Rubens Pereira Ribeiro
Thiago Mafei

1. Boas práticas em programação Java

“Aprender Java: Boas práticas para um bom projeto de software”, Artigo publicado por Ronaldo Lanhellas, do site Devmedia, aborda alguns princípios básicos e muito importantes para o desenvolvimento de um bom projeto, código e sistema. Traz também algumas dicas muito interessantes para que se possa trabalhar de forma produtiva, organizada e padronizada. Apesar de existirem diversos métodos e características para desenvolver um bom projeto de software, este artigo foca em alguns dos mais importantes e muita das vezes ignorados, por pressa, inexperiência ou falta de conhecimento necessário.

O objeto é focar em boas práticas, não sobre explicação de códigos ou ferramentas. O artigo mostra como aplicar tais características. Mostra em Java, mas, o foco não é a técnica e sim o conceito, então você poderá simplesmente aplicar em outra linguagem Orientada a Objetos.

2. Generalizar para reutilizar

A primeira característica é a generalização ao máximo do seu código para torná-lo reutilizável, quantas vezes forem necessárias. Generalizar, em outras palavras, é tornar seu sistema poderoso o suficiente para adaptar-se a várias situações, mesmo aquelas que você nem pensou que podem ocorrer.

Pensando no seguinte cenário: você deve desenvolver uma classe chamada 'CaixaSurpresa' que será utilizada em diversos locais do seu sistema. Na verdade, são outros desenvolvedores que irão utilizá-la, mas você é o responsável por criá-la. Nessa Caixa Surpresa pode haver diversos tipos de objetos: bolas, tesouras, papéis e assim por diante. Sua classe deve ter um método para retornar uma lista de objetos de acordo com o que for solicitado.

Então você não sabe quais os objetos podem ser retornados (são inúmeros) e é inviável e impraticável ficar criando um "*get*ObjetoX(), *get*ObjetoY()...". Nossa solução foi usar a técnica de '*Generics*' do Java para solucionar esse problema, mas se você tiver outra ideia que também resolva e torne seu código Genérico o suficiente para resolver tal problema, vá em frente.

3.Centralizar para facilitar

O autor aponta que é muito comum vermos projetos com diversos códigos espalhados, que fazem exatamente a mesma coisa. Isso ocorre principalmente quando estamos trabalhando com projetos que possuem mais de uma pessoa envolvida, pois uma desenvolve uma lógica em uma local e outra pensando que já não foi desenvolvido, criando a mesma coisa, mas em outro local.

A dica aqui é centralizar lógicas que serão utilizadas em muitos locais do sistema em uma classe *Helper*. Assim sendo, todos procurarão dentro do seu *Helper* antes de criar um novo método que faça a mesma coisa. Sistemas bem projetos tem classes como: CalculosDeImpostos*Helper*, ConversorDeDatas*Helper*, ConversorDeUnidades*Helper* e assim por diante. Dá para se entender, apenas pelo nome, o que cada classe faz, então se você é novo em um projeto que já tem um tempo de "estrada", obviamente você procurará dentro da classe CalculosDeImpostos*Helper* antes de criar um método para calcular um ICMS de um produto, certo? Não é regra, mas geralmente as classes *Helper's* tem seus métodos com a assinatura "*public static*".

4.Classes enxutas

Outro ponto importante é, nada de classes com dez mil linhas de códigos para mostrar que você teve muito trabalho. Trabalho mesmo você terá após cinco anos, quando precisar dar manutenção nesta classe e não encontrar o que está procurando. Use o velho conceito de "Dividir para Conquistar", quando você perceber que sua classe está ficando muito grande, comece a criar classes auxiliares, *helpers*, mas tire toda carga extra da sua classe principal.

Um exemplo muito simples: se você está construindo uma classe que fará um CRUD de Funcionários, você não deve colocar nela cálculos de folha de pagamento, impressão de crachá, entre outras funcionalidades que fogem ao foco inicial que era apenas um CRUD.

5.Comentários Úteis

Para Ronaldo Lanhellas, este é um erro comum e quase frequente em muitos códigos onde você vê milhões de comentários, mas 90% deles são inúteis.

Comentário é um artifício ótimo quando utilizado da forma correta, porém a utilização de comentários desnecessários acaba tornando seu código ainda mais complexo de entender, pois ele se torna mais extenso, ou seja, a leitura se torna impraticável.

6.Não 'formate' (identar) o código de outra pessoa

Essa dica vale para quem trabalha em um mesmo projeto com mais pessoas, o que acaba causando problemas caso você não saiba a maneira certa de se portar quando analisa ou refatora o código de outro profissional.

É muito comum, ao estar trabalhando com classes desenvolvidas por outras pessoas, você perceber falta de identação, não por descuido do outro profissional, mas talvez pela configuração da IDE dele ser diferente da sua, ou seja, a identação que ele aplica é diferente da identação que você aplica. O aconselhável jamais identar códigos de terceiros, a não ser que o mesmo permita.

Imagine você que desenvolve a classe Y e passa para outro profissional criar um método ABC, então quando ele devolve a classe Y, você percebe que sua classe está totalmente diferente, apenas porque ele fez uma identação completa do código.

Pode parecer até absurdo tal característica ser foco em nosso artigo, mas na verdade ela é indispensável no dia a dia do trabalho em grupo, caso contrário, você terá muita dor de cabeça com esse tipo de problema.

O foco principal do artigo foi demonstrar pontos chaves para o desenvolvimento de bons projetos, tentando abstrair ao máximo a tecnologia e ferramentas. Assim você pode aplicar esses conceitos nas mais diversas linguagens, não apenas em Java como foi demonstrado neste artigo.

O mais interessante neste artigo é a explicação e exemplificação de assuntos e “Manias” que podem prejudicar todo um projeto.