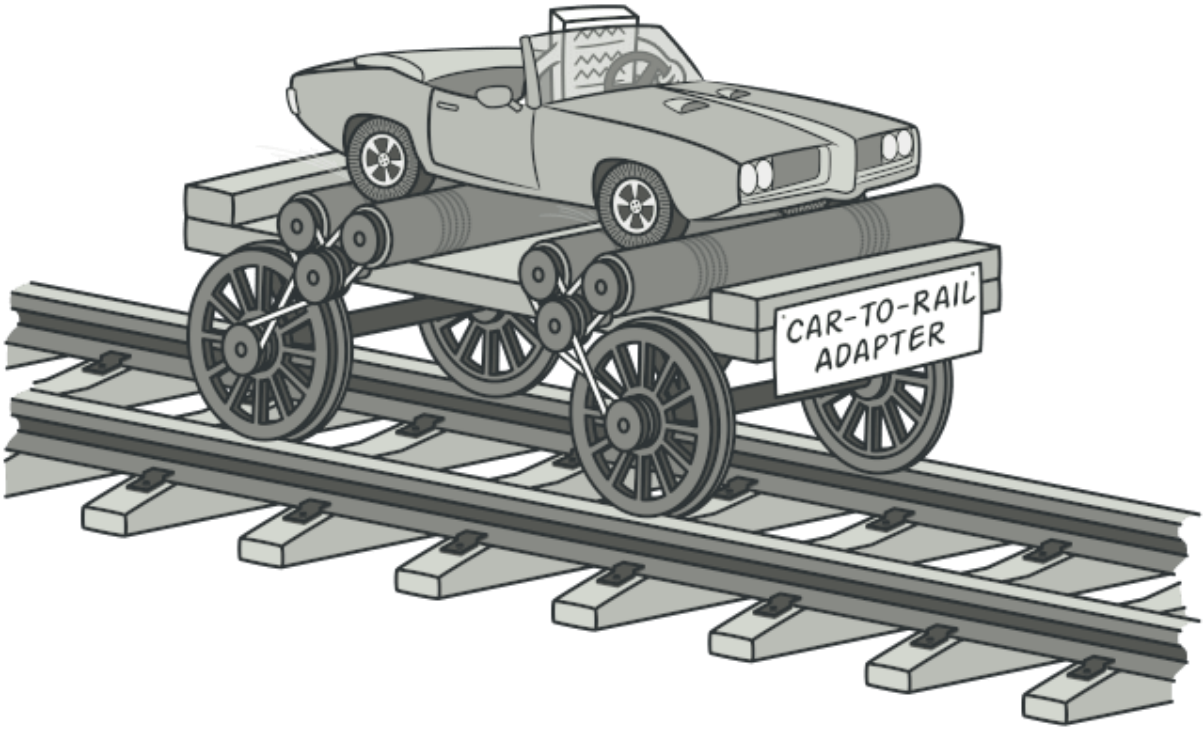
Adaptador

Também conhecido como: Capa

Intenção

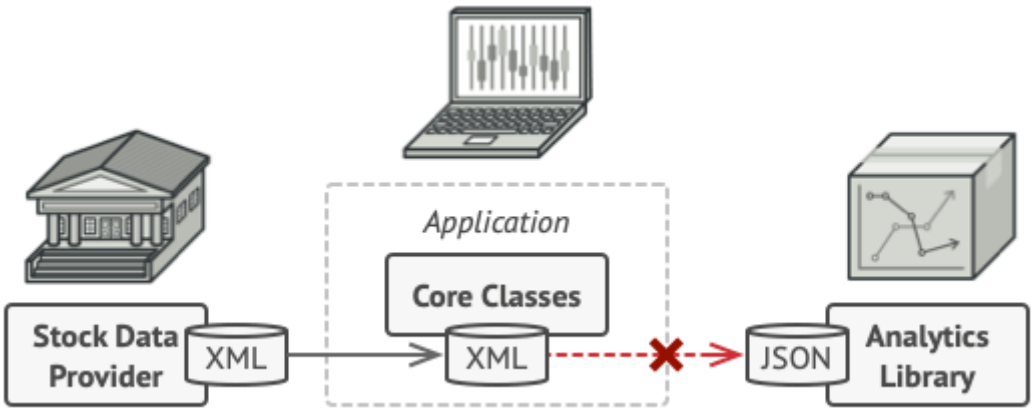
O adaptador é um padrão de design estrutural que permite que objetos com interfaces incompatíveis colaborem.



Problema

Imagine que você está criando um aplicativo de monitoramento do mercado de ações. O aplicativo baixa os dados de ações de várias fontes no formato XML e, em seguida, exibe gráficos e diagramas bonitos para o usuário.

Em algum momento, você decide melhorar o aplicativo integrando uma biblioteca de análise inteligente de terceiros. Mas há um problema: a biblioteca de análise só funciona com dados no formato JSON.



Você não pode usar a biblioteca de análise "no estado em que se encontra" porque ela espera os dados em um formato incompatível com seu aplicativo.

Você pode alterar a biblioteca para trabalhar com XML. No entanto, isso pode interromper algum código existente que depende da biblioteca. E pior, você pode não ter acesso ao código-fonte da biblioteca em primeiro lugar, tornando essa abordagem impossível.

Solução

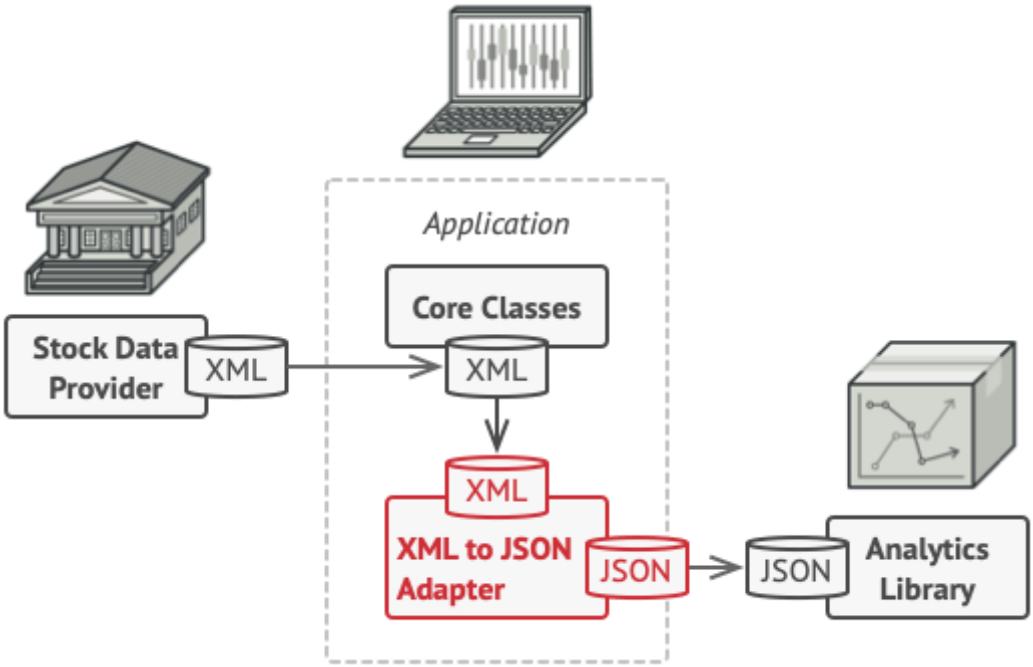
Você pode criar um *adaptador*. Este é um objeto especial que converte a interface de um objeto para que outro objeto possa entendê-lo.

Um adaptador encapsula um dos objetos para ocultar a complexidade da conversão que ocorre nos bastidores. O objeto encapsulado nem mesmo está ciente do adaptador. Por exemplo, você pode encapsular um objeto que opera em metros e quilômetros com um adaptador que converte todos os dados em unidades imperiais, como pés e milhas.

Os adaptadores podem não apenas converter dados em vários formatos, mas também ajudar objetos com diferentes interfaces a colaborar. Veja como funciona:

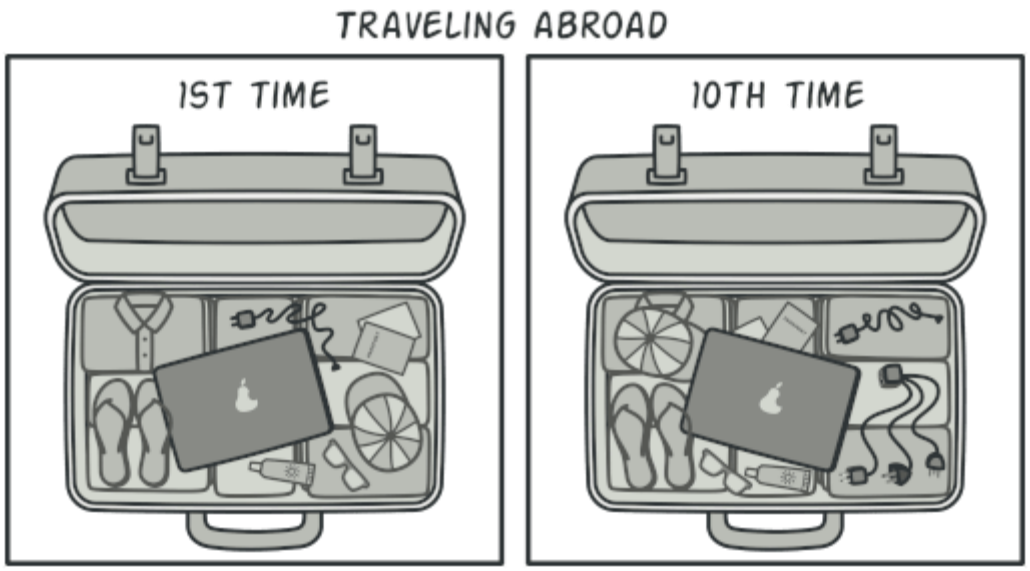
1. O adaptador recebe uma interface, compatível com um dos objetos existentes.
2. Usando essa interface, o objeto existente pode chamar com segurança os métodos do adaptador.
3. Ao receber uma chamada, o adaptador passa a solicitação para o segundo objeto, mas em um formato e ordem que o segundo objeto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

Às vezes, é até possível criar um adaptador bidirecional que possa converter as chamadas em ambas as direções.



Vamos voltar ao nosso aplicativo de mercado de ações. Para resolver o dilema de formatos incompatíveis, você pode criar adaptadores XML para JSON para cada classe da biblioteca de análise com a qual seu código trabalha diretamente. Em seguida, você ajusta seu código para se comunicar com a biblioteca somente por meio desses adaptadores. Quando um adaptador recebe uma chamada, ele converte os dados XML de entrada em uma estrutura JSON e passa a chamada para os métodos apropriados de um objeto de análise encapsulado.

Analogia do mundo real



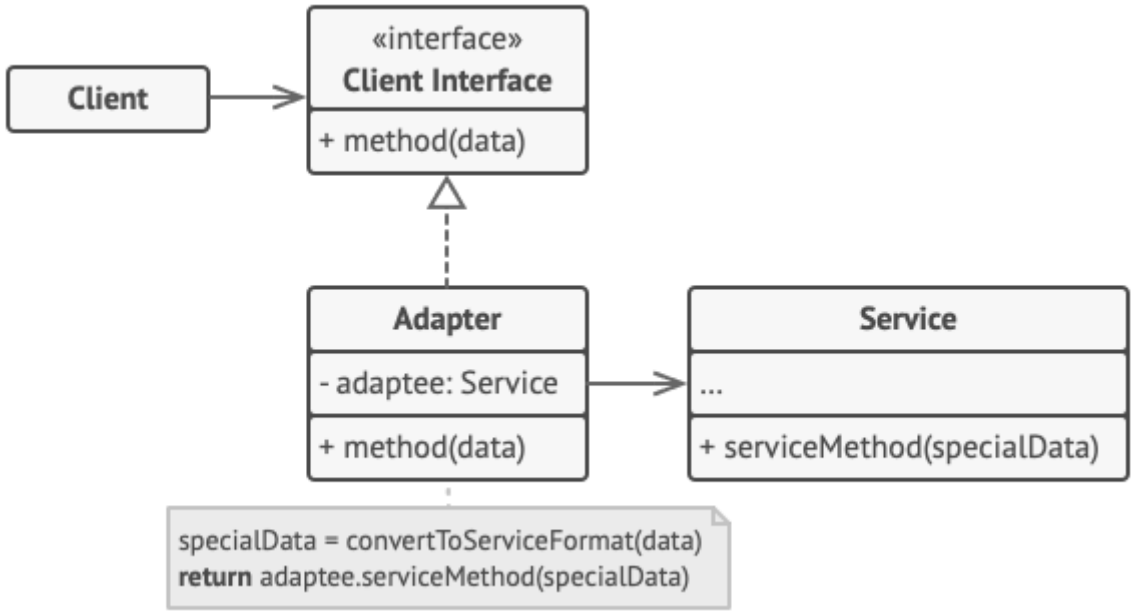
Uma mala antes e depois de uma viagem ao exterior.

Ao viajar dos EUA para a Europa pela primeira vez, você pode ter uma surpresa ao tentar carregar seu laptop. Os padrões de plugue e soquetes de alimentação são diferentes em diferentes países. É por isso que seu plugue americano não cabe em uma tomada alemã. O problema pode ser resolvido usando um adaptador de plugue de alimentação que tenha o soquete de estilo americano e o plugue de estilo europeu.

Estrutura

Adaptador de objeto

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



O **Cliente** é uma classe que contém a lógica de negócios existente do programa.

A **Interface do Cliente** descreve um protocolo que outras classes devem seguir para poder colaborar com o código do cliente.

O **Serviço** é uma classe útil (geralmente 3rd party ou legado). O cliente não pode usar essa classe diretamente porque ela tem uma interface incompatível.

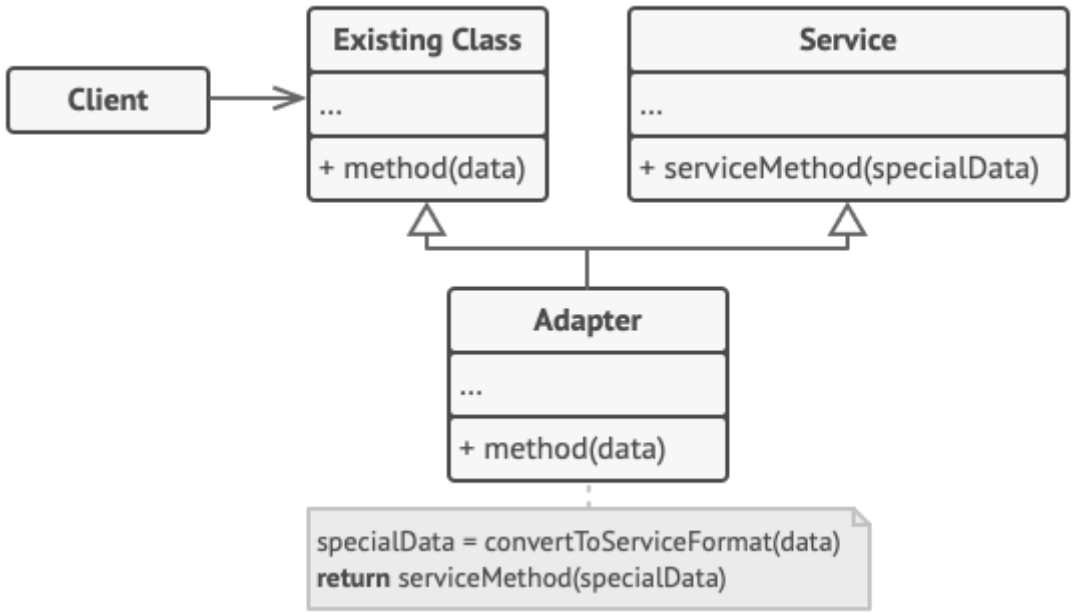
O **Adapter** é uma classe capaz de trabalhar com o cliente e o serviço: ele implementa a interface do cliente, enquanto encapsula o objeto de serviço. O adaptador recebe chamadas do cliente por meio

da interface do cliente e as converte em chamadas para o objeto de serviço encapsulado em um formato que ele possa entender.

O código do cliente não é acoplado à classe do adaptador concreto, desde que funcione com o adaptador por meio da interface do cliente. Graças a isso, você pode introduzir novos tipos de adaptadores no programa sem quebrar o código do cliente existente. Isso pode ser útil quando a interface da classe de serviço é alterada ou substituída: você pode simplesmente criar uma nova classe de adaptador sem alterar o código do cliente.

Adaptador de classe

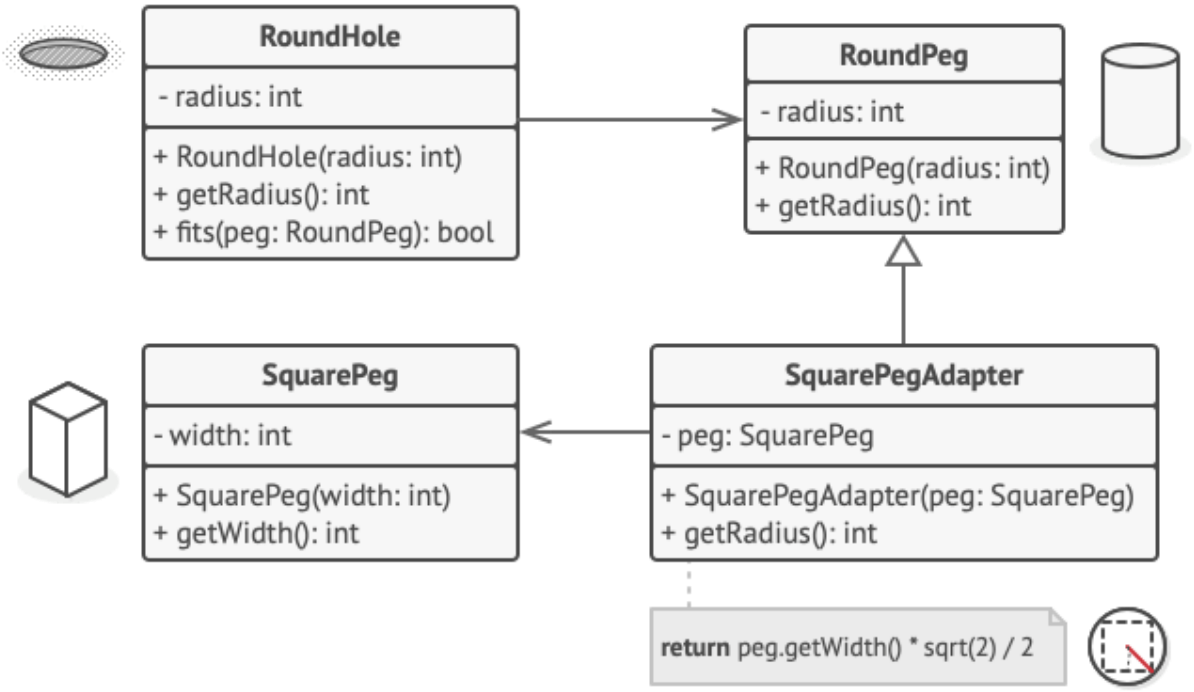
Essa implementação usa herança: o adaptador herda interfaces de ambos os objetos ao mesmo tempo. Observe que essa abordagem só pode ser implementada em linguagens de programação que dão suporte a várias heranças, como C++.



O **Adaptador de Classe** não precisa encapsular nenhum objeto porque herda comportamentos do cliente e do serviço. A adaptação acontece dentro dos métodos substituídos. O adaptador resultante pode ser usado no lugar de uma classe de cliente existente.

Pseudocódigo

Este exemplo do padrão **Adaptador** é baseado no conflito clássico entre pinos quadrados e furos redondos.



Adaptação de pinos quadrados a furos redondos.

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

```
// Say you have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Return the radius of the hole.

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

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

    method getRadius() is
        // Return the radius of the peg.
```

```
// But there's an incompatible class: SquarePeg.
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Return the square peg width.


// An adapter class lets you fit square pegs into round holes.
// It extends the RoundPeg class to let the adapter objects
act
// as round pegs.
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg

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

    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the
adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2


// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible
types)
```

```
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

Use a classe Adapter quando quiser usar alguma classe existente, mas sua interface não for compatível com o restante do código.
O padrão Adapter permite criar uma classe de camada intermediária que serve como um tradutor entre seu código e uma classe herdada, uma classe de terceiros ou qualquer outra classe com uma interface estranha.

Use o padrão quando quiser reutilizar várias subclasses existentes que não possuem alguma funcionalidade comum que não pode ser adicionada à superclasse.
Você pode estender cada subclasse e colocar a funcionalidade ausente em novas classes filhas. No entanto, você precisará duplicar o código em todas essas novas classes, o que **cheira muito mal**.

A solução muito mais elegante seria colocar a funcionalidade ausente em uma classe de adaptador. Em seguida, você encapsularia objetos com recursos ausentes dentro do adaptador, obtendo os recursos necessários dinamicamente. Para que isso funcione, as classes de destino devem ter uma interface comum e o campo do adaptador deve seguir essa interface. Essa abordagem é muito semelhante ao padrão **Decorator**.

Como implementar

1. Certifique-se de ter pelo menos duas classes com interfaces incompatíveis:
    a. Uma classe *de serviço* útil, que você não pode alterar (geralmente de terceiros, legada ou com muitas dependências existentes).
    b. Uma ou várias classes de *cliente* que se beneficiariam do uso da classe de serviço.
2. Declare a interface do cliente e descreva como os clientes se comunicam com o serviço.

3. Crie a classe do adaptador e faça com que ela siga a interface do cliente. Deixe todos os métodos vazios por enquanto.
4. Adicione um campo à classe do adaptador para armazenar uma referência ao objeto de serviço. A prática comum é inicializar esse campo por meio do construtor, mas às vezes é mais conveniente passá-lo para o adaptador ao chamar seus métodos.
5. Um por um, implemente todos os métodos da interface do cliente na classe do adaptador. O adaptador deve delegar a maior parte do trabalho real ao objeto de serviço, manipulando apenas a interface ou a conversão de formato de dados.
6. Os clientes devem usar o adaptador por meio da interface do cliente. Isso permitirá que você altere ou estenda os adaptadores sem afetar o código do cliente.

Prós e contras

Princípio de Responsabilidade Única. Você pode separar a interface ou o código de conversão de dados da lógica de negócios primária do programa.
Princípio Aberto/Fechado. É possível introduzir novos tipos de adaptadores no programa sem quebrar o código do cliente existente, desde que eles funcionem com os adaptadores por meio da interface do cliente.
A complexidade geral do código aumenta porque você precisa introduzir um conjunto de novas interfaces e classes. Às vezes, é mais simples apenas alterar a classe de serviço para que ela corresponda ao restante do código.

Relações com outros padrões

- **O Bridge** geralmente é projetado antecipadamente, permitindo que você desenvolva partes de um aplicativo independentemente umas das outras. Por outro lado, o **Adapter** é comumente usado com um aplicativo existente para fazer com que algumas classes incompatíveis funcionem bem juntas.
- **O adaptador** 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* dá suporte à composição recursiva, o que não é possível quando você usa o *Adaptador*.

- Com o **Adaptador**, você acessa um objeto existente por meio de uma interface diferente. Com o **Proxy**, a interface permanece a mesma. Com o **Decorator**, você acessa o objeto por meio de uma interface aprimorada.
- **O Facade** define uma nova interface para objetos existentes, enquanto o **Adapter** tenta tornar a interface existente utilizável. *O Adapter* geralmente envolve apenas um objeto, enquanto o *Facade* trabalha com um subsistema inteiro de objetos.
- **Ponte**, **Estado**, **Estratégia** (e até certo ponto **Adaptador**) têm estruturas muito semelhantes. De fato, todos esses padrões são baseados na composição, que é delegar trabalho a outros objetos. No entanto, todos eles 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.