What is the Adapter pattern, and what problem does it solve in software design?	Describe the structure of the Adapter pattern. What are the main components involved?	What are the benefits of using the Adapter pattern in software development? Provide some practical examples.
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. It solves problems like: How can a class be reused that does not have an interface that a client requires? How can classes that have incompatible interfaces work together? How can an alternative interface be provided for a class?	The main components of the Adapter pattern are: Target Interface: This is the interface that the client expects to work with. It defines the methods that the client uses to interact with the target. Adaptee Interface: This is the interface of the existing class that needs to be adapted to the target interface. It may have a different method signature or return types compared to the target interface. Adapter: This is a class that implements the target interface and has a reference to an object of the adaptee class. The adapter translates the method calls from the target interface to the adaptee interface. There are two main types of Adapter patterns: Object Adapter Pattern: In this pattern, the adapter contains an instance of the class it wraps. The adapter makes calls to the instance of the wrapped object. This is useful when you need to adapt an object to an interface without changing the object's class. Class Adapter Pattern: This pattern uses multiple polymorphic interfaces, implementing or inheriting both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages that do not support multiple inheritance of classes.	The Adapter pattern offers several benefits, including: Code Reusability: It allows the reuse of existing code by creating an adapter that makes it compatible with the interfaces expected by new code. This is particularly useful when you want to use classes or components that provide valuable functionality but don't conform to the desired interface 3. Integration of Existing Code: The Adapter pattern enables seamless integration of existing components into new systems without modifying their original code. This is crucial when you have existing code or components with interfaces that are incompatible with the interfaces expected by new code or systems 3. Third-Party Library Integration: When incorporating third-party libraries or APIs into a project, and their interfaces do not match the rest of the system, adapters make it possible to use external components by providing a compatible interface for the rest of the application 3. Client-Server Communication: In client-server applications, when the client expects a specific interface while the server provides a different one, adapters help in translating requests and responses between client and server, ensuring smooth communication despite interface differences 3. Examples of the Adapter pattern include: Language Translation: A classic example is the language translation scenario, where an adapter acts as a translator between two friends who speak different languages. This analogy illustrates how the Adapter pattern bridges the gap between incompatible interfaces, enabling communication 3. Client-Server Communication: In software development, when building client-server applications, adapters can translate requests and responses between the client and server, ensuring that despite interface differences, the communication is smooth 3.
The Adapter pattern is a structural design pattern used to allow objects with incompatible interfaces to work together.	Target: is the interface that the client code expects to interact with. It defines the operations that the client can perform. Adaptee: This is the existing interface that needs to be adapted to work with the client code. It's the interface that the Adapter will adapt. Adapter: This is the class that bridges the gap between the Target and the Adaptee. It implements the Target interface and internally uses an instance of the Adaptee to perform the required operations. The Adapter translates the requests from the client code into calls to the Adaptee's interface.	Flexibility and Reusability, Interoperability, Simplified Maintenance, Incremental Development. A logging library with a specific interface, where you want to switch to a different logging library with a different interface. Instead of rewriting the entire application to work with the new library, you can create an adapter that translates calls from the existing interface to the new one.
The Adapter pattern facilitates integration of components with incompatible interfaces, enabling reuse and flexibility. It translates interface calls between classes, allowing them to work together seamlessly, vital for integrating new components into existing systems without altering original code.	The Adapter pattern involves four key components: the Target interface, which the client interacts with; the Client, which expects the Target interface; the Adaptee, whose interface needs adaptation; and the Adapter, which translates between Target and Adaptee. This facilitates compatibility between incompatible interfaces, enabling the client to utilize the Adaptee's functionality seamlessly through the Adapter.	The Adapter pattern offers numerous benefits in software development, including enhanced compatibility, reusability, improved organization, and flexibility. Practical examples include database drivers converting application queries, GUI libraries standardizing event handling, middleware facilitating communication between diverse applications, and wrapping APIs to shield applications from changes. By seamlessly integrating components with incompatible interfaces, the Adapter pattern fosters efficient and adaptable system design.
It's a structural pattern that allows objects with incompatible interfaces to colaborate. The adapter design pattern solves problems like how to reuse a class that does not have an interface that is needed, how alternative interfaces can be provided for a class, the evolution of an interface, etc.	Object Adapter: 1 - Client: class that contains the logic of the program. 2 - Client Interface: describes the protocol that other classes needed to follow. 3 - Service: class that the client can't access directly, usually used by a 3rd party. 4 - Adapter: class that is able to work with the client interface and the service, making a connection between them. Class Adapter doesn't need to wrap any objects because it inherits behaviors from both client and the service.	Single Responsability Principle: separate the interface and the data conversion from the service. Open/Close Principle: introduce new adapters without influencing the existing program. Example: Media player application that supports MP3 files that can also extend and play FLAC files.
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. The main problem the Adapter pattern solves is the need to make two existing interfaces work together, which may have different methods, parameters, or behaviors. Instead of modifying the existing codebase to make the interfaces compatible, the Adapter pattern provides a clean and flexible solution by introducing an intermediary adapter class.	The Adapter pattern allows objects with incompatible interfaces to collaborate. It consists of four main components: - Client: The class that relies on the Target interface. -Target: The interface that the Client expects. - Adaptee: The existing interface that needs to be adapted. - Adapter: The intermediary class that implements the Target interface and wraps the Adaptee, translating requests from the Client to calls to the Adaptee.	Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user. At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format. By creating an adapter, it will convert the interface of one object so that another will understand it. Some of its benefit's are: Integration: It enables the seamless integration of incompatible interfaces, allowing components with different interfaces to work together without modification. Reusability: Adapter classes encapsulate adaptation logic, promoting code reusability across the system by facilitating the use of the same adapter for multiple interfaces. Maintainability: By keeping interface adaptations within adapter classes, the pattern maintains a clean, modular codebase, making it easier to understand, extend, and maintain the system over time.
O padrão Adapter é um padrão de design estrutural que permite que objetos com interfaces incompatíveis colaborem. Ele funciona como uma ponte entre duas interfaces incompatíveis, encapsulando a lógica de transformação de uma interface para outra. Este padrão resolve o problema quando há a necessidade de usar uma classe cuja interface não é compatível com o restante do código da aplicação sem alterar o código existente.	Adapter consiste em três componentes principais: Inclui uma interface "Target", um "Adapter" para fazer a classe existente trabalhar com a interface do target, e um "Adaptee" que é a classe existente com interface incompatível.	O padrão aumenta a compatibilidade entre classes, permite flexibilidade e possibilita a reutilização de código. Por exemplo, é útil ao integrar novas bibliotecas que não correspondem à interface do sistema existente.
Bridge é um padrão de design estrutural que permite dividir uma classe grande ou um conjunto de classes intimamente relacionadas em duas hierarquias separadas – abstração e implementação – que podem ser desenvolvidas independentemente uma da outra. Problemas que este padrao resolve: - integrar componentes com interfaces incompatíveis, promovendo a reutilização de código	O padrão Adapter tem três principais componentes: - Target: É a interface que o código cliente espera utilizar. - Adapter: É a classe que conecta o alvo com o adaptado. Ele implementa a interface do alvo e usa internamente uma instância do adaptado para realizar as operações necessárias. - Adaptee: É a classe existente com uma interface incompatível que precisa ser adaptada para se adequar ao alvo.	Benefícios do uso do padrão Adapter: - Reutilização de código: Permite integrar classes existentes em novos sistemas sem modificar seu código original. - Flexibilidade: Facilita a interoperabilidade entre diferentes partes de um sistema, permitindo que componentes com interfaces incompatíveis trabalhem juntos. - Manutenção simplificada: Isola as mudanças necessárias em uma única classe adaptadora, minimizando o impacto nas demais partes do sistema. Exemplo pratico: - Adaptar uma classe existente para atender aos requisitos de uma nova API ou framework.

O padrão Adaptador permite que interfaces que não são compatíveis possam colaborar entre si, servindo como uma ligação entre duas interfaces que não se harmonizam diretamente. Ele realiza essa tarefa convertendo a interface de uma classe na interface desejada por um cliente.
Um dos principais desafios resolvidos pelo padrão Adaptador é a necessidade de fazer com que classes existentes possam interagir com outras sem que seja necessário alterar seu código-fonte. Esta solução é especialmente valiosa durante a integração de novos componentes ou bibliotecas em um sistema já estabelecido, onde as interfaces dos elementos existentes e os novos podem ser inconciliáveis.
Dessa forma, o padrão Adaptador oferece uma forma de ajustar as interfaces dos novos componentes para que se encaixem perfeitamente no sistema existente.

The Adapter pattern acts as an intermediary between two classes, converting the interface of one class so that it can be used with the other. This enables classes with incompatible interfaces to work together. The Adapter pattern implements an interface known to its clients and provides access to an instance of a class not known to its clients. An adapter object provides the functionality of an interface without having to know the class used to implement that interface.
The main components are the following:
Target: The interface that clients communicate with.
Adaptee: The existing interface that needs adaptation.
Adapter: The bridge between Target and Adaptee. It implements the Target interface and translates its method calls to method calls on the Adaptee.

O padrão Adapter em desenvolvimento de software oferece vários benefícios, tais como a capacidade de integrar interfaces incompatíveis, facilitando a reutilização de código e a interoperabilidade entre diferentes sistemas. Além disso, ele permite a fácil adaptação de classes existentes sem modificar seu código-fonte original. Dois exemplos práticos são: adaptadores de plugues elétricos, que permitem conectar aparelhos com diferentes tipos de tomadas a uma fonte de energia; e adaptadores de mídia, que convertem formatos de áudio ou vídeo para que possam ser reproduzidos em diferentes dispositivos. Esses exemplos ilustram como o padrão Adapter pode simplificar a integração de sistemas heterogêneos e melhorar a interoperabilidade no desenvolvimento de software.

The benefits of using the Adapter pattern in software development include:

Integration of Existing Code: It allows you to integrate existing components seamlessly into new systems without modifying their original code.
Reuse of Existing Functionality: It enables you to reuse existing code by creating an adapter that makes it compatible with the interfaces expected by new code.
Interoperability: The Adapter pattern acts as a bridge, allowing systems with incompatible interfaces to collaborate effectively.
Client-Server Communication: Adapters help in translating requests and responses between client and server, ensuring smooth communication despite interface differences.
Third-Party Library Integration: Adapters make it possible to use external components by providing a compatible interface for the rest of the application.

Practical examples of using the Adapter pattern include integrating third-party libraries or APIs into a project whose interfaces do not match the rest of the system, making different systems or components work together when they have different interfaces, and translating requests and responses between client and server in client-server applications.

The structure of the Adapter pattern involves several key components:

Target Interface: The interface expected by the client.

Adaptee Interface: The interface of the existing class or component that needs to be adapted.

Adapter: A class that implements the target interface and translates calls to the target interface into calls to the adaptee interface.

Client: The code that uses the adapter to interact with the adaptee through the target interface.

The Adapter pattern is a structural design pattern that enables objects with incompatible interfaces to collaborate. It converts the interface of a class into another interface that clients expect, allowing classes to work together that couldn't otherwise because of incompatible interface. This pattern is particularly useful in scenarios where you have existing code or components with interfaces that are incompatible with the interfaces expected by new code or systems and when you want to reuse classes or components that provide valuable functionality but don't conform to the desired interface.

É um padrão de desenho estrutural que permite a interação de objetos com interfaces diferentes.

Um adapter é um objeto especial que converte uma interface de um objeto de modo a que outro objeto a possa entender.
Estrutura: O adapter é uma classe que implementa a interface de um objeto ao qual se pretende converter o formato para uso de por exemplo uma biblioteca externa, recebe chamadas por parte do cliente, através da sua interface e traduz em chamadas para o serviço a ser adaptado.

Benefícios: Podemos separar a interface e o código de conversão de dados da code base principal. Podemos adicionar novos adapters num programa já existente sem modificar diretamente o código principal dos componentes envolvidos.
Exemplo: Pelo mundo, existem diversos tipos de tomadas. Ao viajarmos para um país onde o modelo de tomadas é diferente do nosso, precisamos de um adaptador para que possamos carregar os nossos dispositivos. O adaptador não altera nem a ficha do nosso carregador, nem a tomada do país destino.

O padrão estrutural adapter tem como pricipal objetivo, a interação entre duas interfaces incompativeis, o adapter funciona como uma ponte entre os dois códigos que irá permitir que eles colaborem.

A estrutura é relativamente simples, contém uma interface que o cliente espera usar, contém uma class que tem funcionabilidades desejadas pelo cliente mas contém uma interface incompativelm e depois existe o foco principal que é o adapter que serve como intermediário.

Alguns dos beneficios do Adapter design pattern sáo a integração de código existente, o re-uso de funcionalidades existentes, inter-operabilidade, integração por terceiros, etc. Alguns cenarios incluem por exemplo: quando se pretende reusar classes ou componentes que providenciam funcionalidades valiosas mas não em conformidade com a interface desejada; quando é preciso fazer diferentes sistemas ou componentes trabalharem juntos, em especial quando têm interfaces diferentes; quando existe imcompatibilidade entre código já existente com as interfaces expectáveis pelo novo código ou sistema.

Its a pattern which aims to allow collaboration between two incompatible classes or objects.

The adapter class, does the formatting of the data passed between the two incompatible objects, while implementing the interface of a Client. The Service and the Client communicate through the Adapter.

Target Interface: This is the interface that the client code expects to work with. It defines the operations or methods that the client code will use.

Adaptee: This is the existing class or component that has the interface incompatible with the client code. The Adaptee needs to be integrated with the client code.

Adapter: This is the class that bridges the gap between the Target Interface and the Adaptee. It implements the Target Interface and internally uses an instance of the Adaptee to perform the required operations. The Adapter translates the requests from the client code into a format that the Adaptee can understand and then forwards them to the Adaptee.

Client: This is the code that interacts with the Target Interface. The client code remains unaware of the existence of the Adaptee and interacts only with the Adapter through the Target Interface.

It follows the Single Responsibility principle and the Open/Closed Principle.
A system that uses miles will adapt to a Client that uses kilometers.

Improved reusability, enhanced flexibility, minimized impact on existing code, simplified client code and interoperability.

Practical examples: Database adapters, File format converters, External API Integration, GUI Widgets...

The Adapter pattern is a structural design pattern that allows objects with incompatible interfaces to work together. The main problem that the Adapter pattern solves is the need to integrate or use existing classes or components that have incompatible interfaces with the client code. Instead of modifying the existing classes to match the client's interface, which might not always be feasible or desirable, the Adapter pattern provides a way to adapt the existing classes to work seamlessly with the client code.

O Adapter pattern é um padrao que permite a interação entre objetos com interfaces incompativeis. Resolvendo assim o problema da incompatibilidade entre certos objetos.

O adapter pattern tem um cliente que ccontem a logica existente do programa, uma interface do cliente que descreve o protocolo que as outras classes devem seguir para "comunicarem" com o cliente,um adapter que serve como tradutor entre o cliente e o serviço.

Alguns dos beneficios passam por seguir o "Single Responsibility Principle" onde separamos a interface da logica primaria do programa. E o "Open/Closed Principle." Em que podemos introduzir novos adapter no programa que este funciona sempre, desde que funcionem com a interface do cliente. Temos como exemplo o nosso codigo que usa ficheiros XML mas como certas bibliotes apenas aceitam ficheiros JSON, usamos um Adapter para ambos poderem comunicar.

The adapter pattern is a strutural pattern that solve the problem of incompatibility between objects

1. Client - contains the business logic
2. Client interface - protocol that other classes must follow to interact with the client
3. Service - usefull class that ins incompatible with the client
4. Adapter - class that is able to work with both the Client and the Service

We dont have to change the code of the service(if we even have access to it) or the client to interact with objects that are not compatible with the Client.

Arrrays as list
InputStreamReader

O padrão Adapter é um padrão de design estrutural que permite que objetos com interfaces incompatíveis colaborem. Ele age como uma ponte entre duas interfaces incompatíveis, permitindo que elas trabalhem juntas. O problema que resolve é a integração de objetos com interfaces incompatíveis, permitindo que eles trabalhem juntos sem modificar o código existente.

- Cliente é uma classe que contém a lógica do programa;
- Interface do cliente, descreve um protocolo que outras classes devem seguir para colaborar com o código do cliente.
- Serviço
- Adapter é uma classe que permite trabalhar tanto com o cliente como com o serviço implementando a interface do cliente e funcionando como um wrapper.

Single Responsibility Principle é respeitado com este padrão assim como o Open/Closed Principle.

Permite que objetos com interfaces incompativeis cooperem. O adaptar faz com que a um dos objetos lhe seja escondida a complexidade desta conversão, sendo que este objeto nem seque vai ter noção de que existe um adapter	- Client: Class que tem a logica do programa - Interface: define as operações e funcionalidades que o cliente pode usar - Serviço/Adaptado: interface existente que é incompativel como o cliente - Adaptador: class responsavel por implementar a interface do cliente e que dá wrap na class serviço. É um middleware entre o cliente e o serviço, traduzindo os requests do cliente em calls compativeis com este Não existe coupling entre o cliente e o adaptador já que o adaptador trabalha com a interface do cliente	- Separar a interface da conversão da lógica do programa-> Single Responsability Principle - Dá para introduzir novos adaptadores no programa, desde que este trabalhem com a interface do cliente-> Open/Closed Principle
The adapter pattern is a pattern that allows incompatible interfaces to work together, allowing for the integration of multiple incompatible services into a single coherent application.	For the adapter pattern to be required, there has to be a client that needs to be adapted to a different service. Then there is an adapter class, that as an attribute accepts an object of the client interface and provides several methods that act upon that object.	The adapter pattern allows for incompatible interfaces to collaborate in creating a new service. An example can be the structuring of a SaaS that uses external APIs to provide various data to the end user. That service would have to adapt all the different interfaces of all the different APIs into a single interface to be provided to the end user.
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.	There are 2 types of adapters: Object Adapter - Let's suppose we have a class A that contains the existing business logic of the program with an interface that describes a protocol that other classes must follow to be able to collaborate with the class' code. We have then the class B that is some useful class (usually 3rd-party or legacy). The class A can't use this class directly because it has an incompatible interface.The Adapter is a class that's able to work with both classes: it implements the client interface, while wrapping the service object. The adapter receives calls from the class A via its interface and translates them into calls to the wrapped class B object in a format it can understand. Class Adapter - The Class Adapter doesn't need to wrap any objects because it inherits behaviors from both classes. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing A class.	Use several non-related and incompatible services in order to create an app that makes them compatible, serving as a translator between them.
The Adapter pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, allowing them to collaborate seamlessly. The main problem the Adapter pattern solves is the inability to directly use a class because its interface doesn't match what is required. This could happen when integrating new code with existing code, or when using third-party libraries with interfaces that don't align with your requirements.	The main components of the adapter pattern are the client, the Client interface, the service, the adapter. The client is a class that contains the existing business logic of the program; the client interface describes a protocol that other classes must follow to be able to collaborate with the client code; the service is some useful class (usually 3rd-party or legacy), the client cant't use this class directly because it has an imcompatible interface; 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 fomat it can understand.	Compatibility - Adapters allow incompatible interfaces to work together Reuse - Adapters enable the reuse of existing classes without modifying their source code Flexibility - Adapters provide flexibility by allowing components to work with different interfaces Encapsulation - Adapters encapsulate the details of converting one interface into another, keeping the client code clean and focused on its own concerns
The Adapter pattern is a structural design pattern used in software engineering. It allows objects with incompatible interfaces to work together. It solves the problem of integrating new code with existing code that has a different interface, without modifying the existing code.	Target Interface: Defines the interface expected by the client. It represents the set of operations that the client code can use. Adaptee: The existing class or system with an incompatible interface that needs to be integrated into the new system. Adapter: A class that implements the target interface and internally uses an instance of the adaptee to make it compatible with the target interface. Client: The code that uses the target interface to interact with objects. It remains unaware of the specific implementation details of the adaptee and the adapter.	It provides Interoperability, Reusability, Flexibility, Encapsulation, Testing. Several pratical examples can be: Database adapters GUI frameworks
O padrão Adapter é um padrão que resolve o problema da existencia de duas ou mais interfaces que trabalham com tipos de dados diferentes e, se for preciso transferir dados de uma interface para outra, o programa vai falhar ou não vai dar os resultados corretos.	O principal componente do padrão Adapter é uma nova interface que faz a tradução dos dados da interface A para os dados da interface B e vice-versa.	Alguns dos benificios deste padrão são: segue o princípio Single Responsibility, pois isola a conversão de dados para uma interface nova, sem ter de dar essa responsabilidade extra ao programa principal; segue o princípio Open/Closed, uma vez que é possível acrescentar novos tipos de adaptadores de dados sem comprometer a funcionalidade do programa principal.
Adapter é um padrão estrutural que permite que objetos com interfaces incompatíveis colaborem, atuando como uma ponte entre duas interfaces incompatíveis. Ele é usado quando você deseja usar uma classe existente, mas sua interface não é compatível com o restante do seu código. O problema que o padrão Adapter resolve no design de software é a necessidade de integrar componentes existentes de maneira integrada em novos sistemas sem modificar seu código original.	The Client is a class that contains the existing business logic of the program. The Client Interface describes a protocol that other classes must follow to be able to collaborate with the client code. 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. 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. 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.	Single Responsibility Principle: You can separate the interface or data conversion code from the primary business logic of the program. Open/Closed Principle: You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
The adapter pattern is a pattern that allows two incompatible interfaces to be used in conjunction.	Adapter - transforms the interface to make it suit what the client needs Adaptee - the original interface Target - the transformed interface Client	.
Este padrão estrutural permite que objetos com interfaces incompatíveis colaborem, através de um objeto especial que converte a interface de um objeto para que um outro objeto a consiga usar. A classe Adapter irá esconder a complexidade da conversão que irá acontecer e os objetos adaptados não saberão da existência desta classe. Esta classe poderá fazer conversão de unidades de medida, por exemplo, ou converter formatos e ordem de um primeiro objeto de forma a que a chamada de um segundo objeto seja efetuada da forma esperada.	O padrão Adapter tem a seguinte estrutura: Cliente-> O cliente tem a lógica atual do programa Interface Cliente-> Descreve um protocolo que outras classes têm de seguir para que possam interagir com o cliente Serviço-> O serviço é uma classe, geralmente de terceiros ou legacy, que o cliente não pode usar porque esta tem uma interface incompatível Adapter-> Esta classe trabalha tanto com o cliente como com o serviço, implementará a interface do cliente enquanto adapta o objeto serviço. Receberá chamadas do cliente através da interface e traduzi-la-ás em chamadas que o serviço consiga entender.	Compatibilidade: Permite que classes com interfaces incompatíveis trabalhem juntas. Isto é particularmente útil ao integrar novo código com código pré-existente ou ao usar bibliotecas de terceiros. Reutilização de Código: Adaptadores podem ser reutilizados para adaptar diferentes classes com interfaces semelhantes. Isso promove a reutilização de código e reduz a duplicação. Flexibilidade: Adaptadores fornecem uma maneira flexível de adaptar interfaces sem modificar o código existente. Isso facilita a manutenção e a extensão do sistema. Encapsulamento: Adaptadores encapsulam os detalhes de como a adaptação é realizada, isolando o cliente das complexidades do processo de adaptação. Interoperabilidade: Facilita a interoperabilidade entre diferentes sistemas ou componentes, fornecendo uma interface comum. APIs de Terceiros: Ao integrar APIs de terceiros numa aplicação, podem ocorrer situações em que a interface da API não corresponda à interface esperada pela aplicação. Adaptadores podem ser usados para adaptar a interface da API de terceiros para se adequar à interface esperada pela aplicação.

É um padrão de projeto estrutural que permite que interfaces incompatíveis trabalhem juntas. O seu objetivo é converter a interface de uma classe existente noutra interface esperada pelos clientes.

Problema é desenvolvimento de software, é ao deparamos com situações onde precisamos fazer com que classes que possuem interfaces distintas trabalhem juntas.O padrão premite resolver esse problema sem a necessidade de modificar o código existente, fazendo com que a colaboração entre objetos com interfaces diferentes seja possível.

O padrão Adapter permite a interoperabilidade entre interfaces incompatíveis, facilitando a integração e a reutilização de classes existentes, sem a necessidade de alterar seu código. Isso contribui para a flexibilidade e a manutenibilidade do software.

A estrutura do padrão Adapter esta dividida em principalmente três componentes:

Target: Define a interface específica que o cliente usa. É a interface que o adaptador vai expor para que possa ser usada pelo cliente como se fosse a interface esperada.

Adapter: Implementa a interface alvo e mantém uma referência ao objeto que precisa ser adaptado. O adaptador faz a tradução entre a interface alvo e a interface existente do objeto a ser adaptado. Quando recebe uma chamada de método da interface alvo, o adaptador converte essa chamada em uma ou mais chamadas para a interface do objeto adaptado, em um formato que ele possa entender.

Adaptee: É a classe existente que precisa ser adaptada. Possui uma interface que precisa ser convertida em outra interface através do adaptador para que possa ser utilizada pelo cliente.

O padrão Adapter oferece várias vantagens as quais:

Compatibilidade entre classes: Permite que classes com interfaces incompatíveis trabalhem juntas. Isso é especialmente útil quando se integra novas bibliotecas.

Reutilização de código: Facilita a reutilização de classes que de outra forma não seriam utilizáveis devido à incompatibilidade de interfaces. Isso permite poupar tempo e recursos.

Princípio de responsabilidade única: O adaptador mantém o princípio de responsabilidade única ao separar a lógica de conversão de interfaces do restante da lógica de negócio das classes envolvidas.

Flexibilidade: Ao desacoplar a interface de uma classe de seu uso, ganhamos flexibilidade para introduzir novos tipos de adaptadores sem alterar o código ou das classes existentes.

É um padrão de design que permite que objetos com interfaces incompatíveis colaborem. Resolve o problema de incompatibilidade entre tipos de dados, criando um objeto especial que converte a interface de um objeto para que um outro objeto possa o entender.

Cliente: Classe que contém a logica atual do programa.
Interface cliente: Descreve o protocolo que as outras classes devem seguir para colaborar com o código do cliente.
Serviço: Uma classe útil que o cliente não pode usar diretamente por incompatibilidade de interface.
Adaptador: Uma classe que é capaz de trabalhar com o cliente e o serviço. Ele implmenta a interface do cliente enquanto envolve o objeto "Serviço".

Expandir um aplicativo usando uma livraria externa em que necessitaria mudanças no código original, por conta de incompatibilidade. Essas mudançcas poderiam quebrar o código. Para contornar esse problema cria-se um adapter, fazendo com que o aplicativo funcione com o serviço.

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

It makes it possible for two incompatible objects two collaborate between each other.

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

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

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.

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.

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.

Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program.
Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

O Adapter pattern é um padrao de design estrutural que permite converter a interface de objeto, de modo que, um outro objeto o consiga perceber. Assim, objetos com interfaces incompatíveis conseguem trabalhar juntos.

Target: a interface que o cliente vai usar
Client: a entidade que vai interagir com objetos através da interface targeted
Adaptee: a classe que precisa de adapçao antes que o cliente a possa usar
Adapter (wrapper): permite que a o adaptee seja compatível com o target

- Aumento da compatibilidade (por exemplo: integração de bibliotecas ou APIs de terceiros)
- Reutilização
- Desacoplamento do Cliente e do Sistema

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate

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.
The Client is a class that contains the existing business logic of the program.
The Client Interface describes a protocol that other classes must follow to be able to collaborate with the client code.
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.

Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program.
Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

Structural design pattern that allows objects with incompatible interfaces to collaborate

Client class: contains the existing business logic of the program
Client interface: Describes protocol that other classes must follow to be able to collaborate with client code.
Service class: Useful third party class that the client cant use directly because it has an incompatible interface.
Adapter class: Class that is able to work with both the client and the service. Implements the client interface while wrapping the service object.

One of the advantages is not having to refactor and change a third party library, which would cost time and money, instead the Adapter class is modified in order to be compatible with the library.
Some other benefits may include: Single Responsability Principle and Open/Closed Principle.
An example would be an application that monitors stock market and the stock data provider provides XML format files so, the core classes of the application are built in order to exclusively work with XML format and an Adapter Class is created to provide compatibility between the main app and a third party analytics app that only receives files in JSON format.

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.
Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user. At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.
You could change the library, but this might break some code that relies on the library
solution:
We can create a special object that converts the interface of one object so that another object can understand it.

The adapter structure is as following:
1 - The client: A class that contains the existing logic of the program
2 - The Client Interface: Describes a protocol that other classes must follow to be able to collaborate with the client code
3 - The Service: Some 3rd part class that the client can't interact with because of an incompatible interface
4 - The Adapter: A class that implements de Client interface, while wrapping the service object. It translates calls from the client into calls for the wrapped service object in a format it can understand

We can separate the interface or data conversion code from the primary business logic of the program and we can introduce new types of adapters into the program withour breaking the existing code, as long as they work with the adapters through the client interface, it also promotes code reasability allowing existing classes to be reused in new systems without modifiying their original code. With adapters we can also achieve loose coupling between components. Finnaly the adapters facilitate unit testing by allowing you to mock or substitute dependencies with adapters.

The adapter can be used in database adapters, GUI Widgets Adapters, Legacy System Integration and in Third-Party API Integration.

Quando surge a necessidade de integrar um componente externo ao nosso
ecossistema e este já existe, podemos reutilizá-lo, mas algumas vezes temos problemas de compatibilidade.
Nem sempre podemos alterar o código da classe que queremos reutilizar, por isso criamos um Adapter que é responsável por:
- Fornecer uma interface compativel com um dos objetos
- Este objeto acede aos métodos da interface
- A cada chamada, o adapter passa o pedido ao segundo objeto, mas com a
formatação adequada.

Client: contém a lógica de negócios existente do programa.
Client Interface/Existing Class: descreve um protocolo que outras classes devem seguir para poder colaborar com o código do cliente.
Service: é alguma classe útil (geralmente de terceiros ou herdada). O cliente não pode usar esta classe diretamente porque ela possui uma interface incompatível.
Adapter: é uma classe capaz de trabalhar tanto com o cliente quanto com o serviço: ela implementa a interface do cliente, enquanto envolve o objeto de serviço. O Adapter recebe chamadas do cliente por meio da interface do cliente e converte-as em chamadas para o objeto de serviço empacotado em um formato que ele possa compreender.

Podemos introduzir novos tipos de adapters no programa sem o código do cliente existente parar de funcionar, desde que funcionem com os adaptadores por meio da interface do cliente.

Exemplo:
Permitir que sistemas novos utilizem bibliotecas ou sistemas legacy sem modificar o código original, como conectar uma nova aplicação de gestão de clientes a um sistema de base de dados antigo.

Um Adaptador serve para que objetos com interfaces incompatíveis consigam trabalhar juntos. O problema do design de software que ele resolve é adaptação de código sem a necessidade de modificar interfases incompatíveis manualmente.	Tem 4 componentes principais envolvidos. O cliente 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 a classe útil que o cliente quer usar mas não pode, porque esta classe possui uma interface incompatível com a do cliente. O Adapter é uma classe que é capaz de implementar a interface do cliente, enquanto envolve o objeto de serviço.	Os benefícios ser extensível pois pode-se adicionar mais novos tipos de adaptadores no programa sem quebrar o código e pode-se separar a interface ou o código de conversão de dados da lógica de negócios principal do programa.
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate, it solves the problem of integrating new components or systems into existing codebases without modifying the existing code. It's useful when leading with legacy code.	Client: The client is the component that needs to interact with a certain interface. Target interface: This is the interface that the client expects. Adaptee: Is the existing class or interface that needs to be adapted to work with the client. Adapter: Is the intermediary class that bridges the gap between the client and the adaptee. It implements the Target interface and wraps the adaptee, allowing the client to communicate with the adaptee through the adapter. The client doesn't know the existence of the adaptee. Adapter translates requests from the client into calls to the adaptee.	Benefits: It is coherent with the single responsibility principle, because you can separate the interface or data conversion code from the primary business logic of the program. It also follows the open/closed principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.
O padrão de software Adapter permite a colaboração entre objetos com interfaces incompatíveis, desta forma, resolve o problema de quando queremos expandir o nosso código libraries e/ou classes com formatos incompatíveis consigam funcionar juntas sem estragar a sua eficiência.	Os principais componentes do padrão Adapter são a classe client, que contém a lógica do programa, esta está ligada a uma interface Client que descreve o protocolo que todas as classes que queiram colaborar com Client têm de seguir. Ao termos uma classe service, que o client não consegue usar diretamente por terem interfaces incompatíves. Desta forma é implementado um Adapter que trabalha com ambos os serviços, implementando a interface client enquanto contém o objeto service, desta forma, traduz as chamadas realizadas pelo client de modo a que o programa externo as reconheça.	As vantagens principais são o facto de que segue o Single Responsibility Principle, visto que separamos a interface ou a conversão de dados da lógica principal do programa enquanto também respeita o Open/Closed Principle, pois somos capazes de introduzir novos tipos de adapters num programa sem quebrar o código existente anterior (código Client).
The adapter patter is a Structural Design Pattern that is used to make it so different and imcompatible interfaces can work together, this is granted through the use of the a single class know as "adapter" that joins the functionalities of these interfaces	2 interfaces, the target, that is the primary interface being used, and the adaptee, that is the incompatible interface, 1 class known as adapter, that acts as a bridge for the interfaces, and the client that uses these interfaces while it continues unaware of the specifics of the adaptee and adapter, through the adapter.	With these pattern we can create compatibility between multiple functions and integrate these existing functions into our new code, besides it also helps translate request and responses in the client-server relationship. A pratical example would be having a legacy program which we want to adapt into a new version which expects a new interface, while still using functionalities of the legacy version.
O padrão Adapter, é um padrão estrutural, que pretende converter a interface de uma classe, de encontro aos requisitos do cliente. Permite assim a colaboração de objetos com interfaces incompatíveis, fornecendo uma nova interface a uma classe já existente. Este pretende resolver o problema de quando surge a necessidade de integrar um componente externo ao nosso ecossistema e este já existe, mas por vezes surgem problemas de compatibilidade, surgindo problemas na sua reutilização. A solução é a criação de um adapter, que é responsável por: • Fornecer uma interface, compatível com um dos objetos; • Este objeto acede aos métodos desta interface; • A cada chamada, o adapter passa o pedido ao segundo objeto, mas com a formatação adequada (processo de adaptação). O objeto "adaptado" não se apercebe da existência do adapter.	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 de terceiros ou herdada). O cliente não pode usar esta classe diretamente porque ela possui uma interface incompatível. O Adapter é uma classe capaz de trabalhar tanto com o cliente quanto com o serviço: ele implementa a interface do cliente, enquanto envolve o objeto de serviço. O adapter recebe chamadas do cliente por meio da interface do cliente e as converte em chamadas para o objeto de serviço empacotado em um formato que ele possa compreender. O código do cliente não é acoplado à classe concreta do adapter, desde que funcione com o adapter por meio da interface do cliente. Graças a isso, você pode introduzir novos tipos de adapters no programa sem quebrar o código do cliente existente. Isso pode ser útil quando a interface da classe de serviço é alterada ou substituída: pode-se simplesmente criar uma nova classe de adaptador sem alterar o código do cliente.	Um benefício é o uso do Princípio de Responsabilidade Única, ao separar a interface ou o código de conversão de dados da lógica de negócios primária do programa. Outra é a utilização do Princípio Aberto/Fechado (Open/Closed), onde se pode introduzir novos tipos de adaptadores no programa sem quebrar o código do cliente existente, desde que funcionem com os adapters através da interface do cliente. Um exemplo do uso deste padrão é ao pensar numa aplicação de monitorização das ações do mercado, que recorre a uma fonte de dados que os fornece em XML. Em determina altura surge a necessidade de integrar uma biblioteca de análise de dados, mas a que está disponível apenas faz o tratamento de dados JSON. Para a conseguirmos integrar sem fazer modificações neste componente, de forma simples e sem o modificar, podemos construir um adapter, acedido pelo componente, que consuta os dados na fonte em XML e os converte para JSON, passando depois esta informação ao novo componente (o cliente).
O Adapter é um structural pattern que tem como objetivo fornecer uma interface de uma classe desenhada para que seja utilizada por uma classe cliente com características específicas. É particularmente útil quando temos duas classes que necessitam de comunicar, no entanto, utilizam formas diferentes para representar informação. Neste caso, o adapter funciona como um "tradutor" para que esta comunicação aconteça da forma mais fluida possível.	O adapter pattern tem um cliente, com a lógica do programa, este tem uma interface com o protocolo que o resto das classes que querem colaborar com o cliente devem seguir. Existe também um serviço, com o qual o cliente não consegue interagir, devido a este ter uma interface incompatível. Para resolver esta incompatibilidade existe o Adapter, uma classe que interage com o cliente e o servidor, que implementa a classe do cliente e envolve o objeto do serviço.	O conjunto de benefícios de utilizar o Adapter Pattern são os seguintes: Princípio da Responsabilidade Única: Pode separar a interface ou o código de conversão de dados da lógica de negócio principal do programa. Princípio Aberto/Fechado: Pode introduzir novos tipos de adaptadores no programa sem quebrar o código de cliente existente, desde que trabalhem com os adaptadores através da interface do cliente. Promove a Reutilização de Código: O padrão Adapter permite que reutilizar várias subclasses existentes que podem estar a faltar alguma funcionalidade comum que não pode ser adicionada à superclasse. Em vez de estender cada subclasse e adicionar a funcionalidade ausente, o que resultaria em duplicação de código, podemos usar um adaptador para adicionar a funcionalidade necessária de forma dinâmica, promovendo a reutilização de código. Flexibilidade na Adição de Novas Funcionalidades: Adicionar novas funcionalidades ao sistema torna-se mais flexível com o uso do Adapter. Podemos facilmente criar novos adaptadores para incorporar novos recursos ou comportamentos, mantendo a estrutura existente intacta e minimizando o impacto nas partes existentes do sistema.
O padrão adapter é um padrão estrutural que permite que objetos com diferentes interfaces colaborem. Resolve o problema que acontece quando se pretende usar objetos com interfaces diferentes (como no exemplo XML e JSON), criando um "adapter" que converte a interface de um dos objetos, numa interface que o outro pode perceber.	O Adapter implementa a interface de um objeto e engloba o outro. A classe Cliente é a que contém a lógica básica do programa. A sua interface descreve o protocol que estipula as regras de utilização da classe Cliente pelas outras classes. A classe Serviço é a classe que contém algum método útil a ser utilizado pela Cliente, mas que não consegue devido à incompatibilidade da sua interface. A classe Adapter é uma classe que consegue trabalhar com as duas anteriores: implementa a interface Cliente, englobando o objeto Service. Esta recebe chamadas do Cliente através da interface Cliente e faz a tradução destas para o objeto englobado Service num formato que este compreende.	Quanto ao exemplo prático, esta é uma possibilidade: temos duas classes compatíveis RoundHole e RoundPeg, que representam um buraco redondo e um pino redondo, respetivamente. Estas classes têm interfaces compatíveis, o que significa que um pino redondo pode se encaixar num buraco redondo se o raio do pino for menor ou igual ao Segue o princípio de responsabilidade única, na medida em que separa a interface que faz as traduções da classe principal (Cliente); Segue também o princípio open/closed, pois é possível introduzir novos tipos de adapters no programa sem quebrar o Cliente existente. Um exemplo prático é a existência de um Cliente que processa dados recebidos por um banco no formato XML, mas quer usar uma libraria de análise dos dados que recebe informação em JSON. Para isto ser possível, é necessária a existência de uma classe Adapter que traduz o formato XML para JSON, ou seja, implementa a interface Cliente (com o formato XML), englobando o objeto JSON, para que seja possível a respetiva tradução.

O padrão adapter é um padrão de design estrutural que permite a colaboração entre objetos incompatíveis. O principal problema que este padrão resolve é quando se pretende criar uma nova classe que queremos que interaja com classes previamente criadas, mas sem ter de alterar o código existente.

Na estrutura do adapter pattern, habitualmente, as componentes são as seguintes:
Cliente, que contem a lógica do programa existente;
A interface do cliente, que descreve a lógica que as classes devem ter para colaborarem com o código existente;
O serviço, a classe que queremos implementar mas não o podemos fazer diretamente por causa da incompatibilidade com a interface do cliente;
O adaptador, que implementa a interface do cliente e ao mesmo tempo traduz os métodos do cliente para que o serviço possa colaborar com ele.

O principal benefício de usar o adapter pattern é a facilidade da reutilização de código existente. Muitas vezes, é nos fornecido código que, não estando necessariamente incorreto, não nos permite implementar certas classes por estas serem incompatíveis com as interfaces do programa. No entanto, utilizando o adapter, podemos fazer com que as classes que queremos implementar possam colaborar com interfaces que normalmente seriam incompatíveis. Para além disto, oferece mais flexibilidade ao código, desacoplando a interface da implementação das classes. Por exemplo, se estivermos a desenvolver uma aplicação em que vamos usar uma biblioteca externa para implementar determinadas funcionalidades mas as interfaces da biblioteca não sao compatíveis com o resto do programa. Usando um adapter, traduzimos as chamadas da aplicação à interface da biblioteca, permitindo-nos fazer isto sem alterar o código que já tinhamos escrito.

The Adapter pattern is the structural design that allows objects with conflicting characteristics and interfaces to still work together. When implementing libraries or other code into an app, there's the risk of joining code that doesn't accept the other, which is what the Adapter pattern solves.

There are three main components:
- The Client interface, defines a protocol that the classes must follow to be able to work with the Client class.
- The Service class, is the class responsible for receiving the code from the Adapter and letting it know how the code must be translated.
- The Adapter, works with previous two and implements the interface and wraps the Service class, translating calls from the client and sending them to the Service in a code that the class can understand without conflicting.

The Adapter class is great when you want to implement or use an external class but it has conflicting code. For example, if there's a library that uses XML but the app is written in JSON there needs to be an adaptation.

O padrão Adapter é um padrão estrutural que resolve o problema de permitir que objetos com interfaces incompatíveis trabalhem juntos. Em design de software, muitas vezes nos deparamos com situações em que uma classe existente não possui a interface necessária para ser utilizada por outra classe ou componente.

O Adapter resolve esse problema agindo como um intermediário entre as duas interfaces incompatíveis. Ele "adapta" a interface de uma classe para que seja compatível com a interface esperada pelo cliente. Dessa forma, o cliente pode interagir com o Adapter sem precisar conhecer os detalhes da implementação da classe adaptada.

Cliente Interface: Este é a interface que é desejada pelo cliente. É a interface que o cliente utiliza para interagir com o sistema.

Adapter: O Adapter é a classe que adapta a interface para a interface do cliente. Implementa a interface para o cliente e mantém uma referência a um objeto do tipo interface.

interface: Este é o objeto existente com uma interface incompatível.

Beneficios:
Princípio de responsabilidade única.
Princípio aberto/fechado

Quando é necessário usar uma classe existente, mas sua interface não for compatível com o resto do código.
Quando for necessario reutilizar diversas subclasses existentes que não possuam alguma funcionalidade comum que não pode ser adicionada a superclasse.

O padrão Adapter é um padrão de design estrutural que permite que objetos com interfaces incompatíveis trabalhem juntos.

Cliente interface, declara o protocolo a ser seguido pelo cliente;
Cliente, implementa a interface "Cliente interface" e quer usar o serviço;
Service é o componente que o cliente deseja usar, porem não diretamente porque as interfaces são incompatíveis;
Adapter responsável por traduzir as chamadas da interface alvo para chamadas que o serviço pode entender.;

Podemos permitir que o cliente use serviços, mesmo que o serviço tenha mudado, sem precisar de alterar o seu código.

Exemplo: aplicativo de software que precisa usar uma biblioteca de terceiros para processamento de dados.

The adapter pattern solves the issue of having two incompatible interfaces in our code. The adapter makes them compatible.

This pattern needs an "Adapter" object that will implement an interface and change it slightly (adapt it) to fit the clients needs.

The adapter pattern allows us to easily reuse interfaces even if they don't exactly fit our neeeds.

Ao manter sistemas complexos com dependências, pode ocorrer uma mudança da licença de uma dessas dependências que faça com que o seu uso seja proibido. No entanto, o sistema está feito de forma a utilizar uma classe daquela dependência em específico.

Existem, no entanto, outras dependências com classes idênticas mas não compatíveis que podem solucionar o problema. Como o sistema é complexo, de forma a não alterar o seu código, pode-se utilizar o padrão de desenvolvimento Adapter, que atua como uma camada de compatibilidade entre o sistema e esta nova dependência.

Este padrão permite a interação entre classes incompatíveis. Os principais componentes envolvidos são: a interface específica que o sistema usa, a classe Adapter que adapta a interface da classe incompatível para a interface que o sistema espera, e a classe a ser adptada.

Assim, o adapter age como uma camada de compatibilidade entre o sistema e a classe a adaptar.

Ao utilizar este padrão consegue-se com que as alterações apenas estejam concentradas na classe Adapter sem modificar o sistema e sem modificar a classe incompatível.

Um exemplo disto seria uma aplicação que faz uso de dados em XML, porém com a troca para outra biblioteca a classe devolve os dados no formato JSON. De forma a não alterar o código fonte do sistema já existente, pode-se criar um adapter que faça a conversão entre estes dois formatos.

O padrão Adapter oferece uma série de benefícios significativos em desenvolvimento de software:

Reutilização de código existente: Permite integrar sistemas legados ou bibliotecas externas com interfaces incompatíveis ao sistema atual, sem necessidade de modificar o código existente. Isso evita reescrever funcionalidades já implementadas, promovendo a reutilização de código.

O padrão Adapter é um padrão de design estrutural que permite que interfaces incompatíveis trabalhem juntas. Ele converte a interface de uma classe em outra interface que um cliente espera encontrar. O Adapter permite que objetos com interfaces incompatíveis cooperem entre si.

O principal problema que o padrão Adapter resolve em software design é quando você precisa integrar um componente existente em um sistema que espera uma interface diferente daquela fornecida pelo componente. Em vez de modificar o componente existente para se adequar à interface do sistema, o padrão Adapter permite criar um adaptador que faz a interface do componente existente ser compatível com a interface do sistema, sem modificar o código do componente ou do sistema. Isso promove a reutilização de código, modularidade e flexibilidade do sistema.

Os principais componentes envolvidos no padrão Adapter são:

Target: Esta é a interface que o código cliente espera interagir. Ela define as operações ou métodos que o cliente usa para se comunicar com o adaptador.

Adapter: O adaptador é uma classe que implementa a interface alvo e envolve uma instância do Adaptee. Ele traduz as chamadas feitas pelo cliente usando a interface alvo em chamadas para a interface do Adaptee.

Adaptee: Este é a classe ou interface existente que precisa ser integrada ao código cliente. Ele tem uma interface incompatível que não pode ser usada diretamente pelo cliente.

Client: O cliente é a classe ou componente que interage com a interface alvo. Ele não tem conhecimento da presença do adaptador e se comunica apenas através da interface alvo.

Desacoplamento: Ajuda a desacoplar o código cliente do código adaptado, permitindo que o cliente não precise conhecer os detalhes de implementação do código adaptado. Isso mantém uma clara separação de responsabilidades entre as diferentes partes do sistema.

Flexibilidade e extensibilidade: Facilita a adição de novas funcionalidades ao sistema sem modificar o código existente. Se uma nova versão de uma biblioteca externa com uma interface incompatível for lançada, um novo adaptador pode ser facilmente criado para integrar essa versão sem afetar o resto do sistema.

Integração de componentes: Facilita a integração de componentes de diferentes fornecedores ou sistemas em um único sistema coeso. Isso é especialmente útil em sistemas distribuídos, onde diferentes componentes podem usar tecnologias diferentes.

Manutenção simplificada: Ajuda a simplificar a manutenção do código, isolando as mudanças nos sistemas adaptados. Se o sistema adaptado for alterado ou substituído, apenas o adaptador correspondente precisa ser atualizado, mantendo o resto do sistema intacto.

Exemplos práticos de uso do padrão Adapter incluem adaptação de interfaces de banco de dados, integração de serviços web e adaptação

O padrão Adapter é um padrão estrutural que permite a colaboração entre diferentes interfaces, que, inicialmente, são incompatíveis. Atua como intermediário entre as duas classes, ao converter a interface de uma classe em outra interface esperada pelo cliente.
O padrão em questão pode ser útil em situações nas quais componentes de um sistema foram desenvolvidos de forma independente, de tal maneira que as classes não estão fortemente acopladas. Também pode ser, útil na reutilização de código.

O padrão Adapter pode ser estruturado em quatro classes: Client, Service e Adapter.
Client: é a classe na qual se interage com o adaptador para utilizar os seus métodos, também pode se utilizar a classe Client e estabelecer uma outra interface a qual interage com as demais classes.
Service: é a interface esperada pelo client e até então incompatível, a qual se deseja adaptar.
Adapter: implementa a interface Service e interage com o Client ao fazer o encapsulamento e abstração necessária para envolver o objeto Service. Recebe chamadas do cliente através da interface do cliente, traduz essas chamadas para um formato compreensível pelo Service e as encaminha para ele

Os benefícios podem ser : reutilização de código, manutenção, integração de sistemas.
Adaptação de diferentes formatos de arquivo para serem processados por um sistema específico.
Adaptar uma API externa para se integrar ao sistema de uma empresa.

The Adapter pattern is essentially a "translator" that converts an object's interface so that an other object can understand it. It functions and a wrapper, wrapping the object to hide complexity when converting the interface. During this process, the object is unaware of what is happening. This solves the problem of incompatibility when creating applications where 1 or more interfaces clash (for example, when 2 interfaces use data in different formats)

As mentioned before, the Adapter functions as a wrapper, implementing an object's interface while wrapping the other object, following the object composition principle.
It's main components are:

- The Target Interface:
  Defines the interface expected by the client. It represents the set of operations that the client code can use. It's the common interface that the client code interacts with.

- The Adaptee - The existing class or system with an incompatible interface that needs to be integrated into the new system. It's the class or system that the client code cannot directly use due to interface mismatches.

- The Adapter - A class that implements the target interface and internally uses an instance of the adaptee to make it compatible with the target interface. It acts as a bridge, adapting the interface of the adaptee to match the target interface.

- The Client - The code that uses the target interface to interact with objects. It remains unaware of the specific implementation details of the adaptee and the adapter. It's the code that benefits from the integration of the adaptee into the system through the adapter.

The Adapter pattern offers several benefits in software development:

Compatibility: The Adapter pattern allows incompatible interfaces to work together. This is particularly useful when integrating existing or third-party code with new systems or when components with different interfaces need to interact.
Reusability: Adapters enable the reuse of existing classes that may not have the interface required by the client. This can help in avoiding the need to modify existing code, thus preserving its integrity.
Flexibility: By introducing adapters, you can introduce new functionality or alter the behavior of existing classes without changing their code. This promotes a more flexible and modular design.
Simplification: Adapters can simplify complex interactions by providing a simplified interface to the client, hiding the complexities of the adapted class.

Pratical Examples:
Unit Conversion: You can create adapters to convert units of measurement, such as from inches to centimeters or from kilometers per hour to miles per hour. This would be useful in applications that need to deal with different unit systems.
Data Mapping: Suppose you have two classes with different data structures, but you want to use them together in your application. You can create an adapter that maps the attributes of one class to the attributes of the other, allowing them to work together without the need for changes to their codes.
Date Formatting: Suppose you have a system that needs to display dates in different formats, such as MM/DD/YYYY or YYYY-MM-DD. You can create adapters to format dates according to different specifications without modifying the core logic of your application.