02

Façades e Singletons

Alguns sistemas são compostos por muitos outros sub-sistemas. Imagine o caso de uma grande empresa, que tem sistemas de faturamento, cobrança, contato ao cliente, etc.

Em muitos casos, esses sistemas precisam se comunicar. O sistema de faturamento precisa gerar uma cobrança, que por sua vez, precisa disparar um contato ao cliente, é um exemplo.

Agora imagine uma classe cliente, tentando consumir todos esses serviços de uma vez? Como fazer?

Uma primeira alternativa é conhecer cada um dos sistemas, e fazer uso dos objetos de domínio de cada sistema:

```
String cpf = /// pega cpf
Cliente cliente = new ClienteDao().buscaPorCpf(cpf);

Fatura fatura = new Fatura(cliente, valor);

Cobranca cobranca = new Cobranca(Tipo.BOLETO, fatura);
cobranca.emite();

ContatoCliente contato = new ContatoCliente(cliente, cobranca);
contato.dispara();
```

Se temos muito sub-sistemas, e mais complexos do que o código acima, pode ser difícil para um cliente fazer uso de todos esses objetos.

Poderíamos então dar uma interface única e mais amigável para esses objetos de domínio. Dessa forma, a classe cliente consumiria apenas essa interface única, sem precisar conhecer cada objeto:

```
public class EmpresaFacade {

    public Cliente buscaCliente(String cpf) {
        return new ClienteDao().buscaPorCpf(cpf);
    }
    public Fatura criaFatura(Cliente cliente, double valor) {
        Fatura fatura = new Fatura(cliente, valor);
        return fatura;
    }

    public Cobranca geraCobranca(Fatura fatura) {
        Cobranca cobranca = new Cobranca(Tipo.BOLETO, fatura);
        cobranca.emite();

        return cobranca;
    }

    public ContatoCliente fazContato(Cliente cliente, Cobranca cobranca) {
        ContatoCliente contato = new ContatoCliente(cliente, cobranca);
        contato.dispara();

        return contato;
```

```
    }
}
```

Esse padrão de projeto, que provê uma "fachada" para os serviços disponibilizados pelos sub-sistemas, é conhecido pelo nome de **Façade**.

Como essa classe provê acesso a todos os outros sub-sistemas, é bem comum também que não haja mais de uma instância dessa classe espalhada pelo sistema.

Neste curso já estudamos o Flyweight, que nos ajuda a controlar a quantidade de instâncias de um objeto. Mas o Flyweight cuida de diversas instâncias, e aqui só temos uma.

Vamos resolver isso com uma única classe, que servirá de "fábrica", e sempre retornará a mesma instância:

```
public class EmpresaFacadeSingleton {

    private static EmpresaFacade instancia;

    public EmpresaFacade getInstancia() {
        if(instancia == null) {
            instancia = new EmpresaFacade();
        }

        return instancia;
    }
}
```

Para pegar uma instância, agora ficou fácil:

```
EmpresaFacade fachada = new EmpresaFacadeSingleton().getInstancia();
```

Para inclusive garantir que ninguém instanciará a classe `EmpresaFacade` diretamente, vamos mudar seu construtor para `protected`. Assim somente o singleton, que está no mesmo pacote, conseguirá!

A implementação acima é o que chamamos de `Singleton`. Ele faz com que só exista uma única instância da classe em todo o sistema!

Mas, apesar dos dois padrões serem bastante populares, seu uso deve ser feito com cautela. O Singleton, por exemplo, quando mal utilizado, acaba por permitir ao usuário a utilização de variáveis globais (que é uma coisa que a programação procedural já mostrou que é problemático).

Uma Façade tende a ser muito acoplada, e ter uma interface gorda. Façades menores podem até ser úteis, mas também devem ser usadas com parcimônia.