Quer ver eu te fazer uma pergunta que te fará pensar o resto do dia?

**Como olhar para um código e dizer que ele é de qualidade?**

Essa é sem dúvida uma pergunta que todos nós tentamos responder todos os dias.

Como bem sabemos, podemos olhar trechos de código por vários pontos de vista diferentes: o quão complexo ele é (muitos ifs, muitas linhas), o quão coeso ele é, o quão acoplado ele é, etc. Sendo tão difícil pensar em qualidade de código, vamos tentar mudar o nível. Quando que um sistema tem qualidade interna, ou seja, do ponto de vista de código?

*Nós gostamos de sistemas que sejam fáceis de mexer. Ou seja, quando o usuário final pede uma mudança, é relativamente fácil de localizar onde ela deve ser feita e, depois de feita, não há propagação de problemas.*

Para que isso aconteça, o código deve estar bem modularizado; cada classe deve ter sua responsabilidade, e as relações entre elas devem estar bem definidas. É aqui que entra a ideia do código sólido, princípios criados por Michael Feathers há bastante tempo.

A brincadeira com o termo “código sólido” vem do acrônimo SOLID. Cada letra representa um dos 5 princípios de orientação a objetos que nos ajudam a manter o código organizado:

•**Single Responsibility Principle (SRP), ou, Princípio da Responsabilidade Única.** Esse princípio diz que as classes devem ser coesas, ou seja, terem uma única responsabilidade. Classes assim tendem a ser mais reutilizáveis, mais simples, e propagam menos mudanças para o resto do sistema.

•**Open Closed Principle (OCP), ou Princípio do Aberto Fechado.**Diz que as classes devem poder ter seu comportamento facilmente estendidas quando necessário, por meio de herança, interface e composição. Ao mesmo tempo, não deve ser necessário abrir a própria classe para realizar pequenas mudanças. No fim, o princípio diz que devemos ter boas abstrações espalhadas pelo sistema.

•**Liskov Substitution Principle (LSP), ou Príncipio da Substituição de Liskov.**Esse princípio diz que precisamos ter cuidado para usar herança. Herança é um mecanismo poderoso, mas deve ser usado com parcimônia, evitando os casos de Gato-estende-Cachorro, apenas por possuírem algo em comum.

•**Interface Segregation Principle (ISP), ou Princípio da Segregação de Interfaces.**Esse princípio diz que nossos módulos devem ser enxutos, ou seja, devem ter poucos comportamentos. Interfaces que tem muitos comportamentos geralmente acabam se espalhando por todo o sistema, dificultando manutenção.

•**Dependency Inversion Principle (DIP), ou Princípio da Inversão de Dependências.**Esse princípio diz que devemos sempre depender de

abstrações, afinal abstrações mudam menos e facilitam a mudança de comportamento e as futuras evoluções do código.

Se conseguirmos seguir todas essas dicas, teremos código fácil de evoluir. As mudanças serão feitas em pontos específicos, e problemas não serão propagados.
O problema é que todos nós sabemos que não é fácil escrever código seco e sólido. Eu fui bem sucinto na explicação de cada um dos princípios. É possível discutir todos eles com muito mais profundidade.
Vale lembrar também o código seco (ou DRY): o programador não deve espalhar o mesmo código por todo o sistema; ele deve ser reutilizado. DRY vem de “Don’t Repeat Yourself”. Todos nós sabemos que repetição de código é um problema gravíssimo; eles dificultam a manutenção e propagam erros muito rapidamente.

http://blog.caelum.com.br/principios-do-codigo-solido-na-orientacao-a-objetos/