Sistema de Biblioteca (Refactorizado con Principios SOLID)

Paso 1: Principio de Responsabilidad Única (SRP)

Cada clase tiene una única razón para cambiar.

- **`GestorDeInventario`** → Maneja el inventario (agregar, eliminar, disponibilidad).
- **`IRegistradorDeEventos`** → Interfaz para registrar eventos.
- **`RegistradorDeArchivo`** → Implementación que guarda los eventos en un archivo.

```
class GestorDeInventario:

    def __init__(self):
        self.inventario = {}

    def agregar_item(self, item):
        if item.id in self.inventario:
            print(f"El item con ID {item.id} ya existe.")
            return
        self.inventario[item.id] = {"item": item, "disponible": True}
        print(f"Item '{item.titulo}' agregado al inventario.")

    def eliminar_item(self, item_id):
        if item_id in self.inventario:
            del self.inventario[item_id]
            print(f"Item con ID {item_id} eliminado del inventario.")
        else:
            print(f"Item con ID {item_id} no encontrado en el
inventario.")

    def es_disponible(self, item_id):
        return self.inventario.get(item_id, {}).get("disponible", False)

    def marcar_como_disponible(self, item_id, disponible):
        if item_id in self.inventario:
            self.inventario[item_id]["disponible"] = disponible
```

```
    def obtener_item(self, item_id):
        return self.inventario.get(item_id, {}).get("item", None)


from abc import ABC, abstractmethod
import datetime

class IRegistradorDeEventos(ABC):
    @abstractmethod
    def registrar(self, evento):
        pass

class RegistradorDeArchivo(IRegistradorDeEventos):
    def registrar(self, evento):
        with open("log_biblioteca_refactor.txt", "a") as f:
            f.write(f"[{datetime.datetime.now()}] {evento}\n")
        print("Evento registrado en el archivo
log_biblioteca_refactor.txt")
```

Paso 2: Principio de Segregación de Interfaces (ISP) y Abierto/Cerrado (OCP)

Permite añadir nuevos tipos de notificación sin modificar código existente.

- **`INotificador`** → Interfaz de notificación.
- **`NotificadorDeEmail`** → Notificación por correo.
- **`NotificadorDeSMS`** → Notificación por SMS.

```
class INotificador(ABC):
    @abstractmethod
    def enviar(self, destinatario, mensaje):
        pass

class NotificadorDeEmail(INotificador):
    def enviar(self, destinatario, mensaje):
        print(f"Notificación por email a '{destinatario}': {mensaje}")

class NotificadorDeSMS(INotificador):
    def enviar(self, destinatario, mensaje):
        print(f"Notificación por SMS a '{destinatario}': {mensaje}")
```

Paso 3: Principio de Sustitución de Liskov (LSP)

Las subclases deben poder sustituir a sus clases base.

- **`IItemBiblioteca`** → Clase base abstracta (ID y título).
- **`Libro`** → Añade autor.
- **`Revista`** → Añade mes de publicación.

```
class IItemBiblioteca(ABC):
    def __init__(self, id, titulo):
        self.id = id
        self.titulo = titulo

class Libro(IItemBiblioteca):
    def __init__(self, id, titulo, autor):
        super().__init__(id, titulo)
        self.autor = autor

class Revista(IItemBiblioteca):
    def __init__(self, id, titulo, mes_publicacion):
        super().__init__(id, titulo)
        self.mes_publicacion = mes_publicacion
```

Paso 4: Principio de Inversión de Dependencia (DIP)

Los módulos de alto nivel dependen de abstracciones, no de implementaciones.

- **`ServicioDePrestamos`** → Gestiona préstamos y devoluciones.
    - Depende de **`IRegistradorDeEventos y INotificador`**.

```
class ServicioDePrestamos:
    def __init__(self, gestor_inventario, registrador, notificadores):
        self.gestor_inventario = gestor_inventario
        self.registrador = registrador
        self.notificadores = notificadores

    def prestar(self, item_id, usuario):
        if not self.gestor_inventario.es_disponible(item_id):
            print("El item no está disponible o no existe.")
            return

        self.gestor_inventario.marcar_como_disponible(item_id, False)
        item = self.gestor_inventario.obtener_item(item_id)

        evento = f"Préstamo: '{usuario}' ha prestado el item
'{item.titulo}'."
        self.registrador.registrar(evento)

        for notificador in self.notificadores:
            notificador.enviar(usuario, f"Has prestado el item
'{item.titulo}'.")

        print(f"Item prestado a {usuario}.")

    def devolver(self, item_id, usuario):
        item = self.gestor_inventario.obtener_item(item_id)
        if not item:
            print("El item no existe.")
            return

        self.gestor_inventario.marcar_como_disponible(item_id, True)

        evento = f"Devolución: '{usuario}' ha devuelto el item
'{item.titulo}'."
        self.registrador.registrar(evento)

        print(f"Item devuelto por {usuario}.")
```

Clase Principal (`main`)

Orquesta todo el sistema (la "Biblioteca" real)

```
def main():
    gestor_inventario = GestorDeInventario()
    registrador_eventos = RegistradorDeArchivo()
    notificador_email = NotificadorDeEmail()
    notificador_sms = NotificadorDeSMS()

    notificadores = [notificador_email, notificador_sms]
    servicio_prestamos = ServicioDePrestamos(
        gestor_inventario=gestor_inventario,
        registrador=registrador_eventos,
        notificadores=notificadores
    )

    print("--- Configurando la biblioteca ---")
    libro1 = Libro("978-0321765723", "La Guía del Programador", "Steve
McConnell")
    revista1 = Revista("R-123", "Revista de Ciencia", "Agosto 2024")

    gestor_inventario.agregar_item(libro1)
    gestor_inventario.agregar_item(revista1)

    print("\n--- Realizando un préstamo ---")
    servicio_prestamos.prestar("978-0321765723", "Carlos")
    servicio_prestamos.prestar("R-123", "Maria")

    print("\n--- Intentando prestar un item no disponible ---")
    servicio_prestamos.prestar("978-0321765723", "Javier")

    print("\n--- Realizando una devolución ---")
    servicio_prestamos.devolver("978-0321765723", "Carlos")
    servicio_prestamos.devolver("R-123", "Maria")

if __name__ == "__main__":
    main()
```