



Today

Adapter Pattern Example ...

Builder Pattern Library System

Abstract Factory Pattern Example

Check Python Init Method

Enhanced Abstract Factory Pattern

Movie Ticket Factory Pattern

Design Patterns Refactoring

Using He for Dog

Isnull vs Coalesce SQL



adapter pattern with python code, you can write code from below domain,

Library Management System,
Parking Lot,
Online Shopping System - E-Commerce,
Stack Overflow,
Movie Ticket Booking System,
Design an ATM,
Airline Management System,
Hotel Management System,
Restaurant Management system,
Design Chess,
Design an Online Stock Brokerage System,
Design a Car Rental System,
Design LinkedIn,
Design Cricinfo,
Design Facebook Social Network,
Design payment gateway



Get citation



The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface the client expects.

I'll provide an example of the Adapter Pattern applied to a Library Management System.

Scenario

In a Library Management System, we have an existing system that works with books. However, a new requirement is to integrate an external eBook management system that has a different interface. The Adapter Pattern can help us make these two systems work together.

Code Example

python

Copy code

Existing book system

class Book:

```
def __init__(self, title, author):  
    self.title = title  
    self.author = author
```

```
def get_title(self):  
    return self.title
```

```
def get_author(self):  
    return self.author
```

class Library:

```
def __init__(self):  
    self.books = []
```

```
def add_book(self, book):  
    self.books.append(book)
```

```
def get_books(self):  
    return self.books
```

External eBook system with a different interface

class EBook:

```
def __init__(self, name, writer):  
    self.name = name  
    self.writer = writer
```

```
def get_name(self):  
    return self.name
```

```
def get_writer(self):
```

```

def get_writer(self):
    return self.writer

# Adapter to make EBook compatible with the Library

class EBookAdapter(Book):
    def __init__(self, ebook):
        self.ebook = ebook

    def get_title(self):
        return self.ebook.get_name()

    def get_author(self):
        return self.ebook.get_writer()

# Usage

# Existing library system
library = Library()

# Adding a regular book
book = Book("1984", "George Orwell")
library.add_book(book)

# External eBook system
ebook = EBook("Digital Fortress", "Dan Brown")

# Using adapter to add eBook to the library
ebook_adapter = EBookAdapter(ebook)
library.add_book(ebook_adapter)

# Display books in the library
for book in library.get_books():
    print(f"Title: {book.get_title()}, Author: {book.get_author()}")

```

Explanation

1. **Book:** The existing class representing a physical book with `title` and `author` attributes.
2. **Library:** The existing library system that manages books.
3. **EBook:** The external eBook class with a different interface (`name` and `writer`).
4. **EBookAdapter:** The adapter class that makes an `EBook` compatible with the `Library` system. It inherits from `Book` and overrides its methods to provide the required interface.
5. **Usage:** We create instances of `Book` and `EBook`, then use the `EBookAdapter` to add an `EBook` to the `Library` system.

This way, the `Library` system can work with both `Book` and `EBook` instances seamlessly.



write another example



Sure! Let's use the Adapter Pattern in the context of a **Parking Lot** system. In this example, we have a parking lot that can manage vehicles, but we need to integrate with a third-party system for electric vehicles that uses a different interface.

Code Example

```

python
# Existing vehicle management system

class Vehicle:
    def __init__(self, license_plate):
        self.license_plate = license_plate

    def get_license_plate(self):
        return self.license_plate

class ParkingLot:
    def __init__(self):
        self.vehicles = []

```

Copy code

```

def add_vehicle(self, vehicle):
    self.vehicles.append(vehicle)

def get_vehicles(self):
    return self.vehicles

# External electric vehicle system with a different interface

class ElectricVehicle:
    def __init__(self, id):
        self.id = id

    def get_id(self):
        return self.id

# Adapter to make ElectricVehicle compatible with the ParkingLot

class ElectricVehicleAdapter(Vehicle):
    def __init__(self, electric_vehicle):
        self.electric_vehicle = electric_vehicle

    def get_license_plate(self):
        # Assume electric vehicle ID can be used as a license plate
        return self.electric_vehicle.get_id()

# Usage

# Existing parking lot system
parking_lot = ParkingLot()

# Adding a regular vehicle
vehicle = Vehicle("ABC123")
parking_lot.add_vehicle(vehicle)

# External electric vehicle system
electric_vehicle = ElectricVehicle("EV456")

# Using adapter to add electric vehicle to the parking lot
electric_vehicle_adapter = ElectricVehicleAdapter(electric_vehicle)
parking_lot.add_vehicle(electric_vehicle_adapter)

# Display vehicles in the parking lot
for vehicle in parking_lot.get_vehicles():
    print(f"License Plate: {vehicle.get_license_plate()}")

```

Explanation

1. **Vehicle:** The existing class representing a vehicle with a `license_plate` attribute.
2. **ParkingLot:** The existing system that manages vehicles.
3. **ElectricVehicle:** The external electric vehicle class with a different interface (uses `id`).
4. **ElectricVehicleAdapter:** The adapter class that makes an `ElectricVehicle` compatible with the `ParkingLot` system. It inherits from `Vehicle` and overrides its method to provide the required interface.
5. **Usage:** We create instances of `Vehicle` and `ElectricVehicle`, then use the `ElectricVehicleAdapter` to add an `ElectricVehicle` to the `ParkingLot` system.

With this adapter, the `ParkingLot` can handle both regular and electric vehicles in a uniform manner.

🔊 📄 🔄 🗑️ ⚡ ⌵