02

Bridges e Adapters

Hoje temos muitos serviços similares que são prestados por diferentes empresas. Um bom exemplo disso é o serviço de mapas. Se quisermos buscar a localização de uma rua, por exemplo, podemos fazer uso do Google Maps, Maplink, entre outras.

Mas, apesar de prestarem os mesmos serviços, cada um o faz de uma maneira diferente, através de uma interface diferente. O Google Maps, por exemplo, para fazer uma busca, precisamos invocar a URL `https://maps.google.com.br/maps?q=endereco+aqui` . Já o Maplink, precisamos consumir um serviço web que faz uso de SOAP.

Vamos então usar o serviço do Google no meio do nosso sistema:

```
// regra de negocio

URL google = new URL(google);
InputStream stream = google.openStream();
// le o stream e pega o conteudo

// faz algo com o mapa
```

Nessa altura, já não precisamos discutir que esse código é problemático. Afinal, se precisarmos trocar a API de mapas, precisaremos mudar em todo lugar que faz uso dessa API.

A solução para isso? Criar uma abstração. Vamos criar então uma interface, `Mapa` , por exemplo, que abstrairá isso:

```
public interface Mapa {
   String devolveMapa(String rua);
}
```

Com a interface, podemos ter N implementações concretas. Vamos fazer a do Google:

```
public class GoogleMaps implements Mapa {
  public String devolveMapa(String rua) {
    // codigo aqui que consome o google
  }
}
```

Com isso, fazemos uso da interface ao longo do nosso programa:

```
Mapa mapa = // uma instância da mapa;
String conteudo = mapa.devolveMapa("Rua Vergueiro, 3185");
```

Com isso, conseguimos mudar facilmente a implementação ao longo do nosso código, já que dependemos de uma interface.

Repare que essa interface é uma "ponte" para a implementação concreta. O nome desse padrão de projeto é justamente esse: **Bridge**.

Mas, às vezes o serviço que precisamos abstrair não é nem um serviço de terceiro, mas sim algum código legado existente. É comum que, no processo de refatoração de um sistema, joguemos fora as classes antigas aos poucos. É comum também que algumas classes não possam ser descartadas, e essas classes "feias" acabam por sujar o novo domínio que está sendo criado, de maneira mais elegante.

Um exemplo comum são as APIs de datas na maioria das linguagens de programação. Elas são geralmente feias, e não muito testáveis.

No Java, para pegarmos a data atual, fazemos `Calendar.getInstance()`. Esse método estático pode complicar desenvolvedores que gostam de testabilidade.

Para resolver isso, podemos colocar uma "camada" por cima dessa API de data. Por exemplo:

```
class Relogio {
  public Calendar hoje() {
    return Calendar.getInstance();
  }
}
```

Veja que o método `hoje()` consome a API legada. Todo o resto do sistema faz uso da classe `Relogio`, que é a nova interface, mais limpa e clara:

```
Calendar agora = new Relogio().hoje();
```

Quando temos um conjunto de classes legadas, que achamos que seu uso vai sujar o novo sistema, criamos um "adaptador" que facilita sua utilização. O nome desse padrão de projetos é **Adapter**.