FICH UNL • FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnologías de Programación

Paradigma Orientado a Objetos

1 – Pecados capitales del diseño

1 - Rigidez

2 - Fragilidad

3 - Inmovilidad

4 - Viscosidad

5 - Complejidad innecesaria

6 – Repetición innecesaria

7 - Opacidad

1 – Pecados capitales del diseño

1 - Rigidez

Tendencia del software a ser difícil de cambiar, aun ante cambios simples.
Ejemplo: Un único cambio causa una cascada de cambios subsecuentes en módulos dependientes. Cuantos más módulos se deben cambiar, más rígido es el diseño

2 - Fragilidad

Tendencia del software a romperse en varios lugares cuando se hace un cambio
Aclaración: Frecuentemente los nuevos problemas surgen en áreas que no tienen relación conceptual con el área que fue cambiada.

3 - Inmovilidad

Un diseño es inmóvil cuando contiene partes que podrían ser útiles en otros sistemas, pero el esfuerzo y riesgo de separarlas del sistema original es muy grande.

1 – Pecados capitales del diseño

4 - Viscosidad

Un software viscoso es aquel que su diseño es difícil de preservar
Existen dos formas:

- viscosidad del software: es alta si la forma de introducir un cambio, hace que sea mas difícil preservar el diseño que vulnerarlo
- viscosidad del ambiente: cuando el ambiente de desarrollo es lento e ineficiente

5 – Complejidad innecesaria

Ocurre cuando el diseño contiene elementos que no son actualmente útiles Ejemplo: Cuando se anticipan cambios a los requerimientos y se construyen “facilidades” para manejar dichos cambios potenciales.
En principio, parece algo bueno que previene pesadillas en futuros cambios o muchas veces, el efecto es opuesto ya que el diseño se satura de mecanismos que nunca se usan y que el software debe mantener

1 – Pecados capitales del diseño

6 – Repetición innecesaria

Cortar y pegar puede ser desastroso para operaciones de edición de código.
Cuando el mismo código aparece una y otra vez, en ligeramente distintas formas, se está necesitando una abstracción..
Cuando hay código redundante en el sistema, los cambios en el sistema pueden ser arduos..

7 - Opacidad

Tendencia de un módulo a ser difícil de entender.
El código puede ser escrito de una manera clara y expresiva, o de una manera compleja y opaca.
A medida que el código evoluciona en el tiempo, llega a ser más y más opaco.

2 – Objetos bien formados

Se refiere a que los objetos cumplen con ciertos principios y pautas de diseño que los hacen coherentes y mantenibles. Algunos criterios comunes para que un objeto sea considerado bien formado son:

Encapsulación: Los objetos deben encapsular su estado interno y ocultarlo de otros objetos. Deben proporcionar una interfaz clara y consistente para interactuar con ellos.

Cohesión: Los objetos deben tener una única responsabilidad claramente definida. Deben estar enfocados en hacer una cosa y hacerla bien.

Acoplamiento: Los objetos deben estar acoplados de manera débil, lo que significa que deben depender lo menos posible de otros objetos. Esto promueve la flexibilidad y la reutilización del código.

Cumplimiento de los principios SOLID: Los objetos bien formados suelen cumplir estos principios.

3 – Principios SOLID

En el paradigma orientado a objetos, existen varias reglas y principios de diseño que ayudan a crear un código limpio, modular y que se pueda mantener. Algunos de los principales son:

1. Principio de Responsabilidad Única (**S**RP)
2. Principio de Abierto/Cerrado (**O**CP)
3. Principio de Sustitución de Liskov (**L**SP)
4. Principio de Segregación de Interfaces (**I**SP)
5. Principio de Inversión de Dependencia (**D**IP)

3 – Principios SOLID

Su autor-mentor es Robert C. Martin y surgieron a comienzos del año 2000.

Se los considera los cinco principios básicos en el diseño y la programación orientada a objetos.
La intención es aplicar estos principios en conjunto para que sea más probable obtener un software fácil de mantener y extender en el tiempo.

Son guías, no son reglas inamovibles

3 – Principios SOLID

1. **<u>Principio de Responsabilidad Única (SRP):</u>** Cada clase o módulo debe tener una única responsabilidad o razón para cambiar. Esto implica que una clase debe tener un solo propósito y estar enfocada en hacer una sola cosa. Corresponde a la "**S" en SOLID.**

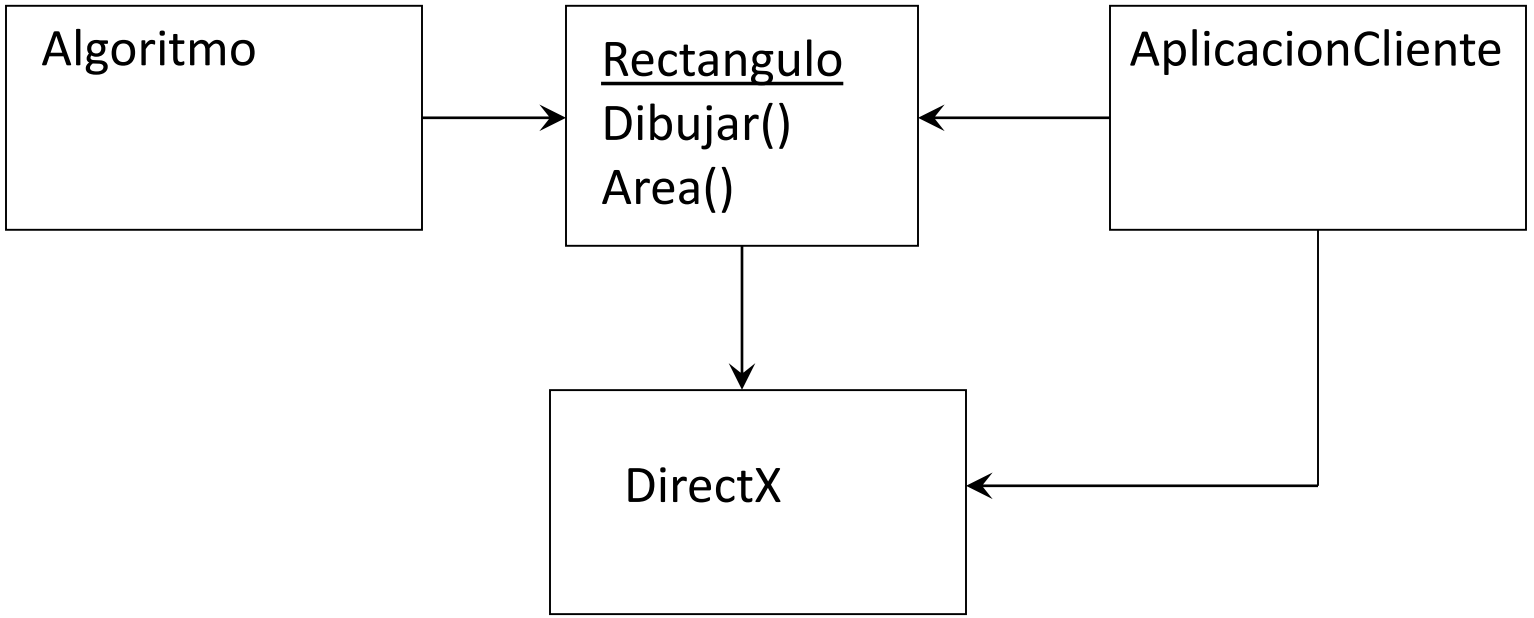
¿Por qué SRP?

- Responsabilidad única = mayor cohesión
- No aplicarlo da como resultado dependencias innecesarias
  - Más razones para cambiar.
  - Rigidez, inmovilidad

3 – Principios SOLID

1. Principio de Responsabilidad Única (SRP):

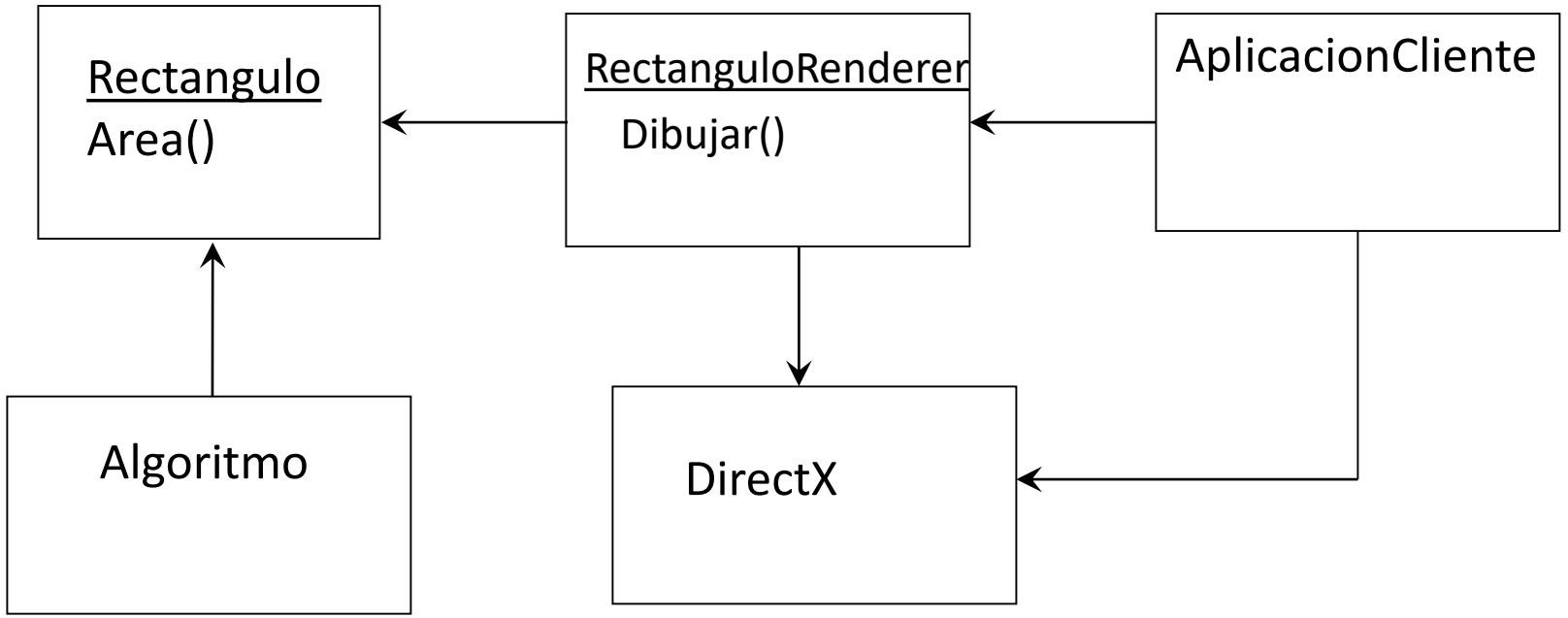
- El rectángulo tiene dos responsabilidades:



- De repente, el algoritmo necesita DirectX

3 – Principios SOLID

1. Principio de Responsabilidad Única (SRP):



3 – Principios SOLID

2. **<u>Principio de Abierto/Cerrado (OCP):</u>** Las entidades de software (clases, módulos, etc.) deben estar abiertas para su extensión pero cerradas para su modificación. Esto significa que se deben poder agregar nuevas funcionalidades o comportamientos sin modificar el código existente. Corresponde a la **"O" en SOLID.**

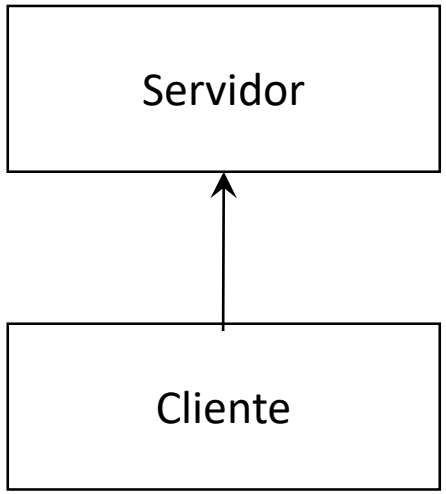
¿Por qué OCP?

Si no se sigue, un solo cambio en un programa da como resultado una cascada de cambios en los módulos dependientes.

El programa se vuelve frágil, rígido, impredecible e inreutilizable.

3 – Principios SOLID

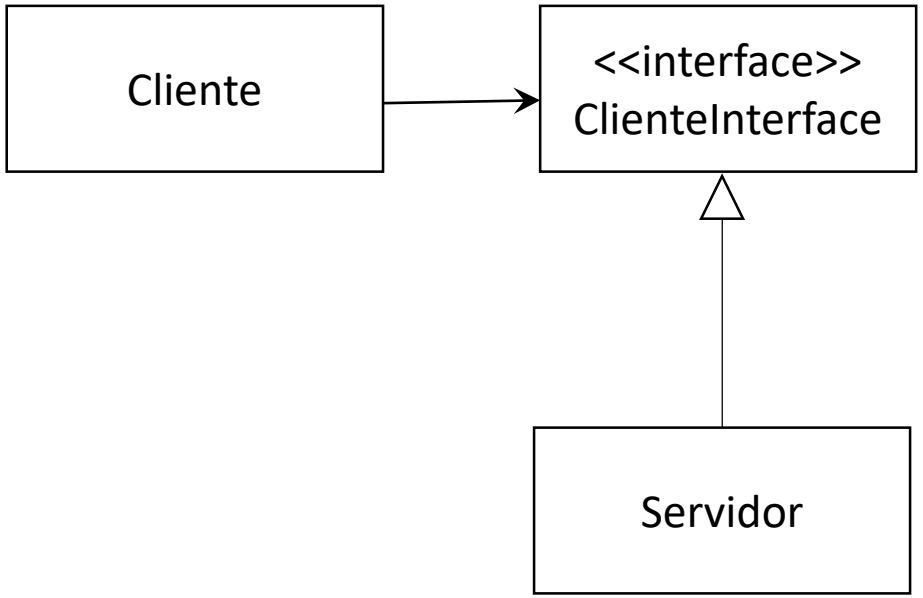
2. **Principio de Abierto/Cerrado (OCP):**



Si por algún motivo la clase o implementación del Servidor es modificada, entonces la clase Cliente también debe ser modificada.

3 – Principios SOLID

2. <u>Principio de Abierto/Cerrado (OCP)</u>



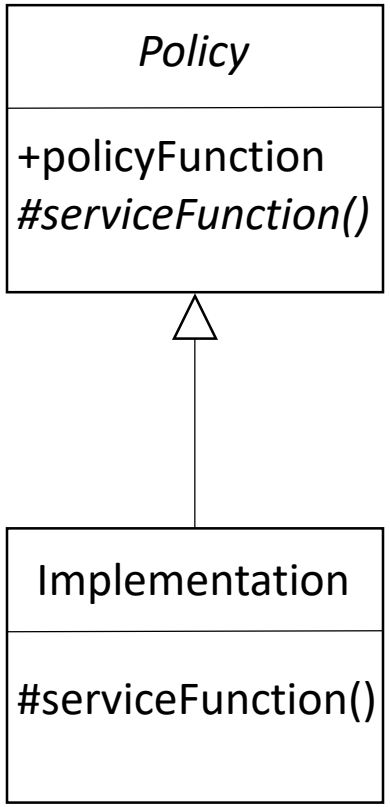
Se agrega una interfaz intermedia, ClienteInterface, entre Cliente y Servidor.

Si, por algún motivo, la implementación del servidor cambia, el cliente probablemente no requiera cambios.

La clase ClienteInterface es cerrada para modificación aunque si está abierto para extensión.

3 – Principios SOLID

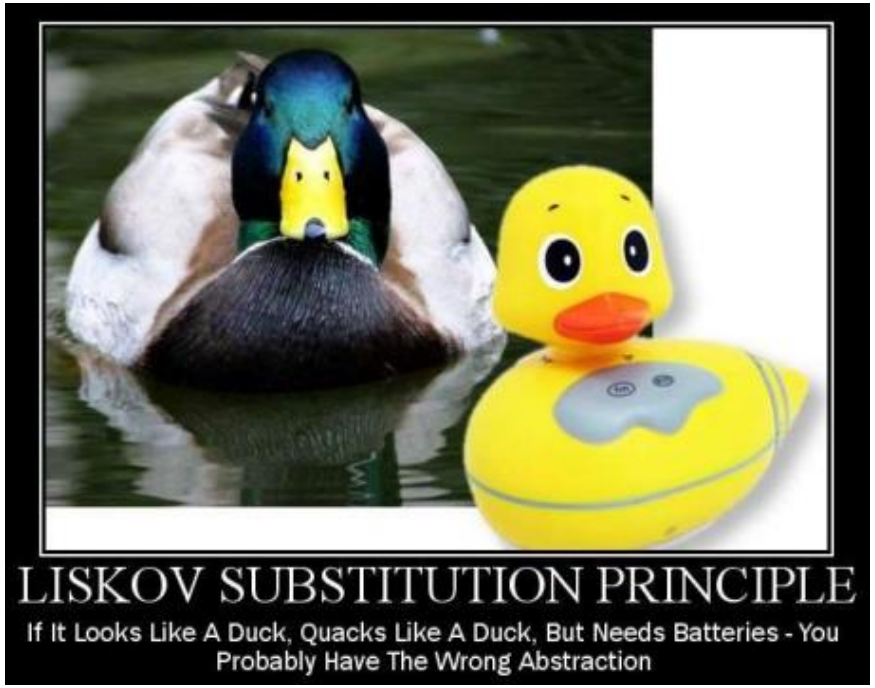
2. <u>Principio de Abierto/Cerrado (OCP)</u>



El patrón Template Method (GoF design patterns) es una alternativa clásica para lograr OCP

3 – Principios SOLID

3. **<u>Principio de Sustitución de Liskov (LSP):</u>** Las clases derivadas deben poder ser sustituidas por sus clases base sin alterar el correcto funcionamiento del programa. Esto garantiza que los objetos de una clase base puedan ser reemplazados por objetos de una clase derivada sin generar errores ni comportamientos inesperados. **Corresponde a la "L"** en SOLID



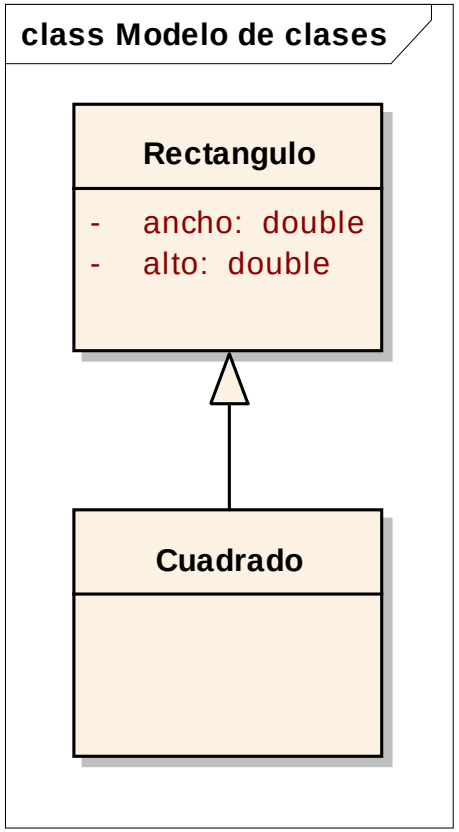
¿Por qué <u>LSP</u>?

No seguir el LSP resulta en un desastre:

- Violaciones de OCP (si/entonces para identificar tipos)
- La prueba unitaria de superclase fallará
- Comportamiento extraño

3 – Principios SOLID

3. Principio de Sustitución de Liskov (LSP):



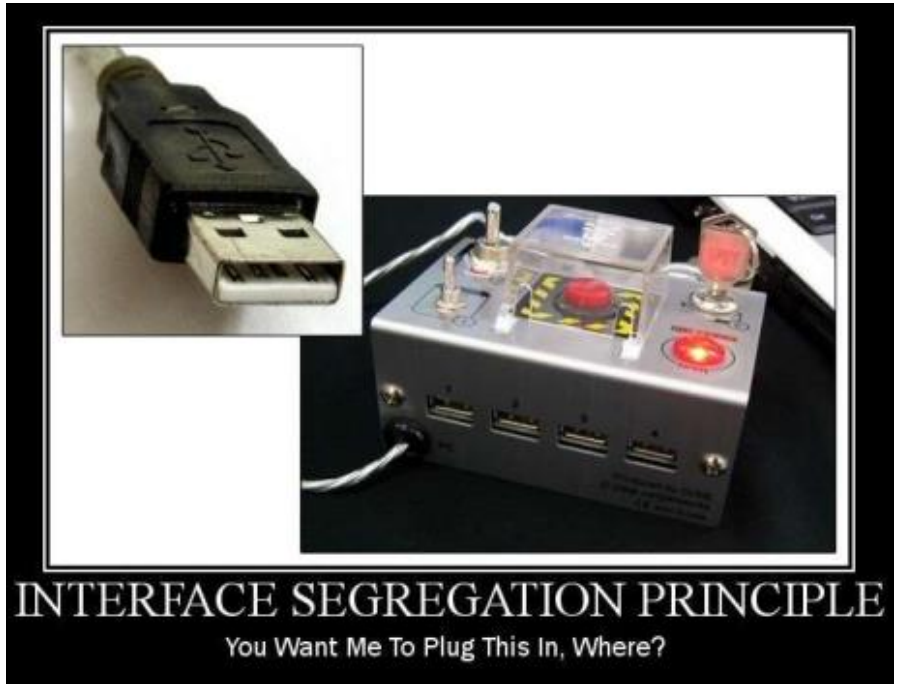
La herencia generalmente se interpreta como una relación “es un”

- un Cuadrado “es un” Rectángulo
- sin embargo:
  - el Cuadrado no necesita tener un Alto y un Ancho, sólo le alcanza con un Lado → desperdicio de memoria
  - a un Cuadrado se le puede setear el Alto y el Ancho → Problema de consistencia !!!!

Solución: Sobreescribir esas propiedades en la clase Cuadrado

3 – Principios SOLID

4. **<u>Principio de Segregación de Interfaces (ISP):</u>** Los clientes no deben depender de interfaces que no utilizan. Este principio promueve la creación de interfaces específicas y cohesivas en lugar de interfaces monolíticas, evitando así la dependencia de funcionalidades innecesarias.



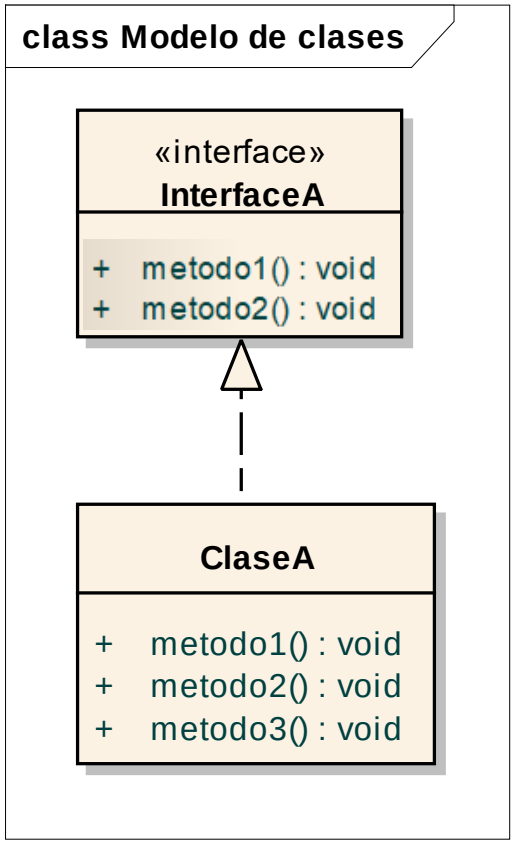
¿Por qué?

De lo contrario, mayor acoplamiento entre diferentes clientes

Básicamente una variación de SRP

3 – Principios SOLID

4. **Principio de Segregación de Interfaces (ISP):**

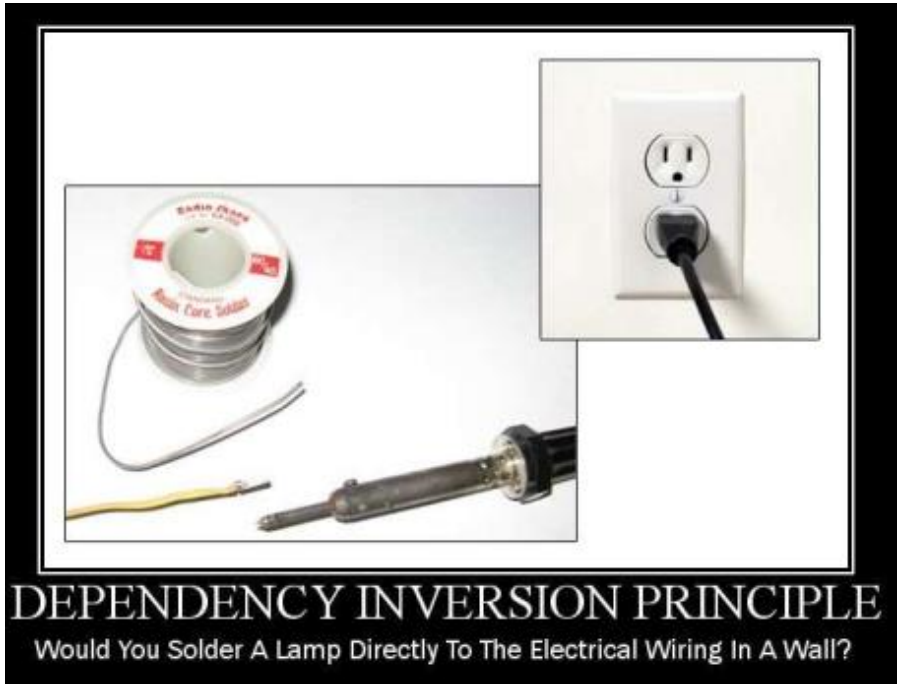


Ejemplo incorrecto

La clase A también tiene un método adicional que no está definido en la interfaz. Esto viola el principio porque la clase está obligada a implementar métodos que no son necesario para su funcionamiento. En este caso, la clase A tiene una dependencia innecesaria de method3()

3 – Principios SOLID

5. **Principio de Inversión de Dependencia (DIP):** Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Este principio promueve la dependencia hacia interfaces o abstracciones en lugar de implementaciones concretas, lo que facilita la flexibilidad y el cambio en el diseño.



¿Por qué DIP?

Incrementar el acoplamiento débil:

- Las interfaces abstractas no cambian
- Las clases concretas implementan interfaces
- Clases concretas fáciles de tirar y reemplazar

Aumentar la movilidad

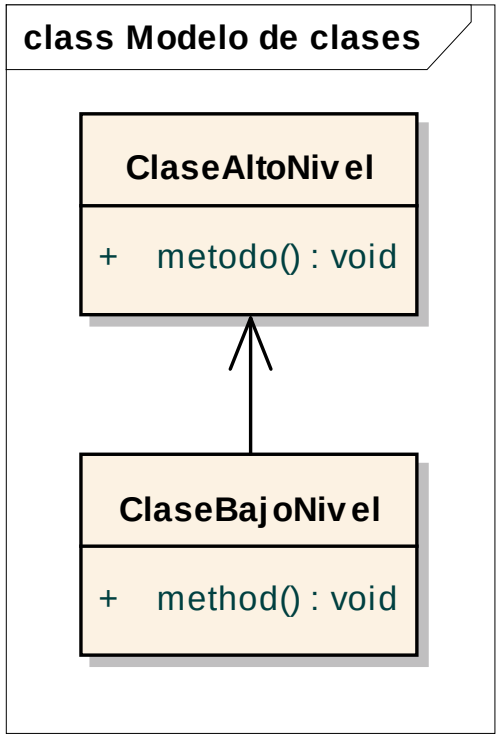
Aumentar el aislamiento

- Disminuir la rigidez
- Aumentar la capacidad de prueba
- Aumentar la capacidad de mantenimiento

Estrechamente relacionado con el Principio de Sustitución de Liskov (LSP)

3 – Principios SOLID

5. <u>Principio de Inversión de Dependencia (DIP):</u>



La ClaseAltoNivel depende directamente de la clase ClaseBajoNivel. Esto viola el principio, ya que las dependencia se establece hacia una implementación concreta en lugar de hacia una abstracción.
Por otra parte, cualquier cambio en ClaseBajoNivel puede afectar directamente a ClaseAltoNivel lo que resulta en un acoplamiento fuerte entre clases.

Solución: agregar una interfaz que represente una abstracción de la funcionalidad necesaria para ClaseAltoNivel

4 – Ley de Demeter

La conocida como ***Ley de Demeter*** o ***del buen estilo***, nos garantiza, durante un desarrollo orientado a objetos una buena escalabilidad, depuración de errores y mantenimiento, ya que ayuda a ***maximizar la encapsulación***. Esto ayuda a **mantener un nivel bajo de acoplamiento**. Es una norma muy simple de seguir.

A menudo, el contenido de la ley se abrevia sólo con una frase:

"Habla sólo con tus amigos"

4 – Ley de Demeter

Un **método M** de un **objeto O** solo debería invocar métodos:

- suyos
- de sus parámetros objetos que cree
- objetos miembros de la clase (atributos)

3 – Ley de Demeter

Una forma de comprender mejor esta ley es dar la vuelta al enunciado y enumerar los casos prohibidos: **no se debe llamar a métodos de los objetos devueltos por otros métodos.**
El caso más común que debemos evitar son las cadenas de métodos, de la forma:

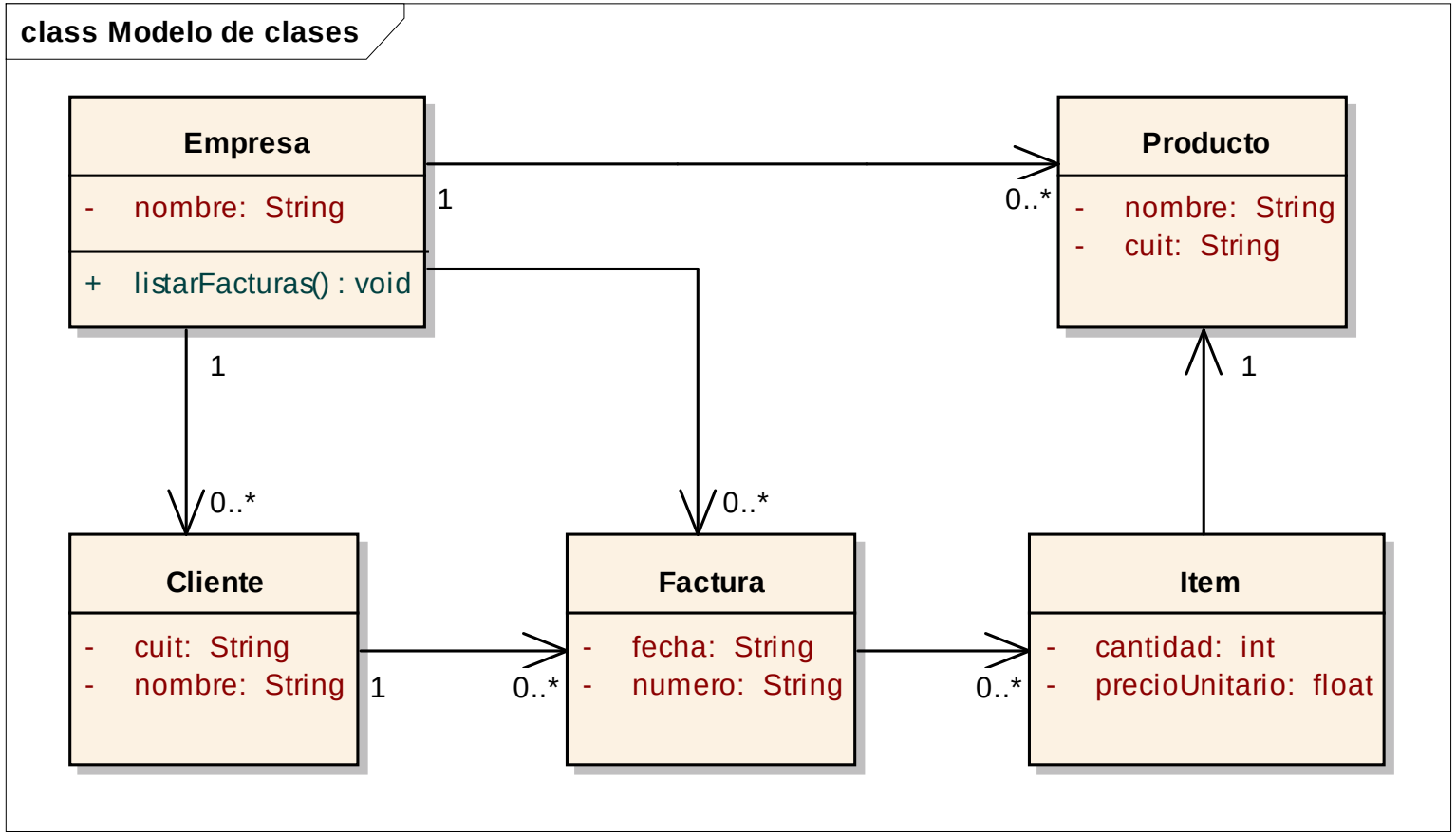
a.obtenerX().obtenerY().obtenerValor();

y sustituirlas por funciones que realicen dicha acción:

a.obtenerXYValor();

4 – Ley de Demeter

Diagrama de clases en UML 2.0 parcial del ejercicio 1 del TP 2



4 – Ley de Demeter

Antes de aplicar la Ley

```
public class Empresa {

    private String nombre;
    private Collection<Producto> cProductos;
    private Collection<Factura> cFacturas;
    private Collection<Cliente> cClientes;

    public void listarFacturas(){
        for (Factura oFactura:cFacturas){
            if(!oFactura.estaAnulada()){
                Float totalFactura = 0;
                Collection<Item> cItems = oFactura.publicarItems();
                for (Item oItem:cItems){
                    totalFactura += oItem.calcularTotalItem();
                }
                System.out.println("Total de la factura " + totalFactura);
            }
        }
    }
}
```

4 – Ley de Demeter

El método listarFacturas() itera sobre la colección de facturas cFacturas y realiza operaciones en cada una de ellas sin solicitar información sobre su estado directamente.

Dentro del bucle, se verifica si la factura está anulada llamando al método estaAnulada() de la clase Factura. Si bien esto podría considerarse una pregunta directa, es posible que dentro del método estaAnulada() exista lógica interna que encapsule los detalles de la anulación de una factura. Por lo tanto, esto no necesariamente viola la Ley de Demeter en su totalidad.

4 – Ley de Demeter

Hay una violación de la Ley de Demeter en la siguiente línea:

Collection<Item> cItems = oFactura.publicarItems();

En lugar de acceder directamente a la colección de ítems de la factura **oFactura.getItems()** se llama a un método **publicarItems()** en la clase **Factura**.

Esto implica que la clase Empresa está accediendo a la estructura interna de Factura en lugar de solicitar los datos necesarios a través de un método de interfaz especifico

4 – Ley de Demeter

Después de aplicar la Ley

```
public class Empresa {

    private String nombre;
    private Collection<Producto> cProductos;
    private Collection<Factura> cFacturas;
    private Collection<Cliente> cClientes;

    public void listarFacturas(){
        for(Factura oFactura : cFacturas) {
            if(!oFactura.estaAnulada()) {
                Float totalFactura = oFactura.calcularTotal();
                System.out.println("Total de la factura " + totalFactura);
            }
        }

    }

}
```

5 – Tell don’t ask - TDA

Es un principio de diseño de software que promueve la encapsulación y la responsabilidad adecuada de las clases y objetos. Se refiere a la idea de que en lugar de preguntar a un objeto por su estado y luego tomar decisiones basadas en esa información, se debe simplemente "decirle" al objeto qué hacer y permitirle que gestione internamente su estado y tome las decisiones adecuadas

Se debe proporcionar al objeto los comandos necesarios para que realice las acciones requeridas. Esto se basa en el principio de encapsulamiento, donde los objetos encapsulan tanto sus datos como su comportamiento

Ayuda a reducir el acoplamiento y la dependencia entre objetos. Facilita el mantenimiento y la extensibilidad del código, ya que los cambios en el estado interno de un objeto no afectarán a otras partes del sistema.

Fomenta un diseño más cohesivo y modular, donde los objetos son capaces de gestionar su propio estado y realizar acciones basadas en los comandos que se les envían, en lugar de ser manipulados externamente a través de preguntas y respuestas sobre su estado.

5 – Tell don’t ask - TDA

¿Cuándo está permitido pedirle un valor a un objeto ?

Se le puede pedir un valor de estado a un objeto (get) siempre que el dato no se utilice para tomar un decisión sobre el estado interno de dicho objeto.

```
// Caso incorrecto - tomo una decisión en base al estado interno de otro obj.
for(Factura oFactura : cFacturas) {
    if(!oFactura.estaAnulada()) {
        Float totalFactura = oFactura.calcularTotal();
        System.out.println("Total de la factura " + totalFactura);
    } else {
        System.out.println("Total de la factura 0 (cero)");
    }
}
// Caso correcto
for(Factura oFactura : cFacturas) {
    System.out.println("Total de la factura " + oFactura.calcularTotal());
}
```

5 – Tell don’t ask - TDA

¿Cuándo está permitido pedirle un valor a un objeto ?

Se le puede pedir un valor de estado a un objeto (get) siempre que el dato no se utilice para tomar un decisión sobre el estado interno de dicho objeto.

```
// Caso incorrecto - tomo una decisión en base al estado interno de otro obj.
for(Factura oFactura : cFacturas) {
    if(!oFactura.estaAnulada()) {
        Float totalFactura = oFactura.calcularTotal();
        System.out.println("Total de la factura " + totalFactura);
    } else {
        System.out.println("Total de la factura 0 (cero)");
    }
}
// Caso correcto
for(Factura oFactura : cFacturas) {
    System.out.println("Total de la factura " + oFactura.calcularTotal());
}
```

6 – Único punto de salida

Un programa debe tener un punto de entrada principal y bien definido desde el cual se inicia la ejecución y, idealmente, solo debe haber un punto de salida donde la ejecución del programa termina. Este principio promueve la claridad, la simplicidad y el mantenimiento del código.

Sugiere que solo debe haber un lugar en el código donde la ejecución del programa se detenga. Esto significa que las diferentes rutas o flujos de ejecución dentro del programa deben converger en un único punto de salida, en lugar de tener múltiples puntos de salida dispersos en el código. Esto facilita el seguimiento y la comprensión del flujo de ejecución del programa.

Tener un único punto de salida también ayuda en la gestión de recursos y la liberación de memoria. Por ejemplo, en lenguajes que utilizan el recolector de basura, el punto de salida único puede ser responsable de liberar los recursos utilizados por el programa antes de que finalice su ejecución.

6 – Único punto de salida

Establece que un programa debe tener un punto de entrada principal y un único punto de salida donde la ejecución del programa termina. Esto promueve la claridad, la simplicidad y el mantenimiento del código, y facilita la gestión de recursos y la comprensión del flujo de ejecución del programa.

```
public Integer prueba(Integer a) {
    // Verificar si "a" es par
    if (a % 2 == 0) {
        // Devolver el doble de "a"
        return a * 2;
    } else {
        // "a" no es par, devolver el mismo valor de "a"
        return a;
    }
}
```

Incorrecto

```
public Integer prueba(Integer a) {
    Integer resultado;

    // Verificar si "a" es par
    if (a % 2 == 0) {
        // Calcular el doble de "a"
        resultado = a * 2;
    } else {
        // "a" no es par, mantener el mismo valor de "a"
        resultado = a;
    }

    return resultado;
}
```

Correcto