

The background of the slide features a light gray pattern of overlapping hexagons and circles, some of which contain smaller hexagons, creating a molecular or network-like aesthetic. The slide is framed by a dark green border at the top and bottom, with the corners of the white content area being clipped at a 45-degree angle.

Principios SOLID-DRY-KISS

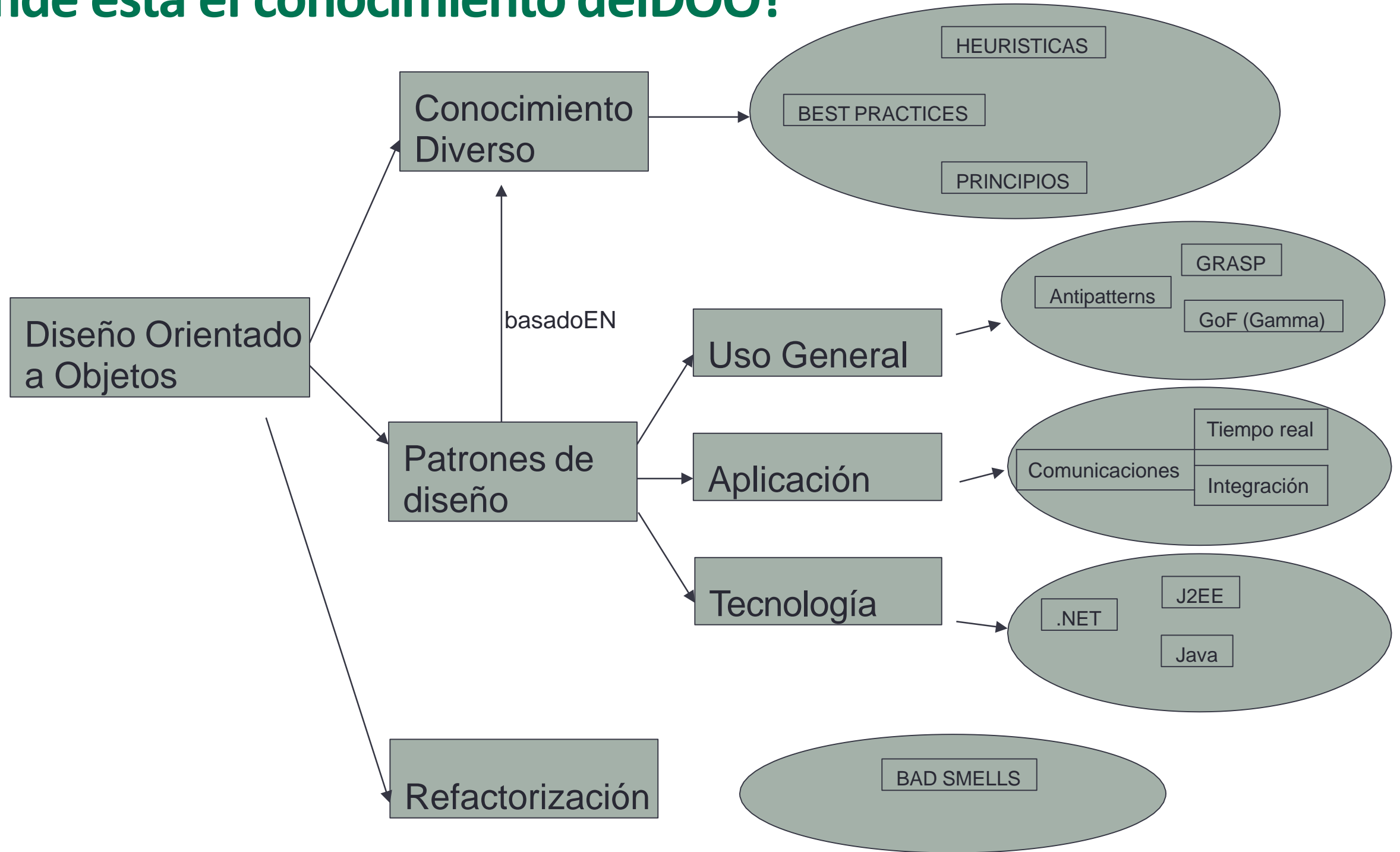
Introducción

“A branch of engineering must take several basic steps in order to become an established profession, highlighting understanding the nature of its knowledge” [1]

Una rama de la ingeniería debe seguir varios pasos básicos para convertirse en una profesión consolidada, destacando la comprensión de la naturaleza de su conocimiento.

¿Cuál es y dónde está el enorme conocimiento práctico en el Diseño Orientado a Objetos (DOO), basado en la experiencia acumulada en el desarrollo de sistemas software durante todos estos años y aplicable en la mayor parte de proyectos?

¿Dónde está el conocimiento delDOO?



Principios SOLID

- **Single-Responsability Principle (SRP)**
- **Open-Close Principle (OCP)**
- **Liskov Substitution Principle (LSP)**
- **Interface Segregation Principle (ISP)**
- **Dependency Inversion Principle (DIP)**

EL cambio

"Los sistemas software cambian durante su ciclo de vida"

- Tanto los buenos como los malos diseños se ven afectados por este requisito.
- Los buenos diseños son estables.

*No importa dónde estés trabajando, qué estés construyendo o qué lenguaje de programación estés usando, siempre habrá una constante que va a estar con nosotros: **el cambio**.*

No importa lo bien diseñes tu aplicación, a medida que pase el tiempo una aplicación deberá crecer y **cambiar** o inevitablemente irá quedando obsoleta.

Lo único que es constante es **el cambio**.

Open-Closed Principle (OCP)

• “

”Los sistemas software cambian durante su ciclo de vida”

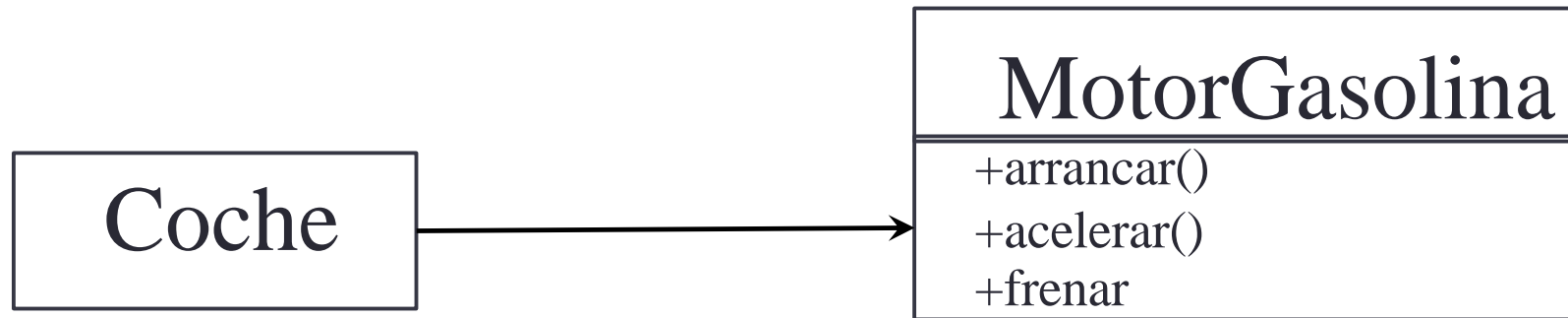
- Tanto los buenos como los malos diseños se ven afectados por este requisito.
- Los buenos diseños son estables.

Las entidades Software deben ser abiertas para extensiones, pero cerradas para modificaciones. [2]



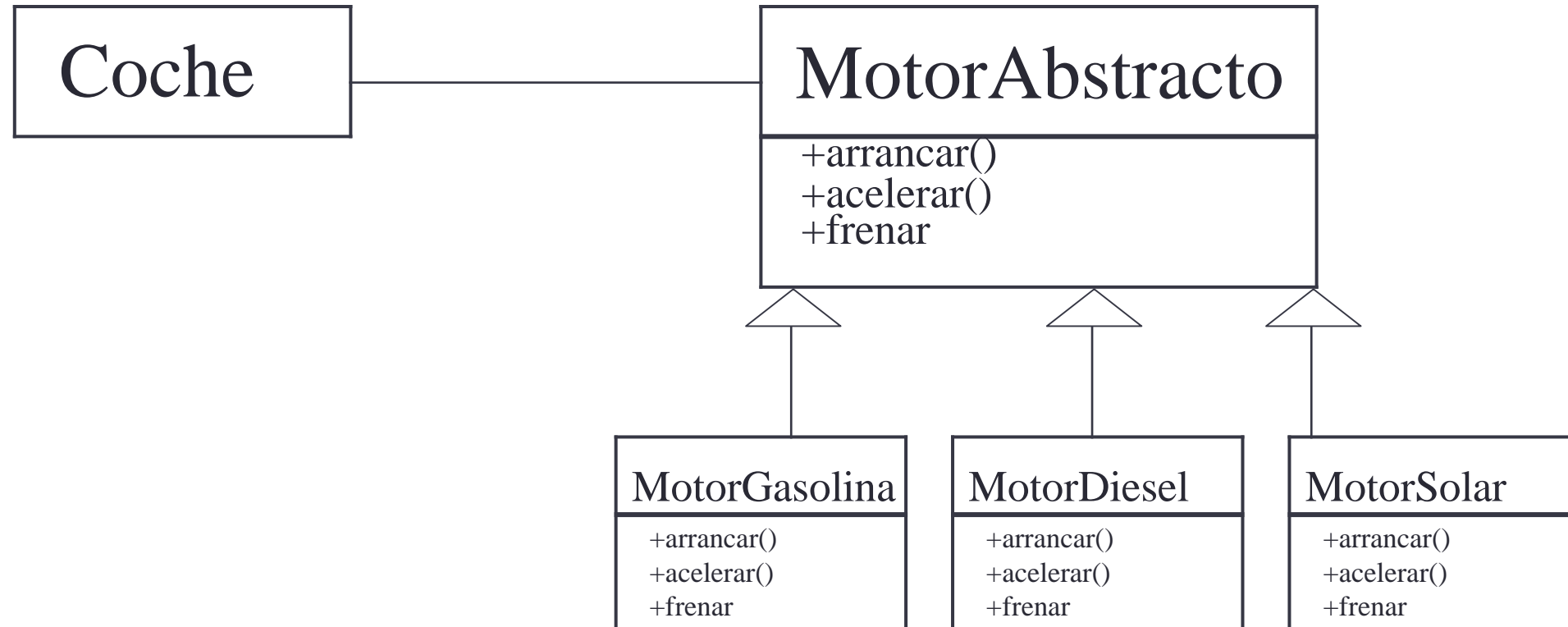
- | Abierto para extensiones
 - 🕷 El comportamiento del módulo puede ser extendido
- | Cerrado para modificaciones
 - 🕷 El código fuente del módulo no debe ser modificado
- *Los módulos deben diseñarse para que puedan ser extendidos sin tener que ser modificados*

Abrir la puerta...



- ¿Cómo hacer que un **Coche** utilice un **MotorGasol** o **MotorSolar**?
- Únicamente cambiando la clase **Coche**!
 - ...en este diseño

... ¡Pero mantenerla cerrada!



- Una clase no debe depender de una clase concreta
- Debe depender de una clase **abstracta**... o una **interfaz**
- ...utilizando **dependencias polimórficas** (llamadas)

Open-Closed Principle (OCP)

- La dependencia “uno a uno” se transforma en una dependencia de “uno a muchos”.
- Programa para la interfaz, no para la implementación.
- Establece una relación a una clase abstracta (o interfaz) sólo en los puntos de variabilidad del programa.

```
graph LR; A[Clase A] --> B[Clase B]
```

Clase A → Clase B

Diseño cerrado/cerrado

```
graph TD; A[Clase A] --> B[Clase abs B]; B1[Clase B1] --|> B; B2[Clase B2] --|> B
```

Clase A → Clase abs B

Clase B1

Clase B2

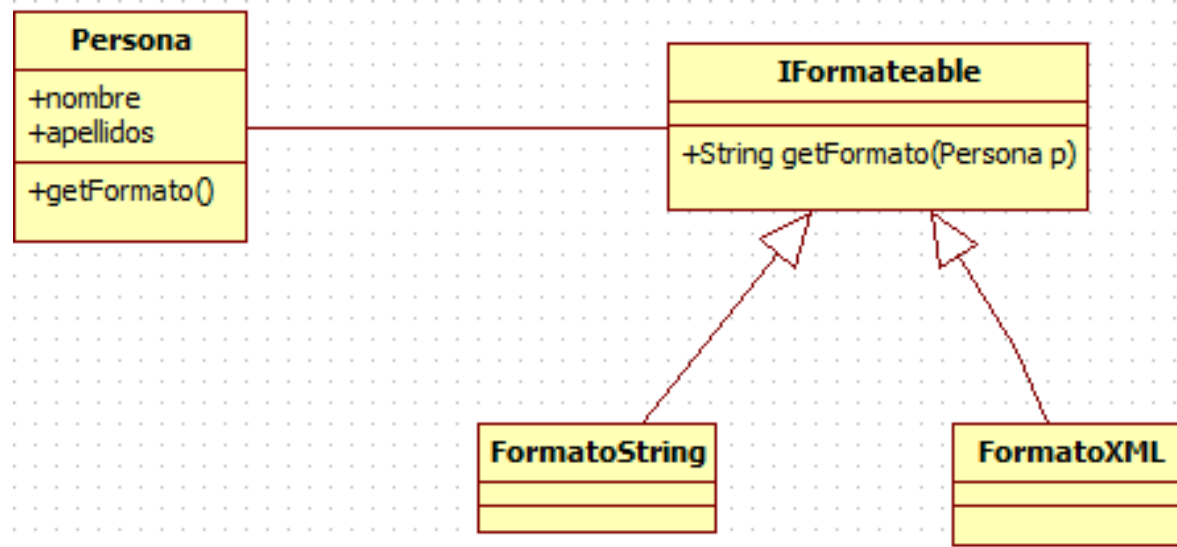
Diseño abierto/cerrado

Ejemplo

```
public class Persona {  
    String nombre;  
    String apellidos;  
  
    public String getPersona(){  
        String s;  
        s=nombre+" "+apellidos;  
        return s;  
    }  
}
```

¿Qué sucede si la representación de la Persona pudiese cambiar a HTML, XML u otro formato?

Solución OCP Composición



```
public interface IFormateable {  
    public String getFormato(Persona p);  
}
```

```
public class Persona {  
    String nombre;  
    String apellidos;  
  
    public String getFormato(IFormateable iformat){  
        String s=iformat.getFormato(this);  
    }  
}
```

```
public class FormatoString implements IFormateable{  
    public String getFormato(Persona p){  
        return p.nombre+" "+p.apellidos;  
    }  
}
```

```
public class FormatoXML implements IFormateable{  
    public String getFormato(Persona p){  
        String xml="<xml> <nombre> " + p.nombre+"</nombre>";  
        xml=xml+"<apellido>"+p.apellidos+"</apellido></xml>";  
        return xml;  
    }  
}
```

El patrón OCP Composición

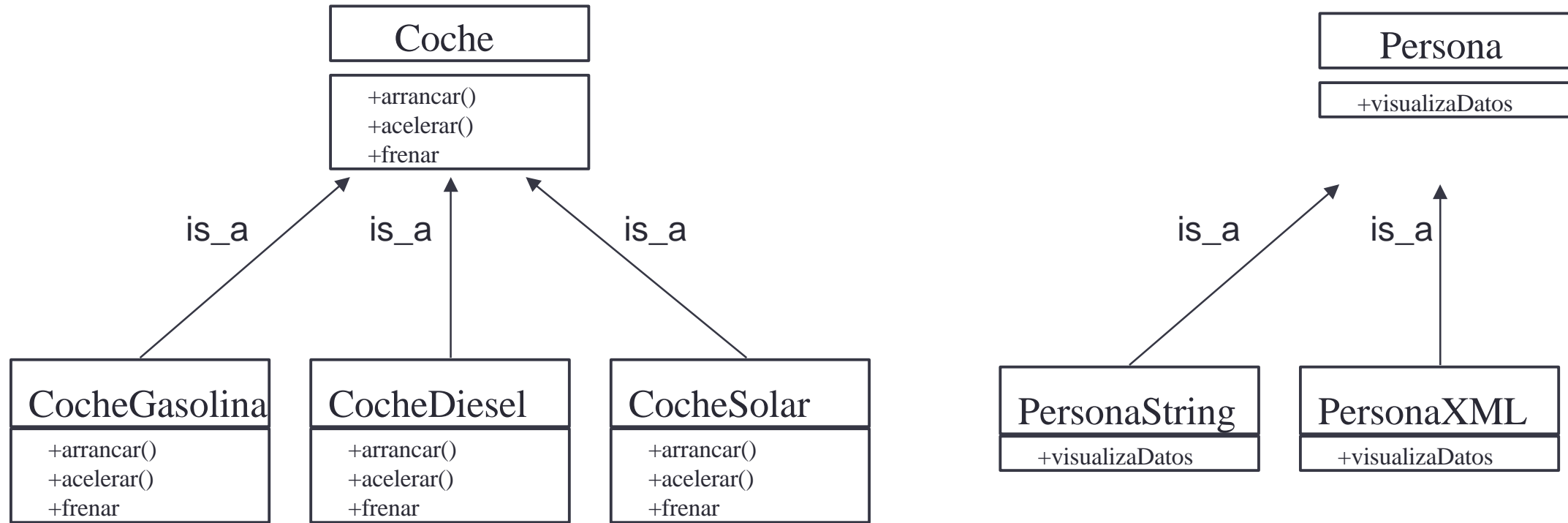
```
public final class ClosedClass {  
    private IMyExtension myExtension;  
    public ClosedClass(IMyExtension myExtension) {  
        this.myExtension = myExtension;  
    }  
    // métodos que usan el objeto de tipo IMyExtension  
    public void doMethod() {  
        myExtension.doStuff();  
    }  
}
```

```
public interface IMyExtension {  
    public void doStuff();  
}
```

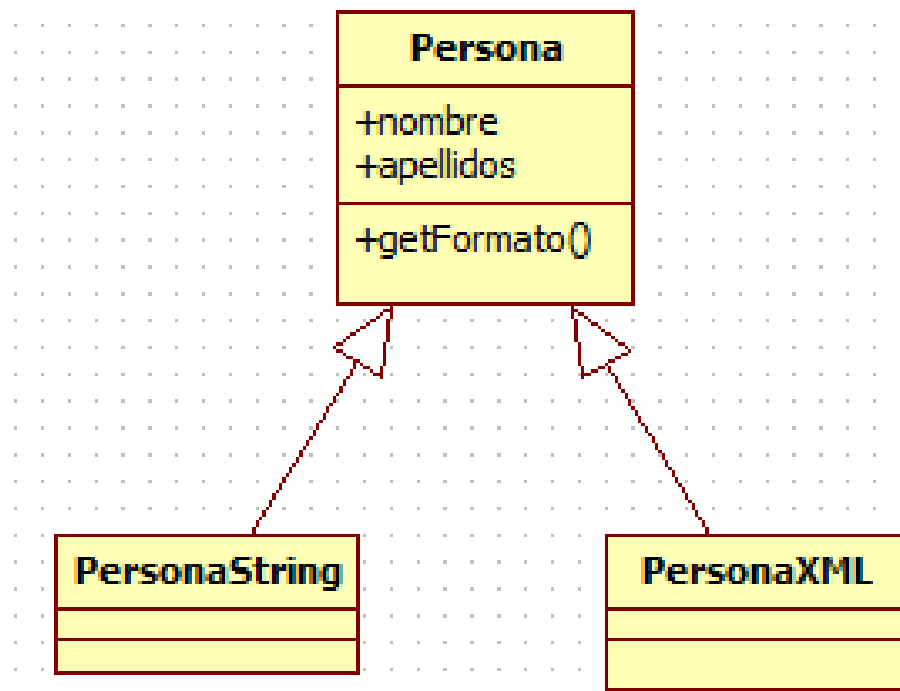
Ejemplo Coche OCP Composición

```
public final class Coche{  
    private IMotor miMotor;  
    public Coche (IMotor myExtension) {  
        this.miMotor= myExtension;  
    }  
    // métodos que usan el objeto de tipo IMotor  
    public void arrancar() {  
        miMotor.arrancar();  
    }  
}
```

Otra alternativa..... herencia



Solución OCP Herencia

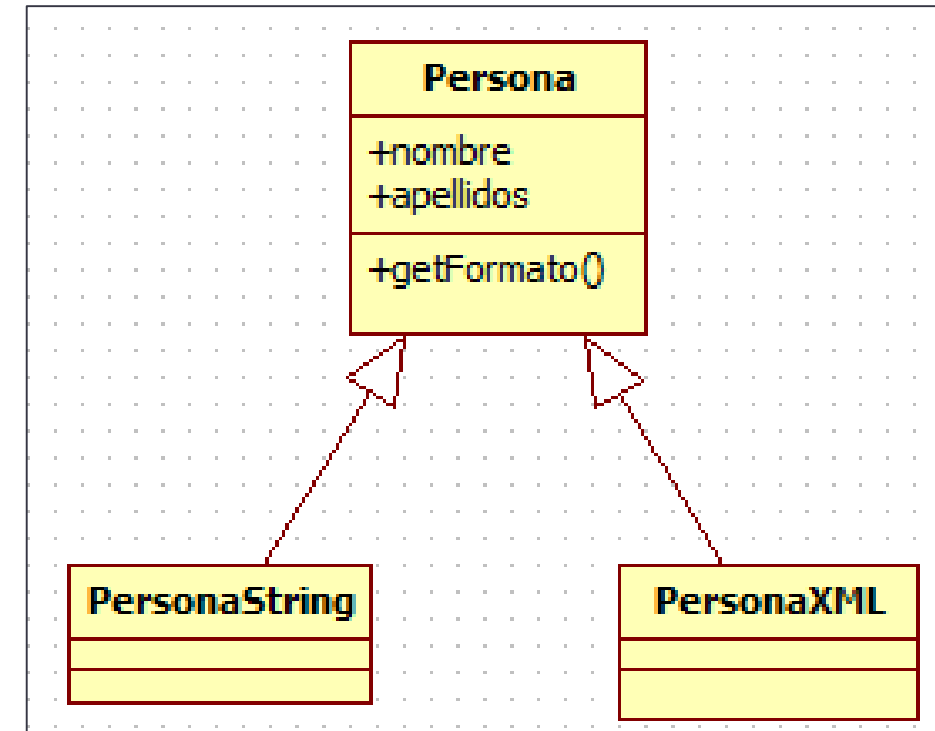
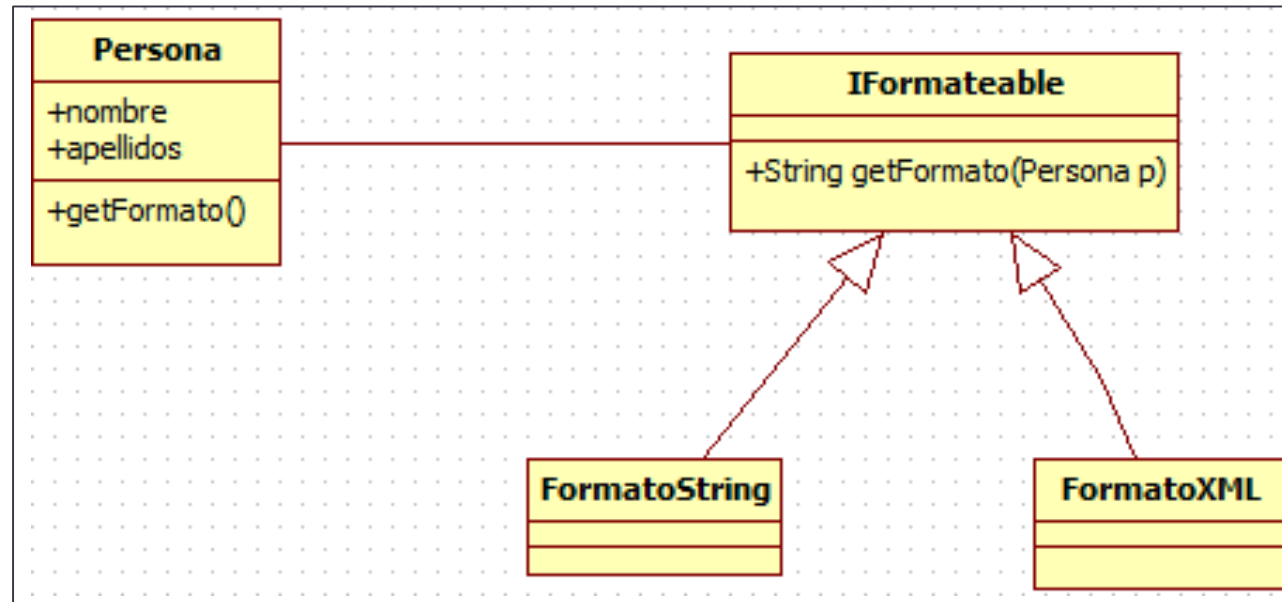


```
public abstract class Persona {
    String nombre;
    String apellidos;

    public abstract String getFormato();
}
```

```
public class PersonaString extends Persona{
    public String getFormato(){
        String s;
        s=nombre+" "+apellidos;
        return s;
    }
}
```

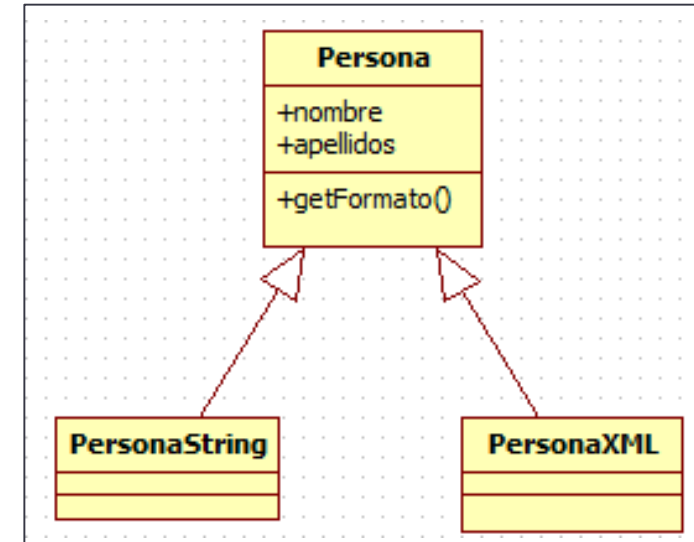
Herencia vs. composición



Herencia vs. composición

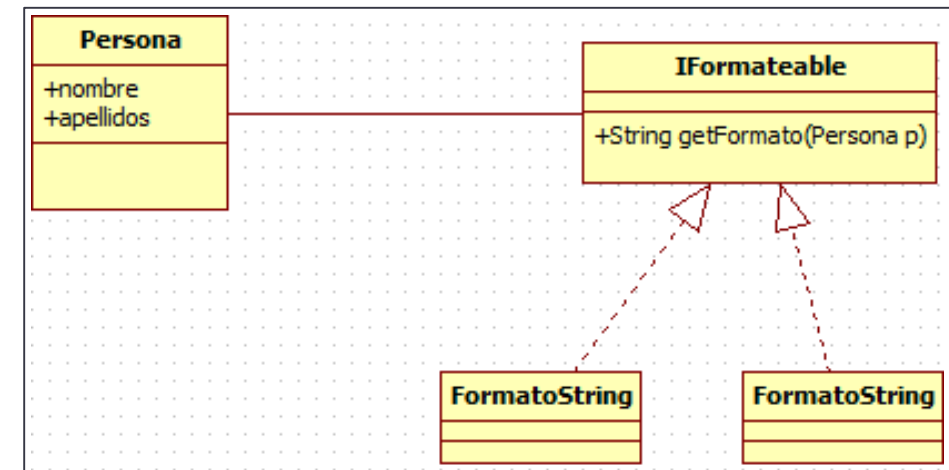
Herencia

- Extensión de **Caja-blanca (herencia)**:
 - No se puede cambiar el comportamiento heredado en run-time (enlace estático)
 - La clase padre define en parte la representación física de las subclases (La herencia rompe la encapsulación)
 - No se puede reutilizar únicamente la subclase.



Herencia vs. composición

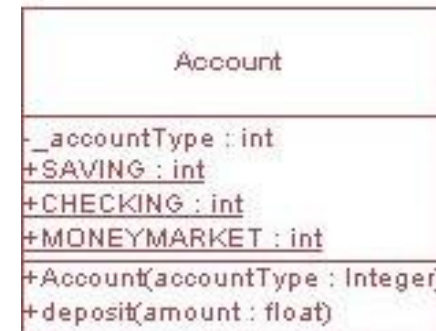
Composición



- **Extensión de Caja-negra (composición) :**
 - La composición se define dinámicamente en *run-time* a través de la adquisición de referencias.
 - Requiere que los objetos tengan interfaces bien definidas.
 - No son visibles los detalles de los objetos.
 - La composición ayuda a mantener cada clase centrada en una tarea.
 - Más objetos, menos complejos.

Ejemplo Composición-Herencia

- Un banco gestiona cuentasCorrientes (Account) (operaciones ingresar, retirar).
- Tenemos 3 tipos de cuentas (credito, debito, monedero).
- Tenemos 3 tipos de clientes (joven, adulto, dorada)



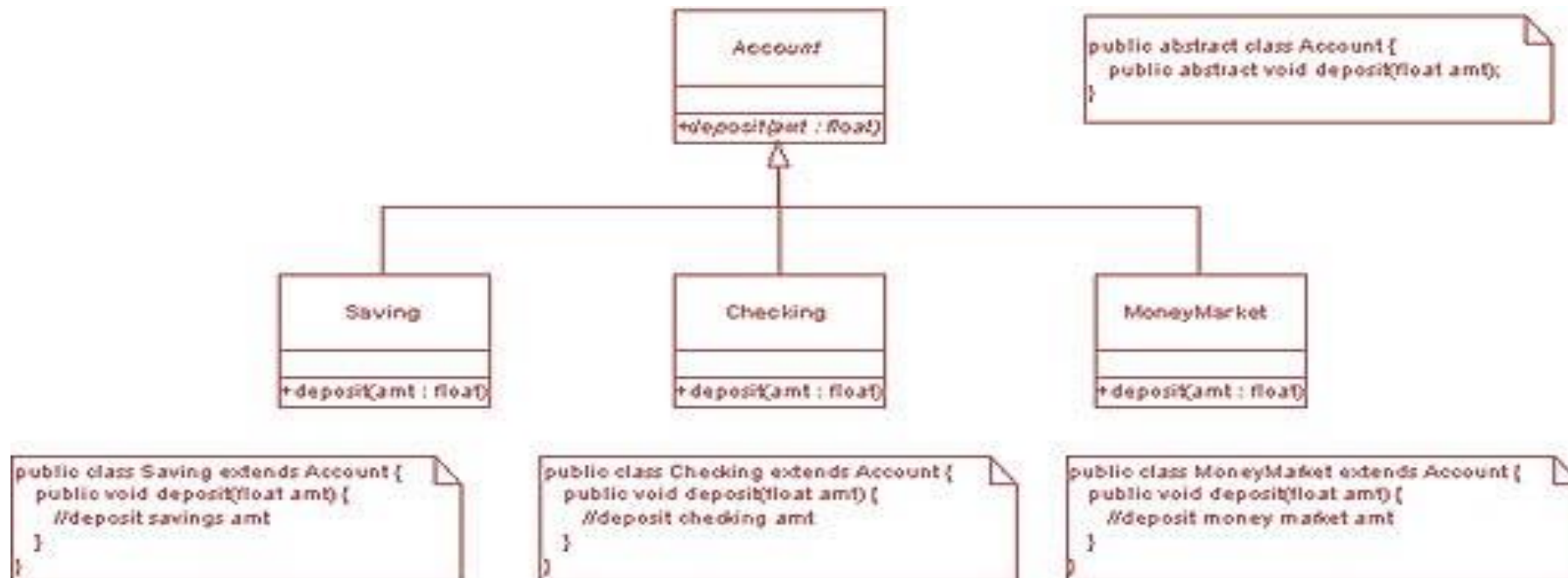
```
public class Account {

    private int _accountType;
    public static final int SAVING = 1;
    public static final int CHECKING = 2;
    public static final int MONEYMARKET = 3

    public Account(int accountType){
        this._accountType = accountType;
    }

    public float calculatePayO {
        if (this._empType == SAVING){
            //Saving deposit.
        } else if (this._empType == CHECKING){
            //Checking deposit.
        } else if (this._empType == MONEYMARKET){
            //Money Market deposit.
        }
    }
}
```

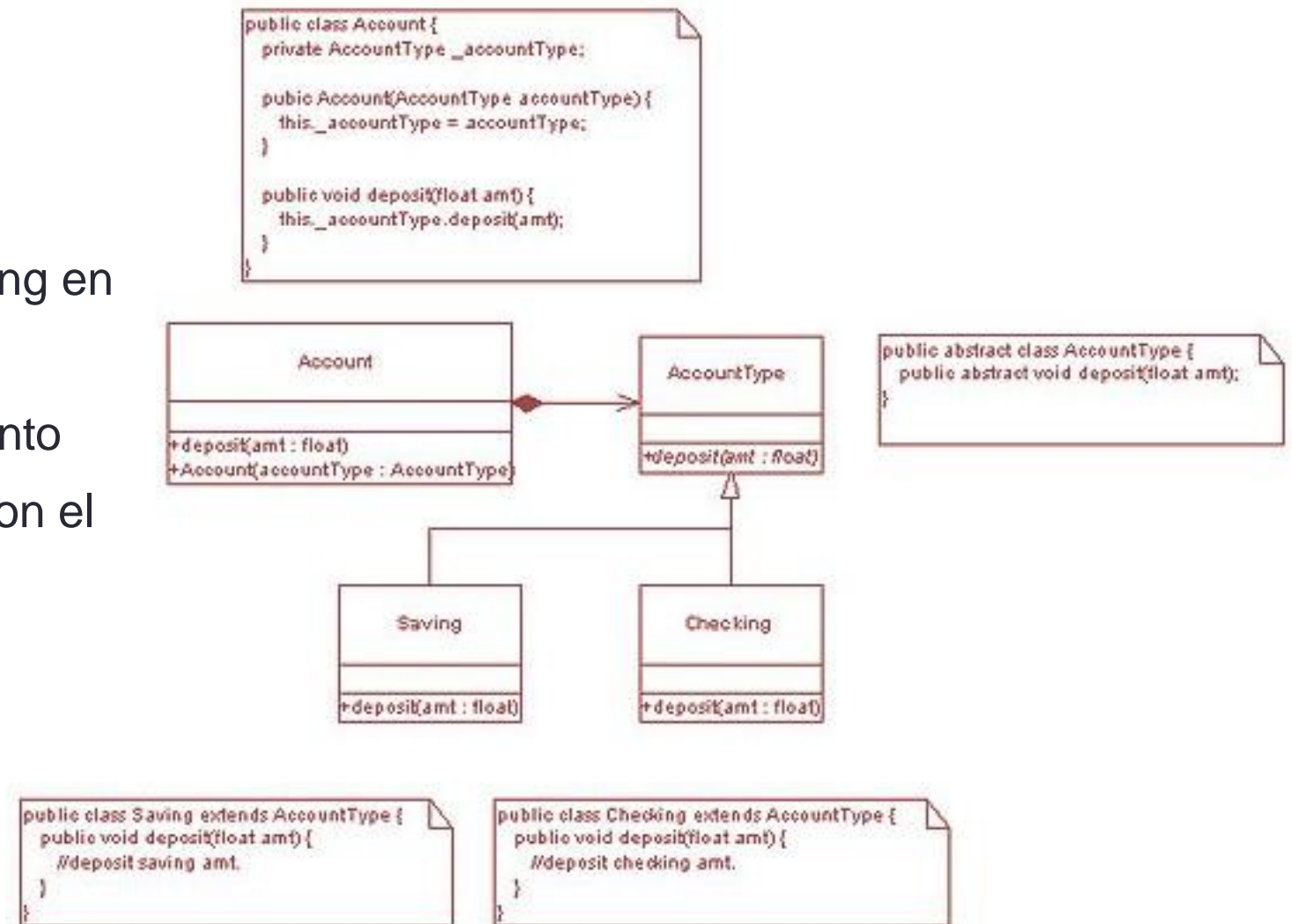
Solución herencia



- ¿Cumple el principio de OCP?
- ¿Puedo reutilizar el comportamiento de Saving en otra clase?
- ¿Qué sucede si añadimos otro comportamiento (`int discount()`) que no depende del criterio con el que se ha creado la jerarquía (p.ej `tipo=joven,normal,mas65`)?

Solución composición

- ¿Cumple el principio de OCP?
- ¿Puedo reutilizar el comportamiento de Saving en otra clase?
- ¿Qué sucede si añadimos otro comportamiento (int discount()) que no depende del criterio con el que se ha creado la jerarquía (p.ej tipo=joven,normal,mas65)?



Cambios en las cuentas existentes

- ¿Qué sucede si...
 - ¿tenemos cuentas diferentes para miembros del banco y para los demás?
 - ¿la cuenta pasa de débito a crédito?
 - ¿la cuenta pasa de joven a adulto?

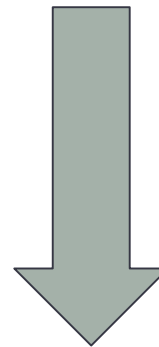
Herencia vs. composición

“Una entidad debe ser abierta para su extensión y parametrización, pero cerrada para su modificación” [3]

- Los patrones de diseño se han definido para soportar la variabilidad de manera flexible.
- Los **patrones de diseño** hacen uso extensivo de la **composición**.
- Usar la **composición** para aquellos puntos de **variabilidad del sistema**.

Del OCP al Siguiente principio

Ningún programa significativo puede ser 100% cerrado. [4]



PRINCIPIO DE RESPONSABILIDAD UNICA (SRP, Single Responsibility Principle)

Principio: Single Responsibility Principle

“Una clase debe tener una única razón para cambiar” [5]

También conocido como High Cohesion (GRASP)

Cohesión es la medida en la que los comportamientos del objeto están relacionados. Un elemento con baja cohesión realiza muchas tareas de diversa índole. La alta cohesión es una característica deseable de los diseños OO.



Imagen bajo licencia CC-BY-2.0.

<https://www.flickr.com/photos/pennuja/5364128968/>

[5] R. Martin. The Single Responsibility Principle.

https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view

Hombre orquesta vs. trompetista



Imagen bajo licencia CC-BY-SA 2.0.

<https://www.flickr.com/photos/84773840@N00/8489287609/>



Imagen bajo licencia CC-BY-SA 2.0.

<https://www.flickr.com/photos/jikatu/8356118191/>

- El trompetista es un especialista. (Cohesión)
- El trompetista puede trabajar fácilmente en varias orquestas, el hombre orquesta es la orquesta. (Reusabilidad)
- Siempre esperamos lo mismo de él, que toque la trompeta. (Reducción de Side effects)
- Si en las interpretaciones de la orquesta algo está mal con la trompeta le pedimos que mejore o lo cambiamos, ¿al hombre orquesta cómo lo cambiamos? Si se acaba la orquesta se acaba el negocio. (Mantenibilidad)

La gran orquesta



Imagen bajo licencia CC-BY-SA 2.0.

<https://www.flickr.com/photos/buenosairesprensa/7983428800/>

“Una orquesta sinfónica, igual que un sistema, se compone de un grupo cohesionado de músicos especializados en un sólo instrumento y mas aún, responsables de tocar una sola partitura de cada pieza musical. Un solo trompetista no puede ejecutar una sinfonía, el hombre orquesta tampoco o con mucha dificultad lo lograría, entonces lo que debemos procurar es tener clases con las funciones necesarias, y que estas a su vez sean "pequeñas", con una única razón para cambiar en la medida de lo posible (no siempre es posible tener clases con una sola razón para cambiar) y orquestadas (es decir, diseñadas) para componer un gran sistema, para interpretar una gran sinfonía.”

Principio: Single Responsibility Principle

- Cada responsabilidad debe residir en una clase separada, ya que cada responsabilidad es una “fuente” de cambio.
- Una clase debe tener un solo motivo de cambio.
- Las clases y métodos que siguen el principio de SRP son más pequeñas y fáciles de entender y mantener. Otra ventaja es que los métodos son más fáciles de testear.

Ejemplo

Necesitamos una clase que descargue un fichero (puede estar en formato csv/json/xml), parsee el fichero y finalmente actualice el contenido en una base de datos o un fichero. Una implementación podría ser:

```
public class Tarea {  
    public void descargarFichero(String ubicacion) {  
        // descarga un fichero  
    }  
    public void parsearFichero(String fichero) {  
        // parsear el contenido del fichero fichero a XML  
    }  
    public void guardarFichero(String fichero) {  
        // almacenar el fichero en la BD  
    }  
}
```

Cuestiones

```
public class Tarea {  
    public void descargarArchivo(String ubicacion) {  
        // descarga un archivo  
    }  
    public void parsearArchivo(String fichero) {  
        // parsear el contenido del archivo fichero a XML  
    }  
    public void guardarArchivo(String fichero) {  
        // almacenar el archivo en la BD  
    }  
}
```

Cuestiones:

- ¿Qué sucede con la reusabilidad de “descargar”, “parsear” o “guardar”?
- ¿Cómo modificarías las clases, de forma que sigas manteniendo en la clase Tarea todas las funcionalidades?

¿Qué sucede con la reusabilidad?

Solución: Crear una clase por responsabilidad

```
public class Downloader {  
    public void descargarFichero(String ubicacion) {  
        // descarga un fichero  
    }  
}
```

```
public class Parser {  
    public void parsearFichero(String fichero) {  
        // parsea el contenido del fichero a XML  
    }  
}
```

```
public class Storer {  
    public void guardarFichero(String fichero) {  
        // almacena el fichero en la BD  
    }  
}
```


Solución: Manteniendo un objeto para cada responsabilidad.

```
public class Tarea {  
    Downloader downloader = new Downloader();  
    Parser parser = new Parser();  
    Storer storer = new Storer();  
    public void descargarFichero(String ubicacion) {  
        // descarga un fichero  
        downloader.descargarFichero(ubicación);  
    }  
    public void parsearFichero(String fichero) {  
        // parsea el contenido del fichero a XML  
        parser.parsearFichero(fichero);  
    }  
    public void guardarFichero(String fichero) {  
        // almacena el fichero en la BD  
        storer.guardarFichero(fichero);  
    }  
}
```


Liskov Substitution Principle (LSP)

- Las claves del OCP: Abstracción y Polimorfismo
 - 🕸 Implementado por herencia
 - 🕸 ¿Cómo podemos medir la calidad de la herencia?

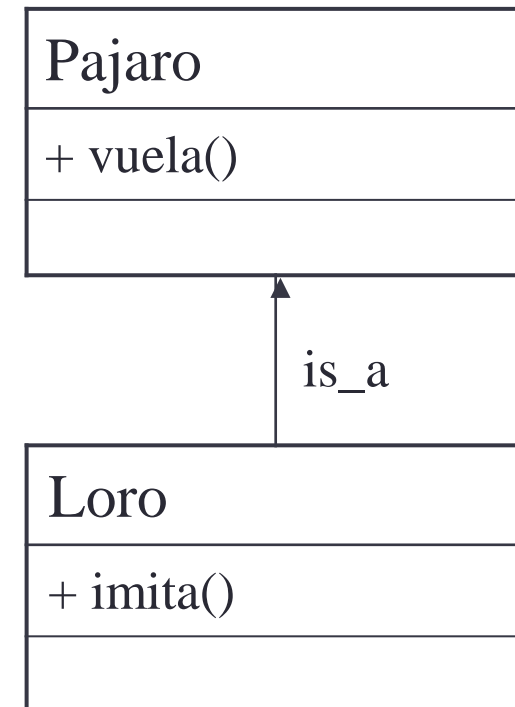
“La herencia debe garantizar que, cualquier propiedad probada para cualquier objeto de la superclase, debe ser válida para cualquier objeto de las subclases” [6]

Los métodos que usan punteros (o referencias) a otras clases, deben ser capaces de utilizar objetos de clases derivadas (de la clase) sin saberlo.

La herencia *Parece* simple

```
class Pajaro {                // tiene plumas, alas, .
    public void vuela();      // Los pajaros pueden volar
};

class Loro extends Pajaro { // Un loro es un pajaro
    public void imita();      // Puede repetir palabras..
};
// ...
Loro miMascota=new Loro();
miMascota.imita();           // Mi mascota siendo loro puede imitar()
miMascota.vuela();          // mi mascota "es un" pajaro, puede volar
```



Los pingüinos no vuelan

```
class Pinguino extends Pajaro {  
    public void vuela() {  
        new Exception("no puedo volar!");  
    }  
};
```



```
void vueloComoPajaro (Pajaro pajaro) {  
    pajaro.vuela();    // OK si loro.  
    // Que pasa si pájaro es pingüino...OOOPS!!  
}
```

- No modela: “*Los pingüinos no pueden volar*”
- Modela “*Los pingüinos pueden volar, pero si lo intenta es un error*”
- Run-time error si intentan volar → no deseable
- *Piensa acerca de la sustituibilidad – Falla el principio de Liskov*

Diseño por contrato

- Comportamiento esperado de un método:
 - **Requisitos esperados** (Precondiciones)
 - **Promesas ofrecidas** (Postcondiciones)

“Cuando redefines un método en una subclase, sólo puedes reemplazar su precondición por una más débil, y la postcondición por una más estricta” [7]

⇒ Los métodos de las subclases no deben exigir más y no prometer menos

```
int Base::f(int x);  
// REQUIERE: x es impar  
// PROMETE: return par int
```

```
int Derived::f(int x);  
// REQUIERE: x es entero  
// PROMETE: par int >50
```

Diseño por contrato

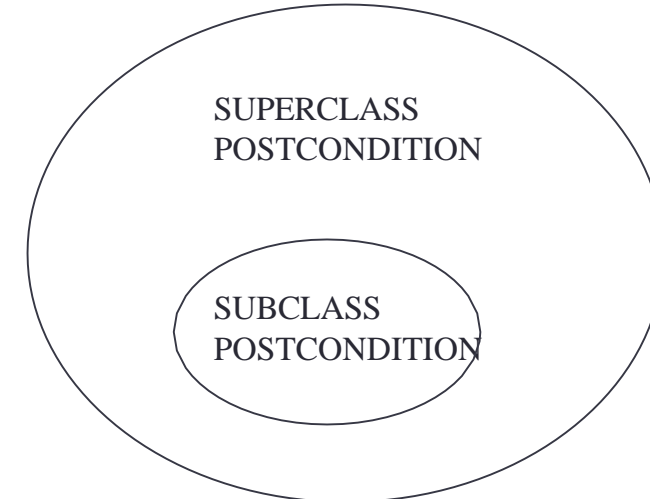
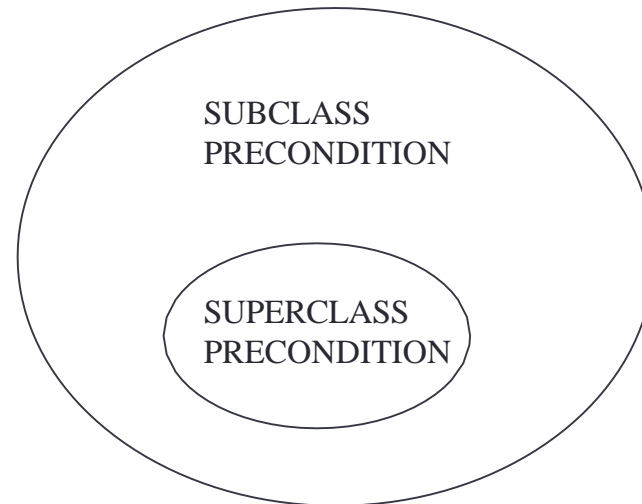
- La subclase conoce mejor la solución que la superclase (más específica). Por tanto al usuario:
 - Le pedirá igual o menos (específico)
 - Le dará igual o más (específico)

Base

```
int f(int x);  
// REQUIERE: x es impar  
// PROMETE: return par
```

Derived

```
int f(int x);  
// REQUIRE: x is int  
// PROMISE: return par>50
```



Diseño por contrato

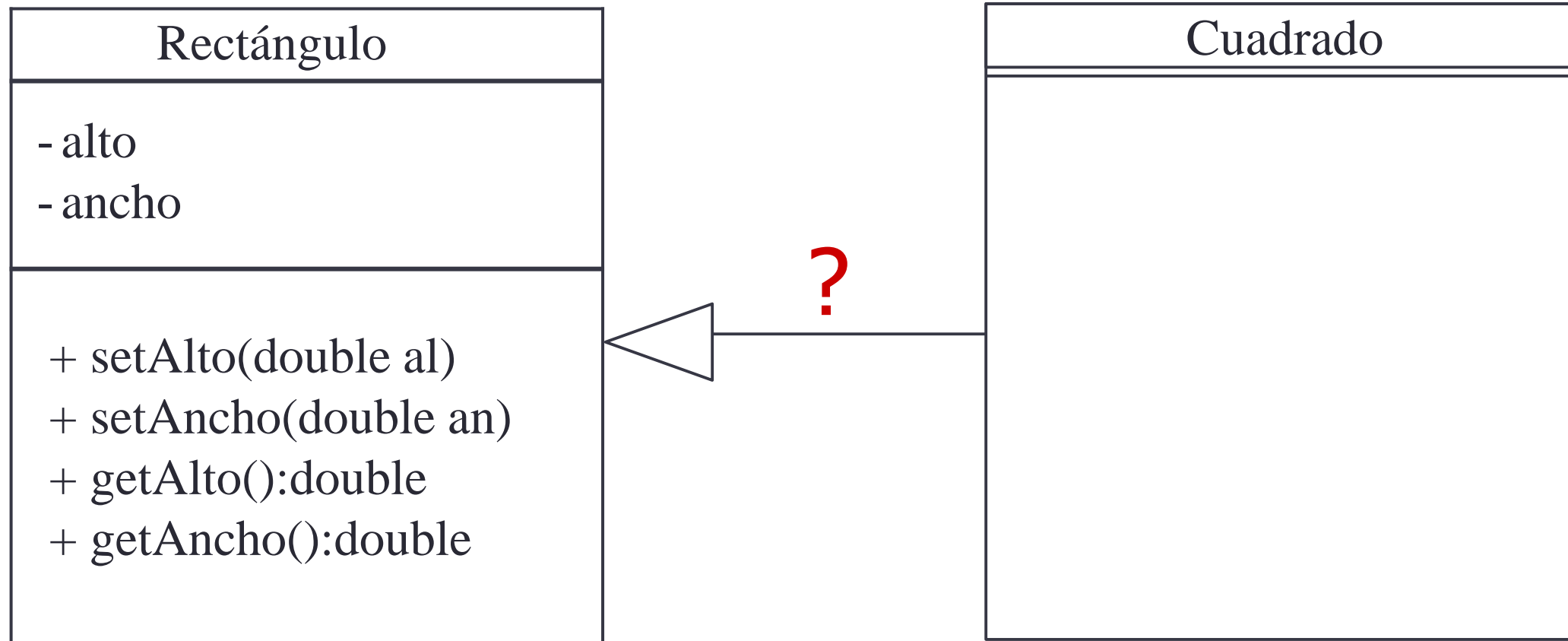
- Dada la clase Base

```
class Base {  
    // REQUIERE: x es impar  
    // PROMETE: return par  
    public void f (int x) ;  
}
```

- Cualquier subclase que especialice Base tiene que tener una precondition menos estricta y una postcondición más acotada.

```
class ClaseInvocadora{  
    public int metodo(Base b){  
        int r=b.f(8); // el valor enviado puede que no sea impar  
        // r tiene que ser par sino la subclase está mal diseñada  
    }  
}
```

¿Cuadrado IS-A Rectángulo?



¿Debería heredar Cuadrado de Rectángulo?

La respuesta es...

- Sobrescribir `setAlto` y `setAncho`

```
class Cuadrado extends Rectángulo {  
    public void setAlto (double al){  
        ancho=al;  
        alto=al;  
    }  
}
```

El problema

```
void calcArea(Rectángulo r) {  
    r.setAncho(5); r.setAlto(4);  
    // ¿Cuánto es el área?  
}
```

- 20!

... ¿Estás seguro? ;-)

Las subclases pueden extender a las superclases sin modificar el **comportamiento**

LSP tiene que ver con la semántica y el reemplazo

- Debe quedar **claramente documentado** el significado y propósito de cada clase y método.
 - La falta de comprensión induce “defacto” a la violación del LSP
- El reemplazo es crucial
 - Siempre que cualquier clase sea referenciada en cualquier código del sistema, cualquier subclase existente o futura debe ser 100% reemplazable.
 - Porque, más pronto que tarde, alguien **sustituirá** la subclase.
 - Es casi inevitable.

¿Es un Rectángulo un Cuadrado?

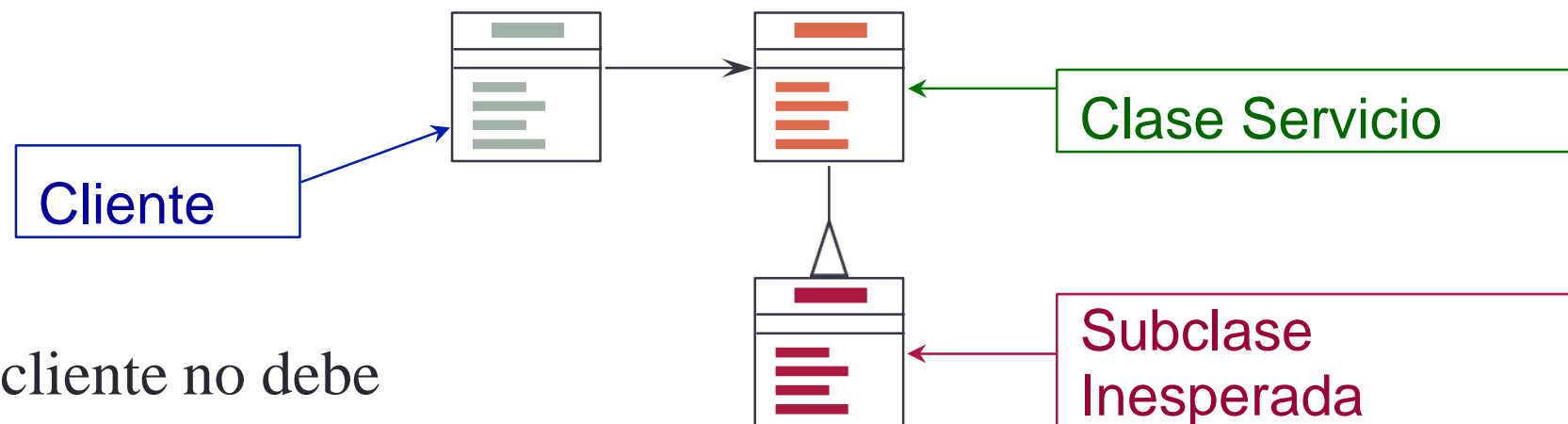
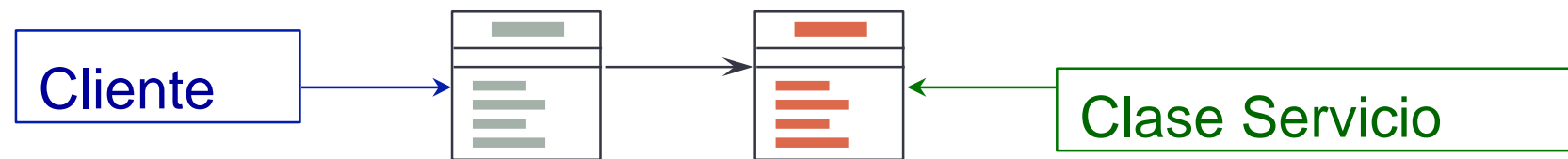
Un Cuadrado podría ser un Rectángulo, pero un objeto Cuadrado no es un objeto Rectángulo. ¿Por qué?



El comportamiento de un objeto Cuadrado no es consistente con el comportamiento de un objeto Rectángulo. Y en el software, el comportamiento es una parte esencial.

Liskov y elreemplazamiento

- Los métodos de una clase (Clase Servicio) invocados por un cliente.
 - Pueden ser substituidos por una subclase **sin afectar** al cliente que le invoca.

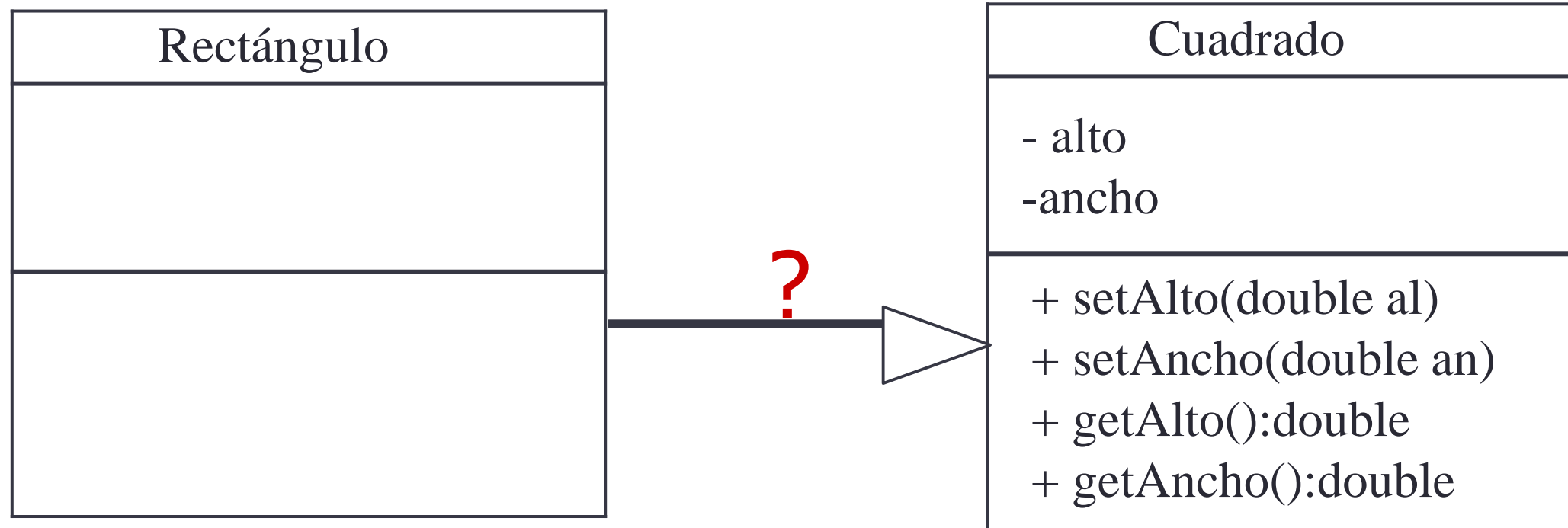


El cliente no debe
conocer las subclases

Heurístico obtenido del LSP

Es incorrecto que una subclase, sobrescriba un método con un método NOP(Nothing Operation)

- **Solución 1:** Relación de herencia inversa
 - ¿Es un Rectángulo un Cuadrado?



POSIBLES SOLUCIONES

“Extraer a otra clase padre las características comunes y hacer que la antigua clase padre y su hija hereden de ella” (*)

```
1  public interface IRectangle {  
2      int getWidth();  
3      int getHeight();  
4      int calculateArea();  
5  }  
6  
7  public class Rectangle extends IRectangle {  
8      ...  
9  }  
10  
11 public class Square extends IRectangle {  
12     ...  
13 }
```

Posibles soluciones II

Podemos solventar esta situación simplemente usando **inmutabilidad**. Consiste en que una vez que se ha creado un objeto, el estado del mismo no puede volver a modificarse.

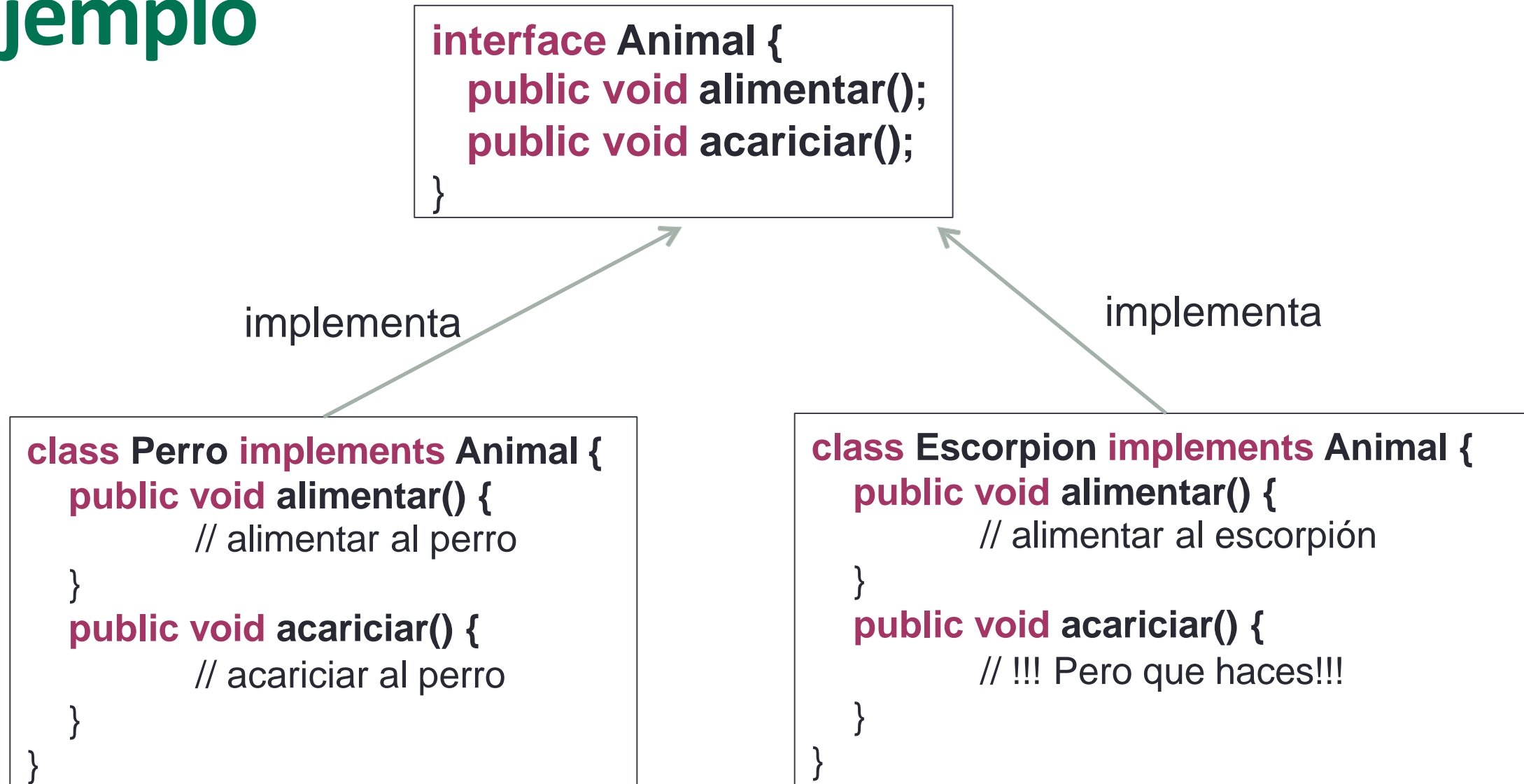
```
1 public class Rectangle {  
2  
3     public final int width;  
4     public final int height;  
5  
6     public Rectangle(int width, int height) {  
7         this.width = width;  
8         this.height = height;  
9     }  
10 }  
11  
12 public class Square extends Rectangle {  
13  
14     public Square(int side) {  
15         super(side, side);  
16     }  
17 }
```

Interface Segregation Principle (ISP)

“Los clientes no deben ser forzados a depender de interfaces que no usan” [9]

- *Varias interfaces específicas son mejores que una interfaz de propósito general.*
- **Consecuencia:**
 - La posibilidad de que cambie una interfaz es proporcional al número de métodos. Cuanto menor sea, mejor.
 - Interface pollution (*Bad Smell*, Fowler)

Ejemplo



Ejemplo

```
interface IMascota{  
    public void acariciar();  
}
```

```
interface Animal {  
    public void alimentar();  
}
```

implementa

implementa

```
class Perro implements Animal, IMascota {  
    public void alimentar() {  
        // alimentar al perro  
    }  
    public void acariciar() {  
        // acariciar al perro  
    }  
}
```

```
class Escorpion implements Animal {  
    public void alimentar() {  
        // alimentar al escorpión  
    }  
}
```

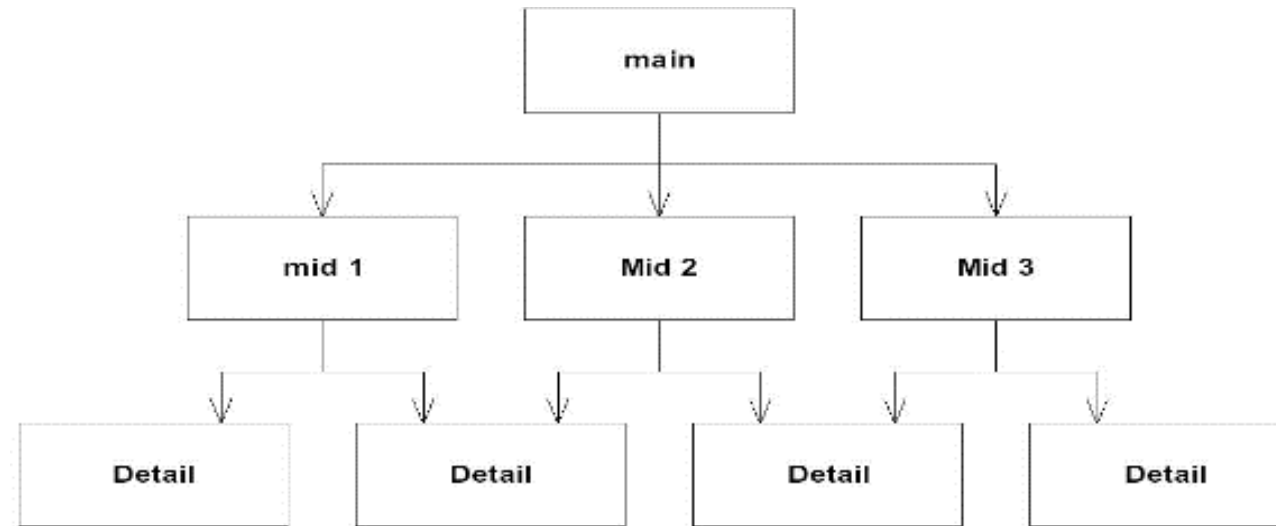
Dependency Inversion Principle (DIP)

- I. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. La dependencia debe basarse en abstracciones.
- II. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones

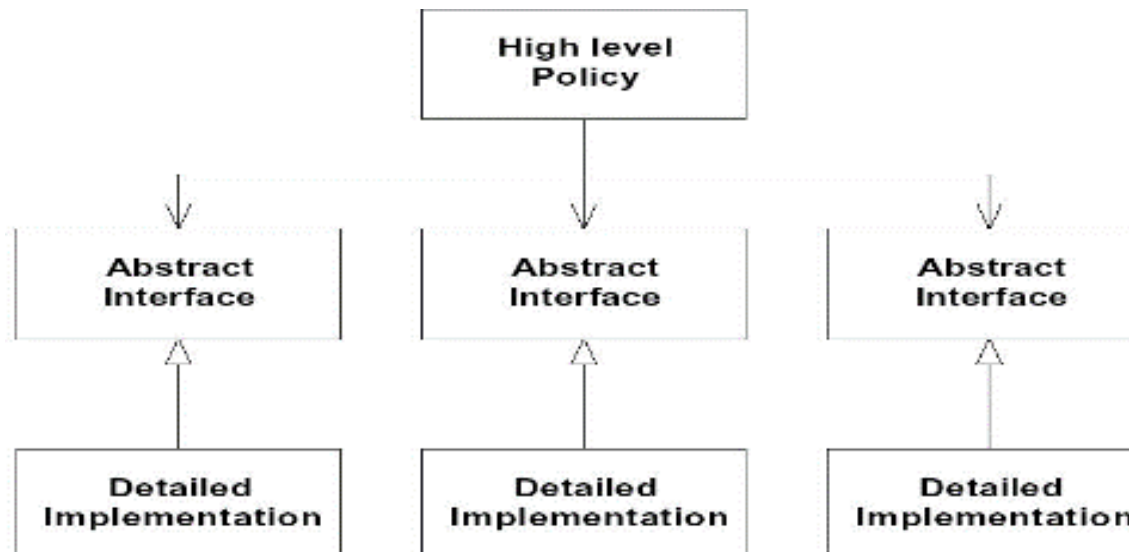
- OCP indica el **objetivo**; DIP indica el **mecanismo**.
- Una superclase no debe conocer ninguna de sus subclases.
- Los módulos con detalles de implementación no son interdependientes, sino que su dependencia se define en base a abstracciones.

Arquitectura Procedural vs. OO

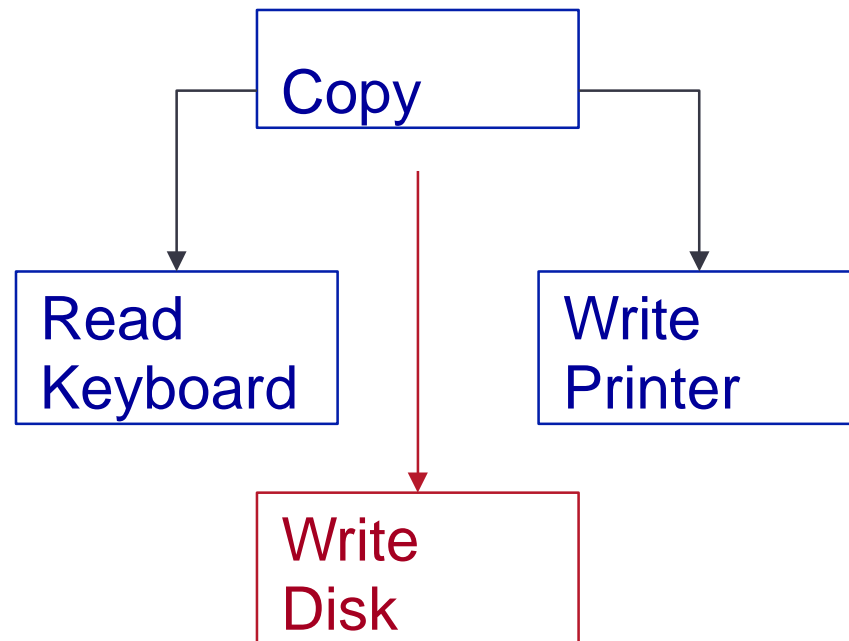
Arquitectura
Procedural



Arquitectura
Orientada a objetos



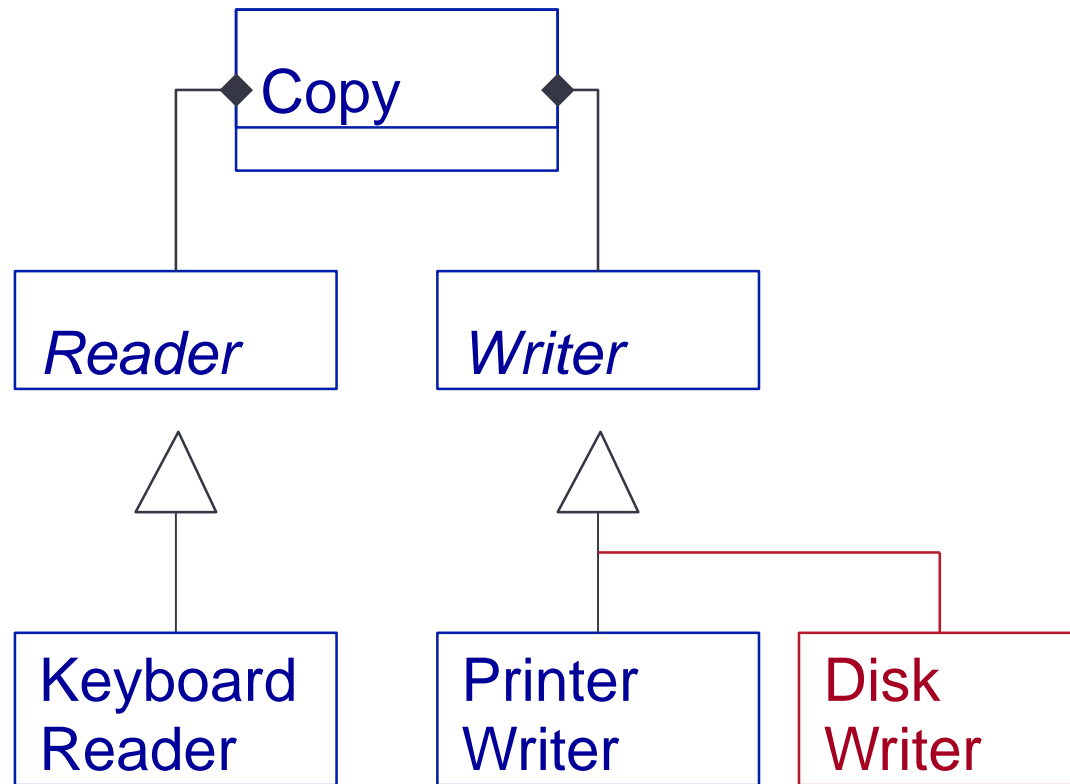
Ejemplo aplicando enfoque procedural



```
enum OutputDevice {printer, disk};  
void Copy(OutputDevice dev){  
    int c;  
    while((c = ReadKeyboard())!= EOF)  
        if(dev == printer)  
            WritePrinter(c);  
        else  
            WriteDisk(c); }
```

```
void Copy(){  
    int c;  
    while ((c = readKeyboard()) != EOF)  
        writePrinter(c);  
}
```

Aplicando el Principio de Inversión de Dependencia (DIP)



```
class Reader {
    public abstract int read();
}
```

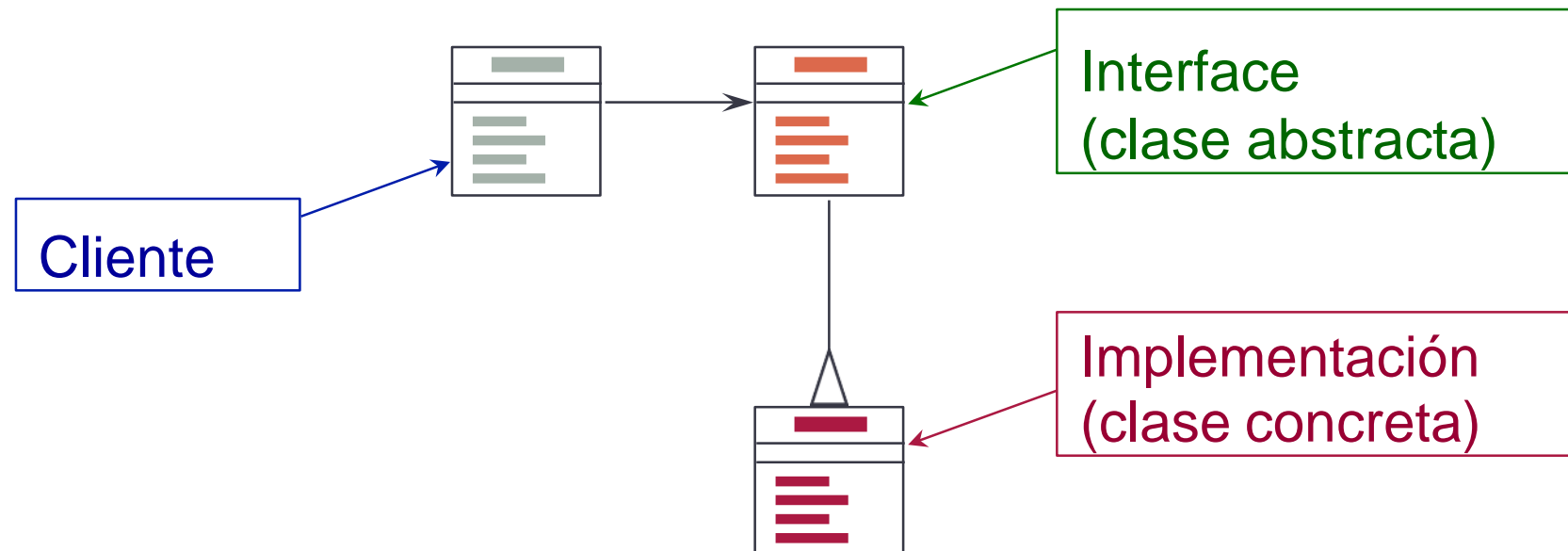
```
class Writer {
    public abstract void write(int i);
};
```

```
void copy(Reader r, Writer w){
    int c;
    while((c = r.read()) != EOF)
        w.write(c);
}
```

Heurísticos relacionados con el DIP

Diseña para la interfaz, no para la implementación!

Utiliza la herencia para evitar en enlace directo entre clases.



Diseña para la interfaz

- **Clases abstractas/interfaces:**
 - Tienden a cambiar menos frecuentemente.
 - Las abstracciones son “puntos bisagra” donde es más sencillo extender/modificar.
 - En general no debería modificarse la clase/interfaz que representa la abstracción (Principio OCP).
- **Excepciones**
 - Algunas clases nunca van a cambiar, por ejemplo
 - Clase *String*. No tiene sentido añadir una capa de abstracción
 - En estos caso es más interesante utilizar la clase directamente
 - Como en Java o C++.

Heurísticos relacionados con DIP

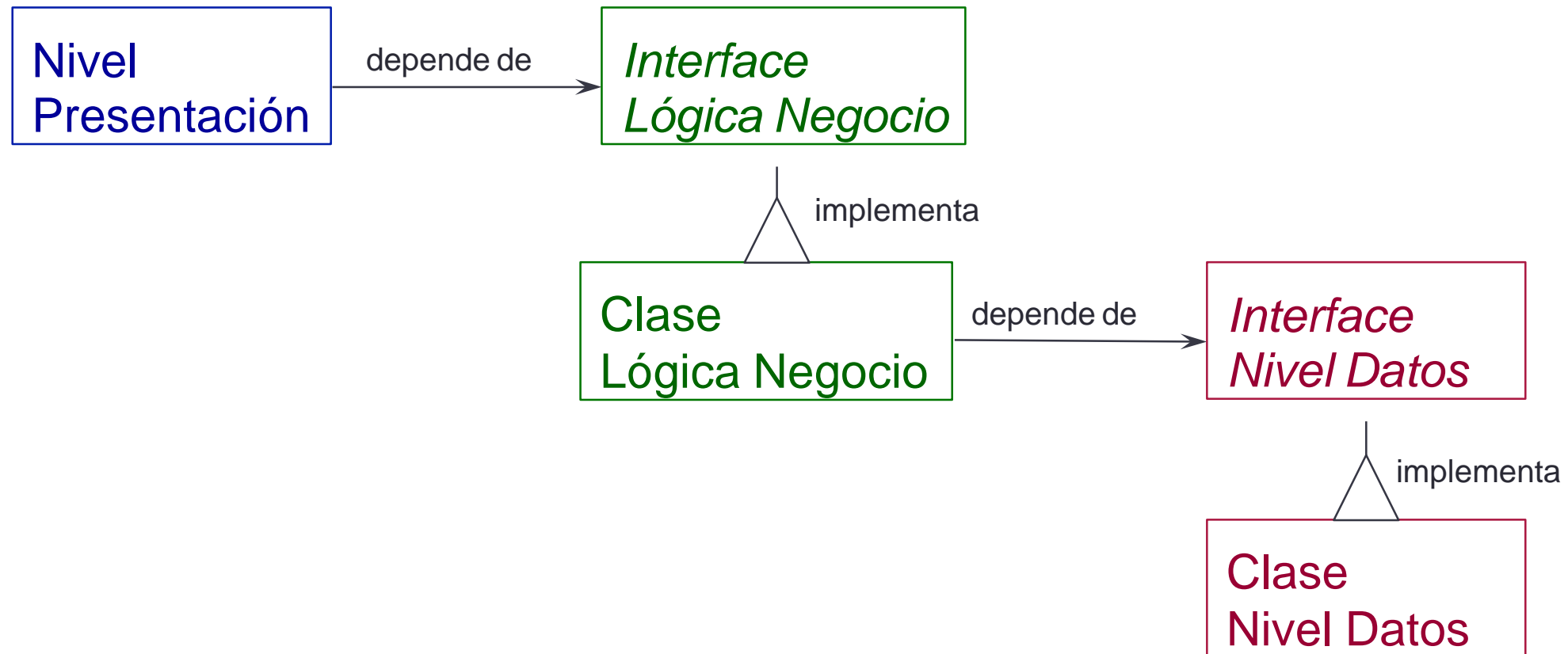
Evita dependencias transitivas

- Evita estructuras en las que las abstracciones de alto nivel dependen de abstracciones de bajo nivel:
 - En el ejemplo, el nivel de presentación depende finalmente del nivel de datos.



Solución a las dependencias transitivas

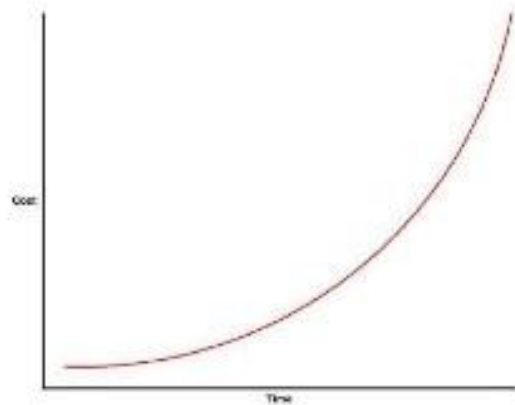
Utiliza la herencia y las clases abstractas para eliminar las dependencias transitivas:



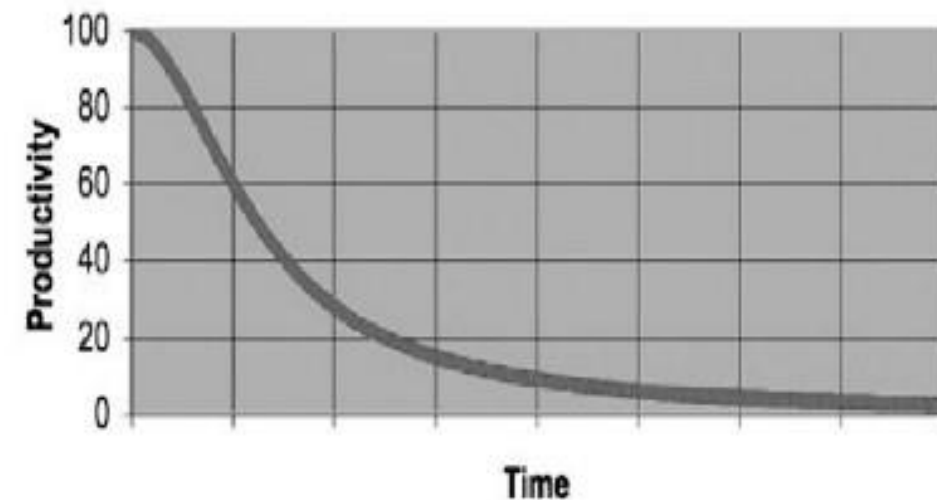
Conclusiones SOLID

- *Los sistemas software cambian durante su ciclo de vida, y los diseños software deben acomodar estos cambios.*
- El coste del cambio aumenta con el tiempo

The cost of change curve



- A la vez que se reduce la productividad.



Conclusiones SOLID

- Los principios SOLID facilitan el diseño OO evolutivo.
- Los principios SOLID son el primer paso para realizar diseños complejos, sin embargo se necesitan técnicas más sofisticadas.

Los patrones de diseño

Para saber más

- Principios SOLID

- <http://www.slideshare.net/bbossola/geecon09-solid>
- <http://www.desarrolloweb.com/manuales/programacion-orientada-objetos-dotnet.html>

- Principio OCP

- <http://www.utopicainformatica.com/2010/09/principio-abierto-cerrado.html>
- <http://danielmazzini.blogspot.com/2010/10/principio-abierto-cerrado-ocp.html>

- Principio SRP

- <http://carlospeix.com/2010/11/principios-solid-1-ejemplo-con-srp-dip-y-ocp/>
 - <http://theartoftheleftfoot.blogspot.com/2010/06/solid-el-principio-de-la.html>
 - <http://joelabrahamsson.com/entry/the-open-closed-principle-a-real-world-example>
-

Para saber más

- Principio LSK
 - <http://javaboutique.internet.com/tutorials/JavaOO/>
- Principio DIP
 - <http://martinfowler.com/articles/injection.html>
 - <http://www.slideshare.net/MarcoManga/dependency-inversion-principle>
- Principio ISP
 - <http://www.oodesign.com/interface-segregation-principle.html>

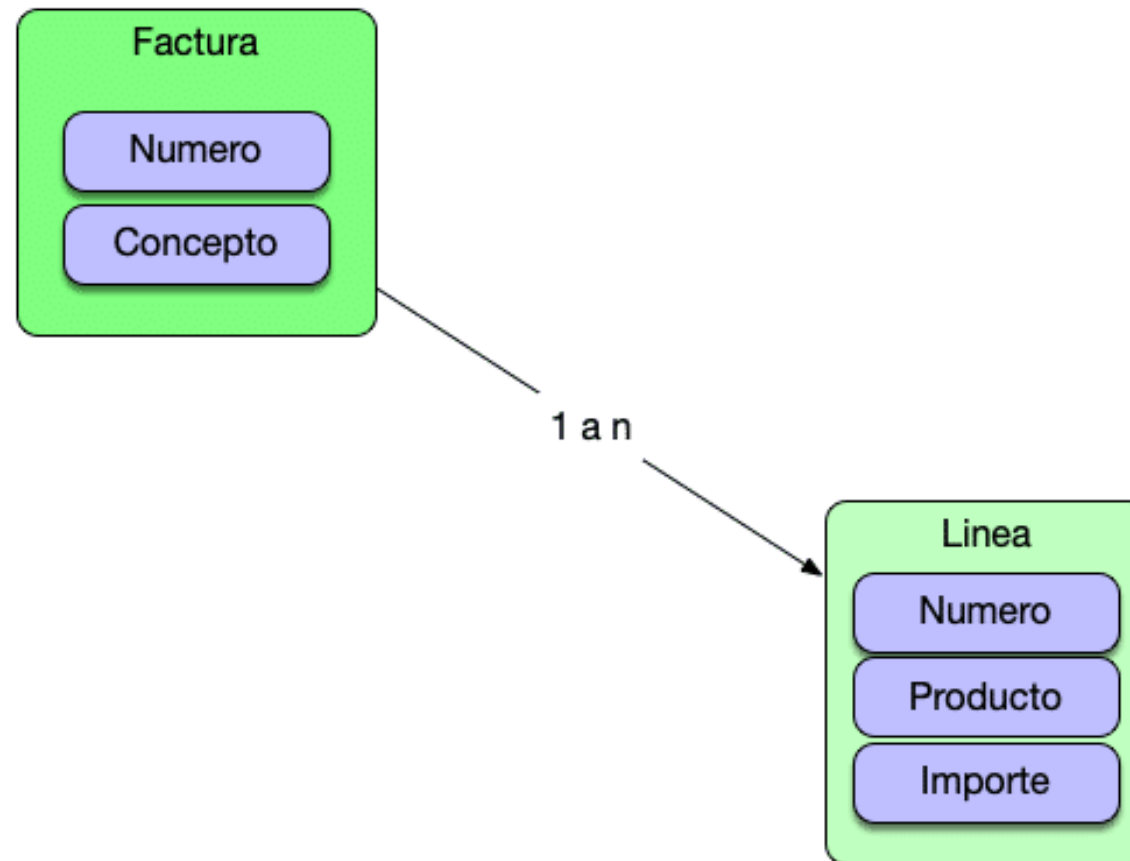
Otros principios

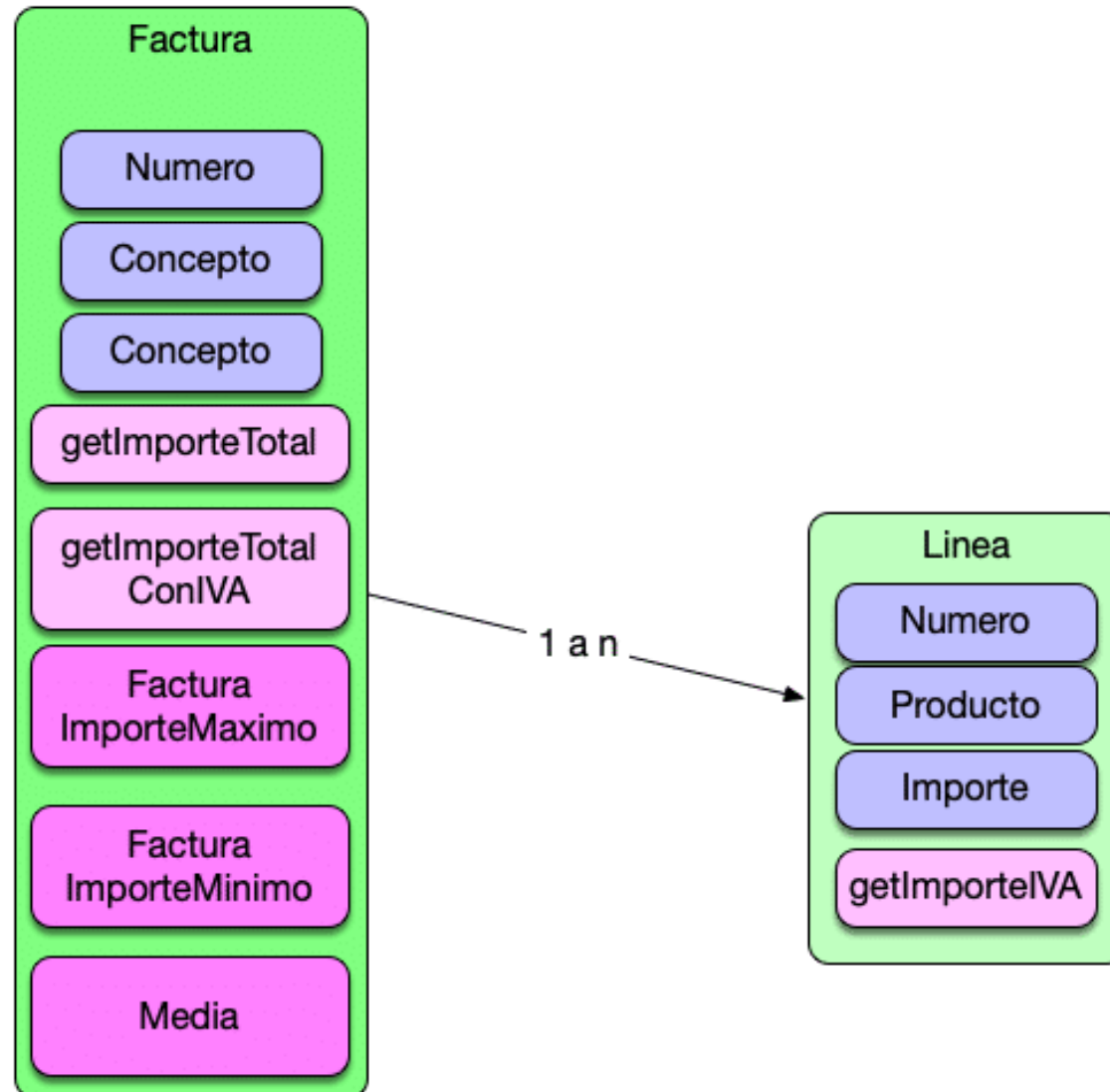
The background of the slide is a light gray with a subtle pattern of overlapping hexagons and circles. Some hexagons are solid white, while others are outlined in white. The circles are also white outlines. The overall effect is a clean, modern, and geometric aesthetic.

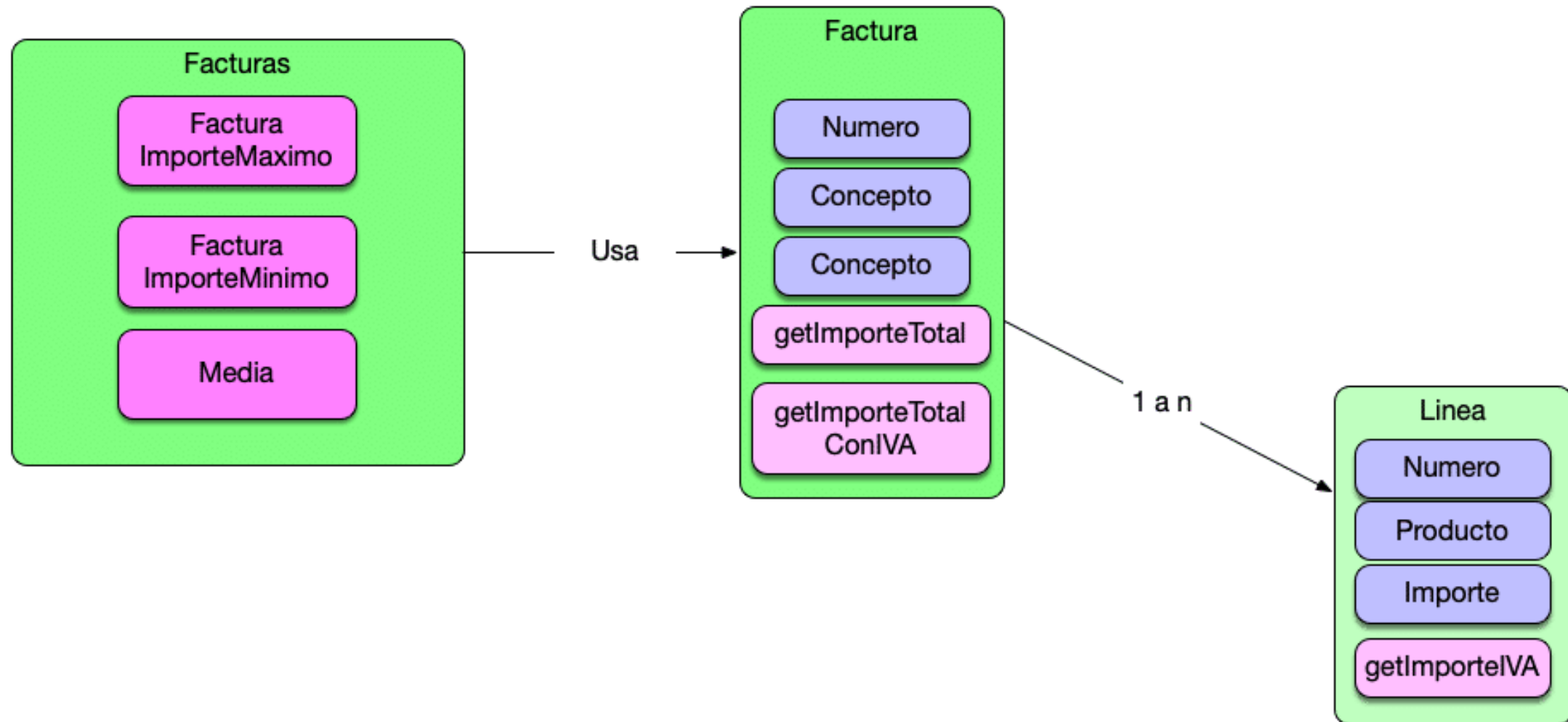
KISS - Keep It Simple Stupid

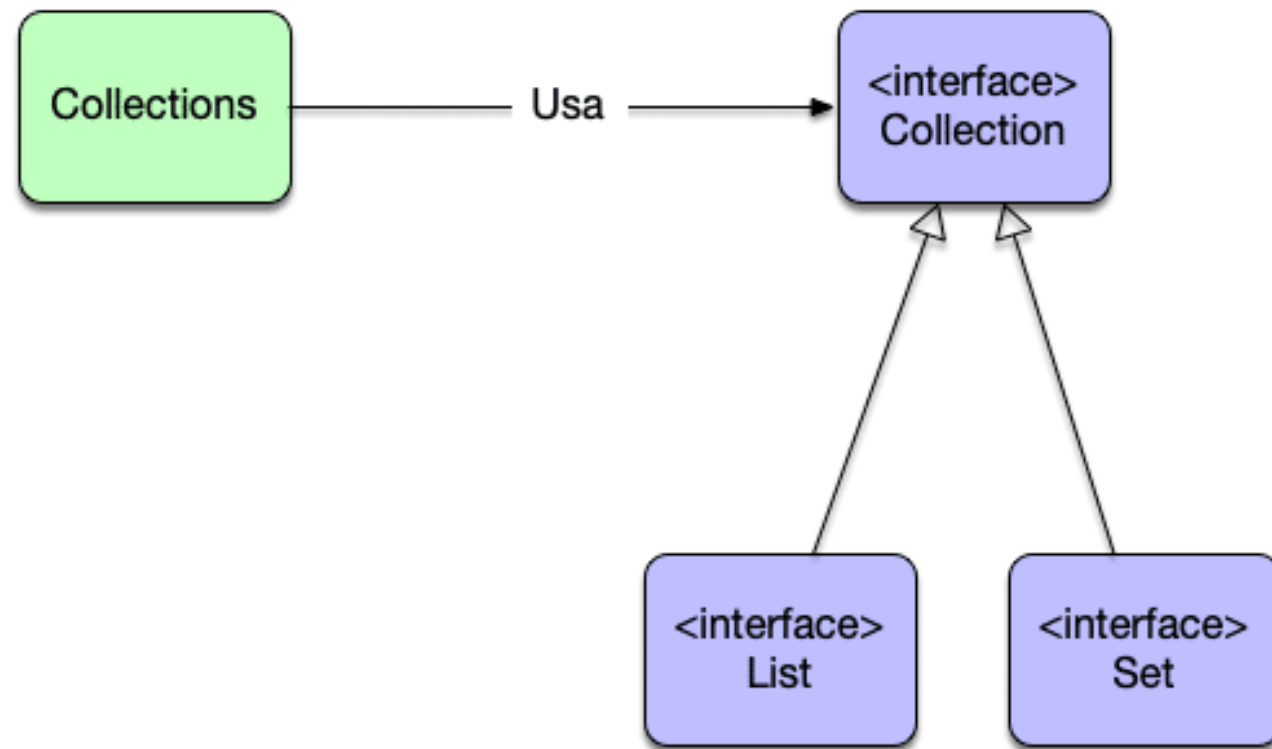
- ▶ Es un principio de diseño y programación en el que la simplicidad del sistema se declara como objetivo o valor principal y debe evitarse una complejidad innecesaria.
- ▶ Siempre se debe usar la solución más simple a un problema. Se trata principalmente de hacer que el código sea lo más simple posible.

Ejemplo









KISS

- ▶ No tiene sentido aumentar infinitamente el nivel de abstracción, hay que poder detenerse en el tiempo
- ▶ No tiene sentido poner en el proyecto funciones redundantes «en reserva» que algún día alguien pueda necesitar (aquí el enfoque según el principio **YAGNI (You Ain't Gonna Need It)** es más correcto)
- ▶ Para qué incluir una biblioteca enorme si solo se necesita un par de funciones de ella
- ▶ La descomposición de algo complejo en componentes simples es un enfoque arquitectónicamente correcto (aquí KISS se hace eco de DRY)
- ▶ No siempre se necesita precisión matemática absoluta o detalles extremos: Los datos pueden y deben procesarse con la precisión suficiente para una solución de alta calidad del problema, y los detalles se dan en la cantidad necesaria para el usuario, y no en el volumen máximo posible.

¿Por qué usarlo?

- ▶ La mayoría de los sistemas de software están excesivamente sobrecargados con funciones prácticamente innecesarias, lo que empeora la conveniencia de su uso por parte de los usuarios y complica su soporte y desarrollo por parte de los desarrolladores.
- ▶ **Seguir el principio KISS permite desarrollar soluciones fáciles de usar y mantener.** En el desarrollo de software, los principios KISS y DRY son dos principios que se recomiendan al implementar un código limpio.

¿Cuáles son los beneficios?

- ▶ Este principio es especialmente necesario para proyectos de software que son medianos y grandes. Entre los beneficios se incluyen:
- ▶ Mayor calidad del código
- ▶ Código más simple de mantener
- ▶ Más flexible y fácil de ampliar, modificar, mejorar.
- ▶ KISS también tiene mucho en común con el principio de separación de interfaces de los cinco principios SOLID

Ejemplo: Microsoft Word

- ▶ ¿A que Word te pega un suston si despliegas todas sus herramientas? Microsoft te da la posibilidad de acceder a todo ello, pero se asegura de mostrarte las herramientas de manera sencilla, con lo que más te puede interesar.
- ▶ En el lado de los sistemas operativos, tienen a personas centradas en la experiencia del usuario que se encargan de asegurarse de que todo pueda estar a los mínimos clics de ratón posibles.

EJEMPLO: GOOGLE

- ▶ Todos estamos de acuerdo en que Google es tremendamente complejo y profundo, ¿verdad?
- ▶ Una de las claves de su éxito respecto a otros competidores ha sido su sencillez. La gente va a Google a buscar cosas, así que se aseguraron de poner la barra de búsqueda, sin distracciones. En un principio, la sencillez de su diseño era por la falta de conocimientos de programación, pero luego descubrieron que era mejor así.

Qué tener en cuenta?

- ▶ Quién es tu target?
- ▶Cuál es tu objetivo?
- ▶ Qué canales dispones?
- ▶ Qué quieres contar?
- ▶ Vuelve a recordar cuál es tu objetivo

Principio DRY

DRY: Don't Repeat Yourself



DRY

- ▶ Es un principio del desarrollo software que establece que «Cada pieza de conocimiento debe tener una única representación autorizada, sin ambigüedades, dentro de un sistema».
- ▶ Si se aplica correctamente, una modificación en un elemento del sistema no requerirá un cambio en otros elementos no relacionados lógicamente.

El problema

- ▶ Como programadores pasaremos gran parte de nuestro tiempo manteniendo software ya existente y raramente estaremos escribiendo código *original*.
- ▶ Ya sea por que estemos realizando cambios, refactorizando para mejorar el programa o arreglando un bug. Por tanto, nos conviene planificar y desarrollar nuestros sistemas pensando en ese mantenimiento desde el principio.

El problema

- ▶ Consideremos un elemento de nuestro sistema con más de una representación. En algún punto durante la vida de nuestro sistema haremos una modificación de alguna de estas representaciones lo que la desfasaría del resto y puede hacer que entren en conflicto. En caso de que no sea así, significa que cada vez que cambiemos una representación de un elemento, debemos buscar el resto de representaciones y actualizarlas en consecuencia. Conforme nuestro sistema evoluciona esto deriva en tremendos dolores de cabeza a la hora de realizar cualquier modificación o mantenimiento.
- ▶ No es algo que se limite solo al código, si no que debe aplicarse al sistema al completo como esquemas de base de datos, test, documentación, etc. Este es el principal problema ya que debemos expresar nuestros elementos de forma que se propaguen por todas las capas del sistema.

Principio DRY y Ortogonalidad

- ▶ Como solución a este problema, Andy Hunt y Dave Thomas formularon en su libro «The Pragmatic Programmer» el principio DRY.
- ▶ Este principio establece la idea o necesidad de que los diferentes elementos del sistema deben de ser únicos, no solo en el código, si no en todas las capas o dominios del sistema.
- ▶ La idea de ortogonalidad se basa en que dos elementos no relacionados conceptualmente no deben de estar relacionados en el sistema. De este modo cambios en uno de los elementos no deben de provocar cambios en otros elementos *ortogonales*. Por ejemplo, un cambio en la interfaz de usuario no debe de provocar cambios en el esquema de la base de datos.

¿Cómo surge la duplicación?

- ▶ La duplicación puede parecer impuesta. El desarrollador puede pensar que el sistema parece requerir duplicación.
- ▶ La duplicación puede ser involuntaria. El desarrollador puede no darse cuenta de que está duplicando información.
- ▶ La duplicación por impaciencia. El desarrollador puede tener flojera y duplicar porque parece lo más fácil.
- ▶ La duplicación por falta de coordinación. Múltiples desarrolladores en un equipo o diferentes equipos pueden duplicar información.

Duplicación en el código

- ▶ Con respecto al código, no se trata de que los pedazos de código NUNCA se deban repetir, a veces es necesario y menos complejo dejar que algunas líneas se repitan en diferentes clases; donde en realidad lo debes aplicar es cuando tratas de hacer representaciones funcionales de tu aplicación: Ejemplo, si tienes una rutina que va a recuperar los datos de cliente y hacer cierta transformación a esos datos y lo harás a menudo en la aplicación, vale la pena hacer una función llamada (por ejemplo) "RecuperaClienteInfo".

Duplicación en el código

- ▶ En muchas ocasiones, encontramos que una función existente hace "casi" lo mismo que queremos, pero necesitamos que haga algo extra o que no lo haga. Si te encuentras en este caso, en lugar de duplicar la función, mejor añádele un parámetro, y utiliza la información de ese parámetro para que la función haga el trabajo que necesitas. Una vez hecha esta operación, el problema que se presenta es rastrear el código existente y modificar por todos lados para añadir ese parámetro; puedes entonces tener varias alternativas:
- ▶ Revisa si tu lenguaje de programación detecta el número de parámetros recibidos y mediante esa función lee el nuevo parámetro y condiciona la nueva lógica.
- ▶ Hacer algo de "refactorización".
- ▶ Aquí tendrías que renombrar las funciones y adaptar tu código, es doloroso, pero a la larga te beneficiarás, sobre todo si planeas que tu programa sobreviva el devenir de los años y la modernización obligada en el futuro.
- ▶ En Programación Orientada a Objetos se hace uso de la sobrecarga de funciones.

Otros: YAGNI

- ▶ Para cerrar el círculo de acrónimos, en este caso de «You aren't gonna need it».
- ▶ Este principio invita a no desarrollar «por si acaso», ¿cuántas veces nos han pedido una funcionalidad determinada y hemos excedido los requisitos en previsión de algo que nunca ocurrió? Pues bien, YAGNI propone hacer exclusivamente lo solicitado y necesario.
- ▶ YAGNI tiene su origen en la programación extrema, dónde prima la simplicidad y dónde se opta por no invertir tiempo en dar solución a necesidades todavía inexistentes. La programación extrema se guía por la afirmación «Do The Simplest Thing That Could Possibly Work», de la cual han creado un acrónimo DTSTTCPW que desde mi punto de vista excede lo razonable y roza el absurdo [1, 2].

1. *Extreme Programming examined*, Giancarlo Succi, Michele Marchesi, 2001, 569 pages, quote: "XP says 'do the simplest thing that could possibly work ' because 'you aren't gonna need it'."

2. ^a [Jump up to:](#) ^b *Object-oriented & classical software engineering*, Stephen R. Schach, 2007, 618 pages, p., quote: "Two acronyms now associated with extreme programming are YAGNI (you aren't gonna need it) and DTSTTCPW (do the simplest thing that could possibly work)."