Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 4.3	Unidad 4.3
Modalidad: Semi -Presencial	**Estratégica Didáctica:** Trabajo individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Via Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	**Fecha Entrega:** A confirmar por el Docente.

MARCO TEÓRICO (Cuestionario reflexivo):

S.O.L.I.D.

Responder el siguiente cuestionario en función de la bibliografía obligatoria.

1. ¿Por qué debes seguir las buenas Prácticas y en qué mejorarías?, justificar con 4 respuestas.

Seguir buenas prácticas es crucial porque el desarrollo de software es una actividad compleja y estas prácticas garantizan la calidad del producto final. Las mejoras específicas incluyen:

Mantenibilidad: El código se divide en componentes pequeños y específicos (gracias a principios como SRP y OCP), lo que lo hace más fácil de entender y modificar sin afectar otras partes del sistema.

Reutilización de código: Al crear interfaces bien definidas y jerarquías coherentes (siguiendo LSP e ISP), los componentes se pueden intercambiar e implementar independientemente, reduciendo problemas de compatibilidad.

Colaboración: La claridad en la estructura y la responsabilidad única de cada componente facilitan que el equipo entienda el código y se asigne tareas de manera efectiva.

Reducción de errores: Una estructura clara y la separación de responsabilidades ayudan a identificar y prevenir problemas antes, minimizando defectos y ahorrando tiempo en pruebas.

2. ¿Que representa el Principio de Responsabilidad única?

Este principio (SRP) establece que una clase debe tener una única razón para cambiar, es decir, una sola responsabilidad . Significa que la clase debe hacer una sola cosa y hacerla bien. El objetivo es promover la cohesión, agrupando funcionalidades relacionadas, lo que facilita el mantenimiento, ya que los cambios en una parte no deberían afectar a otras.

3. ¿Qué es el Principio de OCP y que evita?

El Principio de Abierto/Cerrado (OCP) dicta que las entidades de software (clases, módulos, funciones) deben estar abiertas para su extensión pero cerradas para su modificación. Esto permite agregar nuevas funcionalidades sin alterar el código fuente existente.
Evita varios problemas críticos:
Riesgo de errores: Al no modificar código que ya funciona, se reduce la probabilidad de introducir nuevos fallos.

Efecto dominó: Limita el impacto de los cambios en otras partes del sistema que dependen de la entidad modificada.

Falta de escalabilidad: Facilita agregar características en sistemas grandes sin riesgo de romper lo existente.

4. Que es el Principio de Sustitución de Liskov, dar ejemplos.

El principio (LSP) establece que las subclases deben ser capaces de reemplazar a sus superclases sin alterar el comportamiento correcto del programa.
Una subclase debe ser una extensión compatible, manteniendo la interfaz y el comportamiento esperado de la clase base.
Ejemplos:

```cpp
#include <iostream>
using namespace std;

// CLASE BASE
class Dispositivo {
public:
    virtual void encender() = 0;
    virtual ~Dispositivo() = default;
};

// SUBCLASES QUE CUMPLEN LSP
class Laptop : public Dispositivo {
public:
    void encender() override {
        cout << "Laptop encendida (pantalla + CPU)\n";
    }
};

class Impresora : public Dispositivo {
public:
    void encender() override {
        cout << "Impresora encendida (calentando cabezal)\n";
    }
};

// FUNCIÓN QUE USA LA CLASE BASE (LSP en acción)
void usarDispositivo(Dispositivo* d) {
    d->encender();  // Funciona con Laptop, Impresora, etc.
}

int main() {
    Dispositivo* d1 = new Laptop();
    Dispositivo* d2 = new Impresora();

    usarDispositivo(d1);  // Correcto
    usarDispositivo(d2);  // Correcto

    delete d1; delete d2;
}
```

5. Que es la Segregación de Interfaz, como lo Implementaría, realizar un diagrama UML (sin un lenguaje en particular)

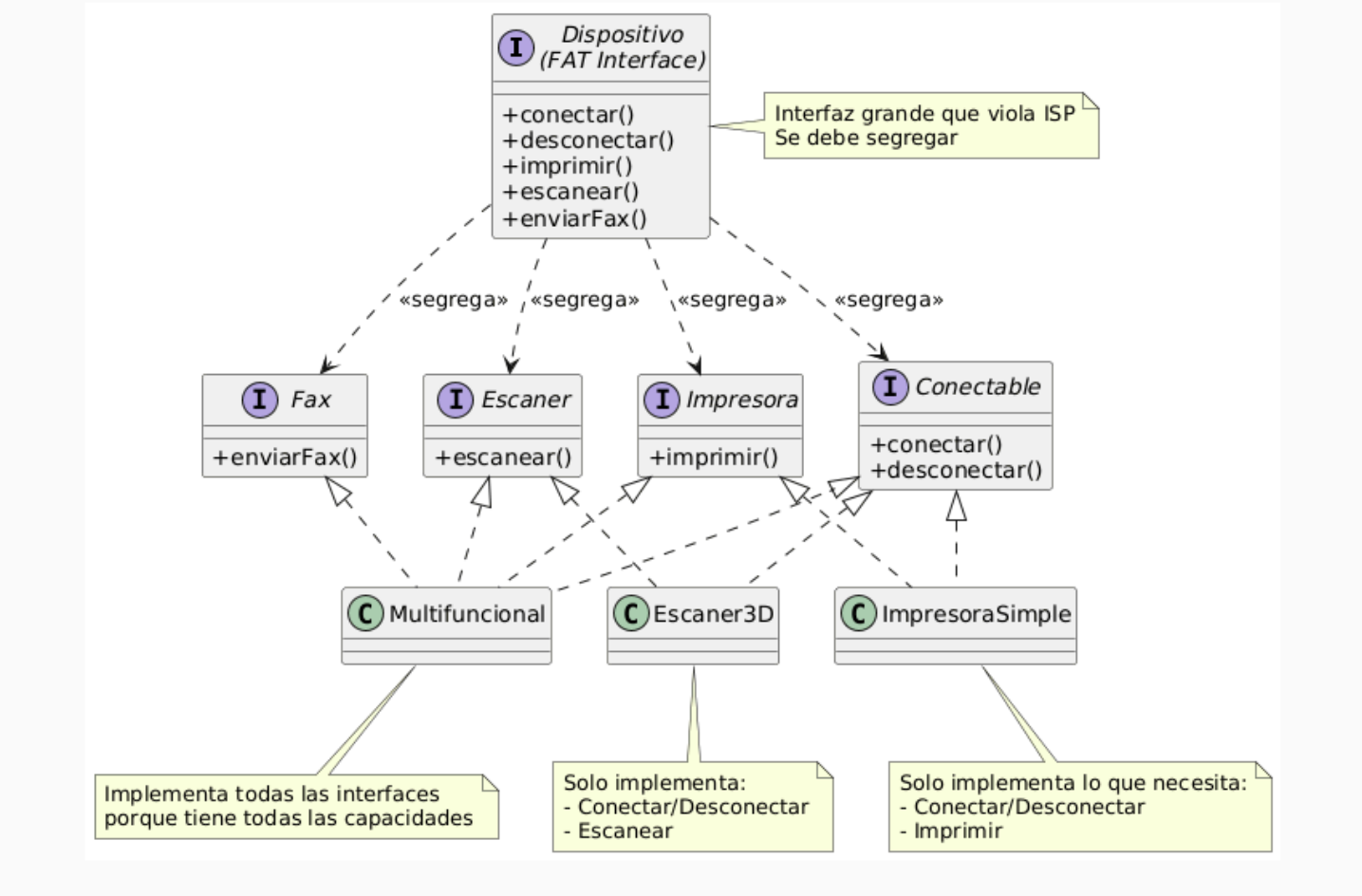
El Principio de Segregación de Interfaz (ISP) establece que una clase no debe verse forzada a implementar interfaces que no utiliza. Las interfaces deben ser específicas y cohesivas para las necesidades del cliente, en lugar de ser interfaces generales y enormes.

Implementación

Se logra dividiendo interfaces grandes ("fat interfaces") en interfaces más pequeñas y específicas. Por ejemplo, en lugar de una interfaz Dispositivo con métodos para conectar(), desconectar(), imprimir(), escanear(), enviarFax(), se pueden crear interfaces separadas como:

Conectable (conectar, desconectar)
Impresora (imprimir)
Escaner (escanear)
Fax (enviarFax)

Esto permite que las clases implementen solo lo que necesitan. Por ejemplo, una impresora simple solo implementaría Conectable e Impresora, mientras que un dispositivo multifunción podría implementar todas las interfaces.



6. ¿Qué entiende con Inversión de Dependencia? ¿Por qué es tan Importante?

El Principio de Inversión de Dependencia (DIP) establece que los módulos de alto nivel no deben depender de los de bajo nivel; ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino los detalles de las abstracciones.

Es importante porque crea código:
Flexible: Permite cambiar la implementación (detalles) sin tocar los módulos de alto nivel.
Testable: Facilita probar módulos de alto nivel de forma aislada usando mocks o stubs de las abstracciones.
Mantenible: Hace el código más fácil de entender y modificar.

7. Qué relación hay entre Acoplamiento, Cohesión y Dependencia con S.O.L.I.D.

Los principios SOLID están diseñados para optimizar estos tres conceptos:

Acoplamiento: SOLID busca reducirlo. SRP reduce el acoplamiento al tener una sola razón de cambio. [cite_start]OCP y DIP lo reducen al depender de abstracciones y extensiones en lugar de modificaciones directas .

Cohesión: SOLID busca aumentarla. [cite_start]SRP y ISP aseguran que las clases e interfaces tengan propósitos bien definidos y métodos relacionados, mejorando la cohesión .

Dependencia: SOLID busca debilitarla y orientarla correctamente. [cite_start]DIP invierte la dependencia para que apunte hacia abstracciones estables en lugar de implementaciones volátiles .

8. ¿Qué es la “Inyección de Dependencia”?

Es una técnica de diseño donde los objetos reciben sus dependencias desde una fuente externa en lugar de crearlas ellos mismos. Esto reduce el acoplamiento, ya que el objeto no necesita conocer la implementación concreta de sus dependencias, solo su interfaz (abstracción)

9. ¿Qué es la delegación en POO?

Es una técnica donde un objeto, en lugar de realizar una tarea por sí mismo, le encarga (delega) esa tarea a otro objeto. Esto ayuda a reducir el acoplamiento y promueve la reutilización, ya que un objeto puede usar la funcionalidad de otro sin heredar de él.

10. Nombrar 5 Buenas Prácticas.

Planificación adecuada: Definir objetivos y requisitos antes de empezar.

Seguir los principios SOLID: Para crear código flexible y mantenible.

Utilizar la inyección de dependencias: Para reducir el acoplamiento y mejorar la testabilidad.

Control de versiones: Usar herramientas como Git para gestionar cambios y colaborar.

Pruebas unitarias y automatizadas: Para asegurar la calidad y facilitar la detección de errores.

Lic. Oemig José Luis.