<!--SOLID-->

SOLID - Principio de Inversión de Dependencias

<Por="Raquel Martinez"/>

}

Definición{

- Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

}

Concepto - Módulos de Alto Nivel {

```python
class App:
    def __init__(self, notificador: Notificador):
        self.notificador = notificador

    def enviar_notificacion(self, mensaje: str):
        self.notificador.enviar(mensaje)
        print("Notificación enviada correctamente.")
```

}

Concepto - Módulos de Bajo Nivel {

```python
class EmailNotificador(Notificador):
    def enviar(self, mensaje: str):
        print(f'Enviando email: {mensaje}')
```

}

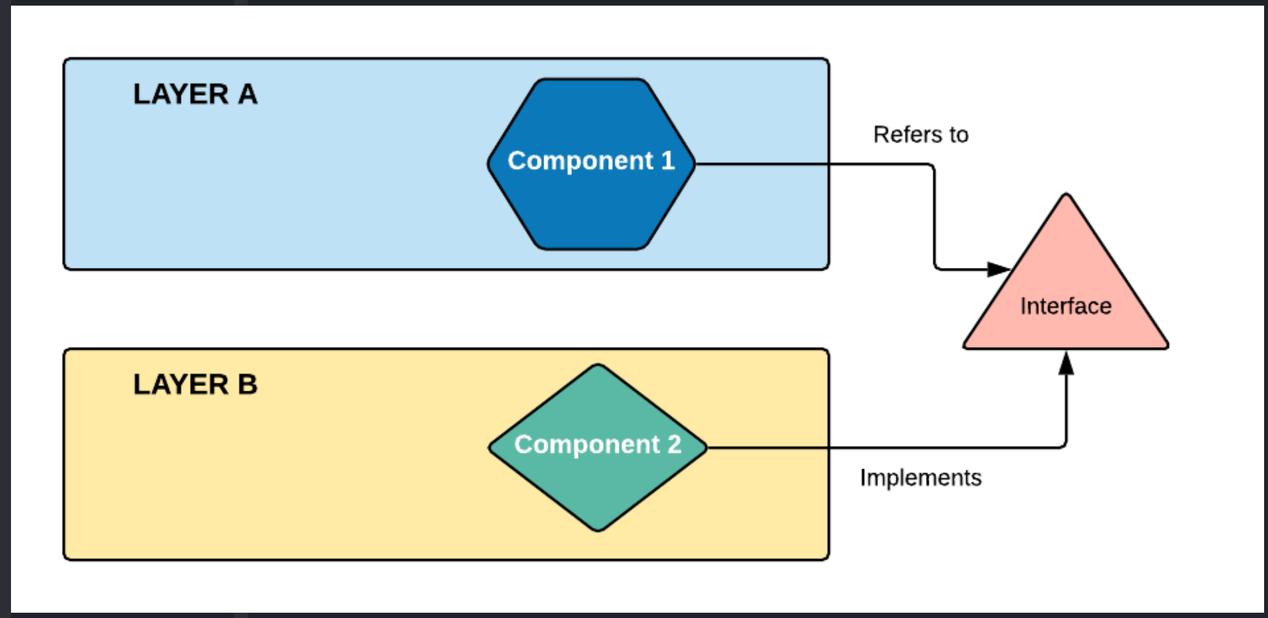
Concepto - Abstracciones {

- Estas son interfaces o clases abstractas que definen los métodos que los módulos de alto y bajo nivel deben implementar.
- Ejemplo: una interfaz Notificador que declara un método enviar(mensaje: str).

}

Beneficios {

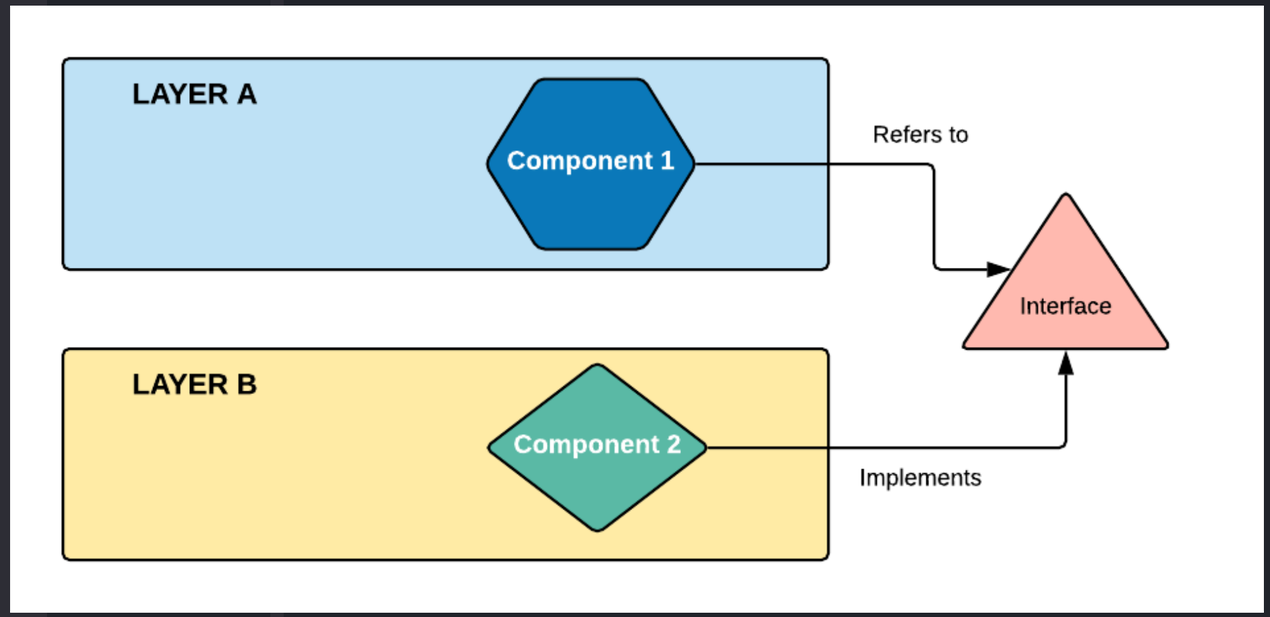
- Reducción del Acoplamiento
- **Mejora de la Modularidad**
- **Facilita el Testing**
- **Mantenibilidad y Extensibilidad**



}

Comparación con Otros Principios SOLID {

- Responsabilidad unica
- Principio Abierto/Cerrado (OCP)
- Principio de Sustitución de Liskov (LSP)
- Principio de Segregación de Interfaces (ISP)



}

NOTA {

DIP es fundamental para la implementación de los otros principios SOLID. Ayuda a mantener el código abierto para la extensión (OCP), garantiza que las implementaciones puedan ser sustituidas (LSP), promueve interfaces específicas (ISP) y mantiene las responsabilidades claras (SRP).

}

Técnicas de Inyección de Dependencias - Inyección de Constructor{

```python
from abc import ABC, abstractmethod

class Notificador(ABC):
    @abstractmethod
    def enviar(self, mensaje: str):
        pass
class EmailNotificador(Notificador):
    def enviar(self, mensaje: str):
        print(f'Enviando email: {mensaje}')
class App:
    def __init__(self, notificador: Notificador):
        self.notificador = notificador

    def enviar_notificacion(self, mensaje: str):
        self.notificador.enviar(mensaje)
        print("Notificación enviada correctamente.")
notificador = EmailNotificador()
app = App(notificador)
app.enviar_notificacion("Hola Mundo!")
```

}

Técnicas de Inyección de Dependencias - Inyección de Setter (Metodo){

```python
class App:
    def __init__(self):
        self.notificador = None

    def set_notificador(self, notificador: Notificador):
        self.notificador = notificador

    def enviar_notificacion(self, mensaje: str):
        if self.notificador:
            self.notificador.enviar(mensaje)
            print("Notificación enviada correctamente.")
        else:
            print("Notificador no configurado.")

app = App()
notificador = EmailNotificador()
app.set_notificador(notificador)
app.enviar_notificacion("Hola Mundo!")
```

}

Técnicas de Inyección de Dependencias - Inyección de Método {

```python
class App:
    def enviar_notificacion(self, notificador: Notificador, mensaje: str):
        notificador.enviar(mensaje)
        print("Notificación enviada correctamente.")

app = App()
notificador = EmailNotificador()
app.enviar_notificacion(notificador, "Hola Mundo!")
```

}

<!--SOLID-->

Codifiquemos! {

<Ejemplo/>

}

<!--SOLID-->

Gracias {

<Por="Raquel Martinez/>

}