UNIVERSIDAD DE LIMA | 60 años

FACULTAD DE INGENIERÍA Y ARQUITECTURA
CARRERA DE INGENIERÍA DE SISTEMAS

PRINCIPIOS DEL DISEÑO ORIENTADO A OBJETOS

UNIDAD 2: DISEÑO E IMPLEMENTACIÓN

Temario

- Principios DRY (Don’t Repeat Yourself)
- Principio Law of Demeter
- Principios SOLID

UNIVERSIDAD DE LIMA | 60 años

Principios en el diseño orientado a objetos

Un principio de diseño representa una guía altamente recomendada para dar forma a la lógica de la solución de cierta manera y con ciertos objetivos en mente

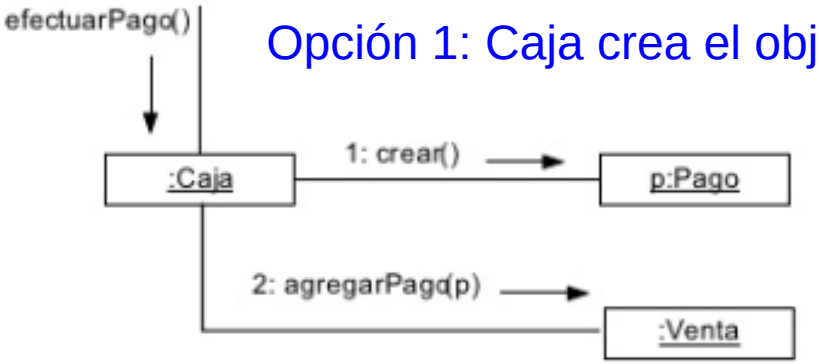
Principio de diseño

- Estos principios nos proporcionan unas **bases para la construcción** de aplicaciones mucho más **sólidas y robustas.**

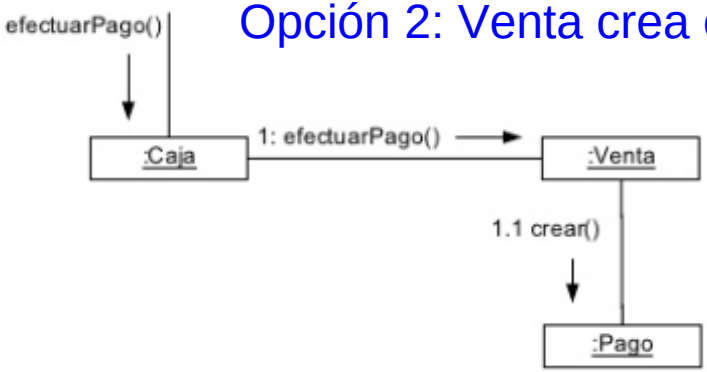
Principio: Bajo acoplamiento

- ¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?
- Una clase con alto (o fuerte acoplamiento) confía en muchas otras clases
  - Los cambios en las clases relacionadas fuerzan cambios locales
  - Son difíciles de entender de manera aislada
  - Son difíciles de reutilizar
- Asignar una sola responsabilidad para que el acoplamiento (innecesario) permanezca bajo.

Principio: Bajo acoplamiento



Opción 1: Caja crea el objeto Pago



Opción 2: Venta crea el objeto Pago

¿Cuál es la propuesta adecuada?

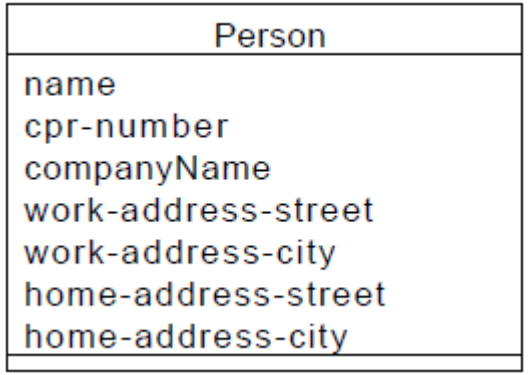
UNIVERSIDAD DE LIMA

Principio: Alta cohesión

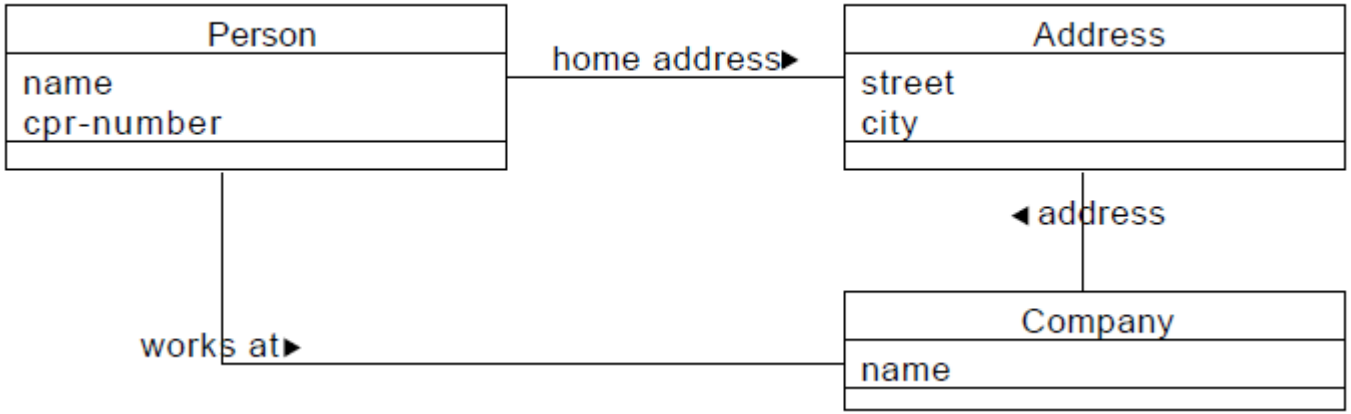
- ¿Cómo mantener la complejidad manejable?
- Asignar una responsabilidad para que la cohesión se mantenga alta.
- Una clase con baja cohesión hace demasiado trabajo:
  - Difíciles de entender
  - Difíciles de reutilizar
  - Difíciles de mantener
  - Delicadas, constantemente afectada por los cambios

¿Cuál tiene mayor cohesión?

UNIVERSIDAD DE LIMA



Baja Cohesión



Alta Cohesión

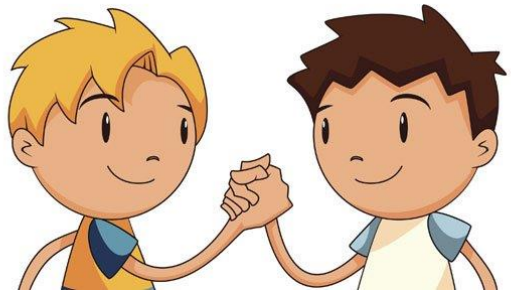
UNIVERSIDAD DE LIMA

Ley de Demeter (Law of Demeter)

- **“Habla solo con tus amigos, no hables con extraños”**

- Con quien puedes hablar:

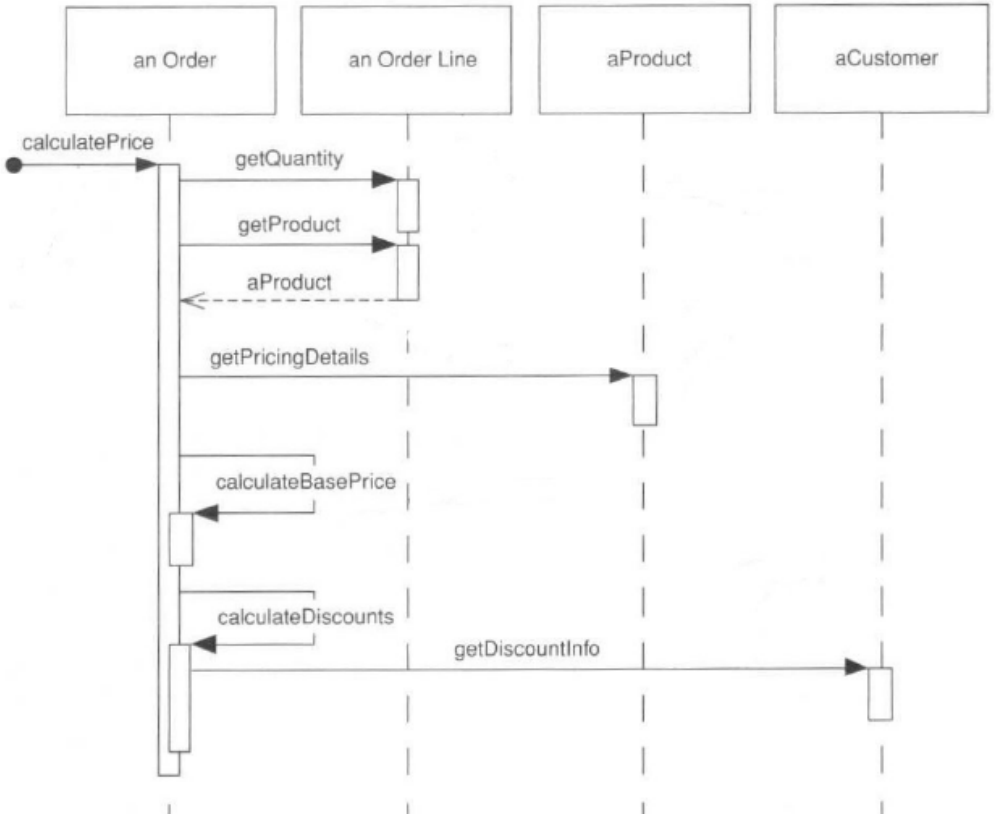
  1. A un objeto conectado mediante un enlace navegable A un objeto recibido como parámetro en esta activación.
  2. A un objeto creado localmente en esta ejecución, o variable local
  3. A mí mismo, el emisor del mensaje.

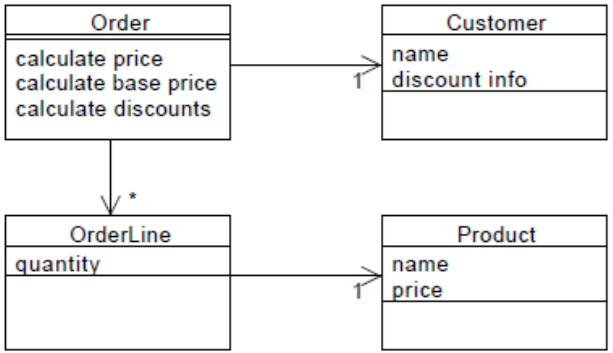


Ley de Demeter (Law of Demeter)

- La Ley de Deméter (LoD) es una regla de estilo simple para diseñar sistemas orientados a objetos
- "No hables con extraños, solamente con los que conoces"
- ¿A quién puedo enviar un mensaje?
  1. A un objeto conectado mediante un enlace navegable (instancia de asociación).
  2. A un objeto recibido como parámetro en esta activación.
  3. A un objeto creado localmente en esta ejecución, o variable local
  4. A mí mismo, el emisor del mensaje.
- Paper: Object-Oriented Programming:An Objective Sense of Style

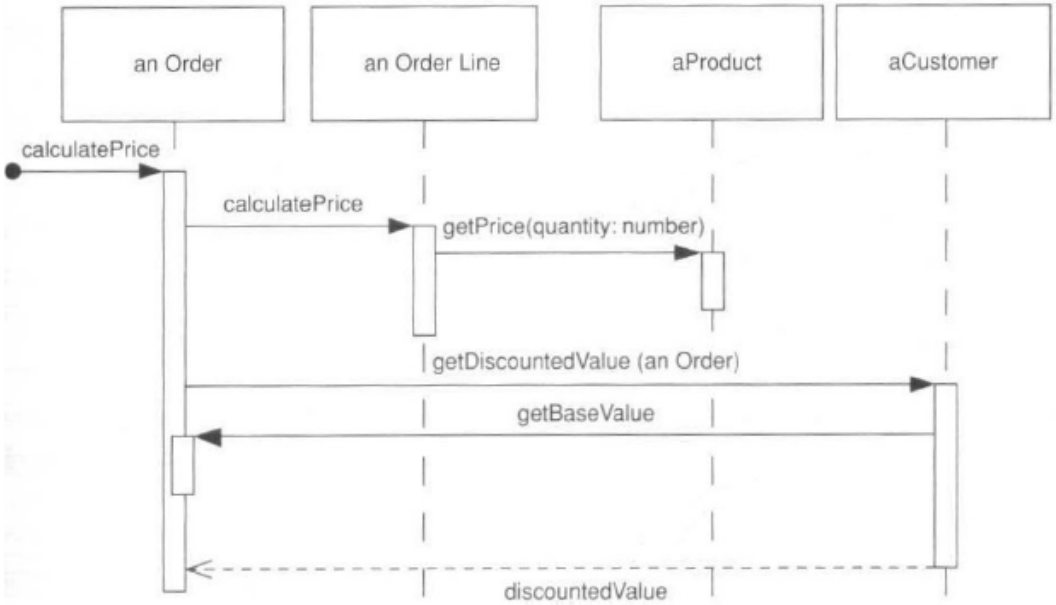
Violación de la Ley de Demeter

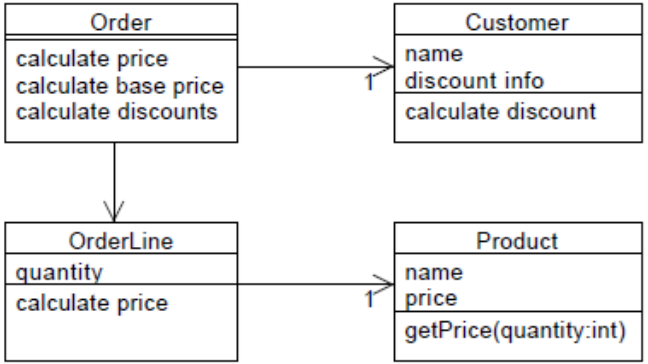




D

Aplicación de la Ley de Demeter





UNIVERSIDAD DE LIMA

Evitar los encadenamientos

```
1  public class DrawEngine
2  {
3    public void Draw(Canvas canvas, Box box)
4    {
5      canvas.SetColor(box.Border.Color);
6      canvas.DrawRectangle(box.Location.X, box.Location.Y,
7                           box.Size.Width, box.Size.Height);
8
9      canvas.SetColor(box.BackColor);
10     canvas.FillRectangle(box.Location.X, box.Location.Y,
11                          box.Size.Width, box.Size.Height);
12   }
13 }
```

Fuente: Ley de Demeter; Tell, Don’t Ask y God Object

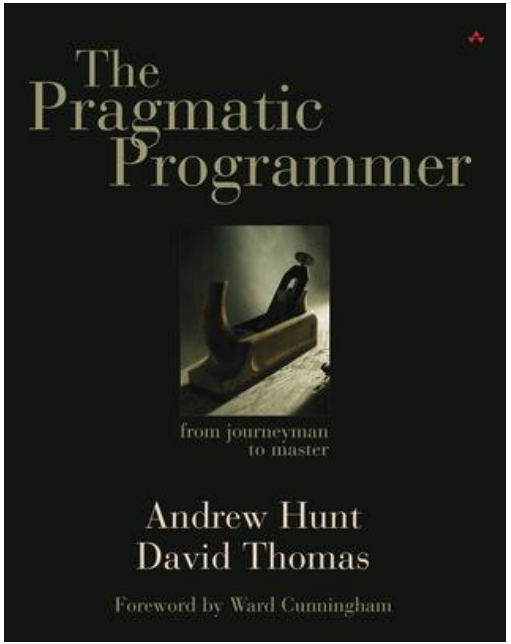
Objetivo: Evitar esto
objectA.getObjectB().doSomething();

Principio DRY

Don’t repeat yourself

- “Todo conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema”

Conocimiento: funcionalidad y/o algoritmo



“El conocimiento de un sistema es mucho más amplio que solo su código. Se refiere a esquemas de base de datos, planes de prueba, el sistema de compilación e incluso documentación”.

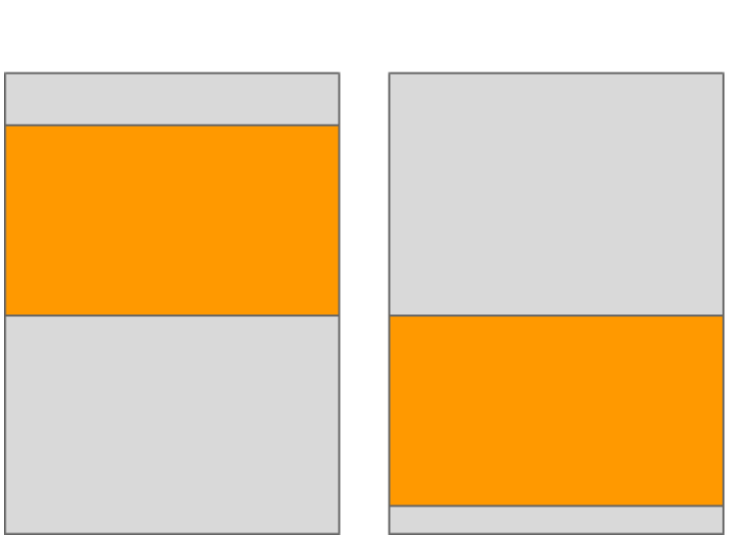
- Dave Thomas -

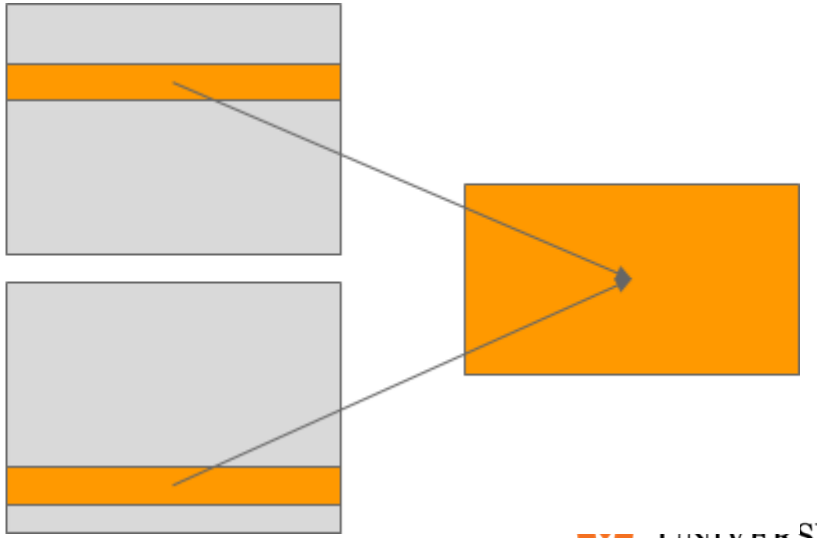
https://media.pragprog.com/titles/tpp20/dry.pdf

UNIVERSIDAD DE LIMA

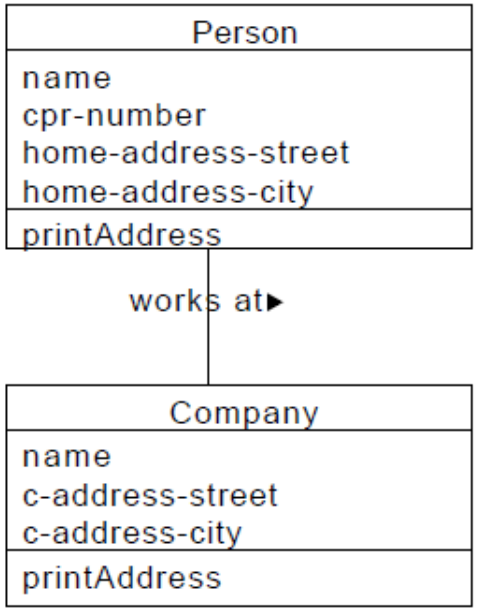
Principio: No te repitas

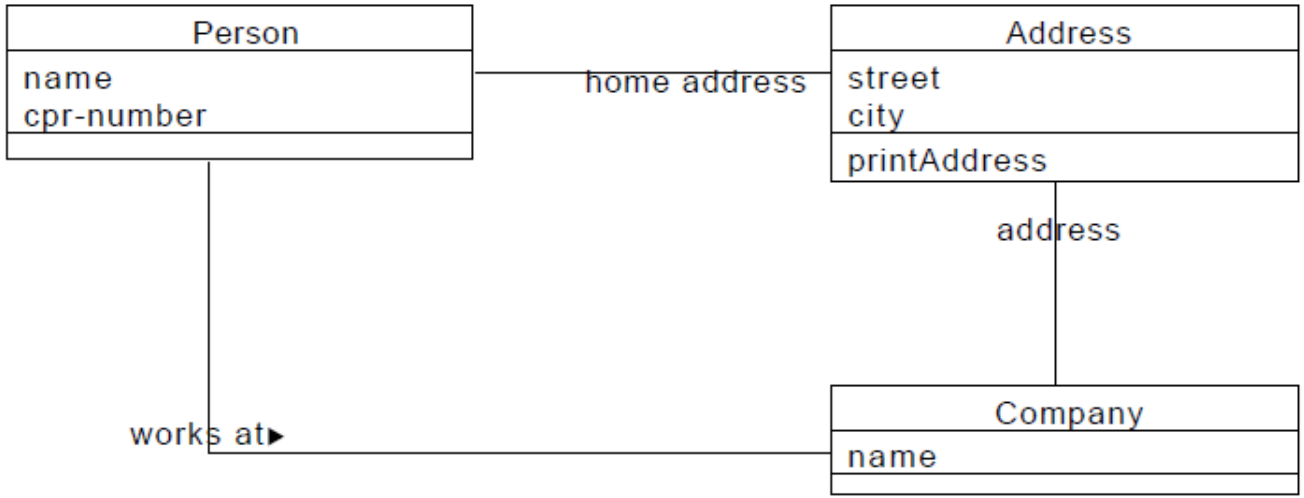
- La aplicación debe evitar especificar el comportamiento relacionado con un concepto particular en múltiples lugares ya que es una fuente frecuente de errores





Ejemplo de código duplicado





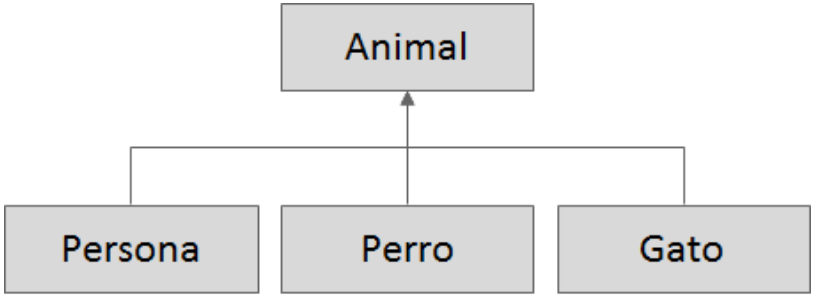
UNIVERSIDAD DE LIMA

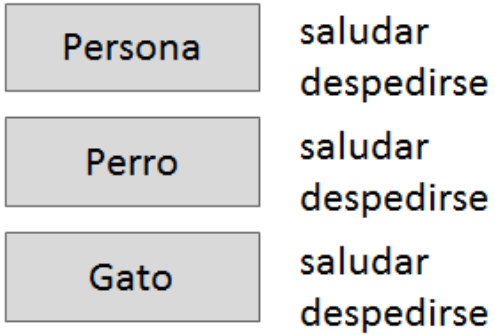
Principio DRY

- Técnicas para evitar duplicados
  - Realizar abstracciones apropiadas
  - Usar Herencia
  - Usar Clases con variables que sean instancias
  - Métodos con parámetros
- Refactorizar para remover duplicados
- Generar artefactos de una fuente común

Duck Typing

- Propiedad de algunos lenguajes de programación de tipeo dinámico.
- Nos permite definir objetos por lo que hacen (sus métodos) más que por lo que son (de quien heredan).
- Contraparte al polimorfismo.





UNIVERSIDAD DE LIMA

DRY

La aplicación debe evitar especificar el comportamiento relacionado con un concepto particular en múltiples lugares ya que es una fuente frecuente de errores

```java
package com.arquitecturajava;

public class Principal {

    public static void main(String[] args) {


        System.out.println("hola");
        System.out.println("hola");
        System.out.println("adios");
        System.out.println("adios");
        System.out.println("hola");
        System.out.println("hola");
        System.out.println("adios");
        System.out.println("adios");
    }

}
```

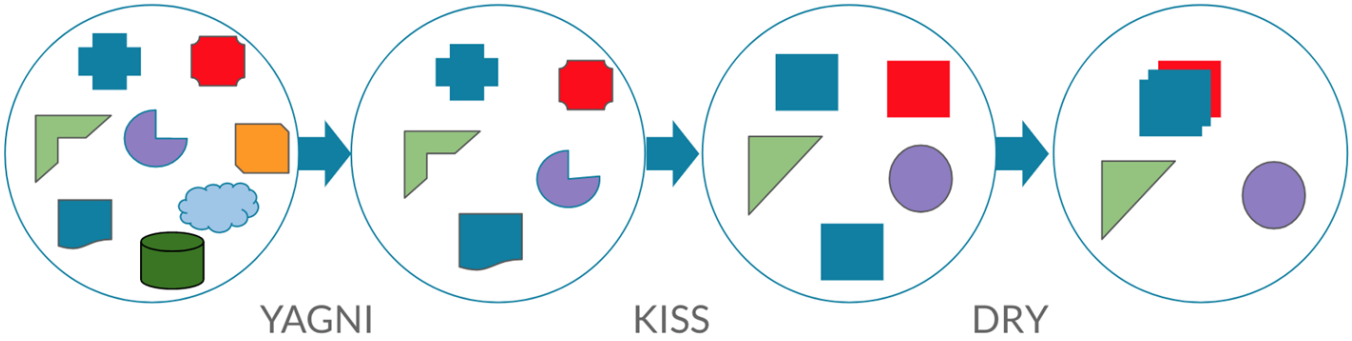
```java
package com.arquitecturajava;

public class Principal {

    public static void main(String[] args) {


        hola();
        System.out.println("adios");
        System.out.println("adios");
        hola();
        System.out.println("adios");
        System.out.println("adios");
    }

    private static void hola() {
        System.out.println("hola");
        System.out.println("hola");
    }

}
```

```java
package com.arquitecturajava;

public class Principal {

    public static void main(String[] args) {


        hola();
        adios();
        hola();
        adios();
    }

    private static void adios() {
        System.out.println("adios");
        System.out.println("adios");
    }

    private static void hola() {
        System.out.println("hola");
        System.out.println("hola");
    }

}
```

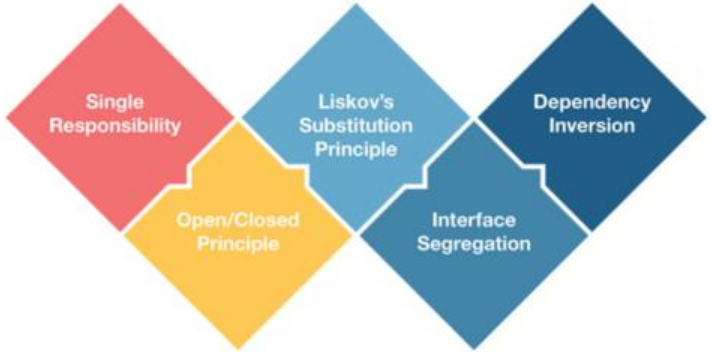
UNIVERSIDAD DE LIMA

Otros principios

YAGNI-KISS-DRY



S.O.L.I.D.



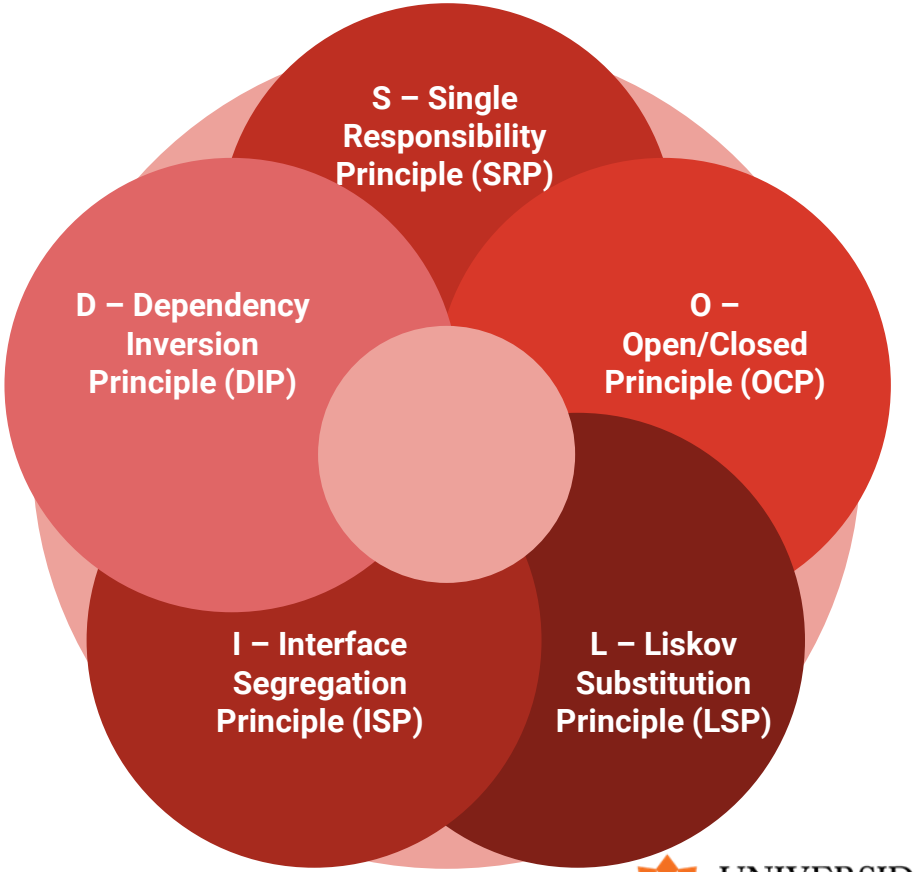
¿Qué son los principios SOLID?

"Principios que facilita a los desarrolladores la labor de crear programas legibles y mantenibles."

Conjunto de principios aplicados al POO que ayudan a desarrollar software de calidad en cualquier lenguaje de programación orientada a objetos. Su intención es eliminar los malos diseños, evitar la refactorización, construir código más eficiente, fácil de testear y mantener. Código con:

- Bajo Acoplamiento
- Alta Cohesión

Los principios SOLID



S

Principio de Responsabilidad Única

Es el Single Responsibility Principle, también conocido por sus propias siglas en inglés SRP.

Este principio establece que cada clase debe tener una única responsabilidad dentro de nuestro software.

- ➔ **Definida**
- ➔ **Concreta**

Definir la responsabilidad única de una clase no es una tarea fácil, será necesario un análisis previo de las funcionalidades y cómo estructuramos la aplicación.

Tips para saber si no estamos cumpliendo con este principio

- En una misma clase están involucradas dos capas de la arquitectura
- Nos cuesta testear la clase
- Por el número de líneas

```
1
2  CalculationService calculationService = new CalculationService();
3  PrintService printService = new PrintService();
4
5  Circle circle = new Circle(5);
6  Square square = new Square(6);
7
8  double result = calculationService.sumAreas(circle, square);
   printService.printResult(result);

}
```

O

Principio abierto / cerrado

El principio de abierto-cerrado (OCP, por sus siglas en inglés).

Este principio establece que una entidad de software (clase, módulo, función, etc) debe quedar abierta para su extensión, pero cerrada para su modificación.

- **Abierto a extensión:** quiere decir tienes que ser capaz de añadir nuevas funcionalidades.
- **Cerrado a modificación:** quiere decir que para añadir la nueva funcionalidad no tienes que cambiar código que ya está escrito.

Beneficios

Las ventajas que nos ofrece diseñar el código aplicando este principio

- Es un software más fácil de mantener al minimizar los cambios en la base de código de la aplicación y de ampliar funcionalidades sin modificar partes básicas de la aplicación probadas.
- También vemos las ventajas a la hora de implementar test unitarios en nuestro software

Ejemplo

```
class CalculationService {
    public void getArea(Polygon p) {
        return p.area();
    }
 }

 class Polygon {
    abstract void area();
 }


class Square extends Polygon {
int side;

    public Square(int side) {
this.side = side;
}

    public void area() {
        return Math.pow(side,2);
    }
}
```

```
    class Polygon {
       int type;
    }
```

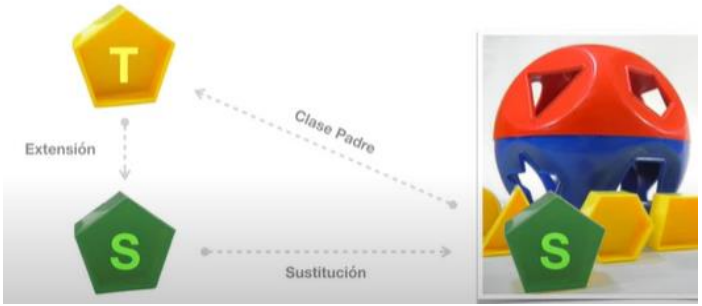
```
class Circle extends Polygon {
int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public void area() {
        return Math.PI * Math.pow(radius,2);
    }
 }
```

```
class Triangle extends Polygon {
int base;
int height;

    public Triangle(int base, int height) {
        this.base = base;
this.height = height;
}

    public void area() {
        return base*height/2;
    }
 }
```

```
    }
```

UNIVERSIDAD DE LIMA

Principio de Sustitución de Liskov



Definición matemática

Si para cada objeto o1 de tipo S hay un objeto o2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o1 es sustituido por o2, entonces S es un subtipo de T.

En otras palabras, el principio nos dice que si en alguna parte de

nuestro código estamos usando una clase y esta es extendida,

tenemos que poder utilizar cualquier clase hija y el programa

debe seguir funcionando

¿Cómo saber si estamos fallando este principio?

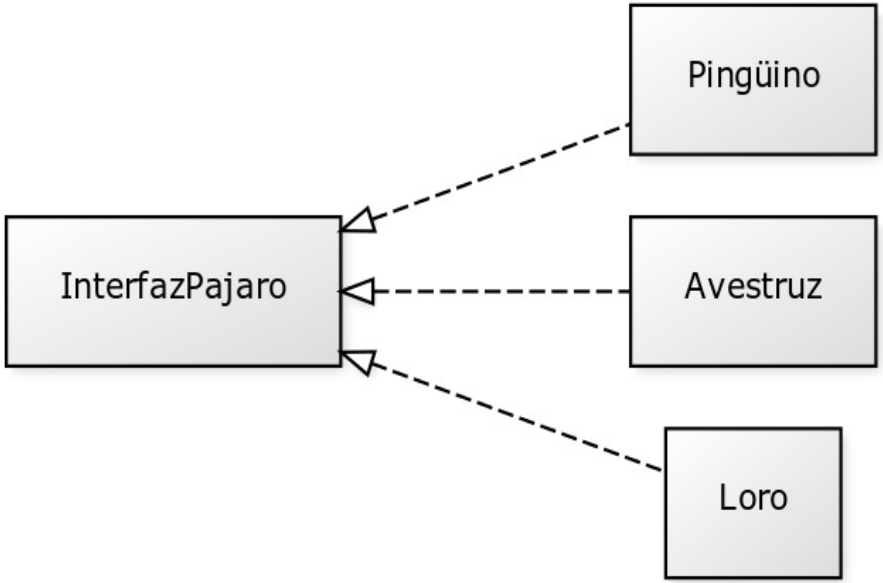
- La principal manera de darnos cuenta de que esto ha sucedido es que creamos una clase que extiende a otra y de repente nos sobra un método
- Si un método sobreescrito no hace nada o lanza una excepción.

```
open class Animal {
    open fun walk() {}
    open fun jump() {}
}

fun jumpHole(a: Animal) {
    a.walk()
    a.jump()
    a.walk()
}

class Elephant : Animal() {
    override fun jump() {
        throw Exception("Un elefante no puede saltar")
    }
}
```

Es así que el principio de Liskov nos ayuda a utilizar las herencias de manera **correcta** y saber como extender las clases. Se debe comprender que no hay una modelización de todos los aspectos de la vida real, así que este principio nos ayudará a hacerlo.

Figura 1

I



Segregación de Interfaces

“Ninguna clase debería depender de métodos que no usa”

“Las interfaces nos ayudan a desacoplar módulos entre sí”

“La problemática surge cuando las interfaces intentar definir más cosas de las debidas” **Fat Interfaces**

¿Cómo detectar que estamos fallando este principo?

Si tenemos una interfaz que tiene algunos métodos cuyas clases hijas no están implementando porque no lo necesitan. entonces estamos violando el principio de segregación

Si la interfaz forma parte del código y se la pueda modificar, entonces lo ideal es dividir la interfaz en varias pequeñas para que cuando las clases la implementen no queden funciones vacías

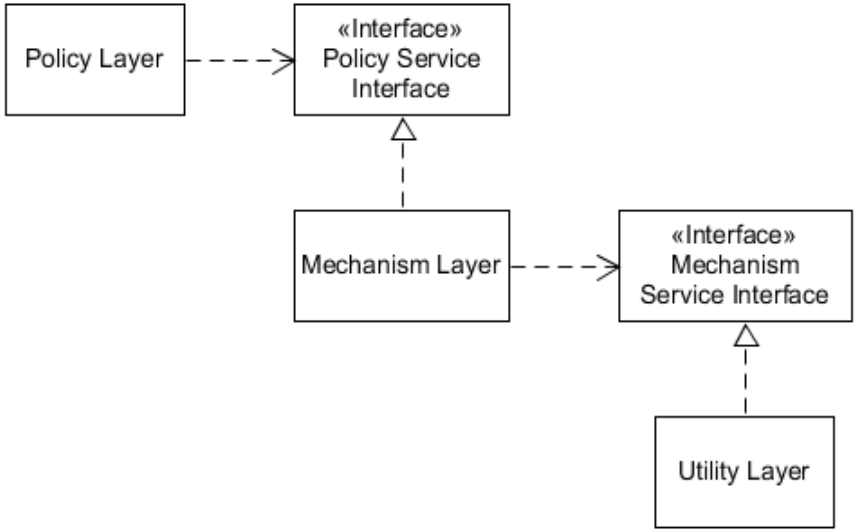
El Principio de Segregación de Interfaces ayuda a poder crear nuestra arquitectura utilizando la composición de protocolos o interfaces, evitando así los *Fat Interfaces* que en la mayoría de los casos obligan a dejar métodos vacíos o lanzar errores en aquellos métodos que no tiene sentido implementar.

```
protocol Animal{
    func run()
}

class Lion: Animal{
    func run(){
        print("Lion corriendo")
    }
}
class Dog: Animal{
    func run(){
        print("Dog corriendo")
    }
}
class Cat: Animal{
    func run(){
        print("Cat corriendo")
    }
}
```

```
protocol Animal {
    func run()
    func speak()
}
class Lion: Animal {

    ...

    func speak() {
        print("Roarrrr 🦁")
    }
}
class Dog: Animal {

    ...

    func speak() {
        print("Guau! 🐶")
    }
}
class Cat: Animal {

    ...

    func speak() {
        print("Miau 🐱")
    }
}
```

```
protocol Animal {

    func run()
    func speak()
    func swim()
}
class Salmon: Animal {

    func run() {
        fatalError("Salmons can NOT run")
    }

    func speak() {
        fatalError("Salmons can NOT speak")
    }

    func swim() {
        print("Swiming... 🐟  ")
    }
}
```

D



Inversión de Dependencias

““Depende de abstracciones, no de clases concretas”.”

clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.

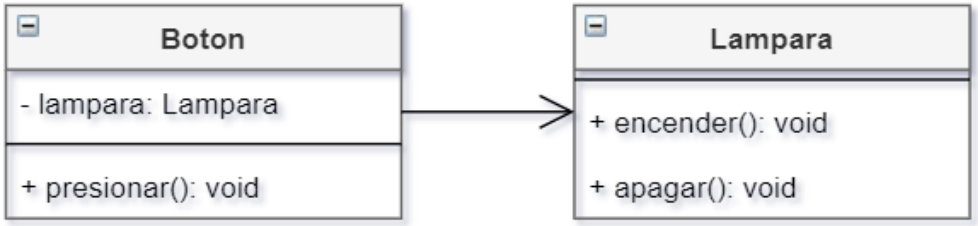
Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

El problema

En la programación vista desde el modo tradicional, cuando un módulo depende de otro módulo, se crea una nueva instancia y la utiliza sin más complicaciones

Las parte más genérica de nuestro código (lo que llamaríamos el dominio o lógica de negocio) dependerá por todas partes de detalles de implementación.
No quedan claras las dependencias
Es muy complicado hacer tests

Ejemplo



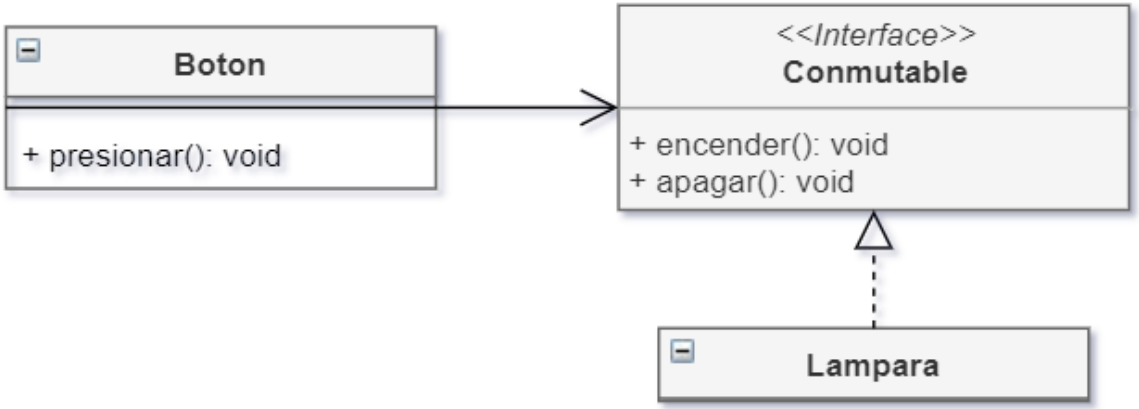
- La política de alto nivel de la aplicación no ha sido separada de la implementación de bajo nivel.
- Las abstracciones no han sido separadas de los detalles
- La política de alto nivel automáticamente depende de los módulos de bajo nivel, las abstracciones automáticamente dependen en los detalles.

Boton depende de Lámpara

Si Lampara cambia, Boton cambiará también.

Boton no es reusable

No podrías usar presionar() para encender una Lavadora, por ejemplo.



Solución

Creemos una capa intermedia en donde definiremos una interfaz abstracta asociada a Boton e implementada por cualquier clase como Lampara.

De este modo:

- **Boton** depende únicamente de abstracciones, puede ser reusado en varias clases que implementen Conmutable. Por ejemplo, Lavadora o Ventilador
- Cambios en **Lampara** no afectarán a **Boton**.
- ¡Las dependencias han sido invertidas! ahora Lampara tiene que adaptarse a la interfaz definida por **Boton**. Lo cual tiene sentido, ya que la idea fundamental es que lo más importante no dependa de lo menos importante.

Referencias Bibliográficas

1. Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.
2. Martin, R. C., Newkirk, J., & Koss, R. S. (2003). Agile software development: principles, patterns, and practices (Vol. 2). Upper Saddle River, NJ: Prentice Hall.