Resenha do capítulo 5 - Princípios de Projeto

Ao ler o capítulo 5, percebi o quanto a organização de software é fundamental para criar sistemas eficientes e de fácil manutenção. A introdução do capítulo fala sobre a busca por soluções eficientes para problemas complexos, e isso me fez refletir sobre a importância de dividir um sistema em partes menores e mais gerenciáveis. O autor enfatiza que sistemas modernos têm uma complexidade cada vez maior, o que exige o uso de abstrações para simplificar a interação entre os componentes. Esse é um ponto crucial que se aplica diretamente à minha jornada de desenvolvimento: muitos dos problemas que enfrentei no meu T.i passado poderiam ter sido evitados se eu tivesse considerado essa abordagem de decomposição desde o início.

Quando o capítulo aborda o conceito de integridade conceitual, ficou claro como é essencial garantir que todas as funcionalidades de um sistema estejam alinhadas de forma coerente. A falta dessa integridade pode gerar um sistema bagunçado e difícil de entender, como ocorreu em alguns dos meus projetos, onde diferentes partes do sistema se sobrepuseram e criaram confusão. A integridade conceitual também está ligada à ideia de que decisões de projeto bem direcionadas evitam sistemas desorganizados, o que é algo que eu poderia ter considerado para melhorar a organização do meu sistema. Vejo como esse princípio poderia ter tornado meu trabalho mais eficiente e menos propenso a falhas, que ocorreram constantemente.

O conceito de ocultamento de informação, introduzido por David Parnas, foi outro ponto que me chamou atenção. A ideia de ocultar detalhes de implementação internos e expor apenas uma interface estável para os usuários de um sistema é um conceito importante. Quando estudei esse princípio, lembrei-me de como meu código muitas vezes acabou sendo difícil de modificar porque detalhes importantes estavam expostos demais. Se eu tivesse implementado o ocultamento de informações de maneira mais eficaz, teria conseguido fazer mudanças mais tranquilas no sistema sem quebrar outras funcionalidades. O exemplo do desenvolvimento de uma classe de controle de estacionamento, com a evolução de uma estrutura pública para uma encapsulada, foi muito bacana para entender melhor como a flexibilidade funciona, pensando em controlar melhor o acesso aos dados.

Sobre o conceito de acoplamento, entendi um pouco melhor como ele afeta a manutenção de um software. No meu T.i passado, algumas classes estavam muito dependentes umas das outras, o que dificultava fazer mudanças. Quando o capítulo falou sobre o acoplamento ruim ser aquele que ocorre com dependências diretas e instáveis, fiquei pensando sobre como poderia ter evitado essa fragilidade. Entender como maximizar a coesão e minimizar o acoplamento me ajudou a ver o quão importante é garantir que as partes do sistema possam evoluir de forma independente, sem causar danos colaterais.

Além disso, os princípios SOLID, que li vagamente enquanto estudava para a prova de programação modular, se conectaram com várias das ideias do capítulo. Eles visam melhorar a flexibilidade e a manutenção do código, princípios que vejo agora como essenciais para qualquer projeto de software. O Single Responsibility Principle, por exemplo, me fez pensar sobre como eu poderia ter mantido as classes mais focadas e com responsabilidades mais claras, o que teria facilitado o trabalho de manutenção e evolução. Quando o capítulo falou sobre o Open/Closed Principle, percebi que muitas vezes tive que modificar o código diretamente, quando, na verdade, deveria ter projetado o sistema de forma que ele fosse aberto para extensão sem modificar a estrutura existente. Esses princípios, ao serem aplicados corretamente, não apenas tornam o código mais fácil de manter, mas também mais flexível e preparado para evoluir.

A Lei de Demeter também me chamou atenção ao sugerir que as classes devem ter o mínimo de conhecimento possível sobre outras classes. Isso faz total sentido quando pensamos em como alterações inesperadas podem afetar o funcionamento do sistema. Aplicando esse princípio, pude perceber que no meu projeto passado, a falta de encapsulamento adequado causou problemas durante modificações, já que as classes estavam "conversando demais" entre si. Em vez de criar dependências excessivas, a Lei de Demeter propõe um design mais isolado e modular, o que facilita a evolução e o teste do sistema.

Por fim, o capítulo aborda as métricas de código fonte, e isso me ajudou a refletir sobre como posso avaliar a qualidade do que estou criando. As métricas de complexidade, como a complexidade ciclomática, são ferramentas valiosas para garantir que o código não se torne excessivamente complicado e difícil de manter. No meus futuros trabalhos, posso começar a aplicar essas métricas para identificar áreas onde o código precisa ser simplificado ou otimizado.

Percebi que a métrica de tamanho mais comum é o número de linhas de código, que conta as linhas de uma função, classe ou sistema. Embora útil para avaliar o tamanho, ela não deve ser usada como indicador de produtividade, pois não leva em consideração a complexidade ou a qualidade do código. Já a coesão de uma classe é medida pelo Lack of Cohesion Between Methods, que analisa a falta de coesão entre os métodos de uma classe. Quanto maior o seu valor, maior a falta de coesão.

Em relação ao acoplamento, o Coupling Between Objects é utilizado para medir o acoplamento estrutural entre classes, quantificando o número de dependências de uma classe com outras, como chamadas de métodos, acessos a atributos ou criação de objetos. Um valor mais alto de CBO indica maior acoplamento, o que pode prejudicar a manutenção do código. Por fim, a complexidade ciclomática, proposta por Thomas McCabe, mede a complexidade de uma função ou método, calculando o número de caminhos independentes no código com base no número

de decisões. Quanto maior a CC, maior a complexidade, o que pode dificultar a manutenção e os testes.

Essas métricas são úteis para identificar áreas problemáticas no código e orientar melhorias, mas. Este capítulo me fez pensar de uma forma diferente principalmente sobre como aplicar esses conceitos para melhorar a forma como estou desenvolvendo. Sobre princípios de encapsulamento, coesão e acoplamento, percebo que posso criar sistemas mais estruturados, e que recebam bem novas funcionalidades, resumindo que sejam preparados para evoluir, o que foi o calcanhar de aquiles do meu trabalho.