Resenha Capitulos 5 e 6

Capitulo 5

O Capítulo 5 do livro Engenharia de Software Moderna, de Marco Tulio Valente, trata dos princípios de projeto aplicados ao desenvolvimento de software, um tema essencial para a construção de sistemas que sejam não apenas funcionais, mas também robustos, flexíveis e manuteníveis. O autor fundamenta a discussão em práticas amplamente reconhecidas na engenharia de software, enfatizando como um bom projeto pode reduzir custos de manutenção, melhorar a legibilidade do código e facilitar futuras extensões do sistema.

1. A importância do bom projeto de software

Antes de apresentar os princípios específicos, o autor contextualiza por que um bom projeto é essencial no desenvolvimento de software. Ele destaca que sistemas mal projetados tendem a ser difíceis de modificar, testar e escalar, levando a problemas como:

Dívida técnica, onde decisões de curto prazo criam dificuldades a longo prazo.
Baixa coesão e alto acoplamento, resultando em código difícil de manter.
Falta de modularidade, dificultando a reutilização de código.
A partir desse contexto, são introduzidos os princípios fundamentais que orientam um projeto de software eficiente.

2. Princípios Fundamentais do Projeto de Software

O capítulo apresenta diversos princípios que ajudam a estruturar sistemas de forma mais eficiente. Entre eles, destacam-se:

1. Integridade Conceitual
   Esse princípio estabelece que um sistema bem projetado deve ter uma visão unificada e coerente. Isso significa que a experiência do usuário e a estrutura do código devem ser consistentes em todas as partes do sistema. Na prática, isso evita que um software pareça uma coleção de "remendos", onde cada funcionalidade foi implementada de maneira diferente.

Exemplo prático:
Uma interface de usuário (UI) bem projetada mantém padrões de navegação e design coerentes, evitando que o usuário tenha que reaprender como interagir com diferentes partes do sistema.

2. Ocultamento de Informação e Modularidade
   O ocultamento de informação (information hiding) sugere que os detalhes internos de um módulo devem ser mantidos privados, expondo apenas as funcionalidades essenciais. Isso favorece a modularidade, permitindo que mudanças internas em um módulo não afetem o restante do sistema.

Exemplo prático:
Considere uma aplicação bancária. O módulo de cálculo de juros pode mudar sua implementação interna, mas, se seus métodos públicos permanecerem os mesmos, outras partes do sistema continuarão funcionando sem precisar ser modificadas.

3. Coesão e Acoplamento
   Esses dois conceitos estão fortemente ligados:

Alta coesão: Um módulo deve ter uma única responsabilidade bem definida. Isso torna o código mais fácil de entender e modificar.
Baixo acoplamento: Os módulos devem ser minimamente dependentes uns dos outros. Isso permite que mudanças em um módulo não impactem outros desnecessariamente.
Exemplo prático:
Em um sistema de e-commerce, o módulo de "pagamento" deve ser altamente coeso (cuidando apenas do processamento de pagamentos) e ter baixo acoplamento com o módulo de "catálogo de produtos". Assim, se houver mudanças na estrutura do catálogo, o sistema de pagamento não será afetado.

4. Princípios SOLID e Outros Conceitos Essenciais
   O capítulo também aborda os princípios SOLID, que são fundamentais para o design orientado a objetos:

Responsabilidade Única (SRP - Single Responsibility Principle)
Cada classe ou módulo deve ter apenas uma razão para mudar. Isso reduz a complexidade e facilita a manutenção.

Exemplo prático:
Um sistema de reservas de voos deve ter classes separadas para "emissão de passagens" e "cálculo de tarifas". Misturar essas responsabilidades criaria dependências desnecessárias.

Aberto/Fechado (OCP - Open/Closed Principle)
Sistemas devem ser abertos para extensão, mas fechados para modificação. Ou seja, é preferível adicionar novas funcionalidades sem alterar código existente.

Exemplo prático:
Um sistema de pagamentos online pode permitir a adição de novos métodos de pagamento (como Pix ou criptomoedas) sem modificar as classes existentes, apenas estendendo-as.

Substituição de Liskov (LSP - Liskov Substitution Principle)
Classes derivadas devem poder ser usadas no lugar das suas classes base sem alterar o comportamento do sistema.

Exemplo prático:
Se um sistema possui uma classe base "Veículo" e classes derivadas como "Carro" e "Bicicleta", ambas devem respeitar as mesmas regras de uso sem causar erros inesperados.

Segregação de Interfaces (ISP - Interface Segregation Principle)
Interfaces devem ser específicas para cada necessidade, em vez de forçar classes a implementar métodos que não utilizam.

Exemplo prático:
Em um software de gerenciamento de impressoras, uma interface "Impressora" deve conter apenas métodos relevantes (como "imprimir" e "digitalizar") em vez de incluir funcionalidades desnecessárias como "fax", que nem todas as impressoras modernas possuem.

Inversão de Dependências (DIP - Dependency Inversion Principle)
Módulos de alto nível não devem depender de módulos de baixo nível, mas sim de abstrações.

Exemplo prático:
Um sistema de notificações pode ser projetado para depender de uma interface genérica "Notificador" em vez de classes concretas como "EmailNotificador" ou "SMSNotificador". Isso facilita a substituição ou adição de novos canais de notificação.

5. Aplicação e Métricas de Qualidade
   Além dos princípios de projeto, o capítulo também discute métricas para avaliar a qualidade do código, como:

Número de linhas de código (LOC): Mede o tamanho do código, mas não é suficiente para determinar qualidade.
Complexidade Ciclomática: Mede a complexidade dos fluxos de decisão dentro do código.
Coesão e Acoplamento: Analisam a estrutura dos módulos e a interdependência entre eles.
Essas métricas ajudam desenvolvedores a identificar possíveis problemas antes que se tornem críticos, permitindo refatorações mais estratégicas.

Conclusão
O Capítulo 5 de Engenharia de Software Moderna oferece uma base teórica e prática essencial para qualquer desenvolvedor que deseja aprimorar suas habilidades de projeto de software. Os princípios abordados — como modularidade, ocultamento de informação, SOLID e métricas de qualidade — são fundamentais para criar sistemas mais fáceis de manter, testar e escalar.

A aplicação desses conceitos na vida real pode ser vista em qualquer software bem projetado: desde plataformas de redes sociais até sistemas financeiros e aplicativos móveis. Adotar essas boas práticas reduz a complexidade do código, melhora a experiência do usuário e evita problemas de manutenção no futuro.

Este capítulo é um guia essencial para desenvolvedores que buscam escrever código mais limpo, sustentável e alinhado com as melhores práticas da engenharia de software.

Capitulo 6

O Capítulo 6 do livro "Engenharia de Software Moderna" é dedicado aos Padrões de Projeto, que são soluções reutilizáveis para problemas recorrentes no desenvolvimento de software. Esses padrões promovem a criação de sistemas mais flexíveis e extensíveis, facilitando a manutenção e evolução do código.

Introdução aos Padrões de Projeto
A origem dos padrões de projeto remonta ao arquiteto Christopher Alexander, que, em 1977, publicou "A Pattern Language", documentando soluções para problemas na construção de cidades e edificações. Inspirados por essa abordagem, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides adaptaram esses conceitos para a engenharia de software, resultando no livro "Design Patterns: Elements of Reusable Object-Oriented Software", publicado em 1995. Nesse contexto, padrões de projeto são definidos como descrições de objetos e classes que se relacionam para resolver um problema de design específico em determinado contexto.

Importância dos Padrões de Projeto
A adoção de padrões de projeto traz diversos benefícios:

Vocabulário Comum: Estabelece uma linguagem padronizada entre desenvolvedores, facilitando a comunicação e a compreensão do design do sistema.

Soluções Testadas: Oferece abordagens comprovadas para problemas comuns, reduzindo o tempo gasto na resolução de desafios de design.

Facilidade de Manutenção: Promove estruturas de código mais organizadas e coesas, facilitando futuras modificações e expansões.

Estrutura do Capítulo
O capítulo detalha dez padrões de projeto, cada um apresentado com:

Contexto: Situação específica onde o padrão é aplicável.

Problema: Desafio de design enfrentado nesse contexto.

Solução: Implementação do padrão para resolver o problema.

A seguir, uma análise mais aprofundada de alguns desses padrões:

1. Fábrica (Factory)
   Contexto: Sistemas que necessitam criar objetos de diferentes tipos de forma dinâmica.

Problema: Instanciar diretamente classes específicas pode tornar o sistema rígido e difícil de expandir para novos tipos.

Solução: Implementar uma classe fábrica com métodos que retornam instâncias de interfaces ou classes abstratas, permitindo que subclasses definam quais objetos serão criados.

Exemplo prático: Em um sistema de comunicação que utiliza diferentes protocolos (TCP, UDP), uma fábrica pode criar objetos de comunicação apropriados com base na configuração, sem que o código cliente precise conhecer os detalhes de cada protocolo.

2. Singleton
   Contexto: Quando é necessário garantir que uma classe tenha apenas uma única instância em todo o sistema.

Problema: Múltiplas instâncias podem causar inconsistências, especialmente quando se trata de recursos compartilhados.

Solução: Definir uma classe com um método que verifica se a instância única já existe; se não, cria e retorna essa instância. O construtor da classe é privado para impedir a criação direta de objetos.

Exemplo prático: Um logger de sistema que registra eventos em um arquivo deve ser único para evitar conflitos de escrita e garantir a consistência dos logs.

1. Adaptador (Adapter)
   Contexto: Integração de classes com interfaces incompatíveis que precisam trabalhar juntas.

Problema: Modificar classes existentes para que sejam compatíveis pode não ser viável, especialmente quando se trata de código de terceiros.

Solução: Criar uma classe adaptadora que implemente a interface esperada pelo cliente e traduza as chamadas de métodos para a interface da classe existente.

Exemplo prático: Em um sistema que controla diferentes modelos de projetores, cada um com sua própria interface, um adaptador pode ser criado para cada modelo, permitindo que todos sejam controlados de maneira uniforme pelo sistema.

2. Fachada (Facade)
   Contexto: Sistemas complexos com múltiplas interações entre subsistemas.

Problema: A complexidade do sistema torna difícil para os clientes interagirem sem um conhecimento profundo de suas partes internas.

Solução: Criar uma classe fachada que fornece uma interface simplificada, delegando as chamadas apropriadas aos componentes internos conforme necessário.

Exemplo prático: Uma biblioteca de manipulação de arquivos que requer múltiplas classes para leitura e escrita pode oferecer uma fachada que simplifica essas operações para o usuário final.

3. Observador (Observer)
   Contexto: Situações onde um objeto precisa notificar outros sobre mudanças em seu estado.

Problema: Manter os objetos dependentes atualizados de maneira eficiente e desacoplada.

Solução: Definir uma interface de observador que outros objetos implementam. O objeto principal mantém uma lista de observadores e os notifica automaticamente sobre quaisquer mudanças de estado.

Exemplo prático: Em uma aplicação de monitoramento climático, sensores de temperatura podem notificar displays e sistemas de alerta sempre que houver uma mudança significativa na temperatura.

Considerações sobre o Uso de Padrões de Projeto
Embora os padrões de projeto ofereçam soluções valiosas, seu uso indiscriminado pode levar à complexidade desnecessária, fenômeno conhecido como "paternite". É crucial avaliar cuidadosamente a necessidade de um padrão específico, garantindo que sua aplicação traga benefícios reais ao projeto. Como destacado no capítulo, o uso excessivo de padrões pode resultar em projetos mais complicados do que o necessário.