implementar métodos que ela não usa. Em vez disso, devemos criar interfaces menores e mais específicas. No contexto de um sistema de impressão, por

exemplo, uma interface Impressora com métodos para imprimir, digitalizar e enviar fax pode ser problemática se uma classe precisar apenas de impressão. Para evitar isso, seria melhor criar interfaces separadas, como Imprimivel, Digitalizavel e EnviaFax, permitindo que as classes implementem apenas o que realmente precisam. Isso é especialmente relevante no design de APIs REST, onde endpoints mais específicos ajudam a manter o sistema mais organizado e fácil de manter.

- Princípio da Inversão de Dependência (DIP): Segundo esse princípio, módulos de alto nível não devem depender de módulos de baixo nível, mas sim de abstrações. Isso significa que, ao projetar um sistema, devemos abstrair detalhes de implementação para reduzir o acoplamento entre componentes. No contexto de um sistema de persistência de dados, em vez de depender diretamente de um banco de dados específico, como MySQL ou PostgreSQL, a aplicação deve depender de uma interface genérica, como RepositorioDados. Isso torna o sistema mais flexível, permitindo que você substitua o banco de dados sem grandes alterações no código.

No geral, a adoção dos princípios SOLID proporciona maior modularidade e facilidade de manutenção, além de garantir que o sistema seja flexível e escalável ao longo do tempo.

Capítulo 6: Padrões de Projeto

O capítulo 6 aborda a importância dos padrões de projeto, que são soluções prontas e eficientes para problemas recorrentes no design de software. Os padrões de projeto ajudam a padronizar a forma como os desenvolvedores resolvem problemas de design, o que facilita a comunicação e manutenção do código. Eles são divididos em três categorias principais: criacionais, estruturais e comportamentais.

- Padrão Singleton (Criacional): O Singleton garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a essa instância. Esse padrão é muito útil para gerenciar recursos compartilhados, como conexões com banco de dados ou logs de sistema, onde só é necessário ter uma única instância para evitar sobrecarga. No mercado, o uso do Singleton é comum em aplicações de grandes empresas, que precisam de um controle centralizado sobre suas operações.

- Padrão Factory Method (Criacional): Esse padrão permite que o sistema crie objetos sem precisar saber exatamente qual classe instanciar. Isso é feito por meio de uma interface, que delega a criação para subclasses. No mercado, ele é amplamente utilizado em frameworks que precisam criar objetos de tipos diferentes, sem saber qual será utilizado em cada momento. Um exemplo clássico é o uso do Factory Method em frameworks de testes, onde o tipo de objeto a ser criado depende da configuração do ambiente.

- Padrão Observer (Comportamental): O Observer estabelece uma relação de dependência entre um objeto e seus observadores, de modo que, quando o estado do objeto muda, todos os seus observadores são notificados. Esse padrão é bastante usado em sistemas de eventos e notificações, como em interfaces gráficas ou sistemas de monitoramento de dados. Um exemplo no mercado seria um sistema de notificação em um e-commerce, onde quando um pedido é atualizado, o sistema notifica todos os envolvidos (clientes, gerentes e equipe de entrega).

- Padrão Decorator (Estrutural): O Decorator permite adicionar funcionalidades a um objeto dinamicamente, sem alterar sua classe original. Esse padrão é útil quando é necessário adicionar comportamentos extras a objetos em tempo de execução. No mercado, ele é frequentemente utilizado para adicionar funcionalidades como logs, cache ou segurança a serviços existentes, sem modificar o código original do sistema. Um exemplo disso seria adicionar uma camada de segurança a uma aplicação existente de e-commerce, sem precisar refatorar todo o código.

- Padrão Strategy (Comportamental): O Strategy define uma família de algoritmos e permite que o algoritmo a ser utilizado seja escolhido em tempo de execução. Esse padrão é ideal para cenários onde a escolha do algoritmo pode variar de acordo com a situação, como algoritmos de ordenação ou compressão de arquivos. No mercado, o Strategy é bastante utilizado em sistemas de processamento de dados em que o tipo de algoritmo pode ser alterado dinamicamente para melhorar o desempenho do sistema.

O uso de padrões de projeto no desenvolvimento de software ajuda a criar sistemas mais modulares e flexíveis, além de promover a reutilização de soluções testadas e eficazes. No mercado, o conhecimento e a aplicação desses padrões são essenciais para criar software de alta qualidade, que seja fácil de entender, manter e expandir.