refactoring.guru

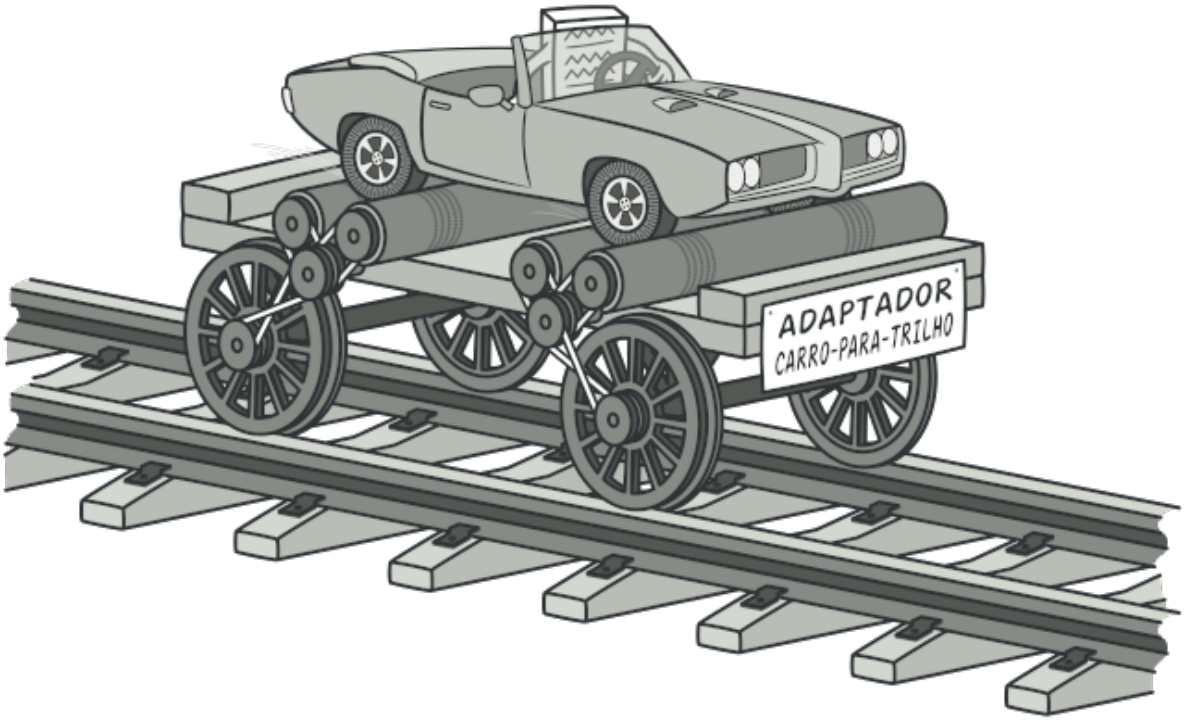
Adapter

11–15 minutes

Também conhecido como: Adaptador, Wrapper

Propósito

O **Adapter** é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

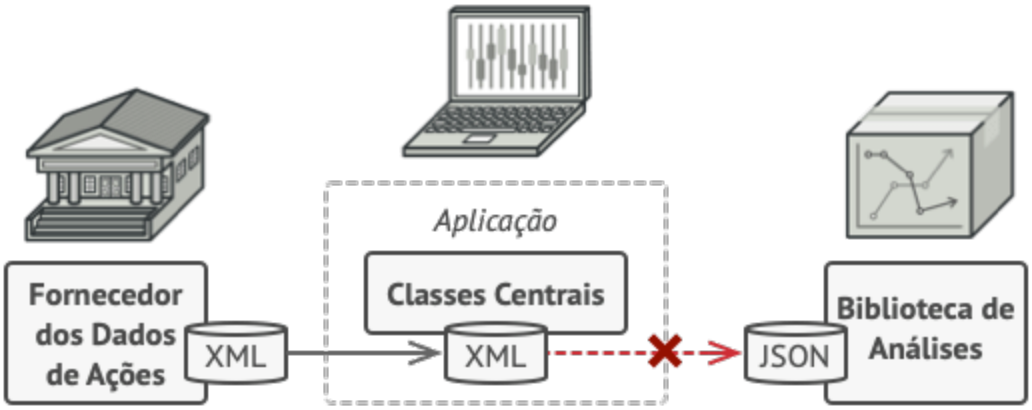


Problema

Imagine que você está criando uma aplicação de monitoramento do mercado de ações da bolsa. A aplicação baixa os dados as

ações de múltiplas fontes em formato XML e então mostra gráficos e diagramas maneiros para o usuário.

Em algum ponto, você decide melhorar a aplicação ao integrar uma biblioteca de análise de terceiros. Mas aqui está a pegadinha: a biblioteca só trabalha com dados em formato JSON.



Você não pode usar a biblioteca “como ela está” porque ela espera os dados em um formato que é incompatível com sua aplicação.

Você poderia mudar a biblioteca para que ela funcione com XML. Contudo, isso pode quebrar algum código existente que depende da biblioteca. E pior, você pode não ter acesso ao código fonte da biblioteca para começo de conversa, fazendo dessa abordagem uma tarefa impossível.

Solução

Você pode criar um *adaptador*. Ele é um objeto especial que converte a interface de um objeto para que outro objeto possa entendê-lo.

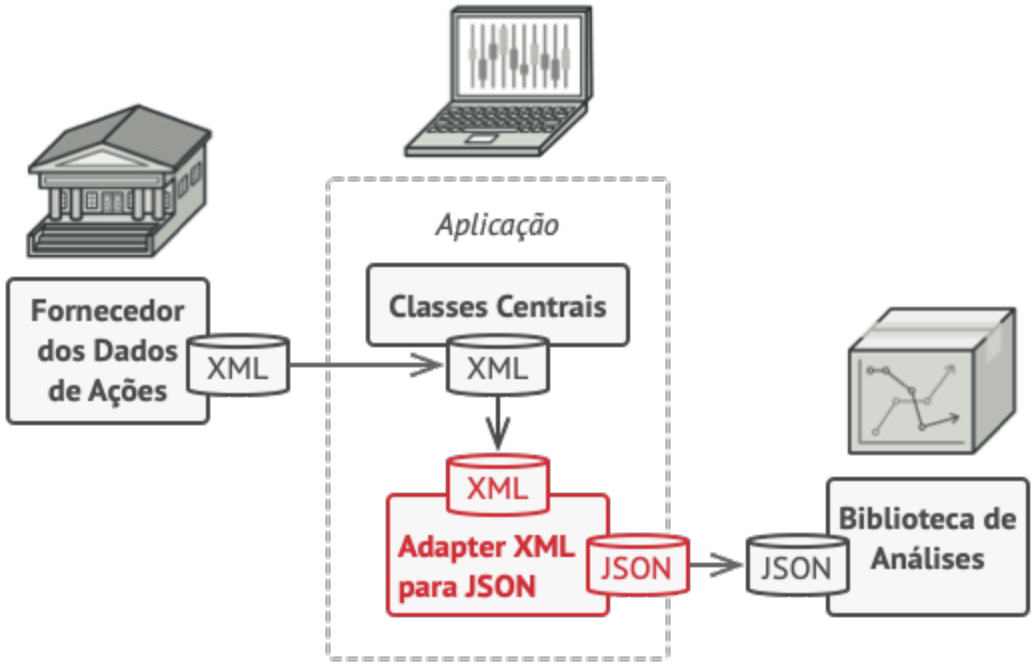
Um adaptador encobre um dos objetos para esconder a complexidade da conversão acontecendo nos bastidores. O objeto encobrido nem fica ciente do adaptador. Por exemplo, você pode

encobrir um objeto que opera em metros e quilômetros com um adaptador que converte todos os dados para unidades imperiais tais como pés e milhas.

Adaptadores podem não só converter dados em vários formatos, mas também podem ajudar objetos com diferentes interfaces a colaborar. Veja aqui como funciona:

1. O adaptador obtém uma interface, compatível com um dos objetos existentes.
2. Usando essa interface, o objeto existente pode chamar os métodos do adaptador com segurança.
3. Ao receber a chamada, o adaptador passa o pedido para o segundo objeto, mas em um formato e ordem que o segundo objeto espera.

   Algumas vezes é possível criar um adaptador de duas vias que pode converter as chamadas em ambas as direções.



Vamos voltar à nossa aplicação da bolsa de valores. Para resolver

o dilema dos formatos incompatíveis, você pode criar adaptadores XML-para-JSON para cada classe da biblioteca de análise que seu código trabalha diretamente. Então você ajusta seu código para comunicar-se com a biblioteca através desses adaptadores. Quando um adaptador recebe uma chamada, ele traduz os dados entrantes XML em uma estrutura JSON e passa a chamada para os métodos apropriados de um objeto de análise encoberto.

Analogia com o mundo real

Uma mala antes e depois de uma viagem ao exterior.

Quando você viaja do Brasil para a Europa pela primeira vez, você pode ter uma pequena surpresa quando tenta carregar seu laptop. O plugue e os padrões de tomadas são diferentes em diferentes países. É por isso que seu plugue do Brasil não vai caber em uma tomada da Alemanha. O problema pode ser resolvido usando um adaptador de tomada que tenha o estilo de tomada Brasileira e o plugue no estilo Europeu.

Estrutura

Adaptador de objeto

Essa implementação usa o princípio de composição do objeto: o adaptador implementa a interface de um objeto e encobre o outro. Ele pode ser implementado em todas as linguagens de programação populares.

1. O **Cliente** é uma classe que contém a lógica de negócio do programa existente.
2. A **Interface do Cliente** descreve um protocolo que outras classes devem seguir para ser capaz de colaborar com o código cliente.
3. O **Serviço** é alguma classe útil (geralmente de terceiros ou código legado). O cliente não pode usar essa classe diretamente porque ela tem uma interface incompatível.
4. O **Adaptador** é uma classe que é capaz de trabalhar tanto com o cliente quanto o serviço: ela implementa a interface do cliente enquanto encobre o objeto do serviço. O adaptador recebe chamadas do cliente através da interface do cliente e as traduz em chamadas para o objeto encobrido do serviço em um formato que ele possa entender.
5. O código cliente não é acoplado à classe concreta do adaptador desde que ele trabalhe com o adaptador através da interface do cliente. Graças a isso, você pode introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente. Isso pode ser útil quando a interface de uma classe de serviço é mudada ou substituída: você pode apenas criar uma nova classe adaptador sem mudar o código cliente.

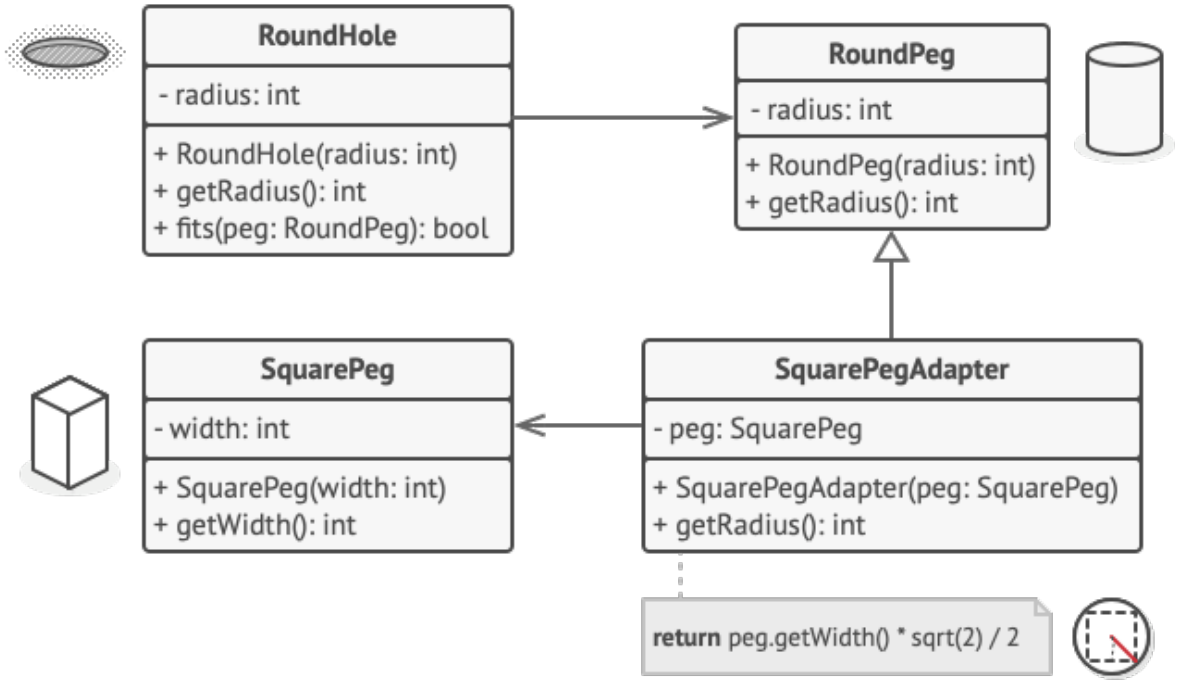
Adaptador de classe

Essa implementação utiliza herança: o adaptador herda interfaces de ambos os objetos ao mesmo tempo. Observe que essa abordagem só pode ser implementada em linguagens de programação que suportam herança múltipla, tais como C++.

1. A **Classe Adaptador** não precisa encobrir quaisquer objetos porque ela herda os comportamentos tanto do cliente como do serviço. A adaptação acontece dentro dos métodos sobrescritos. O adaptador resultante pode ser usado em lugar de uma classe cliente existente.

Pseudocódigo

Esse exemplo do padrão **Adapter** é baseado no conflito clássico entre pinos quadrados e buracos redondos.



Adaptando pinos quadrados para buracos redondos.

O adaptador finge ser um pino redondo, com um raio igual a metade do diâmetro do quadrado (em outras palavras, o raio do menor círculo que pode acomodar o pino quadrado).

```
// Digamos que você tenha duas classes com interfaces
// compatíveis: RoundHole (Buraco Redondo) e RoundPeg (Pino
// Redondo).
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Retorna o raio do buraco.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Retorna o raio do pino.


// Mas tem uma classe incompatível: SquarePeg (Pino Quadrado).
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Retorna a largura do pino quadrado.


// Uma classe adaptadora permite que você encaixe pinos
// quadrados em buracos redondos. Ela estende a classe
RoundPeg
```

```
// para permitir que objetos do adaptador ajam como pinos
// redondos.
class SquarePegAdapter extends RoundPeg is
    // Na verdade, o adaptador contém uma instância da classe
    // SquarePeg.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // O adaptador finge que é um pino redondo com um raio
        // que encaixaria o pino quadrado que o adaptador está
        // envolvendo.
        return peg.getWidth() * Math.sqrt(2) / 2


// Em algum lugar no código cliente.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
// Isso não vai compilar (tipos incompatíveis).
hole.fits(small_sqpeg)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
```

```
hole.fits(large_sqpeg_adapter) // false
```

Aplicabilidade

Utilize a classe Adaptador quando você quer usar uma classe existente, mas sua interface não for compatível com o resto do seu código.

O padrão Adapter permite que você crie uma classe de meio termo que serve como um tradutor entre seu código e a classe antiga, uma classe de terceiros, ou qualquer outra classe com uma interface estranha.

Utilize o padrão quando você quer reutilizar diversas subclasses existentes que não possuam alguma funcionalidade comum que não pode ser adicionada a superclasse.

Você pode estender cada subclasse e colocar a funcionalidade faltante nas novas classes filhas. Contudo, você terá que duplicar o código em todas as novas classes, o que cheira muito mal.

Uma solução muito mais elegante seria colocar a funcionalidade faltante dentro da classe adaptadora. Então você encobriria os objetos com as funcionalidades faltantes dentro do adaptador, ganhando tais funcionalidades de forma dinâmica. Para isso funcionar, as classes alvo devem ter uma interface em comum, e o campo do adaptador deve seguir aquela interface. Essa abordagem se parece muito com o padrão Decorator.

Como implementar

1. Certifique-se que você tem ao menos duas classes com interfaces incompatíveis:

- Uma classe *serviço* útil, que você não pode modificar (quase sempre de terceiros, antiga, ou com muitas dependências existentes).
- Uma ou mais classes *cliente* que seriam beneficiadas com o uso da classe serviço.

2. Declare a interface cliente e descreva como o cliente se comunica com o serviço.
3. Cria a classe adaptadora e faça-a seguir a interface cliente. Deixe todos os métodos vazios por enquanto.
4. Adicione um campo para a classe do adaptador armazenar uma referência ao objeto do serviço. A prática comum é inicializar esse campo via o construtor, mas algumas vezes é mais conveniente passá-lo para o adaptador ao chamar seus métodos.
5. Um por um, implemente todos os métodos da interface cliente na classe adaptadora. O adaptador deve delegar a maioria do trabalho real para o objeto serviço, lidando apenas com a conversão da interface ou formato dos dados.
6. Os Clientes devem usar o adaptador através da interface cliente. Isso irá permitir que você mude ou estenda o adaptador sem afetar o código cliente.

Prós e contras

- *Princípio de responsabilidade única*. Você pode separar a conversão de interface ou de dados da lógica primária do negócio do programa.
- *Princípio aberto/fechado*. Você pode introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente,

desde que eles trabalhem com os adaptadores através da interface cliente.

- A complexidade geral do código aumenta porque você precisa introduzir um conjunto de novas interfaces e classes. Algumas vezes é mais simples mudar a classe serviço para que ela se adeque com o resto do seu código.

Relações com outros padrões

- O Bridge é geralmente definido com antecedência, permitindo que você desenvolva partes de uma aplicação independentemente umas das outras. Por outro lado, o Adapter é comumente usado em aplicações existentes para fazer com que classes incompatíveis trabalhem bem juntas.
- O Adapter fornece uma interface completamente diferente para acessar um objeto existente. Por outro lado, com o padrão Decorator, a interface permanece a mesma ou é estendida. Além disso, o *Decorator* oferece suporte à composição recursiva, o que não é possível quando você usa o *Adapter*.
- Com Adapter, você acessa um objeto existente por meio de uma interface diferente. Com Proxy, a interface permanece a mesma. Com Decorator, você acessa o objeto por meio de uma interface aprimorada.
- O Facade define uma nova interface para objetos existentes, enquanto que o Adapter tenta fazer uma interface existente ser utilizável. O *Adapter* geralmente envolve apenas um objeto, enquanto que o *Facade* trabalha com um inteiro subsistema de objetos.
- O Bridge, State, Strategy (e de certa forma o Adapter) têm

estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.