



observer 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



Sure! Let's use the Observer pattern to design a simple notification system for a **Library Management System**. The Observer pattern is useful here to notify multiple components of the system (such as patrons or staff) about changes like new book arrivals or overdue books.

## Observer Pattern Overview

1. **Subject:** The class that maintains the state and notifies observers of any changes.
2. **Observer:** The interface or abstract class for objects that need to be notified of changes in the Subject.
3. **ConcreteObserver:** A class that implements the Observer interface and reacts to updates.
4. **ConcreteSubject:** A class that implements the Subject interface and notifies observers of state changes.

## Example Code

Here's a Python implementation for a simple Library Management System using the Observer pattern:



python

Copy code

```
from abc import ABC, abstractmethod

# Observer Interface
class Observer(ABC):
    @abstractmethod
    def update(self, message: str):
        pass

# Concrete Observer
class Patron(Observer):
    def __init__(self, name: str):
        self.name = name

    def update(self, message: str):
        print(f"{self.name} received notification: {message}")

# Subject Interface
class Subject(ABC):
    def __init__(self):
        self._observers = []

    def attach(self, observer: Observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer: Observer):
        if observer in self._observers:
            self._observers.remove(observer)

    def notify(self, message: str):
```

```

        for observer in self._observers:
            observer.update(message)

# Concrete Subject
class Library(Subject):
    def __init__(self):
        super().__init__()

    def add_book(self, book_title: str):
        message = f"New book added: {book_title}"
        print(message)
        self.notify(message)

    def book_overdue(self, book_title: str):
        message = f"Book overdue: {book_title}"
        print(message)
        self.notify(message)

# Example Usage
if __name__ == "__main__":
    # Create library and patrons
    library = Library()
    alice = Patron("Alice")
    bob = Patron("Bob")

    # Attach patrons to the library
    library.attach(alice)
    library.attach(bob)

    # Library actions
    library.add_book("Python Programming 101")
    library.book_overdue("Data Structures and Algorithms")

    # Detach a patron
    library.detach(alice)
    library.add_book("Advanced Python Programming")

```

## Explanation

1. **Observer Interface:** `Observer` defines the `update` method, which is called when the `Subject` state changes.
2. **Concrete Observer:** `Patron` implements the `Observer` interface and responds to notifications.
3. **Subject Interface:** `Subject` manages a list of observers and provides methods to attach, detach, and notify observers.
4. **Concrete Subject:** `Library` extends `Subject` and includes methods to add books and handle overdue notifications, sending updates to its observers.

This design allows the `Library` to notify all attached `Patron` objects about important events, such as new book arrivals or overdue books, keeping them informed and responsive to changes in the system.

< 2/2 >

give another example



Certainly! Let's implement the Observer pattern for a **Movie Ticket Booking System**. In this example, we'll have observers that get notified about different events like ticket availability and booking confirmations.

## Example Code

```

python
Copy code

from abc import ABC, abstractmethod

# Observer Interface
class Observer(ABC):
    @abstractmethod
    def update(self, message: str):
        pass

```

```

# Concrete Observer
class Customer(Observer):
    def __init__(self, name: str):
        self.name = name

    def update(self, message: str):
        print(f"{self.name} received notification: {message}")

# Subject Interface
class Subject(ABC):
    def __init__(self):
        self._observers = []

    def attach(self, observer: Observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer: Observer):
        if observer in self._observers:
            self._observers.remove(observer)

    def notify(self, message: str):
        for observer in self._observers:
            observer.update(message)

# Concrete Subject
class MovieTheater(Subject):
    def __init__(self):
        super().__init__()

    def notify_ticket_availability(self, movie_title: str):
        message = f"Tickets available for movie: {movie_title}"
        print(message)
        self.notify(message)

    def confirm_booking(self, movie_title: str):
        message = f"Booking confirmed for movie: {movie_title}"
        print(message)
        self.notify(message)

# Example Usage
if __name__ == "__main__":
    # Create movie theater and customers
    theater = MovieTheater()
    john = Customer("John")
    mary = Customer("Mary")

    # Attach customers to the movie theater
    theater.attach(john)
    theater.attach(mary)

    # Movie theater actions
    theater.notify_ticket_availability("Inception")
    theater.confirm_booking("Inception")

    # Detach a customer
    theater.detach(mary)
    theater.notify_ticket_availability("The Matrix")

```

## Explanation

1. **Observer Interface:** `Observer` defines the `update` method that observers implement to receive notifications.
2. **Concrete Observer:** `Customer` implements the `Observer` interface and responds to notifications from the `MovieTheater`.
3. **Subject Interface:** `Subject` maintains a list of observers and provides methods to attach, detach, and notify them.
4. **Concrete Subject:** `MovieTheater` extends `Subject` and includes methods to notify about ticket availability and booking confirmations.

In this system, when tickets become available for a movie or a booking is confirmed, the `MovieTheater` class will notify all attached `Customer` objects. This allows each customer to react accordingly, such as checking their booking status or making further arrangements.

write observer design pattern for cloud watch like system who track system usage , if cpu usage is more than 60 % than it notify users, or send msg or email



Let's design an Observer pattern for a **Cloud Watch-like System** that tracks CPU usage and notifies users if the usage exceeds a certain threshold, like 60%. In this example, we'll use a simple notification system that can send messages or emails when the CPU usage is high.

## Example Code

```
python Copy code

from abc import ABC, abstractmethod
import random
import time

# Observer Interface
class Observer(ABC):
    @abstractmethod
    def update(self, message: str):
        pass

# Concrete Observer: EmailNotifier
class EmailNotifier(Observer):
    def __init__(self, email: str):
        self.email = email

    def update(self, message: str):
        print(f"Sending email to {self.email}: {message}")

# Concrete Observer: SMSNotifier
class SMSNotifier(Observer):
    def __init__(self, phone_number: str):
        self.phone_number = phone_number

    def update(self, message: str):
        print(f"Sending SMS to {self.phone_number}: {message}")

# Subject Interface
class Subject(ABC):
    def __init__(self):
        self._observers = []

    def attach(self, observer: Observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer: Observer):
        if observer in self._observers:
            self._observers.remove(observer)

    def notify(self, message: str):
        for observer in self._observers:
            observer.update(message)

# Concrete Subject: CloudWatch
class CloudWatch(Subject):
    def __init__(self):
        super().__init__()

    def check_cpu_usage(self):
        # Simulate random CPU usage
        cpu_usage = random.randint(0, 100)
        print(f"Current CPU usage: {cpu_usage}%")

        if cpu_usage > 60:
            self.notify(f"Alert: CPU usage is high at {cpu_usage}%")

# Example Usage
if __name__ == "__main__":
    # Create CloudWatch instance
    cloud_watch = CloudWatch()
```

```
# Create observers
email_notifier = EmailNotifier("admin@example.com")
sms_notifier = SMSNotifier("+1234567890")

# Attach observers
cloud_watch.attach(email_notifier)
cloud_watch.attach(sms_notifier)

# Simulate periodic CPU checks
for _ in range(5):
    cloud_watch.check_cpu_usage()
    time.sleep(2) # Wait for 2 seconds before the next check
```


## Explanation

1. **Observer Interface:** `Observer` defines the `update` method, which is called by the `Subject` when it needs to notify its observers.
2. **Concrete Observers:**
  - `EmailNotifier` sends email notifications.
  - `SMSNotifier` sends SMS notifications.
3. **Subject Interface:** `Subject` maintains a list of observers and provides methods to attach, detach, and notify them.
4. **Concrete Subject:** `CloudWatch` extends `Subject` and includes a method `check_cpu_usage` to simulate CPU usage monitoring. If the CPU usage exceeds 60%, it sends notifications to all attached observers.

In this system, the `CloudWatch` instance periodically checks the CPU usage. If the usage exceeds the threshold, it notifies all attached observers, which can be email or SMS notifiers. This design allows the system to be easily extended to support other types of notifications or thresholds.

🔊 📄 🔄 🗑️ ⚡

Enhanced Assistant features

 **Upgrade plan**  
Get GPT-4, DALL-E, and more

 Message ChatGPT



ChatGPT can make mistakes. Check important info.

?