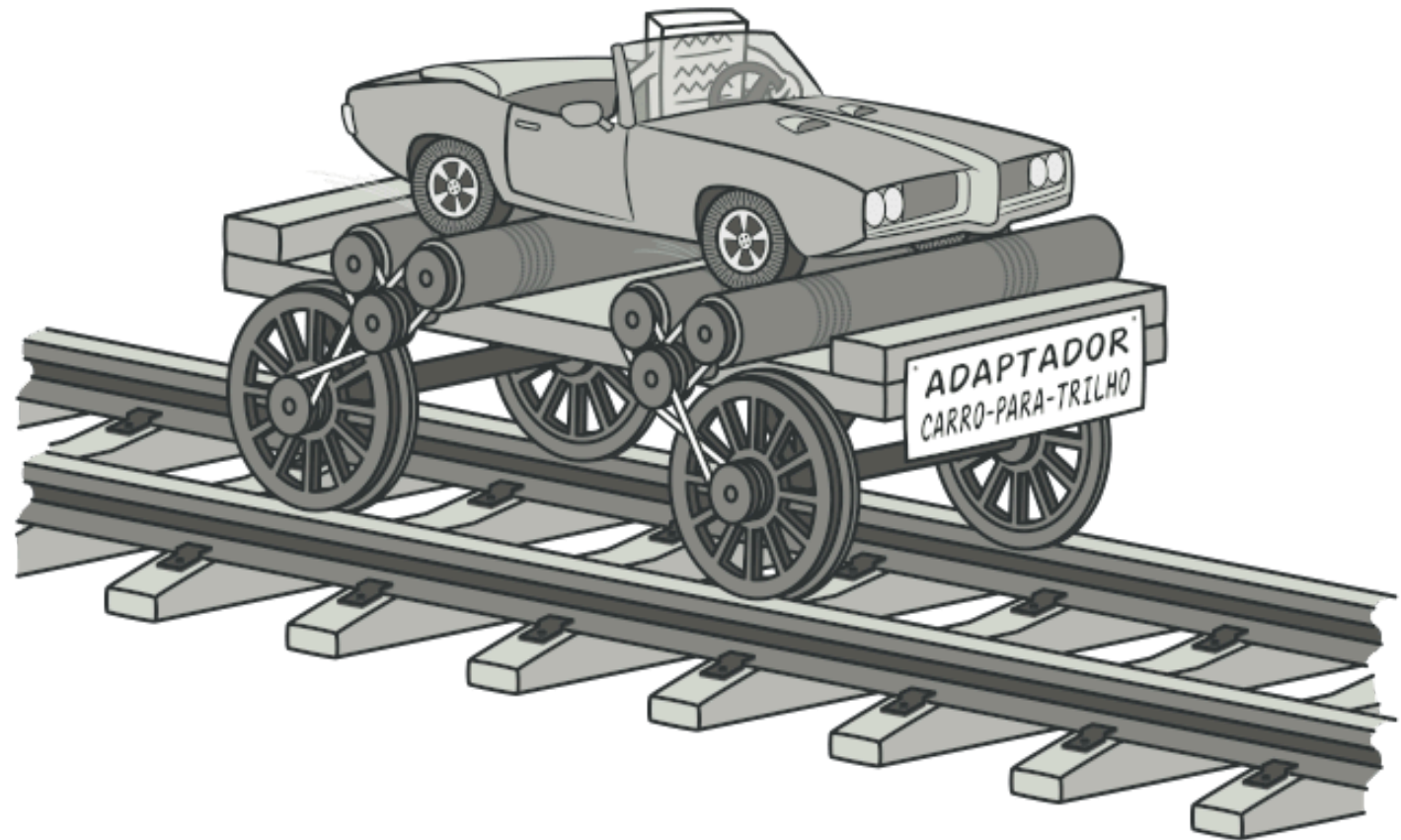


TÓPICO 08: ADAPTER

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

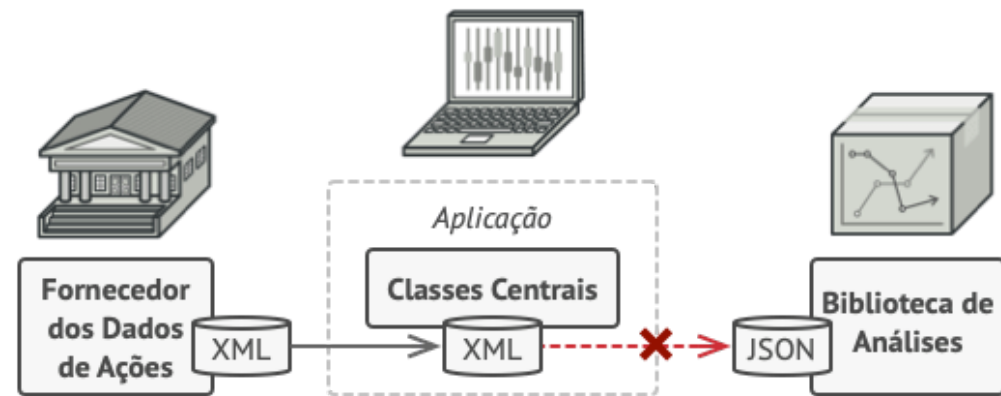
Adapter é um padrão de projeto **estrutural** que permite que objetos com **interfaces incompatíveis** colaborem entre si, funcionando como um **tradutor** entre o código cliente e uma classe existente (de terceiros ou legada).



Problema

Em muitos sistemas precisamos usar **bibliotecas de terceiros** ou código **legado**. Nem sempre a interface deles é compatível com o que o nosso código espera:

- o **cliente** espera uma interface (ex.: temperatura em °C)
- o **serviço externo** fala outra "língua" (ex.: retorna °F, outra assinatura)
- normalmente **não podemos modificar** o serviço (SDK fechado)



Neste exemplo, um painel de clima precisa consultar **dois fornecedores externos**, cada um com a **própria API**:

```
public class PainelClima {  
    public void exhibir(String cidade, String provedor) {  
        if (provedor.equals("accuweather")) {  
            AccuWeatherApi api = new AccuWeatherApi();  
            int f = api.getTemperature(cidade);           // retorna °F  
            System.out.println("Temperatura: " + (f - 32) * 5 / 9 + "°C");  
        } else if (provedor.equals("openweather")) {  
            OpenWeatherApi api = new OpenWeatherApi();  
            double c = api.getTemp(cidade);              // outra assinatura  
            System.out.println("Temperatura: " + c + "°C");  
        }  
    }  
}
```

O problema é que essa **conversão manual** tende a se repetir em **vários pontos** do sistema:

- código **acoplado** à classe concreta de cada fornecedor
- a regra de conversão ($^{\circ}\text{F} \rightarrow ^{\circ}\text{C}$) é **duplicada**
- adicionar um novo provedor exige mexer em **todo lugar** que consulta o clima
- testar cada provedor isoladamente fica difícil

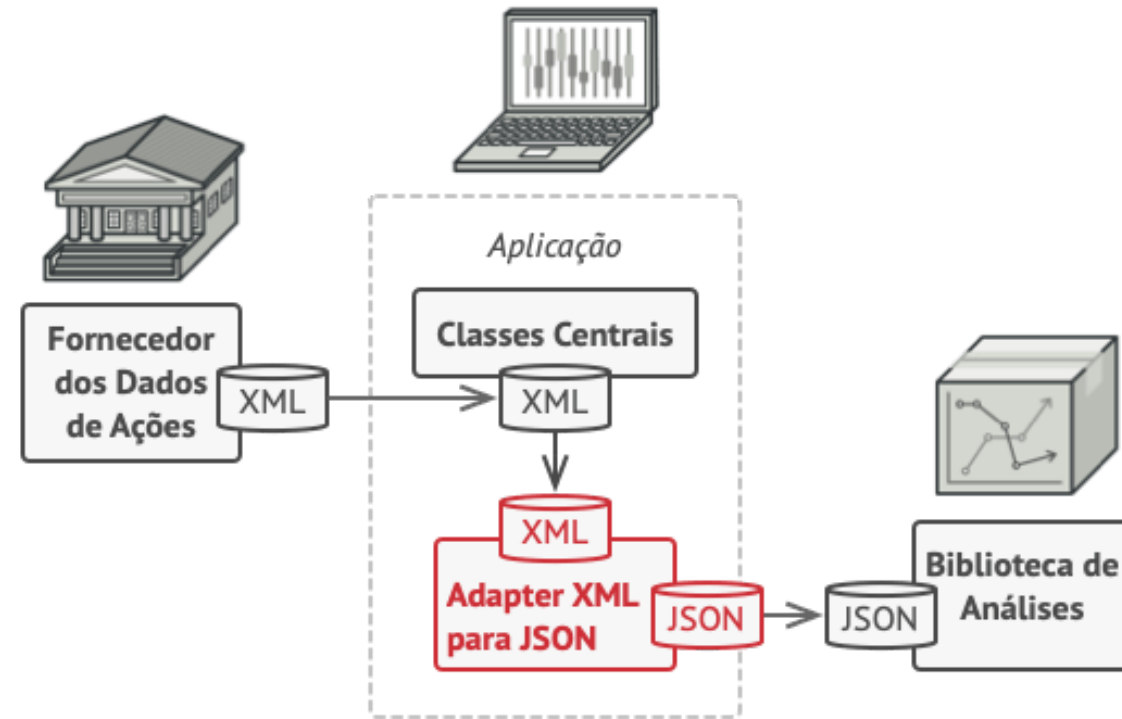
Solução

Vamos criar um **adaptador**: uma classe que **implementa a interface esperada pelo cliente** e **encapsula o serviço incompatível**, traduzindo as chamadas.

- **Interface do cliente**: o contrato que o nosso código usa
- **Serviço**: classe de terceiros/legada, que não pode ser alterada
- **Adaptador**: implementa a interface e **converte** as chamadas para o serviço
- O cliente continua falando **somente com a sua interface**

A adaptação acontece **por fora**, sem tocar no serviço:

- `PainelClima` pede `obterTemperatura(cidade)` em °C
- o adaptador sabe como chamar o fornecedor certo e **converter o resultado**
- cada fornecedor externo ganha o **seu próprio adaptador**
- trocar de fornecedor = **trocar o adaptador**, nunca o cliente



Implementação

1. Primeiro, definimos a **interface do cliente** (contrato que nossa aplicação usa):

```
public interface PrevisaoService {  
    int obterTemperatura(String cidade); // sempre em °C  
}
```

2. O serviço externo (SDK de terceiros) tem uma **interface incompatível** — e não podemos alterá-lo:

```
public class AccuWeatherApi {    // SDK de terceiros
    public int getTemperature(String cidade) {
        // retorna a temperatura em °F
        return 82;
    }
}
```

3. Criamos o **adaptador**: ele implementa `PrevisaoService`, encapsula o SDK e faz a conversão:

```
public class AccuWeatherAdapter implements PrevisaoService {  
  
    private final AccuWeatherApi api;  
  
    public AccuWeatherAdapter(AccuWeatherApi api) {  
        this.api = api;  
    }  
  
    @Override  
    public int obterTemperatura(String cidade) {  
        int fahrenheit = api.getTemperature(cidade); // API incompatível  
        return (fahrenheit - 32) * 5 / 9;           // converte para °C  
    }  
}
```

Utilização

O cliente passa a depender **apenas da interface**, sem conhecer o SDK:

```
public class PainelClima {  
    private final PrevisaoService previsao;  
  
    public PainelClima(PrevisaoService previsao) {  
        this.previsao = previsao;  
    }  
  
    public void exhibir(String cidade) {  
        System.out.println("Temperatura: " +  
            previsao.obterTemperatura(cidade) + "°C");  
    }  
}
```

A escolha do fornecedor fica isolada em **um único ponto**:

```
PrevisaoService previsao = new AccuWeatherAdapter(new AccuWeatherApi());  
new PainelClima(previsao).exibir("Criciúma"); // usa a interface  
  
// novo fornecedor = novo adaptador, o PainelClima não muda  
PrevisaoService open = new OpenWeatherAdapter(new OpenWeatherApi());  
new PainelClima(open).exibir("Criciúma");
```

- conversão **centralizada** no adaptador
- testar é fácil: dá para injetar um *fake* de PrevisaoService

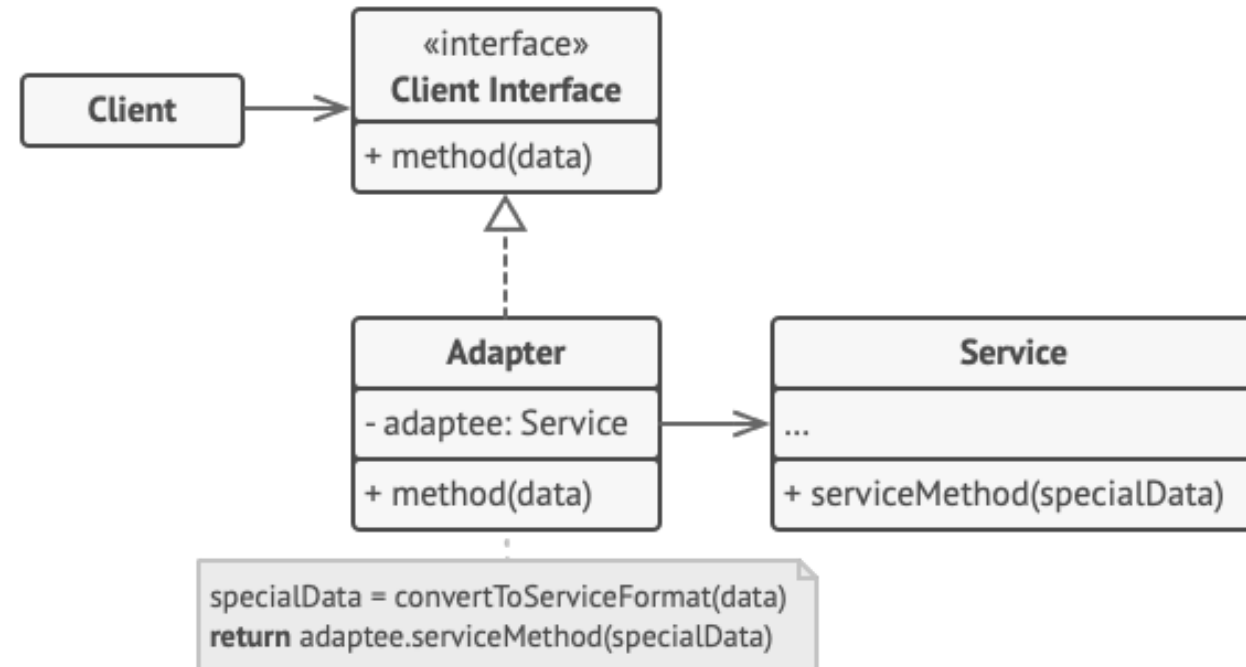
Adaptador de objeto × de classe

Há duas formas de implementar:

Adaptador de objeto (composição):

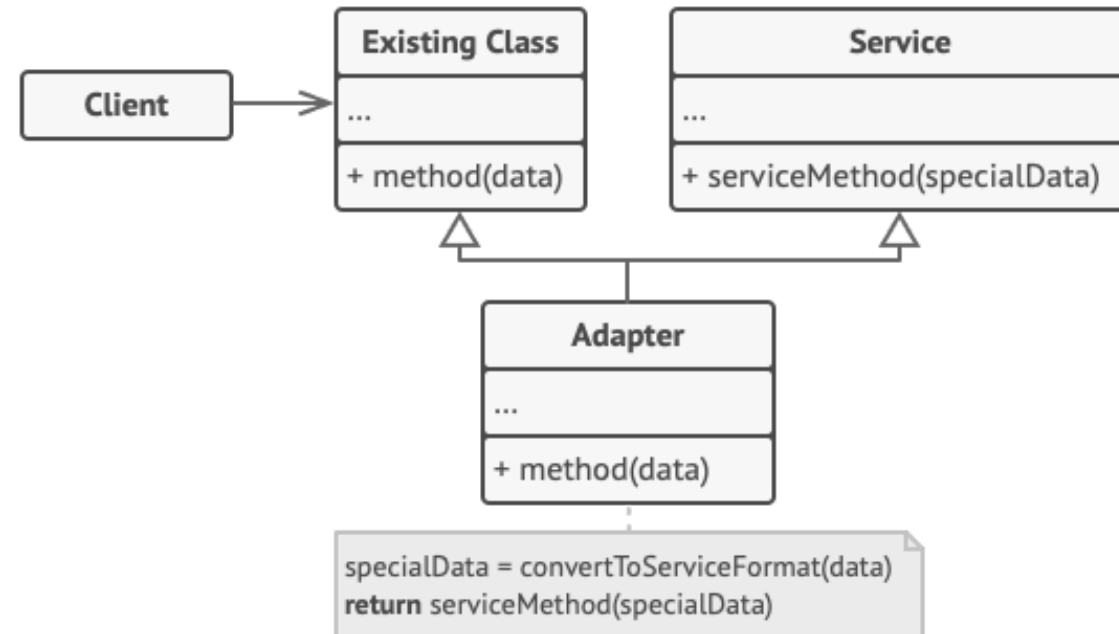
- o adaptador implementa a interface do cliente e **encapsula** o serviço
- funciona em **qualquer linguagem**

usamos essa abordagem



Adaptador de classe (herança):

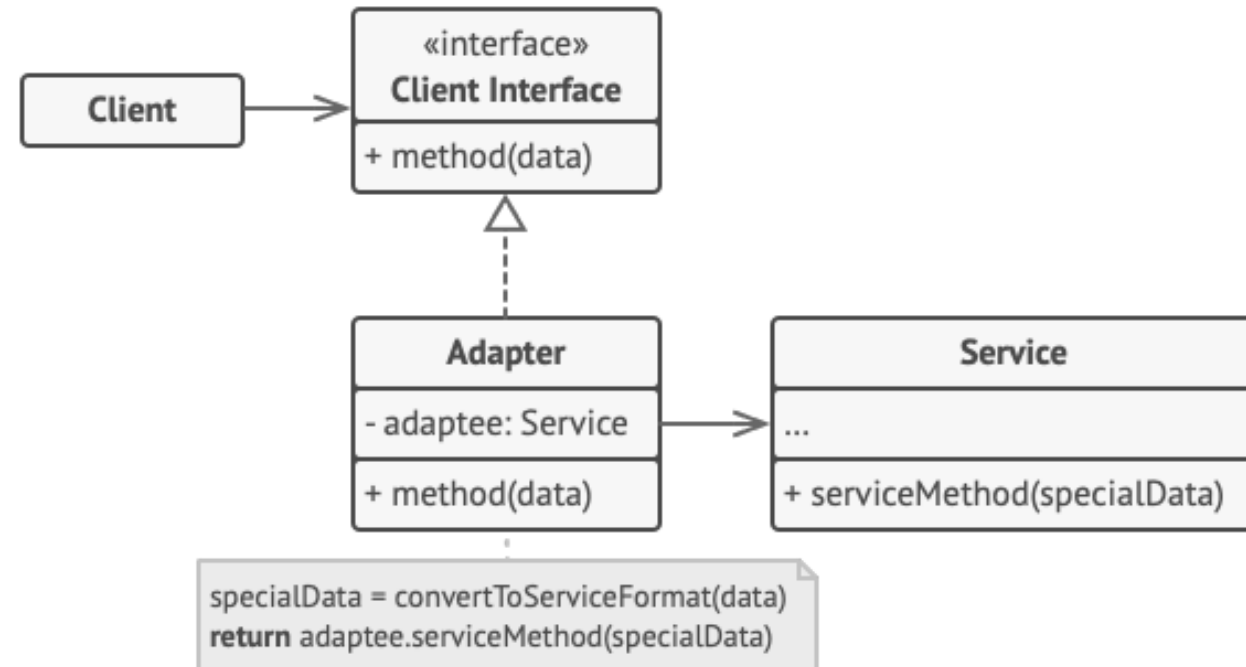
- o adaptador **herda** do serviço e implementa a interface
- exige **herança múltipla** (ex.: C++).
Não funciona em Java e outras linguagens que não permitem herança múltipla



Diagramas

A estrutura envolve:

- **Client**: usa a interface do cliente
- **Target**: interface esperada pelo cliente
- **Adaptee**: serviço de terceiros/legado incompatível
- **Adapter**: implementa o Target e encapsula o Adaptee



Analogias

- **Tomada em outro país:**
 - o laptop usa um plugue brasileiro, mas a tomada é europeia
 - o **adaptador de tomada** encaixa os dois lados
 - ninguém precisa "mudar o laptop" nem "mudar a tomada"
- O padrão faz o mesmo entre **código cliente** e **serviço legado**



Aplicações

- Usar uma **biblioteca de terceiros** ou código **legado** com interface incompatível
- Unificar o acesso a **vários fornecedores/sistemas externos** atrás de uma interface
- Quando **não se pode modificar** o serviço (SDK fechado)
- Reutilizar classes legadas sem reescrever o cliente

Exemplos típicos:

- consultar **APIs de clima** de fornecedores diferentes
- integrar **gateways de pagamento** de terceiros
- wrappers do JDK: `Arrays.asList` (array → `List`), `InputStreamReader` (bytes → caracteres)

Prós

- **SRP**: a conversão de interface/dados fica isolada do cliente
- **OCP**: novos adaptadores entram sem quebrar o código cliente
- **Desacopla** o cliente de bibliotecas de terceiros
- Reaproveita **código legado** sem reescrevê-lo

Contras

- **Mais classes** (interfaces + adaptadores)
- Às vezes é mais simples **adequar o serviço** do que adaptá-lo
- Cuidado para não criar **camadas sem necessidade** (wrapper vazio)
- O adaptador **não adiciona comportamento** — só traduz

Regra: vale a pena quando o serviço é de terceiros ou **não pode ser alterado**.

Resumo

- Adapter: **traduz interfaces incompatíveis** entre cliente e serviço
- O cliente continua usando **a sua interface**
- **Objeto adaptador** (composição) é o padrão em Java
- Ideal para integrar **SDKs de terceiros e código legado**
- Diferença do Bridge: Adapter é usado **em código existente**; Bridge é **projetado antes**