Engenharia de Software Moderna

Marco Tulio Valente
https://engsoftmoderna.info/

Capítulo 5: Princípios de Projeto

Entendimento do Capítulo

O capítulo 5 do livro "Engenharia de Software Moderna" aborda os princípios fundamentais para o projeto de software, destacando como lidar com a complexidade que surge no desenvolvimento de sistemas. A ideia principal é que, para resolver problemas complexos, precisamos dividi-los em partes menores e mais simples, garantindo que essas partes possam ser desenvolvidas de forma independente. Além disso, o capítulo apresenta conceitos importantes como integridade conceitual, ocultamento de informação, coesão e acoplamento, além de introduzir os famosos princípios SOLID.

Uma coisa que me chamou atenção logo no início foi a citação sobre decomposição de problemas. Isso faz mais sentido, relacionando ao meu estágio, já percebi que tentar resolver tudo de uma vez só pode ser confuso. Dividir as tarefas em partes menores não só facilita o trabalho, mas também deixa o código mais organizado e fácil de entender.

Outra coisa que achei interessante foi como o autor explica que a complexidade dos sistemas modernos é inevitável, mas podemos combatê-la criando abstrações. Por exemplo, no desenvolvimento iOS, usamos bastante classes, protocolos e frameworks como UIKit ou SwiftUI para esconder detalhes complexos e focar apenas no que precisamos usar. Isso ajuda muito, principalmente quando estamos trabalhando em equipe ou lidando com códigos que outra pessoa escreveu.

Aplicação no Mercado (e no meu estágio)

Pela minha experiencia no mercado, percebo que muitos dos conceitos apresentados no capítulo são extremamente úteis no dia a dia. Vou dar alguns exemplos práticos de como esses princípios podem ser aplicados no meu contexto:

1. **Decomposição de Problemas**: muitas vezes recebo tarefas que parecem grandes e complicadas, como "implementar uma tela de login". Se eu tentasse fazer tudo de uma vez, provavelmente ficaria perdido. Em vez disso, aprendi a dividir o problema em partes menores, como:

- Criar a interface da tela (campos de texto, botões, etc.).
- Implementar a validação dos dados inseridos pelo usuário.
- Configurar a integração com a API para autenticação.
- Tratar erros, como credenciais inválidas ou problemas de conexão.
- Elaborar testes unitários de coberturas.

Essa abordagem modular não só facilita o desenvolvimento, mas também permite que diferentes partes sejam testadas separadamente. Por exemplo, posso testar a validação dos campos sem precisar configurar a API.

2. **Integridade Conceitual**: Integridade conceitual significa que o sistema deve ser consistente, tanto na forma como funciona quanto na forma como é implementado. No meu estágio, isso aparece muito na padronização de componentes visuais. Por exemplo, se usamos um botão azul com bordas arredondadas em uma tela, ele deve ter o mesmo estilo em todas as outras telas. Caso contrário, o aplicativo parece inconsistente e confuso para o usuário.

A falta de integridade conceitual causou problemas. Por exemplo se algumas telas usavam `UITableView` para exibir listas, enquanto outras usavam `UICollectionView`, mesmo quando o comportamento esperado era o mesmo. Isso gerava inconsistências no código e tornava a manutenção mais difícil. Agora, sempre que possível, tentamos padronizar as soluções para evitar esse tipo de problema.

3. **Ocultamento de Informação**: Esse princípio fala sobre esconder os detalhes internos de uma classe ou módulo e expor apenas o que é necessário. No desenvolvimento iOS, isso é algo que vejo muito ao trabalhar com APIs. Por exemplo, imagine que temos uma classe `APIManager` responsável por fazer requisições HTTP. Em vez de expor todos os detalhes, como URLs ou cabeçalhos, criamos métodos públicos como `login(email: String, password: String)` ou `fetchUserProfile()`. Assim, quem usa essa classe não precisa saber como ela funciona internamente, apenas como chamá-la.

Um outro exemplo seria integrar um serviço de pagamento no app. Criamos uma classe para encapsular toda a lógica de comunicação com o gateway de pagamento, expondo apenas métodos como `processPayment()` e `refundPayment()`. Isso facilita muito o uso dessa funcionalidade em outras partes do app, sem que precisássemos nos preocupar com os detalhes técnicos.

4. **Coesão e Acoplamento**: Esses dois conceitos andam juntos e são superimportantes. Coesão significa que uma classe ou módulo deve ter uma única responsabilidade, enquanto acoplamento diz respeito à dependência entre diferentes partes do sistema. Existem problemas com classes que fazem coisas demais. Por exemplo, uma classe que deveria apenas gerenciar o carrinho de compras também estava calculando impostos e enviando notificações. Isso dificulta a manutenção, porque qualquer mudança em uma parte poderia quebrar outra.

Para resolver isso, dividimos as responsabilidades em classes diferentes: uma para o carrinho, outra para cálculos de impostos e uma terceira para notificações. Isso deixa o código mais organizado e fácil de modificar no futuro.

5. **Princípios SOLID**: Os princípios SOLID são como um guia para escrever código limpo e bem estruturado.

- **Responsabilidade Única**: Para uma funcionalidade de busca no app. Antes, tínhamos uma classe que fazia tudo: recebia a entrada do usuário, consultava a API e formatava os resultados. Refatorar o código para separar essas responsabilidades em classes diferentes, como `SearchInputHandler`, `SearchService` e `SearchResultFormatter`.
- **Inversão de Dependências**: Esse princípio recomenda depender de abstrações (protocolos) em vez de implementações concretas. No caso da busca, criei um protocolo `SearchServiceProtocol` e fazemos a classe `SearchViewController` depender dele. Isso torna mais fácil trocar a implementação da busca (por exemplo, para usar um mock nos testes).

Exemplos Práticos

Exemplo 1: Decomposição no Desenvolvimento iOS

No capítulo, o autor usa o exemplo de um compilador para mostrar como dividir um problema em partes menores. No meu caso, algo parecido acontece ao implementar uma tela de listagem de produtos. Podemos dividir o problema assim:

- **Camada de Dados**: Criar um modelo `Product` e uma classe `ProductService` para buscar os dados da API.
- **Camada de Apresentação**: Criar uma `ProductViewModel` para preparar os dados para exibição.
- **Interface do Usuário**: Configurar um `UITableView` ou `UICollectionView` para exibir os produtos.

Essa divisão facilita muito o desenvolvimento, porque cada parte pode ser trabalhada e testada separadamente.

Exemplo 2: Ocultamento de Informação

No capítulo, o autor menciona o exemplo de um estacionamento onde a estrutura de dados interna (uma tabela hash) é encapsulada. Ao implementar um cache local para imagens no app, em vez de expor diretamente o dicionário usado para armazenar as imagens, criei métodos como `getImage(forKey:)` e `saveImage(_:forKey:)`. Isso permitiu alterar a implementação interna (por exemplo, trocar o dicionário por um banco de dados) sem impactar o restante do código.

Exemplo 3: Princípio Aberto/Fechado

O capítulo cita o exemplo da classe `Collections` em Java, que é aberta para extensões, mas fechada para modificações. Algo parecido ao criar um protocolo `Themeable` para aplicar temas no app. As classes que adotam esse protocolo podem implementar seus próprios estilos, mas o protocolo em si permanece inalterado.