Resenha Engenharia de Software Moderna: Capítulo 5

O capítulo 5 do livro Engenharia de Software Moderna se inicia esclarecendo o conceito do que é um projeto de software. Projeto de software é a solução, cuja aplicação será feita por meio de software para solucionar um problema. Mas para esta definição fazer sentido, é necessário definir o que é problema, sendo, no caso de software como produto, a dor de um cliente ou do P.O. Porém, muitos desses problemas tem um grau de complexidade muito alto para se projetar uma solução como um todo, assim, cria-se a necessidade de decompor o problema em problemas menores, para que cada uma dessas soluções possa ser implementada de maneira independente. Por exemplo, ao se projetar um sistema para um estacionamento, é necessário elicitar todas as funções que esse sistema deve comportar para sanar a dor do cliente, tais como: estacionar, manter histórico de clientes, calcular o lucro, dentre outras. Essa decomposição tem o fim de combater a complexidade do sistema.

A integridade conceitual por sua vez defende que, um sistema não deve ser *bloated* com diversas funcionalidades que não comunicam entre si. No caso do estacionamento, seria um mal exemplo de integridade conceitual a presença de duas tabelas distintas, com uma mostrando o lucro diário do estacionamento em BRL como um inteiro, e outra tabela distinta mostrando o lucro semanal em USD como um double. Em software, também pode se dar exemplos de classes distintas em um mesmo sistema utilizando semânticas distintas para variáveis, ou APIs diferentes para funções similares, criando assim um sistema que não é coeso consigo mesmo.

Outra parte importante em um software bem projetado é a presença do Ocultamento de Informações. Conceito esse que foi discutido pela primeira vez em 1972, por David Parnas, que antes mesmo da programação orientada a objetos ter sido oficialmente cunhada, já discutia a modularização de componentes de software. E que traz uma série de vantagens como o desenvolvimento em paralelo, que, uma vez que, módulos do sistema são separados e ocultos um do outro, desenvolvedores diferentes serão capazes de trabalhar de forma independente e paralela. Outro benefício que o ocultamento de informações traz ao sistema é o de flexibilidade a mudanças, pois, com os módulos contendo um certo grau de independência, é mais fácil de se refatorar um único módulo sem, ou com, baixa necessidade de se alterar outros módulos do software. Além de também trazer a facilidade de entendimento, uma vez que as funções de cada módulo estiverem claras, se torna menos laborioso o processo de entendimento do sistema como um todo. Porém para se obter todas essas vantagens é

necessário o uso de modificadores de acesso para variáveis e métodos, deixando-os, respectivamente, privados e públicos. Assim transformando os métodos de uma classe que possam ser chamados por código externo de interfaces, que por sua vez devem ser estáveis e com baixa necessidade de refatoração.

O acoplamento é uma força de coesão entre duas classes, ou seja, o quanto uma classe consegue conversar e usar de métodos e atributos de outra classe, e no geral, é algo desejável em um projeto de um software complexo. Porém, existem dois tipos de acoplamento, os que não são somente aceitáveis, mas também necessários para o pleno funcionamento de um sistema, que é quando, uma classe B utiliza apenas interfaces de uma classe A. E o ruim, que é quando uma classe possui várias funções atreladas a si, e com baixa estabilidade, capacidade de modificação e ou refatoração. Como por exemplo, retornando ao estacionamento, um bom exemplo de acoplamento é aquele que ocorre quando uma classe que calcula o lucro recebe apenas as informações necessárias da classe vaga para realizar suas operações. E um exemplo do mal acoplamento seria ter essas duas funções presentes em uma mesma classe.

Porém, o para se atingir todos os objetivos e no final obter um software bem projetado, é necessário que todos os seus componentes e decisões de projeto sejam suscetíveis a mudanças. Porém, se houver um grau muito alto de encapsulamento e um grau muito baixo de acoplamento, as classes param de dialogar com o sistema, assim, criando a necessidade de se ter métodos públicos estáveis (interfaces)apesar dos atributos privados, pois, assim, com eles, uma espécie de ponte é criada entre a classe encapsulada e o resto do sistema.

Principios SOLID, é um acrônimo para uma série de regras criadas para projetos de software, sendo elas:

Single Responsibility Principle (Cada parte do sistema cuida dela e somente dela)

Open/Closed Principle (Classe deve ser fechada a modificações e aberta a extensões)

Liskov Substitution Principle (Que permite que classes derivadas de uma outra classe possam ser utilizadas no lugar da classe original)

Interface Segregation Principle (define que interfaces têm que ser enxutas, coesas e específicas para a sua função)

Dependency Inversion Principle (Prefira interfaces a classes para realizar determinadas funções)

Métricas de código fonte, embora, atualmente, não sejam mais tão utilizadas, ainda fazem parte de princípios de projeto, sendo elas:

Tamanho, quantas linhas de código uma determinada classe tem, que pode ser utilizada como métrica, analisando o total de linhas escritas e ignorando linhas vazias. Porém, linhas escritas nem sempre são sinônimos de produtividade ou de desempenho da classe, uma vez que, Ken Thompson, um dos criadores do sistema operacional UNIX, considera que seu dia de trabalho mais produtivo foi aquele que ele deletou mais de mil linhas de código de um sistema.

Coesão, uma das maneiras mais conhecidas de se metrificar a coesão de um código é por meio de um método chamado de LCOM (lack of cohesion between methods). Que por sua vez mede a falta de coesão entre métodos, de uma maneira que, quanto maior o LCOM, maior vai ser a falta de coesão de um sistema.

Acoplamento, há o CBO, que mede o acoplamento estrutural entre duas classes de objetos.

Complexidade, o conceito de complexidade relaciona-se com a dificuldade de manter e testar uma função. Principalmente em relação a ifs e afins, pois cada decisão no fluxo de código adiciona uma unidade de complexidade. Que pode ser medida com a fórmula CC = N + 1;