

Tecnologías de Programación



Paradigma Orientado a Objetos

III – Reglas del buen diseño



Reglas del Buen Diseño

- COHESION – ACOPLAMIENTO
- PRINCIPIO DE RESPONSABILIDAD UNICA
- BREAK/CONTINUE
- UNICO PUNTO DE SALIDA
- OBJETOS BIEN FORMADOS
- HERENCIA / CLASE ABSTRACTA
- LEY DE DEMETER
- TELL DON'T ASK (TDA)

Toda *ley* o *directriz* de diseño debería ser eso una directriz.
Siempre hay excepciones a estas reglas.



Cohesión

La cohesión tiene que ver con la forma en la que agrupamos unidades de software en una unidad mayor. Por ejemplo, la forma en la que agrupamos funciones en una librería, o la forma en la que agrupamos atributos en una clase, o la forma en la que agrupamos clases en un paquete, etc...

Se suele decir que cuanto más cohesionados estén los elementos agrupados, mejor.

Acoplamiento

El término "**acoplamiento**" hace alusión al grado de dependencia que tienen dos unidades de software.

¿Qué es una unidad de software? Cualquier pieza de software que realice algún cometido. Por ejemplo: una función, un método, una clase, una librería, una aplicación, un componente, etc...



Principio de responsabilidad única

- Los METODOS solo deben hacer una cosa, deben hacerlo bien y deben ser lo único que hagan.
- También se puede aplicar este principio a una CLASE.



MÉTODOS

- El nombre de los métodos debe ser descriptivo, debe describir de la mejor manera su cometido.
- Cuanto más reducido y concreto sea el método, más sencillo será elegir su nombre.
- El número ideal de argumentos para un método es cero. Después uno y dos, siempre que sea posible evite la presencia de tres o más argumentos.

Utilización de Break/Continue en ciclos

```
Integer i = 0;  
// Un ciclo "infinito":  
while(true) {  
    i++;  
    Integer j = i * 27;  
    if(j == 1269) break; // Fuera del Ciclo  
    if(i % 10 != 0) continue; // Regreso al Inicio del Ciclo  
    System.out.println(i);  
}
```



Único punto de salida

```
//Este método devuelve el doble de a
//si a es par, y devuelve a tal cual si no lo es.

public Integer prueba(Integer a) {
    if (a % 2 == 0)
        return a * 2;
    else
        return a;
}
```




Objetos Bien Formados

- Cuando NO existe un tiempo entre la creación de un objeto y la adquisición de lo que ese objeto necesita para llevar a cabo su función (métodos)
- Un objeto bien formado se encuentra en estado consistente desde su creación.
- Hay que usar Constructores con las necesidades mínimas para el funcionamiento del objeto.



Herencia

- Siempre que codifiquemos una gerarquía de herencia entre clases, las clases generales deben ser abstracta. Las únicas clases concretas son las especializadas o particulares.



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”



Demeter II

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

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



Demeter III

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:

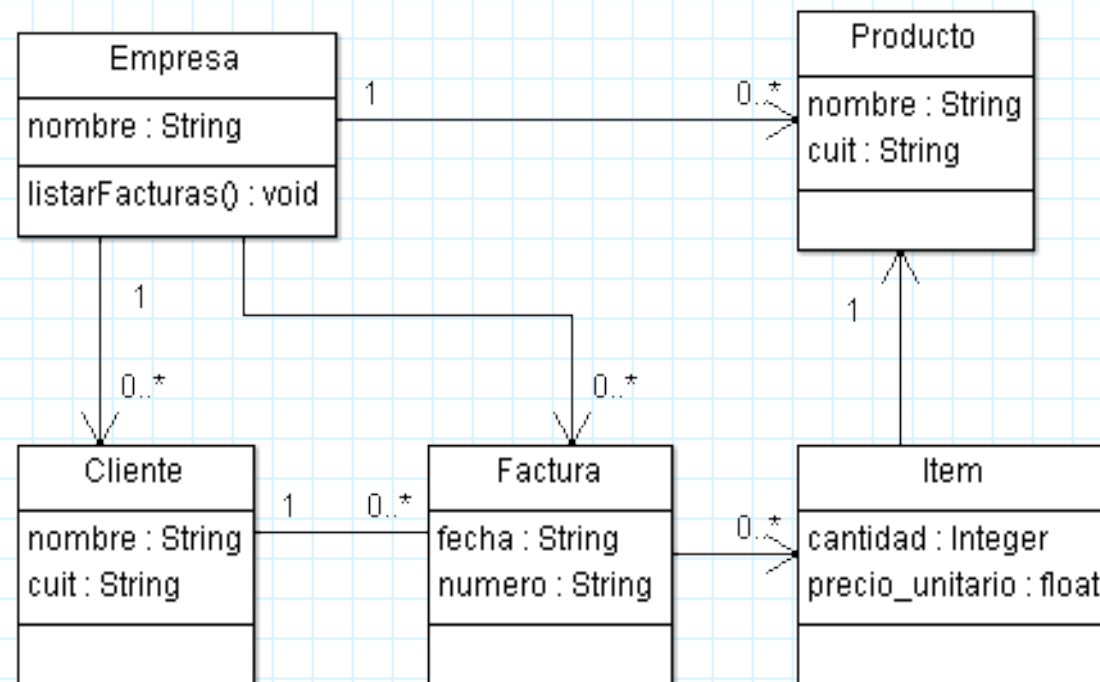
a.obtenerX().obtenerY().obtenerValor();

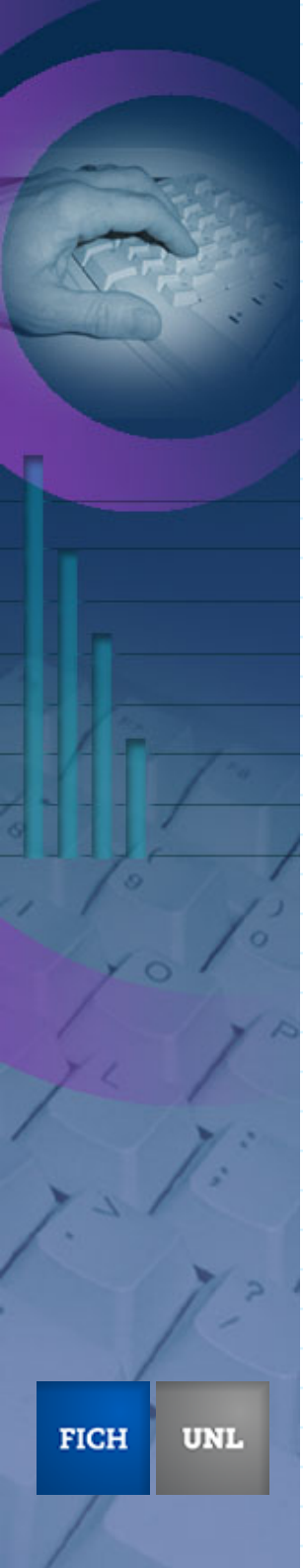
y sustituirlas por funciones que realicen dicha acción:

a.obtenerXYValor();

Demeter - Ejemplo

Tomamos el ejercicio 1 del TP 2.





```
public class Empresa {

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

    public void listarTotalFacturas() {

        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);
            }
        }
    }

    for(Factura oFactura : cFacturas) {

        if(!oFactura.estaAnulada()) {
            Float totalFactura = oFactura.calcularTotal();
            System.out.println("Total de la factura " + totalFactura);
        }
    }
}
```



Tell don't ask - TDA (Dile, no le preguntes)

Esta es una Regla del buen Diseño más restrictiva que Demeter.

Establece que en toda comunicación entre objetos, el emisor del mensaje debe decirle **que hacer** al receptor, no puede **pedirle** algún atributo para tomar una decisión sobre el estado interno del objeto y a continuación decirle que hacer.

Pasando en limpio, el único que puede tomar una decisión en base al **estado de alguno de sus atributos** es el objeto que posee dichos atributos.

Un objeto no puede tomar una decisión en base al estado de un atributo que no es suyo.



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 una 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());  
}
```