Capítulo 5: Princípios de Projeto
Este capítulo inicia destacando a importância do projeto de software como uma atividade fundamental para decompor problemas complexos em partes menores e mais gerenciáveis. A complexidade dos sistemas modernos exige estratégias eficazes de decomposição, permitindo que diferentes componentes sejam desenvolvidos de forma independente, mas coesa.

Um dos conceitos centrais discutidos é a **Integridade Conceitual**, que enfatiza a necessidade de consistência e uniformidade nas decisões de projeto. Isso garante que o sistema tenha uma aparência e comportamento homogêneos, facilitando sua compreensão e manutenção. Para alcançar essa integridade, é crucial que as decisões de projeto sejam guiadas por uma visão clara e unificada, muitas vezes liderada por um arquiteto de software ou uma equipe coesa.

Outro princípio fundamental é o **Ocultamento de Informação**, que sugere que os módulos de um sistema devem expor apenas o necessário, mantendo detalhes internos escondidos. Isso reduz o acoplamento entre componentes e aumenta a modularidade, permitindo que alterações internas não afetem outras partes do sistema. Por exemplo, ao desenvolver uma biblioteca de processamento de pagamentos, os detalhes de implementação das transações devem ser encapsulados, expondo apenas métodos públicos para iniciar, confirmar ou cancelar pagamentos.

A **Coesão** e o **Acoplamento** são discutidos como métricas para avaliar a qualidade do projeto. A coesão refere-se ao grau em que as responsabilidades de um módulo são relacionadas entre si. Módulos altamente coesos realizam tarefas específicas e bem definidas. Por outro lado, o acoplamento diz respeito ao grau de dependência entre módulos; um baixo acoplamento é desejável, pois indica que os módulos podem evoluir de forma independente. No desenvolvimento de uma aplicação de comércio eletrônico, por exemplo, é benéfico que o módulo de gerenciamento de estoque seja altamente coeso, lidando apenas com funções relacionadas ao controle de produtos, e tenha baixo acoplamento com o módulo de interface do usuário.

O capítulo também apresenta diversos princípios de projeto, como:

- **Responsabilidade Única**: cada classe ou módulo deve ter uma única responsabilidade ou motivo para mudar.
- **Segregação de Interfaces**: clientes não devem ser forçados a depender de interfaces que não utilizam.
- **Inversão de Dependências**: módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações.
- **Prefira Composição a Herança**: promove o uso de composição de objetos em vez de hierarquias de herança para reutilização de código.
- **Lei de Demeter**: um módulo deve ter conhecimento limitado sobre outros módulos, interagindo apenas com seus "amigos" diretos.
- **Aberto/Fechado**: entidades de software devem ser abertas para extensão, mas fechadas para modificação.
- **Substituição de Liskov**: objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento esperado do sistema.

A aplicação desses princípios pode ser observada no desenvolvimento de sistemas bancários. Por exemplo, ao implementar diferentes tipos de contas (corrente, poupança, investimento), é preferível usar composição para adicionar comportamentos específicos a cada tipo de conta, em vez de criar uma complexa hierarquia de herança. Isso torna o sistema mais flexível para futuras extensões, como a introdução de novos tipos de contas ou serviços.

Capitulo 6: Padrões de Projeto

O Capítulo 6 explora os padrões de projeto, soluções reutilizáveis para problemas recorrentes no desenvolvimento de software. Os padrões ajudam a melhorar a estrutura e flexibilidade do código, facilitando manutenção e expansão.

Os padrões de projeto foram popularizados pela "Gang of Four" (GoF), que os categorizaram em três grupos:

1. **Criacionais**: focam na instanciação de objetos.
2. **Estruturais**: tratam da composição das classes.
3. **Comportamentais**: definem como os objetos interagem.

Padrões Criacionais

Os padrões criacionais tratam da criação de objetos de maneira flexível e reutilizável. Alguns dos principais são:

- **Singleton**: Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso. Em aplicações de logging, um singleton assegura que todas as partes do sistema gravem logs no mesmo arquivo.
- **Factory Method**: Define uma interface para criar objetos, permitindo que subclasses decidam qual objeto instanciar. Útil em frameworks que precisam criar objetos sem especificar suas classes diretamente.
- **Abstract Factory**: Cria famílias de objetos relacionados sem especificar suas classes concretas. É muito usado no desenvolvimento de interfaces gráficas, onde diferentes sistemas operacionais precisam de implementações distintas para botões e menus.

Padrões Estruturais

Os padrões estruturais lidam com a forma como classes e objetos são compostos para formar estruturas maiores e reutilizáveis. Alguns exemplos são:

- **Adaptador (Adapter)**: Converte a interface de uma classe em outra esperada pelo cliente. Um exemplo é um sistema de pagamento que precisa integrar vários provedores com interfaces diferentes.
- **Fachada (Facade)**: Fornece uma interface unificada para um conjunto de interfaces, simplificando a interação com sistemas complexos. Pode ser usado para encapsular vários serviços de um sistema de ERP.

- **Decorator**: Permite adicionar funcionalidades a um objeto dinamicamente. Em sistemas de notificação, pode-se adicionar diferentes canais (e-mail, SMS, push) a uma notificação sem modificar sua estrutura base.

Padrões Comportamentais

Os padrões comportamentais definem como objetos interagem e se comunicam. Alguns exemplos são:

- **Observer**: Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados. Muito utilizado em eventos de interfaces gráficas e em sistemas de notificações.
- **Strategy**: Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. Um exemplo seria um sistema de pagamento que pode alternar entre diferentes estratégias como cartão de crédito, boleto e Pix.
- **Command**: Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes operações. Isso é útil em sistemas de desfazer/refazer, como editores de texto e sistemas de controle de versão.

Aplicações no Mercado

A aplicação de padrões de projeto no mercado é vastíssima. No desenvolvimento de um aplicativo de transporte, por exemplo, pode-se usar **Factory Method** para criar diferentes tipos de veículos (carro, moto, bicicleta) com base na escolha do usuário. O padrão **Fachada** pode ser empregado para unificar chamadas a APIs de mapas e serviços meteorológicos, simplificando a interação com o código.

Outro exemplo está no desenvolvimento de jogos, onde o **Singleton** pode ser usado para armazenar o estado do jogo (como pontuação e configurações), garantindo que haja apenas uma instância acessível em toda a aplicação. O padrão **Observer** também pode ser aplicado para atualizar o HUD (heads-up display) do jogo quando eventos ocorrem, como coleta de itens ou mudanças na barra de vida.

Em sistemas de e-commerce, o padrão **Decorator** pode ser usado para adicionar descontos e taxas de envio a um carrinho de compras dinamicamente, sem modificar a lógica principal. Já o padrão **Command** pode ser útil para gerenciar pedidos, permitindo que ações como "Cancelar Pedido" ou "Reembolsar" sejam desfeitas facilmente.

Em resumo, os padrões de projeto são ferramentas essenciais para o design de sistemas escaláveis e flexíveis. Seu uso adequado melhora a reutilização de código, reduz dependências desnecessárias e facilita a manutenção de grandes sistemas. Dominar esses padrões permite criar software mais eficiente e sustentável, garantindo soluções robustas e fáceis de evoluir ao longo do tempo.