1. **SOLID – D – Dependency Inversion**

# EJEMPLO DE APLICACION CON NOTIFICACION

from abc import ABC, abstractmethod

# Abstraccion  para el servicio de notificacion ( interface )

class Notificador(ABC):

    @abstractmethod

    def enviar(self, mensaje: str):

        pass

# Implementacion del servicio de notificacion para correo electronico

# Clase de BAJO NIVEL --> (incluye detalles)

class EmailNotificador(Notificador):

    def enviar(self, mensaje: str):

        print(f"Enviando email: {mensaje}")

# Implementacion del servicio de notificacion para SMS

# Clase de BAJO NIVEL --> (incluye detalles)

class SmsNotificador(Notificador):

    def enviar(self, mensaje: str):

        print(f"Enviando sms: {mensaje}")

# Clase o modulo de ALTO NIVEL que maneja la logica de negocios

class App:

    def \_\_init\_\_(self, notificador: Notificador):  # inversion DE DEPENDENCIAS

        self.notificador = notificador

    def enviar\_notificacion(self, mensaje: str):

        self.notificador.enviar(mensaje)

        print(f"Notificacion enviada correctamente")

# MODO DE USO

email\_notificador = EmailNotificador()

app\_con\_email = App(email\_notificador)  # INYECCION DE DEPENDENCIAS

app\_con\_email.enviar\_notificacion("Este es un mensaje de prueba de correo electronico")

sms\_notificador = SmsNotificador()

app\_con\_sms = App(sms\_notificador)  # INYECCION DE DEPENDENCIAS

app\_con\_sms.enviar\_notificacion("Este es un mensaje de prueba de sms")

email\_notificador = EmailNotificador()

app\_con\_email = App(email\_notificador)

app\_con\_email.enviar\_notificacion("Este es un mensaje de prueba de correo electronico")

from abc import ABC, abstractmethod

# Definir la ABSTRACCION del servicio de almacenamiento de productos

class AlmacenamientoProductos(ABC):

@abstractmethod

def agregar\_producto(self, nombre: str, cantidad: int):

pass

@abstractmethod

def obtener\_stock(self, nombre: str) -> int:

pass

# Implentacion del almacenamiento de productos

# METODO DE BAJO NIVEL --> detallado

class MemoriaAlmacenamientoProductos(AlmacenamientoProductos):

def \_\_init\_\_(self):

self.inventario = {}

def agregar\_producto(self, nombre: str, cantidad: int):

if nombre in self.inventario:

self.inventario[nombre] += cantidad

else:

self.inventario[nombre] = cantidad

def obtener\_stock(self, nombre: str) -> int:

return self.inventario.get(nombre, 0)

# METODO DE ALTO NIVEL -- logica de negocios

class GestorProductos:

def set\_almacenamiento(self, almacenamiento: AlmacenamientoProductos):

self.almacenamiento = almacenamiento

def agregar\_producto(self, nombre: str, cantidad: int):

self.almacenamiento.agregar\_producto(nombre, cantidad)

print(f" Producto { nombre} agregado al inventario")

def obtener\_stock(self, nombre: str):

stock = self.almacenamiento.obtener\_stock(nombre)

print(f"Stock de {nombre}:{stock}")

return stock

almacenamiento\_memoria = MemoriaAlmacenamientoProductos()

gestor\_productos = GestorProductos()

gestor\_productos.set\_almacenamiento(almacenamiento\_memoria)  # INYECCION DE DEPENDENCIAS

gestor\_productos.agregar\_producto("Camisa", 2)

gestor\_productos.agregar\_producto("Pantalon", 10)

gestor\_productos.obtener\_stock("Camisa")

gestor\_productos.obtener\_stock("Pantalon")

gestor\_productos.agregar\_producto("Camisa", 1)

gestor\_productos.agregar\_producto("Pantalon", 1)

gestor\_productos.obtener\_stock("Camisa")

gestor\_productos.obtener\_stock("Pantalon")