

# Criterios de Buen Diseño

FI.UBA


75.10 - Técnicas de Diseño

# Diseño

- Cual es el significado de Diseño?
- Cual es la diferencia si lo comparamos con el análisis?



Análisis es acerca  
del **QUE**



Diseño es acerca  
del **COMO**

- Por qué necesitamos un (buen) diseño?



Para manejar  
el cambio

Para tener un  
delivery rápido

Para lidiar con  
la complejidad

- Como sabemos que el diseño es malo?



- Debemos definir algunos criterios
- Son estos síntomas de un mal diseño?



**RIGIDEZ**



**INMOBILIDAD**



**FRAGILIDAD**

- Por que es que los diseños se vuelven rígidos, frágiles e inmóviles?



**INCORRECTAS DEPENDENCIAS  
ENTRE MODULOS**

- Características de un buen diseño



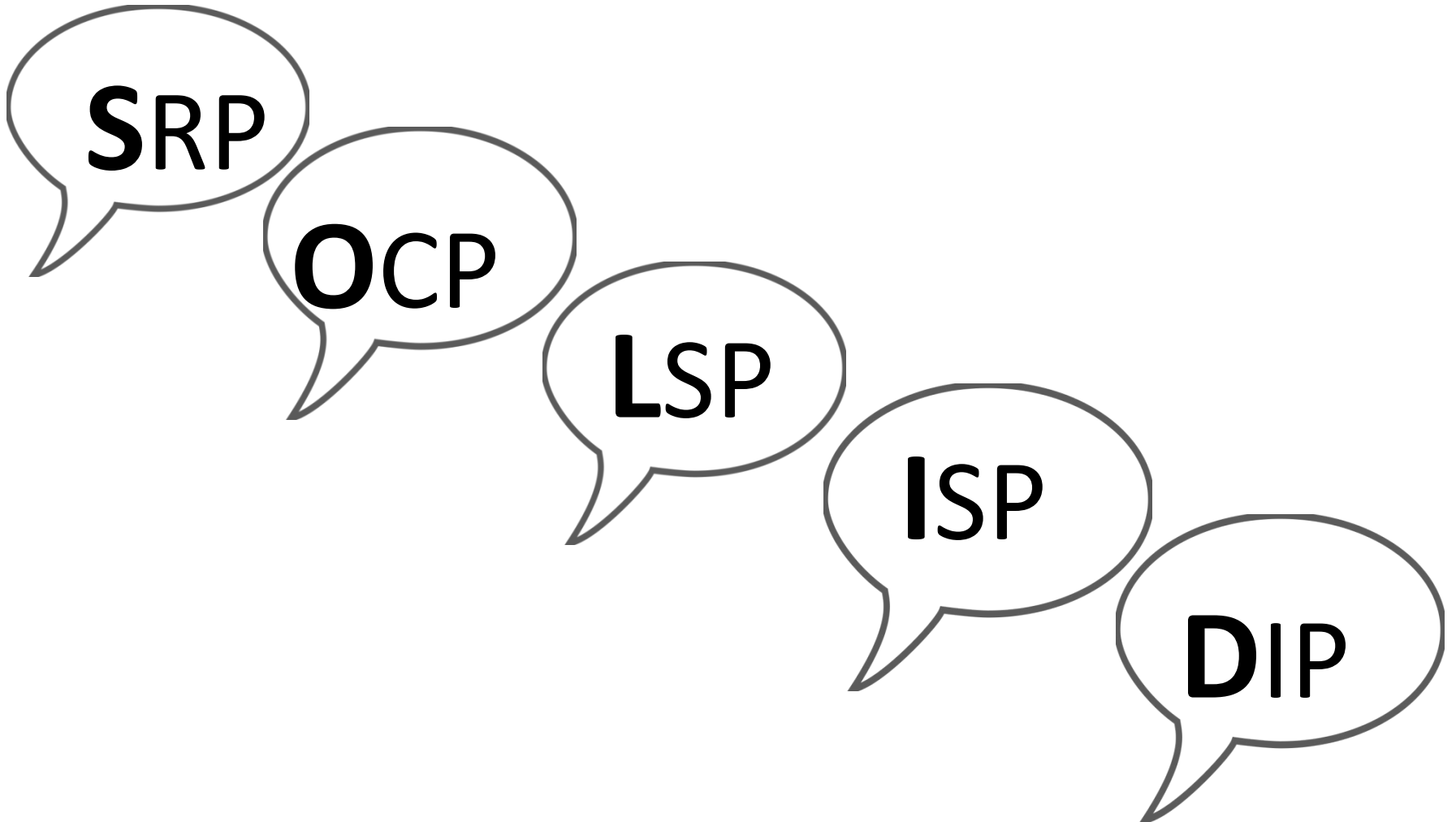
**ALTA  
COHESION**



**BAJO  
ACOPLAMIENTO**

# S.O.L.I.D.

- Como lograr un buen diseño







# SOLID

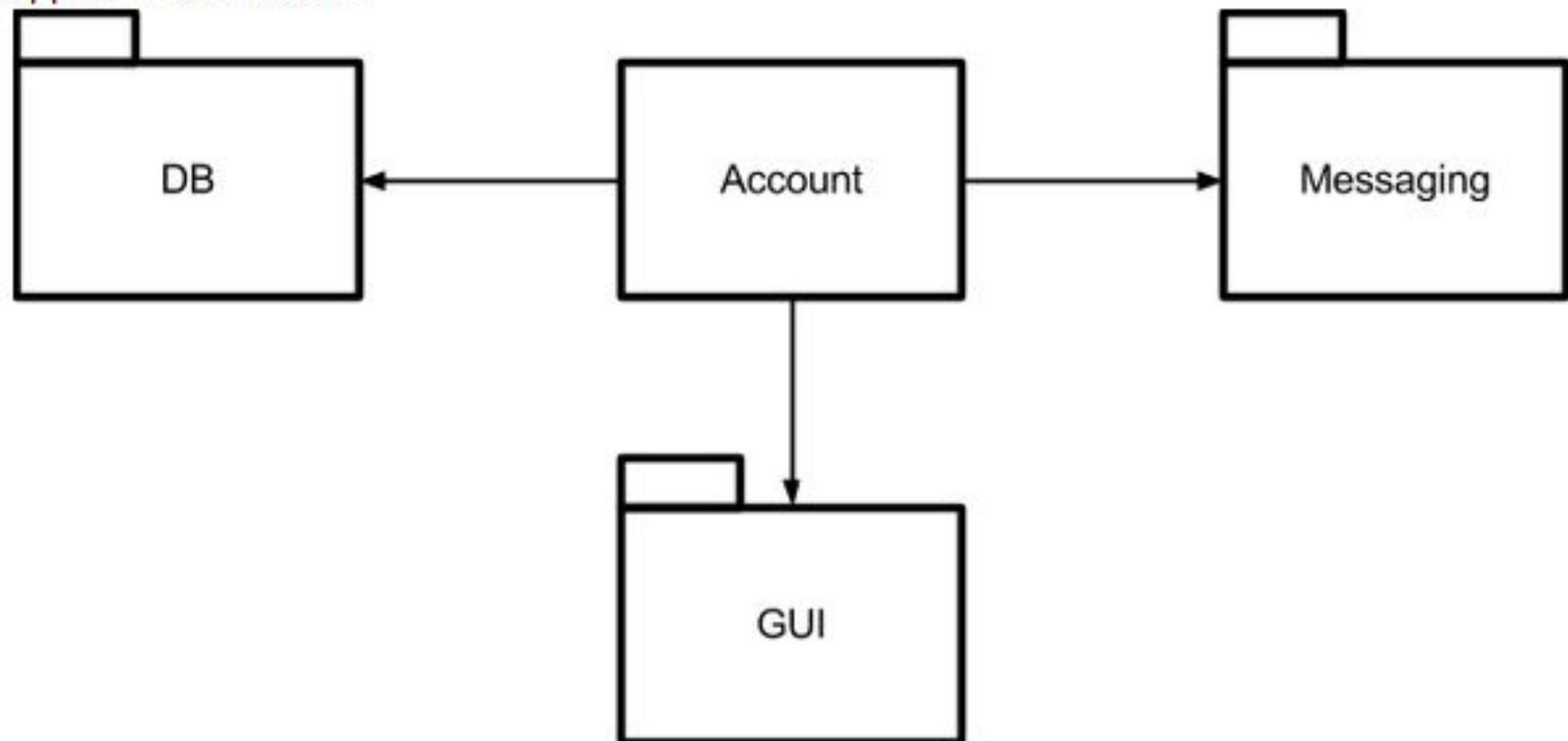
Software Development is not a Jenga game



# Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

Supongamos una clase Account que representa un Cuenta de un banco o de alguna app de E-Commerce.



En el diagrama se ve que la clase Account tiene 3 dimensiones distintas de cambio (por todo lo que usa) más la dimensión inherente a la entidad que se esta modelando. Es decir 4 posibles razones en total.

```
public class OrderController
{
    ...

    public ActionResult CreateForm()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(OrderCreateRequest request)
    {
        if (!ModelState.IsValid)
        {
            /* View data preparations */
            return View();
        }
        using (var context = new DataContext())
        {
            var order = new Order();
            // Create order from request
            context.Orders.Add(order);

            // Reserve ordered goods
            ...(Huge logic here)...

            context.SaveChanges();

            //Send email with order details for customer
            var smtp = new SMTP();
            // Setting smtp.Host, UserName, Password and other parameters
            smtp.Send(client, order);
        }
        return RedirectToAction("Index");
    }
    ... (many more methods like Create here)
}
```

```
public class OrderService
{
    public void Create(OrderCreateRequest request)
    {
        // all actions for order creating here
        using (var context = new DataContext())
        {
            var order = new Order();
            // Create order from request
            context.Orders.Add(order);
            // Reserve ordered goods
            ...(Huge logic here)...
            context.SaveChanges();

            //Send email with order details for customer
            var smtp = new SMTP();
            // Setting smtp.Host, UserName, Password and other parameters
            smtp.Send();
        }
    }
}

public class OrderController
{
    public OrderController()
    {
        this.service = new OrderService();
    }
    [HttpPost]
    public ActionResult Create(OrderCreateRequest request)
    {
        if (!ModelState.IsValid)
        {
            return View();
        }
        this.service.Create(request);
        return RedirectToAction("Index");
    }
}
```

Enviar un correo electrónico, en realidad, no es una parte del flujo de creación del pedido principal.

Incluso si la aplicación no logra enviar el correo electrónico, la orden sigue creándose correctamente.

Además podría surgir una nueva opción en el área de configuración de usuario que les permite optar por no recibir un correo electrónico después de realizar un pedido con éxito, o incluso otros medios.

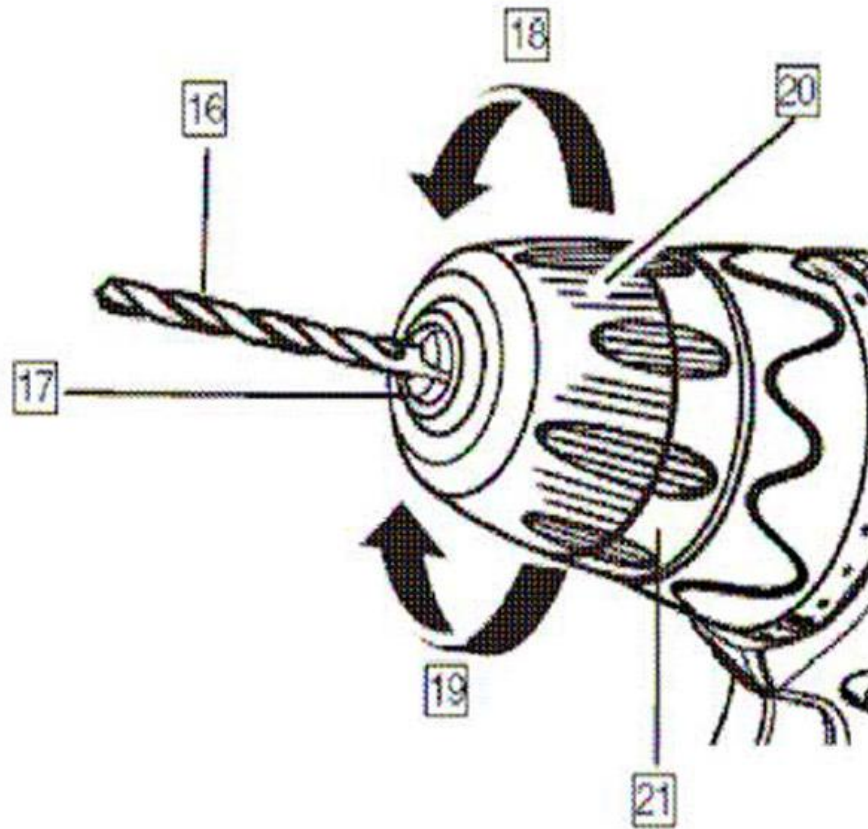
Puede pensarse en un Helper para delegar el envío, o para desacoplar más un, un Observer o Eventos.



# Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.





Es una herramienta poderosa muy útil, nos reduce el trabajo, pero está abierta para extensión, mientras está cerrada para su modificación.



```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

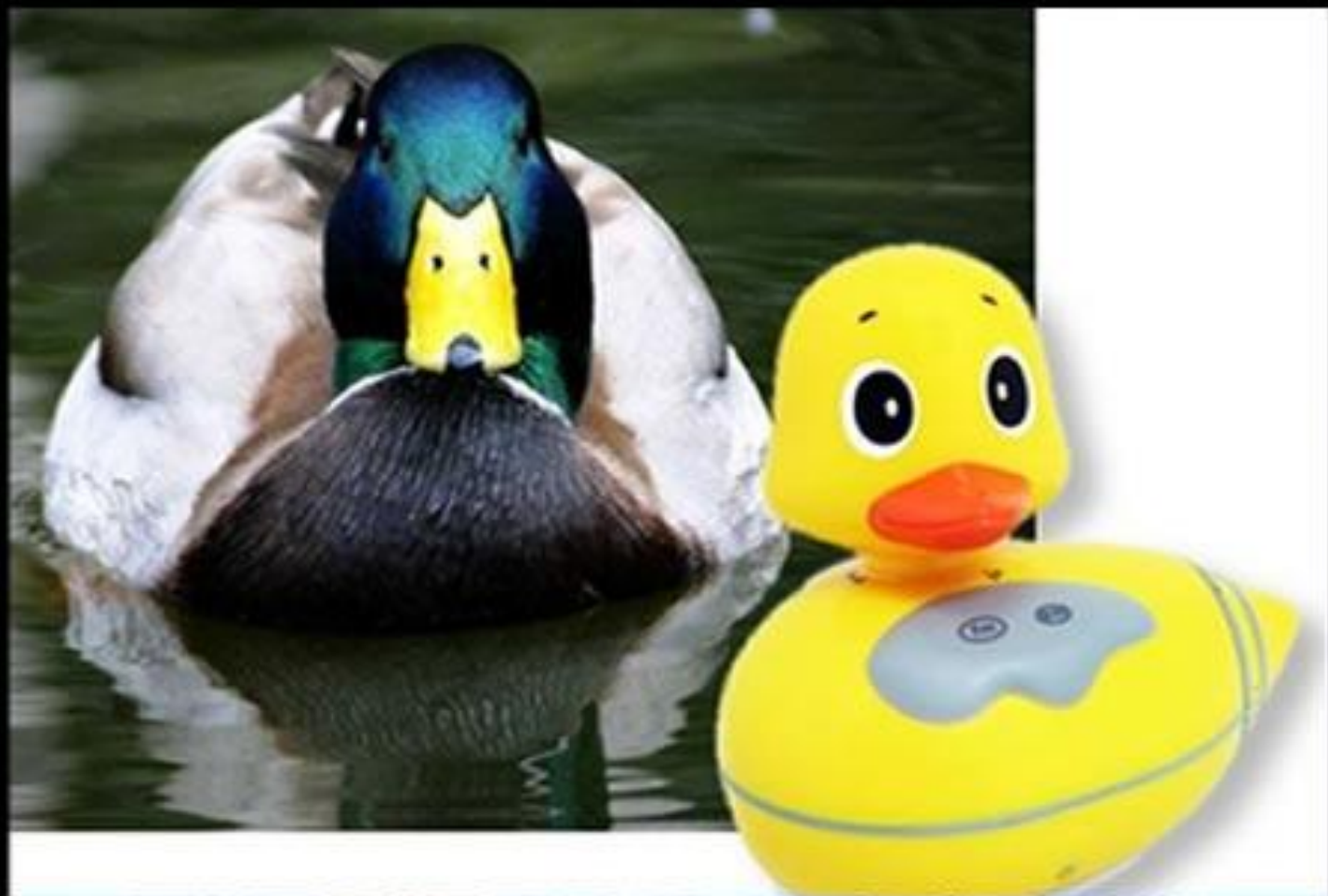
```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }
        return area;
    }
}
```

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape; area +=
rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape; area += circle.Radius * circle.Radius *
Math.PI;
        }
    }
    return area;
}
```

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

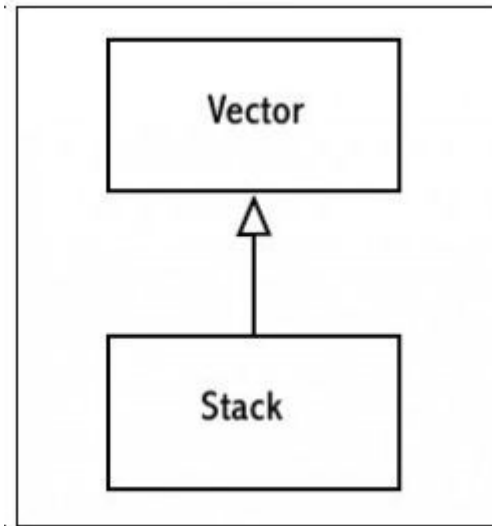
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}

public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }
    return area;
}
```



# Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.



```
Vector<String> vectorStack = new Stack<String>();
vectorStack.addElement("one");
vectorStack.addElement("two");
vectorStack.addElement("three");
vectorStack.removeElementAt(1);
System.out.println(vectorStack.size());
// throws Exception si quiere ser un Stack. (o sino quizas imprime: 2)
```



# Interface Segregation Principle

You want me to plug this in *where*?

IBird

Eat()

Walk()

Chirp()

Fly()

IBird

Eat()

Walk()

Chirp()

IFlyingBird

Fly()



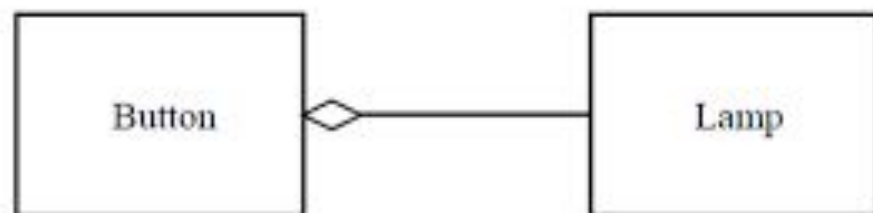
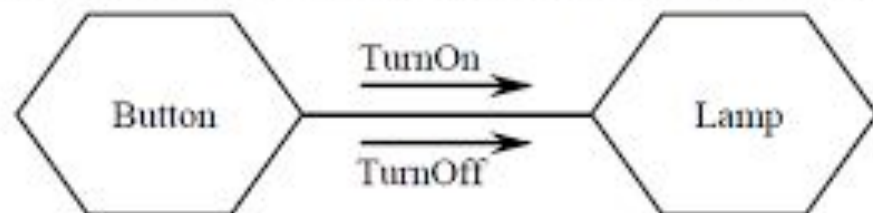


# Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?



**Figure 5: Naive Button/Lamp Model**



**Listing 5: Naive Button/Lamp Code**

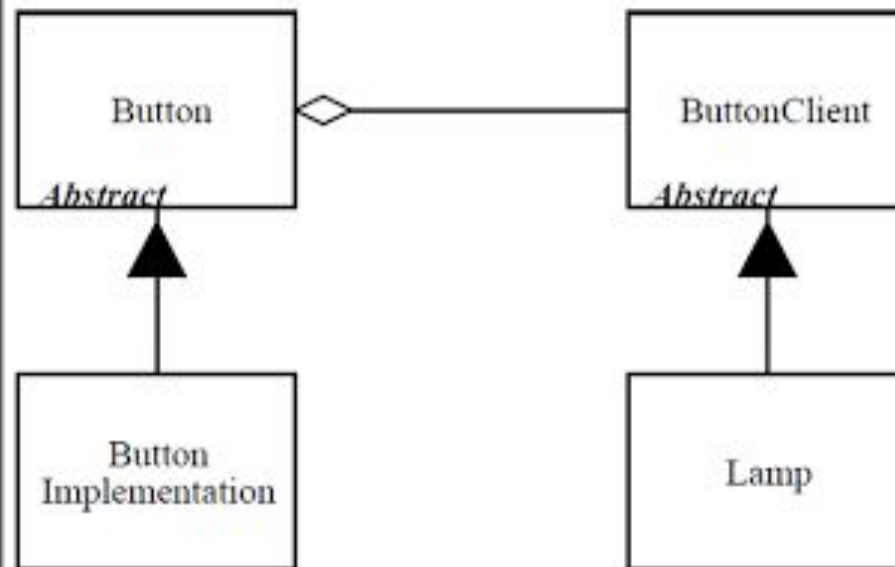
```
-----lamp.h-----
class Lamp
{
public:
    void TurnOn();
    void TurnOff();
};

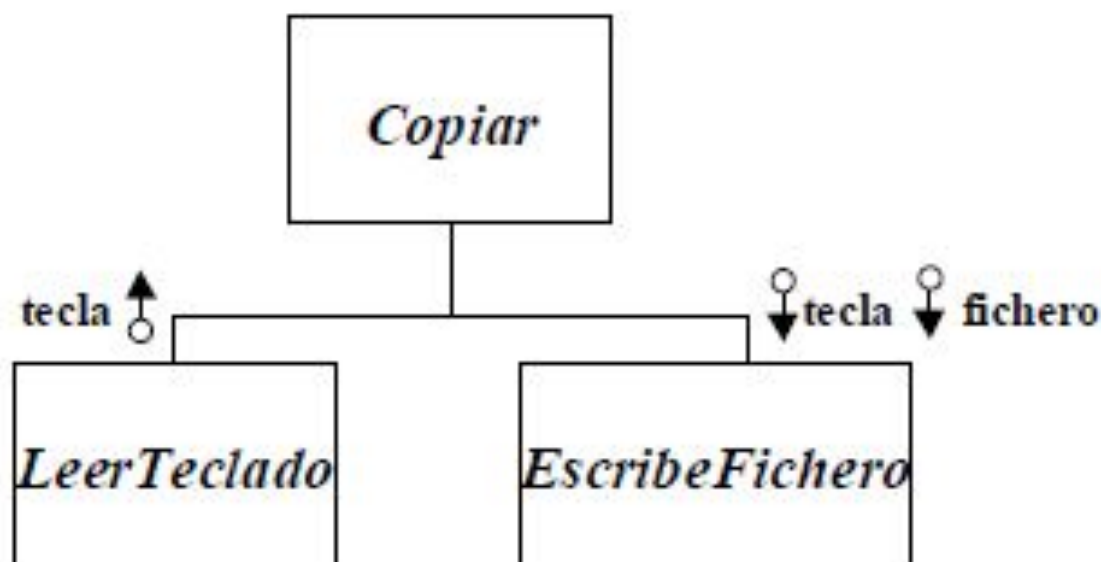
-----button.h-----
class Lamp;
class Button
{
public:
    Button(Lamp& l) : itsLamp(&l) {}
    void Detect();
private:
    Lamp* itsLamp;
};

-----button.cc-----
#include "button.h"
#include "lamp.h"

void Button::Detect()
{
    bool buttonOn = GetPhysicalState();
    if (buttonOn)
        itsLamp->TurnOn();
    else
        itsLamp->TurnOff();
}
```

**Figure 6: Inverted Button Model**



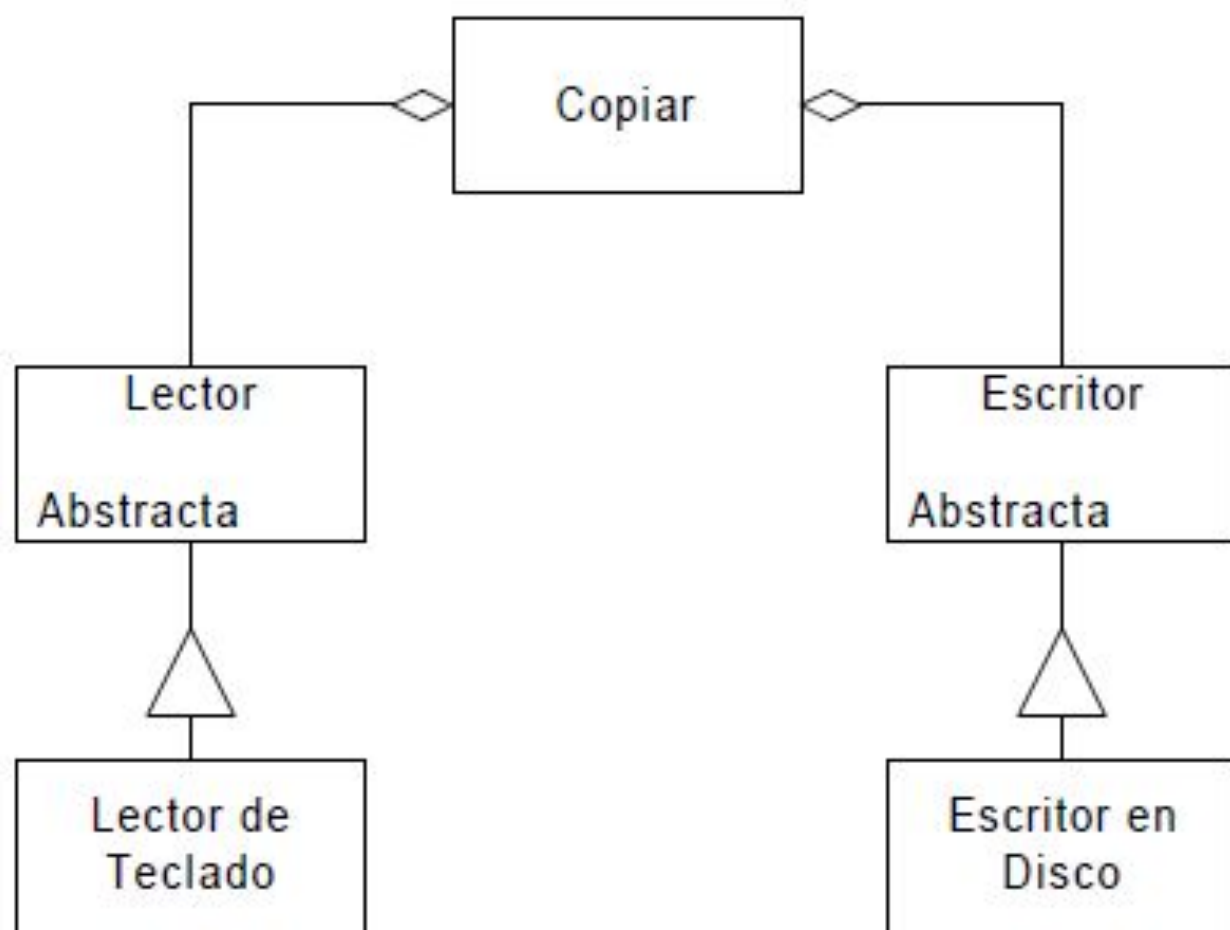


### Listing1. The Copy Program

```
void Copy() {  
    · · int c;  
    · · while((c=ReadKeyBoard())!=EOF)  
    · · · WritePrinter(c);  
}
```

## Listing2: The “Enhanced” Copy Program↵

```
void Copy() {↵  
    · · int c;↵  
    · · while((c=ReadKeyBoard())!=EOF)↵  
    · · · if(dev==printer)↵  
        WritePrinter(c);↵  
    · · · Else↵  
    · · · WiteDisk(c);↵  
}↵
```



# Conclusiones

- Un buen diseño es necesario para lidiar exitosamente con los cambios
- Los principales fuerzas que conducen el diseño deberían ser ALTA COHESION y BAJO ACOPLAMIENTO
- Los principios S.O.L.I.D. nos ponen en el camino correcto

# Bibliografía

## Principios SOLID – ObjectMentor.COM

[http://www.objectmentor.com/omSolutions/oops\\_what.html](http://www.objectmentor.com/omSolutions/oops_what.html)

- OCP — The Open Closed Principle
- LSP — The Liskov Substitution Principle
- DIP — The DIP — The DIP — The Dependency DIP — The Dependency DIP — The Dependency Inversion DIP — The Dependency Inversion DIP — The Dependency