02

Problemas do alto acoplamento

Transcrição

Um dos principais assuntos discutidos quando começamos a trabalhar com Orientação a Objetos é a questão de manter o baixo acoplamento e a alta coesão.

Crescemos no mundo OO ouvindo isso de todos os lados, mas de fato, qual o real problema que o alto acoplamento nos traz? Primeiro, precisamos entender o que é um código acoplado.

Um código com alto acoplamento é quando uma classe conhece muitos detalhes de suas dependências, de modo que qualquer alteração em uma das dependências faz com que a classe que possui a dependência tenha que ser alterada.

Em nosso código, temos um exemplo claro do quão o alto acoplamento pode ser maléfico. A classe `DAO` requisita dentro do método `existe` um objeto do tipo `EntityManager`, através da classe `JPAUtil`, onde centralizamos a criação dos nossos `EntityManager`s.

Suponha que a classe `JPAUtil` precise agora instanciar `EntityManager`s de algum outro banco de dados e não somente do único banco presente no arquivo `persistence.xml`. Precisaríamos, neste caso, que a classe `JPAUtil` fosse alterada para ter dois tipos de `EntityManagerFactory`, uma para cada unidade de persistência que trabalhamos.

A classe `JPAUtil` precisará saber a partir de qual `EntityManagerFactory` ela criará as instâncias de `EntityManager`. Uma possível estratégia, seria passar na chamada do método `getEntityManager` uma `String` ou algo similar, que diga à classe `JPAUtil` qual `EntityManagerFactory` utilizar para criar o `EntityManager`. Teríamos um código parecido com este:

```
public class JPAUtil {

    private static EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("livraria");

    private static EntityManagerFactory emf2 =
        Persistence.createEntityManagerFactory("banco");

    public EntityManager getEntityManager(String factory) {
        if ("notas".equals(factory)) {
            return emf.createEntityManager();
        } else if("notas2".equals(factory)) {
            return emf2.createEntityManager();
        }
    }
}
```

Automaticamente, teremos um erro de compilação dentro da classe `DAO` e não somente nela, mas em todas as classes que fazem referência a classe `JPAUtil`, e que na atual abordagem que estamos usando, podem vir a se tornar muitas. Um claro indício que nosso código está acoplado demais.

Um outro teste simples seria introduzir um novo parâmetro no construtor do `DAO`, por exemplo:

```
public DAO(Class<T> classe, boolean loga) {
    this.classe = classe;
}
```

Logo recebemos vários erros de compilação. Estamos espalhando a criação do nosso `DAO` pelos *beans* do JSF, praticamente sem controle. Mais um claro indício que nosso código está acoplado demais.

Estratégias para diminuir o acoplamento

Vimos que a maneira que estamos trabalhando atualmente em nosso design está mais atrapalhando do que ajudando. Isso graças ao alto grau de acoplamento que obtivemos ao longo do tempo. Mas como resolver isso? Como desacoplar o código e não afetá-lo com qualquer mudança?

A fábrica de `EntityManager` que criamos e chamamos de `JPAUtil` pode ser considerada uma forma de diminuir o acoplamento em nossas classes. Ela nos poupa de termos que criar o `EntityManagerFactory` toda vez que precisarmos de um.

Porém, como vimos anteriormente, ela nos ajuda mas ainda assim o acoplamento é grande.

O grande problema é que o código da classe `DAO` busca a dependência que precisa para realizar suas tarefas. Isso também aplica-se para os *beans* do JSF, todos eles buscam as dependências. E se ao invés de buscarmos esta dependência, recebêssemos ela? A classe `DAO` passaria agora a receber qualquer `EntityManager` , não importando de onde ele veio. Em nossos testes, poderíamos criar o `EntityManager` com base em um banco de dados em memória, por exemplo, o **HyperSql Database** (HSQLDB), ou ainda nem usar um banco de dados, simulando o objeto.

Inversão de controle

Queremos arrumar a nossa camada de persistência, mas infelizmente o JSF **sozinho** não consegue resolver esses problemas de acoplamento.

Precisamos de um *framework* que consegue gerenciar o `EntityManager` e o nosso `DAO` . Todos esses *new*s espalhados pela aplicação devem sumir!

O *framework* que tem todo esse poder dentro da especificação JavaEE é o **CDI**. O "**C**ontext and **D**ependency **I**njection for the Java EE platform" (CDI) é uma especificação que diz como devemos trabalhar com nossas dependências. Ele assume o gerenciamento delas, define quanto tempo vivem e quando criar.

Nosso objetivo então é integrar o JSF com CDI para arrumar a nossa camada de persistência, mas antes de realmente arrumar a casa, vamos integrar os dois *frameworks*.