Programación orientada a objetos Individual - POO - Unidad 3
Actividad: Taller sobre Principios SOLID en Java - Inversión de Dependencias
Tutor: JOHN CARLOS ARRIETA ARRIETA DOCENTE

Taller: Principio de Inversión de Dependencias (DIP - Dependency Inversion Principle)

Objetivo del Taller:

En este taller, los estudiantes aprenderán el **Principio de Inversión de Dependencias (DIP)**, uno de los principios fundamentales del diseño orientado a objetos en SOLID. Este principio establece que **los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones**. El taller aborda definiciones formales y cotidianas, sus beneficios, desventajas, conceptos previos, ejemplos correctos e incorrectos, y ejercicios propuestos para afianzar el conocimiento.

---

1. Definición Formal del DIP:

El **Principio de Inversión de Dependencias (Dependency Inversion Principle)** establece que **los módulos de alto nivel no deben depender de módulos de bajo nivel, sino que ambos deben depender de abstracciones**. Además, **las abstracciones no deben depender de los detalles; los detalles deben depender de las abstracciones**.

- **Definición Técnica:** "Las dependencias deben ir de las implementaciones hacia las abstracciones, y no al revés."

Imagina que tienes un restaurante (aplicación OO) donde los gerentes (módulos de alto nivel) no deberían preocuparse por cómo los cocineros (módulos de bajo nivel) preparan cada plato. En su lugar, los gerentes solo deberían definir los estándares (interfaces) que los cocineros deben seguir. Los detalles específicos de cómo se preparan (métodos) los platos dependen de los cocineros, no de los gerentes (inversión de dependencias).

---

2. Beneficios del DIP:

- **Aumenta la flexibilidad del Código:** Al depender de abstracciones, el código se puede cambiar o mejorar fácilmente sin afectar a los módulos de alto nivel.
- **Fomenta la Reusabilidad:** Las abstracciones permiten crear componentes que se pueden reutilizar en diferentes contextos.
- **Reduce el Acoplamiento:** Al depender de interfaces, se minimizan las dependencias directas entre los módulos de alto nivel y los módulos de bajo nivel.

---

3. Desventajas del DIP:

- **Mayor Complejidad Inicial:** Implementar el DIP puede requerir un diseño más sofisticado utilizando interfaces o clases abstractas, lo cual puede ser desafiante para programadores novatos.
- **Incremento del Número de Abstracciones:** Puede ser necesario crear muchas interfaces o clases abstractas, lo que puede aumentar la cantidad de código.

---

4. Conceptos Relacionados que se Deben Dominar:

- **Abstracción:** Definir el comportamiento sin especificar los detalles de implementación.
- **Inyección de Dependencias:** Patrón que permite inyectar dependencias a través del constructor, métodos o propiedades, en lugar de crearlas dentro de una clase.
- **Interfaz:** Definición de un contrato que especifica el comportamiento que las clases deben implementar.

---

5. Ejemplos de Aplicación del DIP:

Ejemplo 1: Sistema de Pago con Dependencia de una Interfaz

- **Situación Inicial:** Una clase `ProcesadorPago` depende directamente de una clase concreta `TarjetaCredito`.
- **Aplicación del DIP:** Crear una interfaz `MetodoPago` e implementar diferentes métodos de pago.

```
// Interfaz MetodoPago
public interface MetodoPago {
    void procesarPago(double monto);
}
```

```
// Clase TarjetaCredito que implementa MetodoPago
public class TarjetaCredito implements MetodoPago {
    @Override
```

```
    public void procesarPago(double monto) {
        System.out.println("Procesando pago con tarjeta de crédito: $" + monto);
    }
}
```

```
// Clase PayPal que implementa MetodoPago
public class PayPal implements MetodoPago {
    @Override
    public void procesarPago(double monto) {
        System.out.println("Procesando pago con PayPal: $" + monto);
    }
}
```

```
// Clase ProcesadorPago (depende de la abstracción MetodoPago)
public class ProcesadorPago {
    private MetodoPago metodoPago;

    public ProcesadorPago(MetodoPago metodoPago) {
        this.metodoPago = metodoPago;
    }

    public void realizarPago(double monto) {
        metodoPago.procesarPago(monto);
    }
}
```

```
// Clase de prueba
public class Main {
    public static void main(String[] args) {
        MetodoPago tarjeta = new TarjetaCredito();
        ProcesadorPago procesador = new ProcesadorPago(tarjeta);
        procesador.realizarPago(100.00);

        MetodoPago paypal = new PayPal();
        procesador = new ProcesadorPago(paypal);
        procesador.realizarPago(200.00);
    }
}
```

Explicación: La clase `ProcesadorPago` no depende de una implementación específica, sino de una abstracción (`MetodoPago`), lo cual permite cambiar o agregar nuevos métodos de pago sin modificar la clase de alto nivel.

Ejemplo 2: Sistema de Notificaciones con Inversión de Dependencias

- **Situación Inicial:** Una clase `Notificador` depende directamente de una clase concreta `EmailService`.
- **Aplicación del DIP:** Crear una interfaz `ServicioNotificacion` para definir el contrato de notificación.

```
// Interfaz ServicioNotificacion
public interface ServicioNotificacion {
    void enviarMensaje(String destinatario, String mensaje);
}

// Clase EmailService que implementa ServicioNotificacion
public class EmailService implements ServicioNotificacion {
    @Override
    public void enviarMensaje(String destinatario, String mensaje) {
        System.out.println("Enviando email a: " + destinatario);
        System.out.println("Mensaje: " + mensaje);
    }
}
```

```
// Clase SMSService que implementa ServicioNotificacion
public class SMSService implements ServicioNotificacion {
```

```
    @Override
    public void enviarMensaje(String destinatario, String mensaje) {
        System.out.println("Enviando SMS a: " + destinatario);
        System.out.println("Mensaje: " + mensaje);
    }
}
```

```
// Clase Notificador que depende de la abstracción ServicioNotificacion
public class Notificador {
    private ServicioNotificacion servicioNotificacion;

    public Notificador(ServicioNotificacion servicioNotificacion) {
        this.servicioNotificacion = servicioNotificacion;
    }

    public void notificar(String destinatario, String mensaje) {
        servicioNotificacion.enviarMensaje(destinatario, mensaje);
    }
}
```

```
// Clase de prueba
public class Main {
    public static void main(String[] args) {
        ServicioNotificacion emailService = new EmailService();
        Notificador notificador = new Notificador(emailService);
        notificador.notificar("juan@example.com", "Hola Juan!");

        ServicioNotificacion smsService = new SMSService();
        notificador = new Notificador(smsService);
        notificador.notificar("123456789", "Hola, este es un mensaje SMS.");
    }
}
```

Explicación: La clase `Notificador` depende de la abstracción `ServicioNotificacion` en lugar de depender de implementaciones concretas como `EmailService` o `SMSService`. Esto facilita el cambio o la adición de nuevas formas de notificación.

---

6. Ejemplos de Violación del DIP:

Ejemplo 1: Clase que Depende de una Implementación Concreta

- **Violación:** La clase `ProcesadorPago` depende directamente de `TarjetaCredito`.

```
public class ProcesadorPago {
    private TarjetaCredito tarjeta;

    public ProcesadorPago(TarjetaCredito tarjeta) {
        this.tarjeta = tarjeta;
    }

    public void realizarPago(double monto) {
        tarjeta.procesarPago(monto);
    }
}
```

Problema: La clase `ProcesadorPago` está fuertemente acoplada a `TarjetaCredito`. Si se desea agregar otro método de pago, se debe modificar esta clase.

Ejemplo 2: Dependencia Directa de Clases Concretas

- **Violación:** La clase `Notificador` depende directamente de `EmailService`.

```java
public class Notificador {
    private EmailService emailService;

    public Notificador(EmailService emailService) {
        this.emailService = emailService;
    }

    public void notificar(String destinatario, String mensaje) {
        emailService.enviarMensaje(destinatario, mensaje);
    }
}
```

Problema: La clase `Notificador` depende de `EmailService` y no puede cambiar fácilmente a otro servicio de notificación.

---

7. Ejercicios Propuestos

Ejercicio 1: Sistema de Autenticación

1. Define una interfaz `ServicioAutenticacion` con un método para autenticar usuarios.
2. Implementa clases `AutenticacionLocal` y `AutenticacionOAuth` que implementen la interfaz.
3. Crea una clase `GestorAutenticacion` que dependa de `ServicioAutenticacion`.

Ejercicio 2: Sistema de Almacenamiento de Archivos

1. Define una interfaz `Almacenamiento` con métodos para guardar y recuperar archivos.
2. Implementa clases `AlmacenamientoLocal` y `AlmacenamientoNube` que implementen la interfaz.
3. Crea una clase `GestorArchivos` que dependa de `Almacenamiento`.

Ejercicio 3: Sistema de Reportes

1. Define una interfaz `GeneradorReporte` con un método para generar reportes.
2. Implementa clases `ReportePDF` y `ReporteExcel` que implementen la interfaz.
3. Crea una clase `GestorReportes` que dependa de `GeneradorReporte`.

Conclusión del Taller:

Este taller ha proporcionado una comprensión profunda del **Principio de Inversión de Dependencias (DIP)** y cómo aplicarlo para crear sistemas desacoplados que dependan de abstracciones en lugar de implementaciones concretas. Al aplicar el DIP, los estudiantes aprenderán a mejorar la flexibilidad y la extensibilidad de sus sistemas orientados a objetos.