**Adapter**

**Definicao**

É um objeto que permite que classes com interfaces diferentes sejam incompatíveis trabalhem em conjunto a partir de um adapter. Logo podemos implementar várias classes sem nos preocupar com a interfaces das mesmas com a existência do adapter.

É um middleware entre aplicações que traduz a comunicação entre elas escondendo a complexidade (grau de transparência é grande garantindo na mesma que o processo como um todo funcione). É possivel ter um two-way adapter que funcione para os dois lados, só funciona em linguagens que herança multipla(C++).

Ex: app que trabalha com Json e outra em XML ou de SQL para PosGreSQL, stock markets com currency diferente, fichas de energia são diferentes nos variados paises (vida real).



Basicamente funciona como um adaptador de corrente eletrica para fazer com que algo funcione num sitio onde nao devia funcionar

Strucutre

Client

The Client is a class that contains the existing business logic of the program

Client interface

The Client Interface describes a protocol that other classes must follow to be able to collaborate with the client code.

Service

The Service is some useful class (usually 3rd-party or legacy). The client can’t use this class directly because it has an incompatible interface.

Adapter

The Adapter is a class that’s able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Por fim

The client code doesn’t get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

## 

## 

## 

## **Pros and Cons**

-Single Responsibility Principle

Podemos separar a interface (ou o conversor da informação) da logica do codigo principal

-Open/Close Principle

Podemos introduzir diferentes adapters sem alterar o codigo que ja existe.

desde que o codigo use as interfaces do cliente

-Complexity increases

* Aumenta pois temos de introduzir mais interfaces e classes, por isso as vezes e mais simples alterar o codigo de forma a que a intruduçao do novo codigo nao afete o antigo

**Relations with Other Patterns**

* [**Bridge**](https://refactoring.guru/design-patterns/bridge) -> desenvolve os componentes de forma a serem independentes uns dos outros
* [**Adapter**](https://refactoring.guru/design-patterns/adapter) -> interface completamente diferente
* **A**[**dapter**](https://refactoring.guru/design-patterns/adapter) acedemos a objetos por uma interface completamente diferente
* [**Adapter**](https://refactoring.guru/design-patterns/adapter) *tenta adaptar uma interface* “wraps just one object”,
* [**Adapter**](https://refactoring.guru/design-patterns/adapter) -> agarra componentes incompatíveis e make them work