Adapter Design Pattern

Last Updated : 03 Jan, 2025

One structural design pattern that enables the usage of an existing class's interface as an additional interface is the adapter design pattern. To make two incompatible interfaces function together, it serves as a bridge. This pattern involves a single class, the adapter, responsible for joining functionalities of independent or incompatible interfaces.

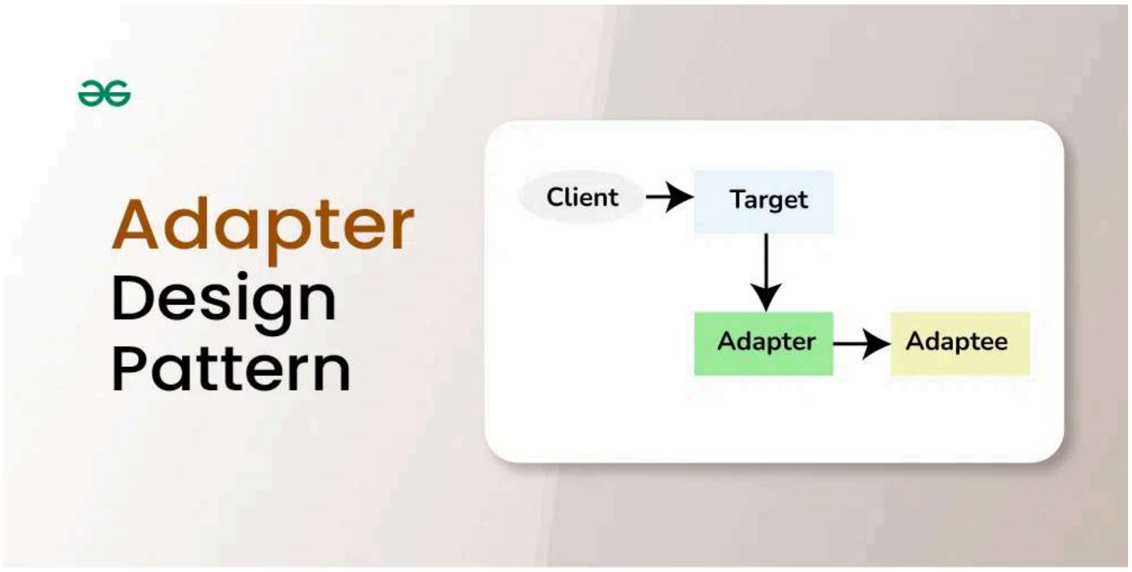


Table of Content

What is Adapter Design Pattern?

Two incompatible interfaces or systems can cooperate by using the adapter design pattern, a structural design pattern. Because of incompatible interfaces, it serves as a bridge between two classes that would not otherwise be able to communicate. The adapter approach is very helpful when attempting to incorporate third-party libraries or legacy code into a new system.

Real-World Example of Adapter Design Pattern

Let's understand this concept using a simple example:

> *Suppose you have two buddies, one of them speaks French exclusively and the other English exclusively. The language barrier prevents them from communicating the way you want them to.*

- You act as an adapter, translating messages between them. Your role allows the English speaker to convey messages to you, and you convert those messages into French for the other person.
- In this way, despite the language difference, your adaptation enables smooth communication between your friends.
- This role you play is similar to the Adapter design pattern, bridging the gap between incompatible interfaces.

Components of Adapter Design Pattern

Below are the components of adapter design pattern:

- **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.
- **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.
- **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.

- **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.

Different implementations of Adapter Design Pattern

The Adapter Design Pattern can be applied in various ways depending on the programming language and the specific context. Here are the primary implementations:

1. Class Adapter (Inheritance-based)

- In this approach, the adapter class inherits from both the target interface (the one the client expects) and the adaptee (the existing class needing adaptation).
- Programming languages that allow multiple inheritance, like C++, are more likely to use this technique.
- However, in languages like Java and C#, which do not support multiple inheritance, this approach is less frequently used.

2. Object Adapter (Composition-based)

- The object adapter employs composition instead of inheritance. In this implementation, the adapter holds an instance of the adaptee and implements the target interface.
- This approach is more flexible as it allows a single adapter to work with multiple adaptees and does not require the complexities of inheritance.
- The object adapter is widely used in languages like Java and C#.

3. Two-way Adapter

- A two-way adapter can function as both a target and an adaptee, depending on which interface is being invoked.
- This type of adapter is particularly useful when two systems need to work together and require mutual adaptation.

4. Interface Adapter (Default Adapter)

- When only a few methods from an interface are necessary, an interface adapter can be employed.
- This is especially useful in cases where the interface contains many methods, and the adapter provides default implementations for those that are not needed.
- This approach is often seen in languages like Java, where abstract classes or default method implementations in interfaces simplify the implementation process.

How Adapter Design Pattern works?

Below is how adapter design pattern works:

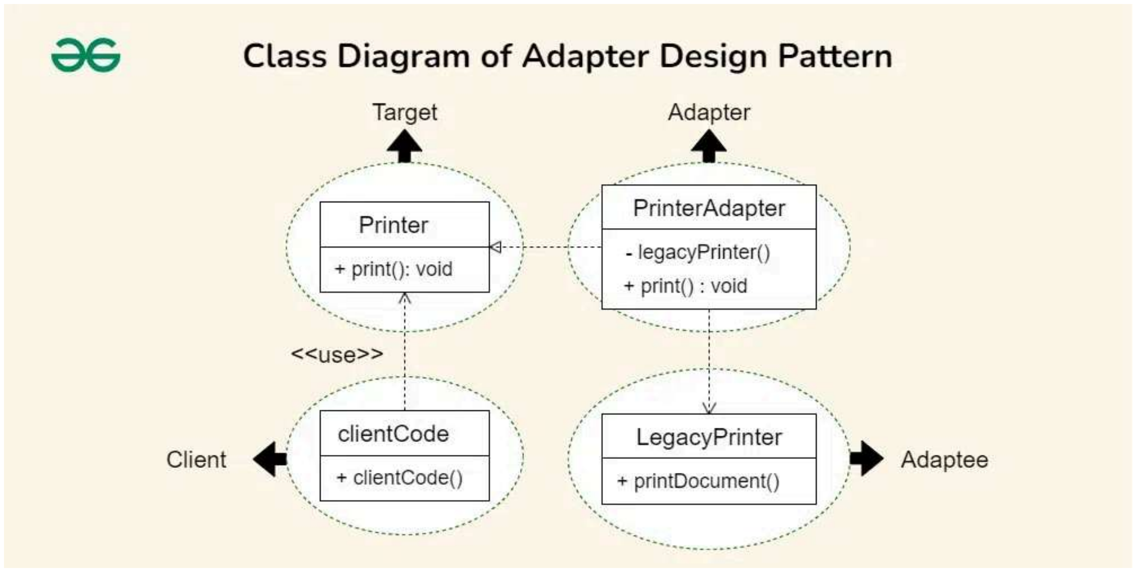
- **Step 1:** The client initiates a request by calling a method on the adapter via the target interface.
- **Step 2:** The adapter maps or transforms the client's request into a format that the adaptee can understand using the adaptee's interface.
- **Step 3:** The adaptee does the actual job based on the translated request from the adapter.
- **Step 4:** The client receives the results of the call, remaining unaware of the adapter's presence or the specific details of the adaptee.

Adapter Design Pattern Example

Let's understand adapter design pattern through an example:

Problem Statement:

> *Let's consider a scenario where we have an existing system that uses a `LegacyPrinter` class with a method named **`printDocument()`** which we want to adapt into a new system that expects a `Printer` interface with a method named **`print()`**. We'll use the **Adapter design pattern** to make these two interfaces **compatible**.*



1. Target Interface (`Printer`)

The interface that the client code expects.

```cpp
// Target Interface

class Printer {
public:
    virtual void print() = 0;
};
```

2. Adaptee (`LegacyPrinter`)

The existing class with an incompatible interface.

```cpp
// Adaptee

class LegacyPrinter {
public:
    void printDocument() {
        std::cout << "Legacy Printer is printing a document." << std::endl;
    }
};
```

3. Adapter (`PrinterAdapter`)

The class that adapts the `LegacyPrinter` to the `Printer` interface.

```cpp
// Adapter

class PrinterAdapter : public Printer {
private:
    LegacyPrinter legacyPrinter;

public:
    void print() override {
        legacyPrinter.printDocument();
    }
};
```

4. Client Code

The code that interacts with the `Printer` interface.

```cpp
// Client Code

void clientCode(Printer& printer) {
    printer.print();
}
```

Complete Code for the above example:

```cpp
// Adapter Design Pattern Example Code

#include <iostream>

// Target Interface
class Printer {
public:
    virtual void print() = 0;
};

// Adaptee
class LegacyPrinter {
public:
    void printDocument() {
        std::cout << "Legacy Printer is printing a document." << std::endl;
    }
};

// Adapter
class PrinterAdapter : public Printer {
private:
    LegacyPrinter legacyPrinter;

public:
    void print() override {
        legacyPrinter.printDocument();
    }
};

// Client Code
void clientCode(Printer& printer) {
    printer.print();
}

int main() {
    // Using the Adapter
    PrinterAdapter adapter;
    clientCode(adapter);

    return 0;
}
```

Output

```
Legacy Printer is printing a document.
```

Pros of Adapter Design Pattern

Below are the pros of Adapter Design Pattern:

- By creating an adapter, you can reuse existing code without needing to modify it. This promotes code reuse and helps maintain a cleaner architecture.
- By separating the issues of interface adaptation, the adapter pattern frees classes to concentrate on their main duties without having to deal with adaptation code that clogs their logic.
- Because you can simply switch out multiple adapters to support different interfaces without altering the underlying system.
- By separating your system from particular implementations, adapters make it simpler to swap out or modify parts without compromising the functionality of other parts.

Cons of Adapter Design Pattern

Below are the cons of Adapter Design Pattern:

- Introducing adapters can add a layer of complexity to your system. Having multiple adapters can make the code harder to navigate and understand.
- The additional layer of indirection may introduce slight performance overhead, especially if the adapter needs to perform complex transformations.
- If not managed properly, the use of adapters can lead to maintenance challenges. Keeping track of multiple adapters for various interfaces can become cumbersome.
- There's a risk of overusing adapters for trivial changes, which can lead to unnecessary complexity. It's critical to assess whether an adapter is actually required in a particular circumstance.
- Only two interfaces can be translated by adapters; if you need to adjust to more than one interface, you might need a lot of different adapters, which could make the design even more difficult.

When to use Adapter Design Pattern?

Use adapter design pattern when:

- We need to connect systems or components that weren't built to work together. The adapter allows these incompatible interfaces to communicate,

making integration smoother.
- Many times, we have existing code or libraries that we want to use, but they don’t match our current system. The adapter helps us incorporate this old code without having to rewrite it.
- As projects grow, new components are frequently added. An adapter allows you to integrate these new pieces without affecting the existing code, keeping the system flexible and adaptable.
- By isolating the changes needed for compatibility in one place, the adapter makes it easier to maintain the code. This reduces the risk of bugs that might arise from changing multiple parts of the system.

When not to use Adapter Design Pattern?

Do not use adapter design pattern when:

- If the system is straightforward and all components are compatible, an adapter may be unnecessary.
- Adapters can introduce a slight overhead, which might be a concern in performance-sensitive environments.
- When there are no issues with interface compatibility, using an adapter can be redundant.
- For projects with a very short lifespan, the overhead of implementing an adapter might not be worth it.

Comment More info Advertise with us

Next Article

Bridge Design Pattern

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you understand the concep...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented and refined over tim...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our...

9 min read

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and represented. Creational...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships betwee...

7 min read

Types of Structural Patterns

Adapter Design Pattern

One structural design pattern that enables the usage of an existing class's interface as an additional interface is the adapter design pattern. To make two incompatible interfaces function together, it serves as a bridge....

8 min read

Bridge Design Pattern

The Bridge design pattern allows you to separate the abstraction from the implementation. It is a structural design pattern. There are 2 parts in Bridge design pattern : AbstractionImplementationThis is a design...

4 min read

Composite Method | Software Design Pattern

Composite Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. The main idea behind the Composite Pattern is to build a tree structure of...

9 min read

Decorator Design Pattern

The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It involves creating...

9 min read

Facade Method Design Pattern

Facade Method Design Pattern is a part of the Gang of Four design patterns and it is categorized under Structural design patterns. Before we go into the details, visualize a structure. The house is the facade, it is...

8 min read

Flyweight Design Pattern

The Flyweight design pattern is a structural pattern that optimizes memory usage by sharing a common state among multiple objects. It aims to reduce the number of objects created and to decrease memory...

10 min read

Proxy Design Pattern

The Proxy Design Pattern a structural design pattern is a way to use a placeholder object to control access to another object. Instead of interacting directly with the main object, the client talks to the proxy, which then...

9 min read

3. Behvioural Design Patterns

Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

3. Types of Behvioural Patterns

Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

Software Design Pattern in Different Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Software Design Pattern Books

Software Design Pattern in Development

Some other Popular Design Patterns

Design Patterns in Different Languages

GeeksforGeeks
Sanchhaya Education Private Limited
Corporate & Communications Address: