Final Laboratorio IV

Sede: Concepción del Uruguay.

Carrera: Tecnicatura Universitaria en Programación.

Cátedra: Laboratorio de Computación IV.

Alumno: Remmer Facundo Tomás.

Legajo: 14137331.

Fecha de entrega: 12/09/2023.

Índice

Tabla de contenido

Introducción

En este informe se define lo que son los patrones de diseño, se detalla su clasificación y su relación con la POO.
Además, se detallan 3 patrones en particular, Singleton, Adpter y Observer para dar resolución a la actividad final de catedra de Laboratorio de Computación IV de la TUP.
Por último, también pedido por la actividad, se brindan y explican 4 ejemplos implementados en Python utilizando los patrones antes mencionados.

Definición de Patrones de Diseño

Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en el código.
Estos patrones no son códigos listos para usar (funciones) o bibliotecas, sino más bien son pautas generales que describen cómo abordar problemas recurrentes de manera efectiva y eficiente. Están diseñados para mejorar la calidad del software, facilitar su mantenimiento, promover la reutilización del código y aumentar la comprensión del diseño.

Diferencia entre Patrón de diseño y Algoritmos

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Pero en realidad un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, mientras que un patrón es una descripción de más alto nivel de una solución, es por esto que el código del mismo patrón aplicado a dos programas distintos puede ser diferente.

Clasificación

Los patrones más básicos y de más bajo nivel suelen llamarse idioms. Normalmente se aplican a un único lenguaje de programación.

Los patrones más universales y de más alto nivel son los patrones de arquitectura. Los desarrolladores pueden implementar estos patrones prácticamente en cualquier lenguaje. Al contrario que los otros patrones más básicos, pueden utilizarse para diseñar la arquitectura de una aplicación completa.

Además, todos los patrones pueden clasificarse por su propósito. Estos son los tres grupos más conocidos:

- Los patrones creacionales: proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
  En otras palabras, definen la mejor manera en que un objeto es instanciado.
  Ejemplos: Factory Method, Abstract Factory, Builder, Singleton.

- Los patrones estructurales: explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura, esto con el fin de ayudarnos a resolver tareas más complejas.
  Ejemplos: Adapter, Bridge, Decorator, Facade, Composite.

- Los patrones de comportamiento: se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.
  Ejemplos: Observer, Strategy, Template Method, Chain of responsiblitity, Iterator, Command, State.

Relación entre un Patrón de Diseño y la POO

Los patrones de diseño y la programación orientada a objetos (POO) están estrechamente relacionados y a menudo se utilizan juntos para diseñar sistemas de software más eficientes, mantenibles y escalables.

La relación se debe principalmente a los siguientes conceptos:

- Abstracción y encapsulación: la POO se basa en la idea de abstracción y encapsulación, donde los objetos representan entidades del mundo real y encapsulan sus datos y comportamientos. Los patrones de diseño utilizan esta abstracción y encapsulación para crear objetos que resuelvan problemas específicos de manera eficiente.
- Herencia y polimorfismo: son conceptos fundamentales de la POO que permiten la reutilización de código y la creación de jerarquías de clases. Algunos patrones de diseño, como el patrón de diseño Factory Method y el patrón de diseño Strategy, hacen un uso intensivo de la herencia y el polimorfismo para crear objetos de manera flexible y extensible.
- Separación de preocupaciones: muchos patrones de diseño promueven la separación de preocupaciones al dividir el sistema en componentes y objetos que se ocupan de tareas específicas. Esto mejora la modularidad y facilita el mantenimiento.
  Ejemplos: el patrón de diseño MVC (Model-View-Controller) y el patrón de diseño Observer.
- Composición sobre herencia: uno de los principios importantes en la POO es "preferir la composición sobre la herencia". Algunos patrones de diseño, como el patrón de diseño Composite aplican este principio para construir objetos complejos combinando objetos más simples en lugar de heredar de ellos.
- Flexibilidad y adaptabilidad: los patrones de diseño, como el patrón de diseño Adapter y el patrón de diseño Strategy, permiten que los sistemas sean flexibles y adaptables a cambios en los requisitos o en el entorno. Esto es esencial en el desarrollo de software a largo plazo.
- Reutilización de código: La reutilización de código es un objetivo importante en la POO, y los patrones de diseño fomentan la reutilización de soluciones probadas para problemas comunes. En lugar de reinventar la rueda, puedes implementar un patrón de diseño existente para resolver un problema específico.

En resumen, los patrones de diseño y la programación orientada a objetos se complementan entre sí. Los patrones de diseño son herramientas que puedes utilizar dentro del marco de la POO para abordar problemas comunes de diseño de manera efectiva y eficiente.
Es eso exactamente lo que busque en la elaboración de los ejemplos que pide la actividad.

Elección de patrones para los ejemplos

Para esta actividad elegí los Patrones de Diseño Singleton (creacional), Adapter (estructural) y Observer (de comportamiento).
A continuación, procedo a explicar en detalle cada uno:

1) Patrón de Diseño Singleton: este patrón implica crear una única instancia de una clase y proporcionar un punto global para acceder a esta instancia.
El uso más común que le podemos dar a este patrón es el de controlar el acceso a recursos compartidos como base de datos, API´s o servicios.

Estructura:

La clase Singleton declara un método estático para obtener la instancia que devuelve la misma instancia de su propia clase.
Además, el constructor debe ocultarse del resto del código de tal manera que la llamada a la instancia sea la única manera de obtener el objeto Singleton.

Aplicabilidad:

- Se utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo; un único objeto de base de datos compartido por distintas partes del programa.
- En otro caso que se debe usar este patrón es cuando se necesite tener un control más estricto de las variables globales.

Ventajas:

- Tener la certeza de que una clase tiene una única instancia.
- Se obtiene un punto de acceso global a dicha instancia (evitando de esta manera la duplicación innecesaria de código).
- Que el objeto Singleton solo se inicia cuando es requerido por primera vez.

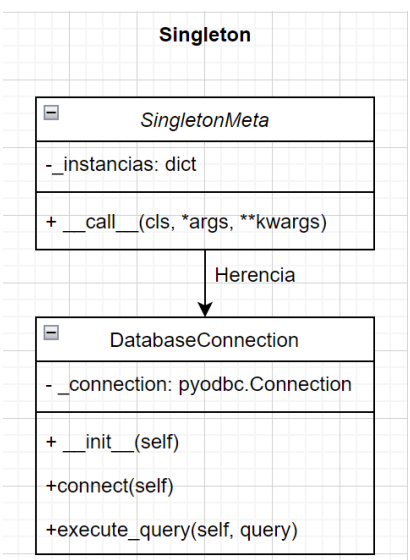
Desventajas:

- Vulnera el principio de responsabilidad única.
- El patrón puede enmascarar un mal diseño, por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros.
- Puede resultar complicado realizar la prueba unitaria del código cliente Singleton porque muchos frameworks de prueba dependen de la herencia a la hora de crear objetos simulados.

Ejemplo:

Para ejemplificar este patrón cree una única instancia de conexión a una base de datos existente. Al ejecutar el código se muestra en la consola una tabla de esa base de datos para verificar que todo funcionó bien.

Diagrama UML:



SingletonMeta es una metaclase que garantiza que solo haya una instancia de DatabaseConnection en el programa. Tiene un método __call__ que se utiliza para crear o devolver la única instancia de DatabaseConnection.

DatabaseConnection es una clase que representa la conexión a la base de datos. Tiene un atributo _connection que almacena la conexión a la base de datos y métodos para inicializar la conexión (__init__ y connect) y ejecutar consultas (execute_query).

Código:

```
#Meta clas epara garantizar que solo haya una conoxion a la base de datos.
class SingletonMeta(type):
    _instancias = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instancias:
            instancia = super().__call__(*args, **kwargs)
            cls._instancias[cls] = instancia
        return cls._instancias[cls]

#Conexion a la base
class DatabaseConnection(metaclass=SingletonMeta):
    _connection = None

    def __init__(self):
        self.connect()

    def connect(self):
        if self._connection is None:
            try:
                import pyodbc
                        connection_string = "DRIVER={ODBC Driver 17 for SQL
Server};SERVER=DESKTOP-OMAF4KB\\SQLEXPRESS;DATABASE=Tienda;UID=DESKTOP-
OMAF4KB\facun;Trusted_Connection=yes"
                self._connection = pyodbc.connect(connection_string)
                print("Conexión exitosa a la base de datos.")
            except Exception as e:
                print("Error de conexión a la base de datos:", str(e))

    def execute_query(self, query):
        if self._connection is not None:
            cursor = self._connection.cursor()
            cursor.execute(query)
            return cursor.fetchall()
        else:
            print("No se puede ejecutar la consulta. La conexión a la base de datos
no está establecida.")

if __name__ == "__main__":

    db_connection = DatabaseConnection()

    if db_connection._connection:
        query = "SELECT * FROM Persona"
```

```
        # Ejecuta consulta y mostra los resultados si la conexión es exitosa
        result = db_connection.execute_query(query)

        if result:
            print("Datos de la tabla Persona:")
            for row in result:
                print(row)
        else:
            print("No se encontraron datos en la tabla Persona.")
    else:
        print("La conexión a la base de datos falló.")
```

La relación entre SingletonMeta y DatabaseConnection se establece a través de la metaclase, lo que garantiza que solo haya una instancia de DatabaseConnection en el programa.

2) Patrón de diseño Adapter: este se usa para permitir que dos interfaces incompatibles trabajen juntas, esto se logra por medio de una clase intermedia que funciona como adaptador.

Estructura:

La interfaz con el cliente describe un protocolo que otras clases deben seguir para poder colaborar con el código.
Servicio es alguna clase útil (normalmente de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.
La clase Adaptadora es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz adaptadora y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.

Aplicabilidad:

- Se utiliza la clase adaptadora cuando se quiera usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Se utiliza el patrón cuando se quiera reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

Ventajas:

- Principio de responsabilidad única. Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.
- Principio de abierto/cerrado. Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

Desventaja:

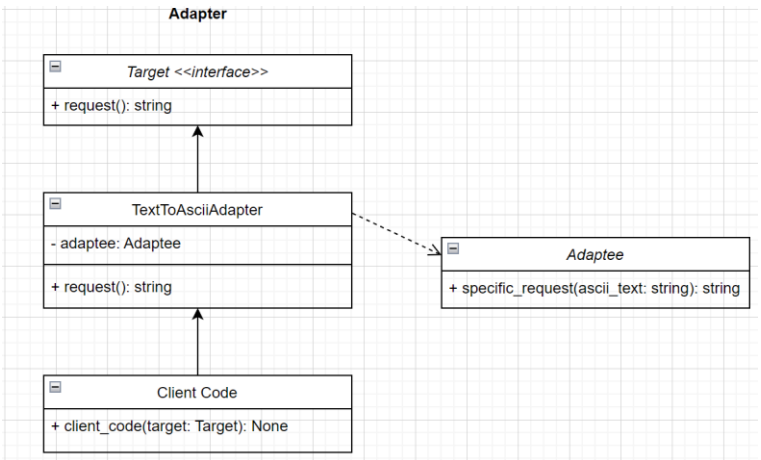
- La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto de tu código.

Ejemplo:

Para este patrón hice un conversor de valores ASCII a texto para poder ejemplificarlo. Al ejecutar el código, el mismo pide al usuario mediante la consola que ingrese un valor en ASCII,

luego lo convierte a texto y lo muestra en consola utilizando la interfaz “incompatible” que solo aceptaba una cadena de texto.

Diagrama UML:



Target es una interfaz (clase abstracta) que define un método request().
Adaptee es una clase que implementa un método specific_request(ascii_text: string).
TextToAsciiAdapter es una clase que hereda de Target e incluye una referencia a un objeto Adaptee. Implementa el método request() para convertir el formato ASCII en texto legible.
Client Code es una función que toma un objeto Target como argumento y llama a su método request().

Código:

```
# Clase abstracta donde defino la interfaz del objetivo que el adaptador debe cumplir.
class Target:
    def request(self) -> str:
        return

# Clase donde represento el objeto a adaptar. Trabaja con ASCII.
class Adaptee:
    def specific_request(self, ascii_text: str) -> str:
        return ascii_text

# Clase donde implemento el adaptador, convirtiendo el formato ASCII en texto legible.
class TextToAsciiAdapter(Target):
    def __init__(self, adaptee: Adaptee):
        self.adaptee = adaptee

    def request(self) -> str:
        ascii_text = self.adaptee.specific_request(input("Ingrese un valor de texto
en formato ASCII: "))
        ascii_values = ascii_text.split()
        text = ''.join(chr(int(value)) for value in ascii_values)
        return f"Texto convertido por el adaptador: {text}"

# Función que simula ser el cliente, utilizando el adaptador para convertir ASCII a
texto.
def client_code(target: "Target") -> None:
```

```
    print(target.request(), end="")

if __name__ == "__main__":
    target = Target()
    adaptee = Adaptee()
    adapter = TextToAsciiAdapter(adaptee)

    client_code(adapter)  # Utilizo el adaptador para convertir y mostrar el texto.
```

3) Patrón de diseño Observer: este patrón trata sobre establecer una relación de dependencia uno-a-muchos entre objetos, de modo que cuando un objeto cambia de estado, todos los objetos dependientes sean notificados y actualizados.

Estructura:

El Notificador envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los notificadores contienen una infraestructura de suscripción que permite a nuevos y antiguos suscriptores abandonar la lista.
Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor.
La interfaz Suscriptora declara la interfaz de notificación. En la mayoría de los casos, consiste en un único método para actualizar.
Los Suscriptores Concretos realizan algunas acciones en respuesta a las notificaciones emitidas por el notificador. Todas estas clases deben implementar la misma interfaz de forma que el notificador no esté acoplado a clases concretas.
El Cliente crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las actualizaciones del notificador.

Aplicabilidad:

- Se utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
- Se utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.

Ventajas:

- Principio de abierto/cerrado. Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).
- Puedes establecer relaciones entre objetos durante el tiempo de ejecución.

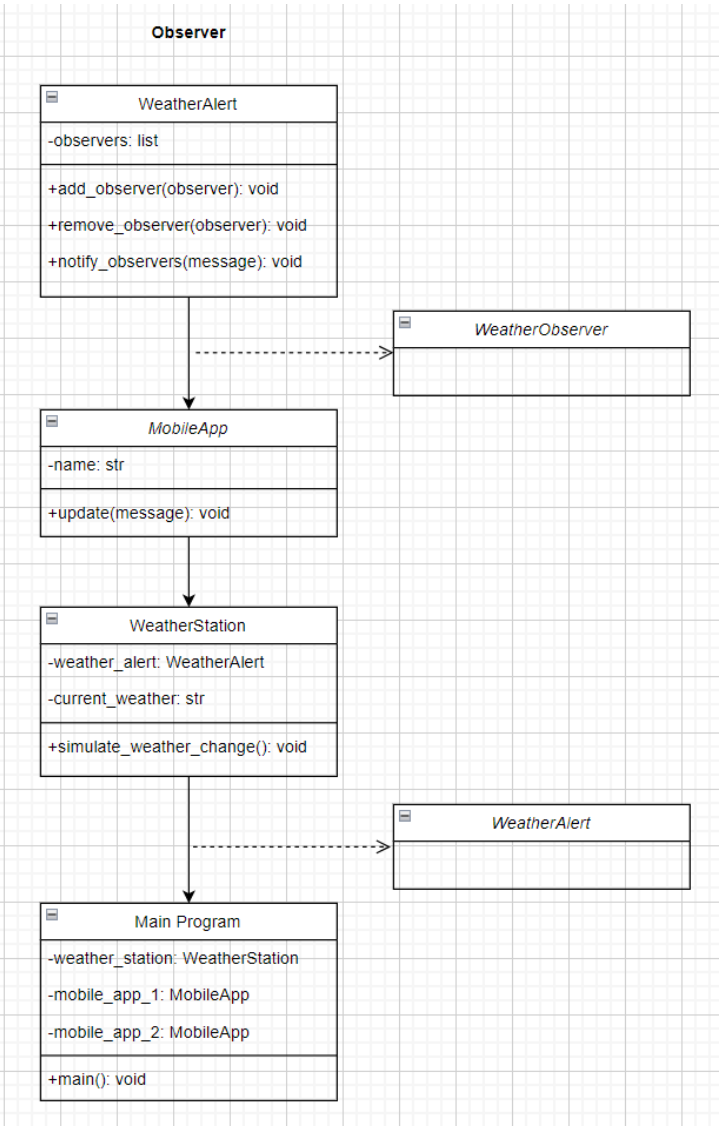
Desventaja:

- Los suscriptores son notificados en un orden aleatorio.

Ejemplo:

Para este patrón simule una app del clima, la cual manda notificaciones a los usuarios cada vez que hay un cambio en el clima (un 1 en un input).
Al ejecutar este código, el mismo le pide al usuario mediante la consola que ingrese un numero (1 o 2). Si ingresa un 1, si simulara un cambio de clima enviando una notificación a los supuestos usuarios, en cambio sí presiona un 2 finaliza la ejecución.

Diagrama UML:



WeatherAlert es una clase que representa un objeto observable y contiene una lista de observadores (observers).
MobileApp es una subclase de WeatherObserver y representa una aplicación móvil que puede recibir notificaciones del estado del tiempo.
WeatherStation representa una estación meteorológica y contiene una instancia de WeatherAlert para notificar a los observadores sobre cambios en el clima.

Código:

```
import sys

#Clase que creo para que represente un objeto observable
class WeatherAlert:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)
```

```
    def remove_observer(self, observer):
        self._observers.remove(observer)

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

class WeatherObserver:
    def update(self, message):
        pass

#Clase que represneta una aplicacion. Hereda de WeatherObserver
class MobileApp(WeatherObserver):
    def __init__(self, name):
        self._name = name

    def update(self, message):
        print(f"{self._name} Nueva notificación de clima 2.0: {message}")

#Clase que representa la estacion meteorologica.
class WeatherStation:
    def __init__(self):
        self._weather_alert = WeatherAlert()
        self._current_weather = "Hoy estará soleado"

    def simulate_weather_change(self):
        if self._current_weather == "Hoy estará soleado":
            self._current_weather = "Hoy estará nublado"
        else:
            self._current_weather = "Hoy estará soleado"
        self._weather_alert.notify_observers(f"{self._current_weather}")


if __name__ == "__main__":
    weather_station = WeatherStation()

    mobile_app_1 = MobileApp("Usuario 1 -")
    mobile_app_2 = MobileApp("Usuario 2 -")

    weather_station._weather_alert.add_observer(mobile_app_1)
    weather_station._weather_alert.add_observer(mobile_app_2)

    print("\nSeleccione una opción:")
    print("1. Simular notificación")
    print("2. Salir")

    while True:
        option = input()
```

```
        if option == "1":
            weather_station.simulate_weather_change()
        elif option == "2":
            sys.exit(0)
        else:
            print("Opción no válida. Por favor, seleccione una opción válida.")
```

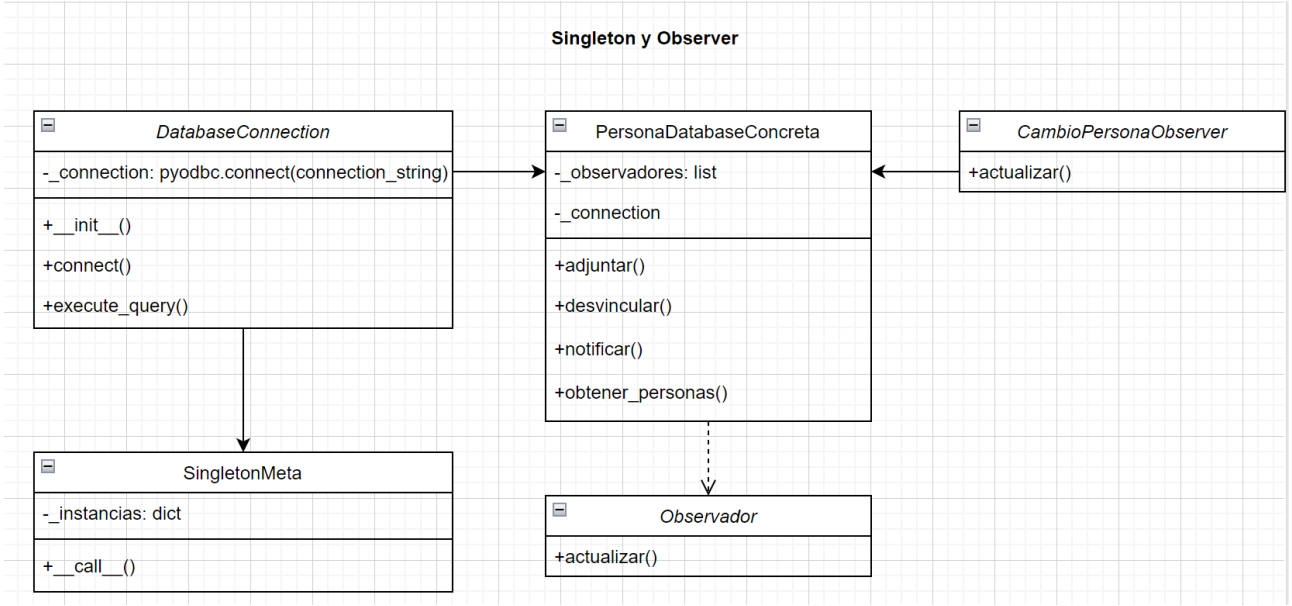
En el programa principal (Main Program), se crea una instancia de WeatherStation y dos instancias de MobileApp, que se agregan como observadores a la WeatherAlert de la estación meteorológica.

4) Patrones de diseño Singleton y Observer juntos

Ejemplo**:**

En este ejemplo combinado utilice el patrón Singleton nuevamente para instanciar una única conexión a base de datos y el patrón Observer para que notifique cada vez que haya cambios en una determinada tabla de esa base.
Cuando se ejecuta el código, el mismo se conecta a la base de datos y comienza a mirar la tabla seleccionada con un refresco cada 5 segundos, si hay cambios en la tabla, muestra los datos de toda la tabla en cuestión en la consola, sino se continúa ejecutando en segundo plano hasta que se presione la tecla “esc”.

Diagrama UML:



DatabaseConnection es una clase Singleton que se encarga de manejar la conexión a la base de datos.
PersonaDatabaseConcreta es una clase que implementa la interfaz PersonaDatabase y se encarga de interactuar con la tabla "Persona" en la base de datos.
CambioPersonaObserver es una clase que implementa la interfaz Observador y se utiliza para observar cambios en la tabla "Persona".
SingletonMeta es una clase metaclass que asegura que solo haya una instancia de DatabaseConnection.

Código:

```
import time
import threading
from abc import ABC, abstractmethod
import pyodbc
import keyboard

# Clase Singleton para la conexión a la base de datos
class SingletonMeta(type):
    _instancias = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instancias:
            instancia = super().__call__(*args, **kwargs)
            cls._instancias[cls] = instancia
        return cls._instancias[cls]

class DatabaseConnection(metaclass=SingletonMeta):
    _connection = None

    def __init__(self):
        self.connect()

    def connect(self):
        if self._connection is None:
            try:
                        connection_string = "DRIVER={ODBC Driver 17 for SQL
Server};SERVER=DESKTOP-OMAF4KB\\SQLEXPRESS;DATABASE=Tienda;UID=DESKTOP-
OMAF4KB\\facun;Trusted_Connection=yes"
                self._connection = pyodbc.connect(connection_string)
                print("Conexión exitosa a la base de datos.")
            except Exception as e:
                print("Error de conexión a la base de datos:", str(e))

    def execute_query(self, query):
        if self._connection is not None:
            cursor = self._connection.cursor()
            cursor.execute(query)
            return cursor.fetchall()
        else:
            print("No se puede ejecutar la consulta. La conexión a la base de datos
no está establecida.")

# Clase Observador que responde a los cambios en la tabla "Persona"
class Observador(ABC):
    @abstractmethod
    def actualizar(self) -> None:
        pass

# Clase Sujeto que representa la tabla "Persona" en la base de datos
```

```python
class PersonaDatabase(ABC):
    @abstractmethod
    def adjuntar(self, observador: Observador) -> None:
        pass

    @abstractmethod
    def desvincular(self, observador: Observador) -> None:
        pass

    @abstractmethod
    def notificar(self) -> None:
        pass

    @abstractmethod
    def obtener_personas(self) -> list:
        pass

class PersonaDatabaseConcreta(PersonaDatabase):
    def __init__(self, connection):
        self._observadores = []
        self._connection = connection

    def adjuntar(self, observador: Observador) -> None:
        # Agrega un observador a la lista de observadores
        self._observadores.append(observador)

    def desvincular(self, observador: Observador) -> None:
        # Elimina un observador de la lista de observadores
        self._observadores.remove(observador)

    def notificar(self) -> None:
        # Notifica a todos los observadores cuando hay cambios en los datos
        for observador in self._observadores:
            observador.actualizar()

    def obtener_personas(self) -> list:
        # Obtiene datos de la tabla "Persona" desde la base de datos
        query = "SELECT * FROM Persona"
        cursor = self._connection.cursor()
        cursor.execute(query)
        return cursor.fetchall()

# Implementación del Observador
class CambioPersonaObserver(Observador):
    def __init__(self, persona_db):
        self._persona_db = persona_db

    def actualizar(self) -> None:
        # Actualiza y mostra los datos cuando se notifican cambios
        personas = self._persona_db.obtener_personas()
```

```
        print("Notificación: Cambios en la tabla Persona")
        for persona in personas:
            print(persona)

def monitorear_cambios(persona_db):
    ultima_actualizacion = None

    while True:
        personas = persona_db.obtener_personas()
        actualizacion_actual = hash(str(personas))

        if ultima_actualizacion is None:
            ultima_actualizacion = actualizacion_actual
        elif actualizacion_actual != ultima_actualizacion:
            # Comprobar si ha habido cambios en los datos y notifica si es así
            print("Notificación: Cambios en la tabla Persona")
            for persona in personas:
                print(persona)
            ultima_actualizacion = actualizacion_actual

        time.sleep(5)  # Espera 5 segundos antes de volver a verificar

if __name__ == "__main__":
    db_connection = DatabaseConnection()

    if db_connection._connection:
        persona_db = PersonaDatabaseConcreta(db_connection._connection)
        cambio_persona_observer = CambioPersonaObserver(persona_db)

        persona_db.adjuntar(cambio_persona_observer)

        # Inicia el monitoreo de cambios en segundo plano
                  monitor_thread    =    threading.Thread(target=monitorear_cambios,
args=(persona_db,))
        monitor_thread.daemon = True
        monitor_thread.start()

        # "Esc" para finalizar
        try:
            keyboard.wait("esc")
            print("Programa finalizado.")
        except KeyboardInterrupt:
            pass
    else:
        print("La conexión a la base de datos falló.")
```

Conclusión

Los patrones de diseño son herramientas de soluciones comprobadas a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.