Tecnología de Programación

Martín L. Larrea

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Redefiniciones

Recordemos que una subclase puede agregar nuevas operaciones o proveer nuevas versiones de las operaciones existentes en la superclase *(redefinir)*

En todos los lenguajes orientados a objetos,
para redefinir un método u operación,
el *signature* debe ser el mismo.

A veces se agrega una nueva versión con tipos diferentes, para sobrecargar el operador.

Sin embargo, pueden existir situaciones en donde
el tipo de los atributos u operaciones
debe/puede cambiar y aún así considerarse redefinición.

Esto depende de cada implementación y del sistema de tipos del lenguaje.

Operaciones redefinibles

La redefinición de operaciones no siempre es directa como en Java

En C++, por ejemplo, pueden redefinirse únicamente
aquellas operaciones declaradas como virtuales.
Este tipo de distinciones también se utiliza en otros lenguajes.

Las operaciones virtuales puras son aquellas que no tienen código (en Java, operaciones abstractas).

```
class A {
  public:
    virtual void f() {…}
    virtual void h()=0
    void g(int n){…}
  …
};
```

```
class B :public A {
  public:
    void f() {…}
    void g(int n){…};
  …
};
```

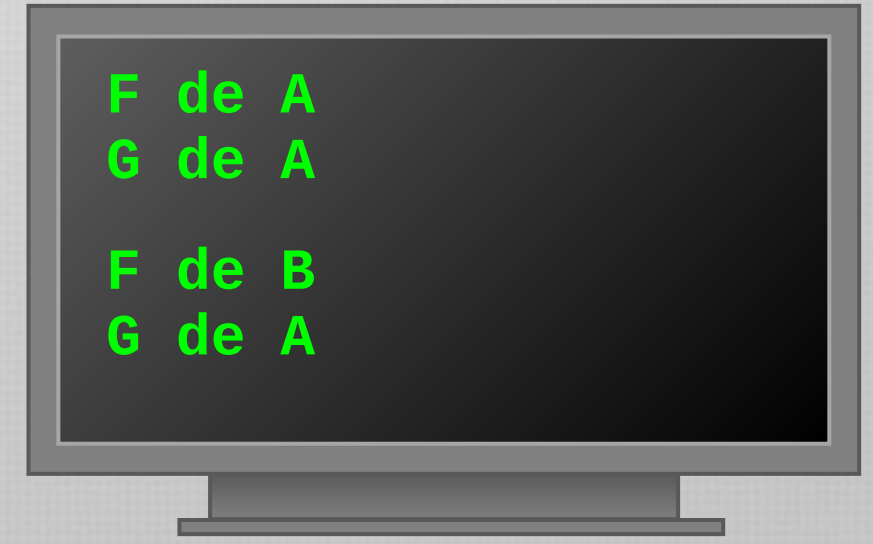
f() esta redefinida.
g() no.
¿Qué cambia en esta distinción?

Operaciones redefinibles - C++

```
class A {
 public:
  virtual void f()
   {cout << “F de A” << endl;}
  void g(int n)
   {cout << “G de A” << endl;}
…
};
```

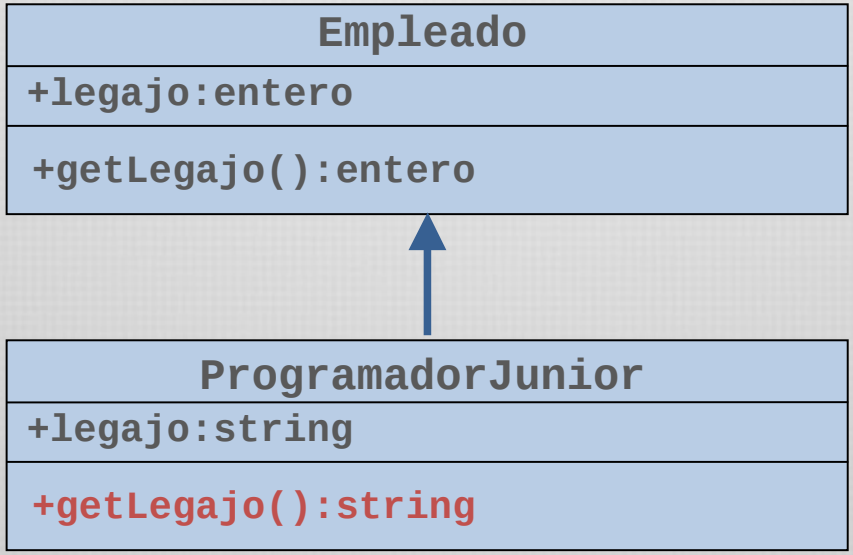
```
class B :public A {
 public:
  void f()
   {cout << “F de B” << endl;}
  void g(int n)
   {cout << “G de B” << endl;}
  …
};
```

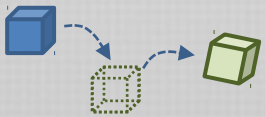
```
…
A *a;
B *b;
A= new A();
B= new B();

a->f();
a->g();
a=b;
a->f();
a->g();
```



No se puede cambiar por cualquier tipo

Claramente, la redefinición del tipo de atributos u operaciones no es completamente libre...

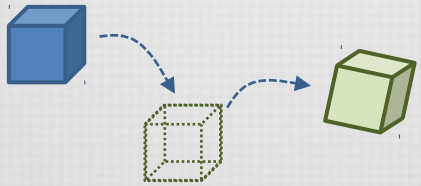




ESTO ES INVALIDO COMO REDEFINICION
Las instancias de ProgramadorJunior no podrían reemplazar a instancias de Empleado.

¿Que hace Java en este caso?

Redefinición y polimorfismo



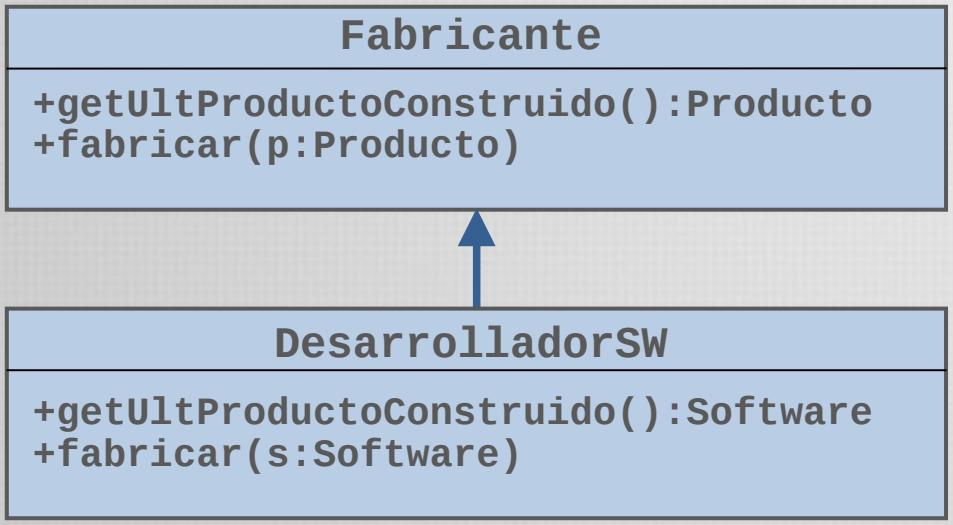
La redefinición no debe *traicionar* a los clientes cuando hay *accesos polimórficos*...

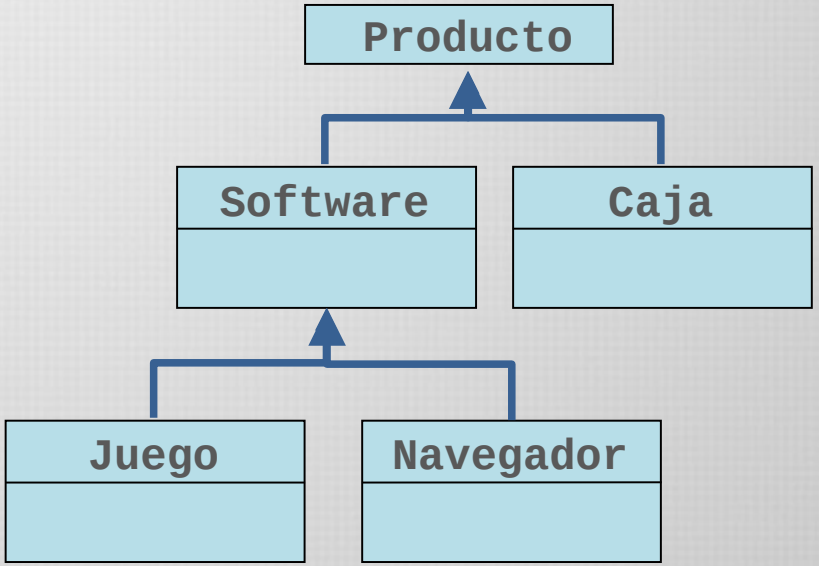
```
f: Fabricante;
p: Producto;

p ← f.getUltProductoConstruido();
```

Aquí se espera obtener
un objeto de tipo
`Producto`

Cualquier clase que herede de **`Fabricante`** y
desee redefinir **`getUltProductoConstruido`**,
debe satisfacer este "*contrato*".
Por lo tanto debe proveer un objeto que sea instancia de
`Producto`

Redefinición y polimorfismo





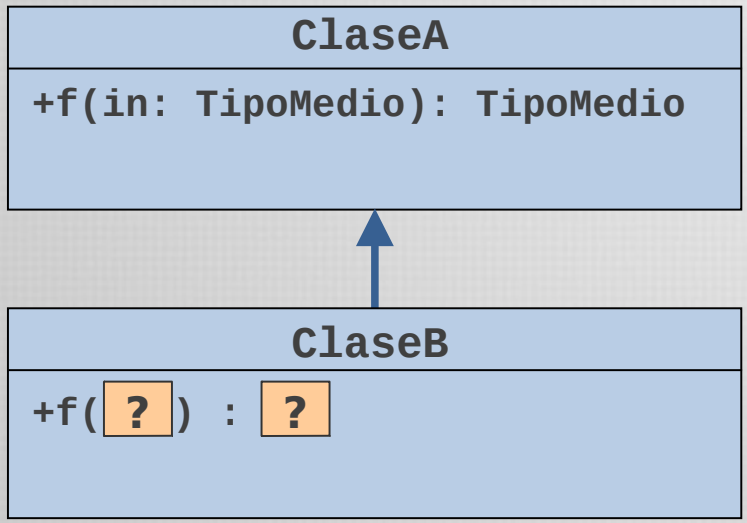

```
f: Fabricante;
d: DesarrolladorSW
p: Producto; s: Software; j:Juego;
c: Caja
...
p ← f.getUltProductoConstruido();
f.fabricar(j);
f.fabricar(c);
f ← d
f.fabricar(c);
p ← f.getUltProductoConstruido();
s ← (Software)f.getUltProductoConstruido();
d.fabricar(j);
```

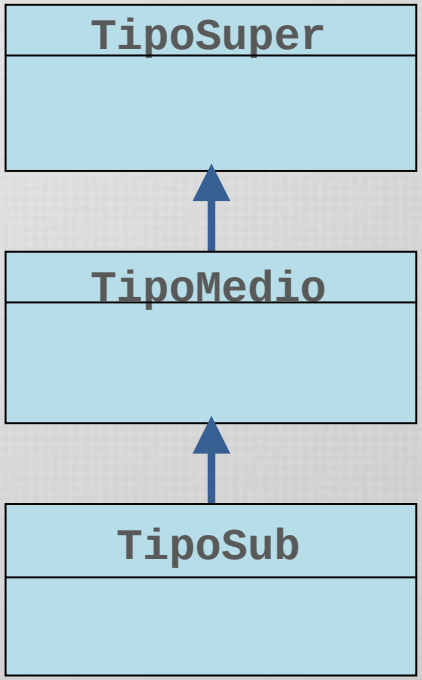
Problemas:

```
...
f.fabricar(c);
f ← d
f.fabricar(c);
```

¿Que hace Java en este caso?

Covarianza, contravarianza, invarianza





Covarianza, contravarianza, invarianza

Es una muestra más del poder de la herencia y lo complicado que es flexibilizarla. Mayor libertad, mas casos críticos a tener en cuenta.

C++ prohibe la redefinición de operaciones al menos que tengan exactamente el mismo encabezado.

Java admite covarianza en el resultado de una función, y en los parámetros.

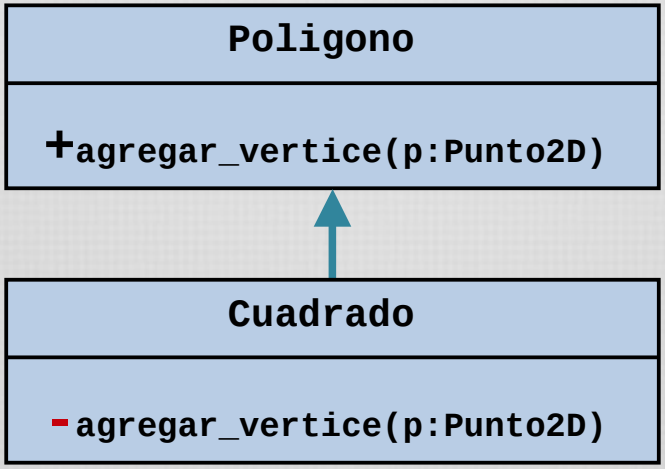
Eiffel utiliza tipos “anclados” (anchored). Puede indicar que ciertos atributos cambian de tipo al cambiar de tipo otras entidades:

```
class Esquiador feature
    companero: like Current

    comparte_hab_con(e: like Current) is
        -- Asigna un compañero
    do
        companero := e
    end
    ...

end -- class Esquiador
```

Esto evita los problemas anteriores, ya que el control de tipos se hace sobre ***like Current***.

Herencia y visibilidades

¿Qué ocurre con la redefinición de visibilidades?




```
p: Poligono; r:Rectangulo; punto1,punto2:Punto2D
...
p.agregar_vertice(punto1);
p ← r
p.agregar_vertice(punto2);
```

Herencia y visibilidades

Como regla general, la visibilidad de las operaciones puede redefinirse únicamente a un grado más permisivo.

privado puede pasar a público
privado puede pasar a protegido
protegido puede pasar a público

Cualquier cambio más restrictivo contradice los principios de sustitución de la orientación a objetos.

Herencia y encapsulamiento

Otro inconveniente: la herencia relativamente *quiebra* el encapsulamiento

```
class Contador {
 private int valor;
    ...
 void inc(){
   valor++;
 }

 void inc2(){
   valor=valor+1;
 }

}
```

```
class ContadorBeep extends Contador{
    ...
    void inc2() {
        inc();
        beep();
    }

}
```

Herencia y encapsulamiento

Otro inconveniente: la herencia relativamente *quiebra* el encapsulamiento

```
class Contador {
 private int valor;
    ...
 void inc(){
   inc2();
 }

 void inc2(){
   valor=valor+1;
 }

}
```

```
class ContadorBeep extends Contador{
    ...
    void inc2() {
        inc();
        beep();
    }

}
```

Una subclase necesita saber bastante de la superclase

Usos de la herencia

No siempre es fácil identificar cuándo utilizar la herencia.

Por lo general conviene observar la relación entre clases.
La más frecuente es la relación "*es un*".

Sin embargo, no hay que confundir con la relación instancia-clase.

Asociamos la herencia con la relación "es-un" (is-a) cuando hablamos de clases como agrupaciones de objetos.

Un aeródromo es un aeropuerto
que acepta aviones de menor tamaño,
bajo regulaciones más flexibles.
Esto puede corresponder a una relación de herencia

Ezeiza es un Aeropuerto,
pero esto corresponde más a la relación instancia-clase

Usos de la herencia

Bertrand Meyer distingue algunos tipos generales de herencia, que califica como “usos validos” de esta técnica.

Tipos de Herencia

Herencia de modelo

Representa la relación “es un” entre las abstracciones del modelo.

(herencia de extensión, de restricción, de subtipo, etc)

Herencia de software

Representa relaciones en el software, sin vinculación necesaria con el modelo.

(herencia de implementación, de facilidades, etc)

Herencia de variación

Representa la descripción de una clase de acuerdo a las diferencias con otra clase.

(variación funcional, variación de tipos)

Problema

```
public void DibujarFigura(Figura s) {

    if (s instanceOf Cuadrado) {
        s.dibujarCuadrado();
    } else
    if (s instanceOf Circulo) {
        s.dibujarCirculo();
    }

}
```

Esta operación debe conocer todos los tipos de datos derivados de la clase **`Figura`** y debe actualizarse cada vez que se agrega un tipo nuevo.

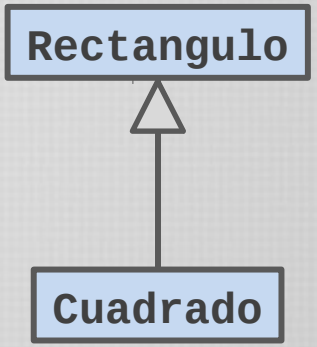
Esta operación no se abstrae de las figuras

(aunque pareciera querer hacerlo, dado el parámetro)

Problema

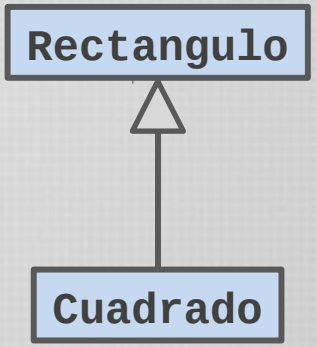
```
class Rectangulo
{
    public:
      void SetAncho(double w) {ancho=w;}
      void SetAlto(double h) {alto=w;}
      double GetAncho() {return ancho;}
      double GetAlto() {return alto;}
    private:
      double ancho;
      double alto;
    };
```

```
class Cuadrado : public Rectangulo {

};
```



Problema

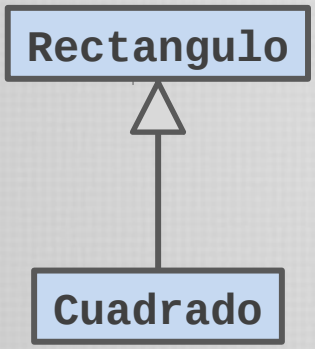
```
class Rectangulo
{
    public:
      void SetAncho(double w) {ancho=w;}
      void SetAlto(double h) {alto=w;}
      double GetAncho() {return ancho;}
      double GetAlto() {return alto;}
    private:
      double ancho;
      double alto;
    };
```




```
class Cuadrado : public Rectangulo {

  void SetAncho(double w) {
       ancho=w; alto=w;
  }
  void SetAlto(double h) {
       ancho=w; alto=w;
  }

};
```

```
c = new Cuadrado();
c->setAncho(3);
c->setAlto(10);
```

Problema



En otro lugar:

```
public double shrink(Rectangulo* r)
{ r->SetAncho(50);
}

c = new Cuadrado();
shrink(c);
```

Como las operaciones no fueron declaradas *virtual*,
entonces el objeto **Cuadrado** queda con los lados diferentes.
Luego, el **Cuadrado** no sustituye al **Rectangulo** como se espera.

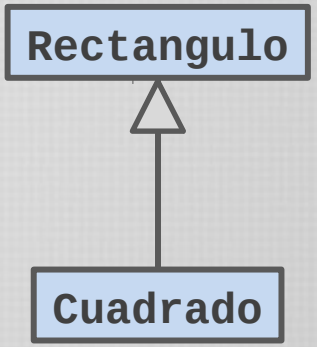
¿solución?
*Declarar las operaciones en **Rectangulo** como virtual.*

o sea, cambios en la clase base porque creamos una clase derivada.
Algo no está bien en el diseño.

Solución?

```
class Rectangulo
{
    public:
      void virtual SetAncho(double w)
    {ancho=w;}
      void virtual SetAlto(double h) {alto=w;}
      double GetAncho() {return ancho;}
      double GetAlto() {return alto;}
    private:
      double ancho;
      double alto;
    };
```

```
class Cuadrado : public Rectangulo {

  void SetAncho(double w) {
       ancho=w; alto=w;
  }
  void SetAlto(double h) {
       ancho=h; alto=h;
  }

};
```




```
c = new Cuadrado();
c.setAncho(3);
c.setAlto(10);
```

Problema

¿Que estuvo mal?
¿Acaso un cuadrado no es un rectángulo?

Un cuadrado es un rectángulo
pero según su uso un **`Cuadrado`** no es un **`Rectangulo`**

En orientación a objetos, la relación "es un" hace referencia al comportamiento.
Comportamiento público, del que dependen los clientes

En el ejemplo anterior,
Cuadrado es una subclase de Rectangulo
pero
Cuadrado no es un subtipo de Rectangulo.

La noción de subtipo se basa en el comportamiento,
no en si figura la palabra "extends" o "implements"

Liskov

Barbara Liskov
Institute Professor - MIT

Primera mujer con un doctorado en Universidad de Stanford
Turing Award – 2008
Doctorado Honorario de Eidgenössische Technische Hochschule
Creó dos lenguajes de programación: *CLU* y *Argus*

Principio de Sustitución

If for each object o_1 of type S there is an object o_2 of type T
such that
for all programs P defined in terms of T,
the behavior of P is unchanged
when o_1 is substituted for o_2
then S is a subtype of T.

Principio de Sustitución

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

Esto implica:

para los métodos:

- Debe haber un método en el subtipo por cada método en el supertipo.
- Los métodos del subtipo no requieren nada adicional para funcionar.
- Los métodos del subtipo no ofrecen menos que los del supertipo.

para las propiedades:

- Todas deben ser garantizadas por el supertipo

Las aserciones nos permitirán trasladar estos requerimientos al código.
Volveremos luego sobre esto.

Principios de diseño

El principio de sustitución es un ejemplo de principio de diseño

Forma parte, además, de un conjunto de principios conocidos como SOLID

S	*Single Responsability Principle*
O	*Open-Closed Principle*
L	*Liskov Substitution Principle*
I	*Interface Segregation Principle*
D	*Dependency Inversion Principle*

Fueron definidos por Robert C. Martin y Michael Feathers

Single Responsability Principle

El principio de Única Responsabilidad establece que una clase debería poseer sólo una responsabilidad puntual en el sistema.

"*Nunca debería haber más de una razón por la que una clase puede modificarse*"

Una clase que descarga videos de Internet y los muestra en un sector de la interfaz.

Una clase Juego que registra los puntajes del jugador y muestra los gráficos del juego.

Una clase que guarda un pedido de mercadería en la Base de Datos y notifica a todas las dependencias de la empresa.

"*Cada responsabilidad es una razón de cambio*"

Open-Closed Principle

El principio de Abierto-Cerrado establece que una
clase debería ser abierta a extensiones, pero cerrada a modificaciones.

Lo hemos visto anteriormente como un Principio de Modularidad por Bertrand Meyer.

Los módulos (clases) que cumplen este Principio exhiben dos cualidades:

Abiertos para Extensión
El módulo puede ser extendido para comportarse de otra forma, según cambios en los requerimientos.

Cerrados para Modificación
No se admiten cambios en el código, ni son necesarios.

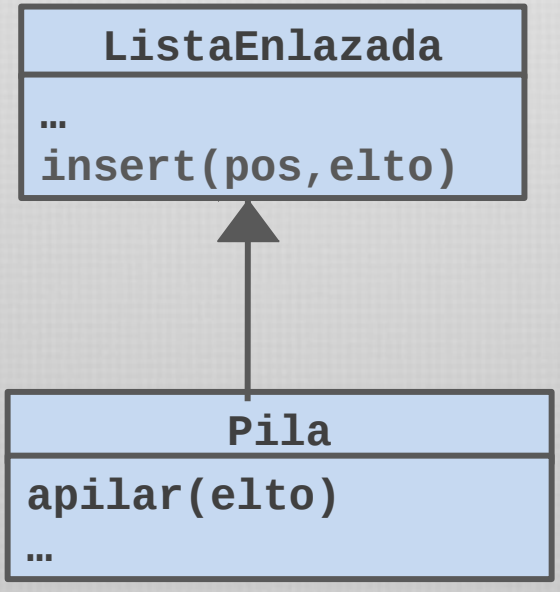
¡El **`instanceOf`** quiebra este principio!

Algunos desarrolladores sugieren incluso evitar el cast.
Por otro lado, algunos desarrolladores sugieren que todos los atributos sean privados.

Interface Segregation Principle

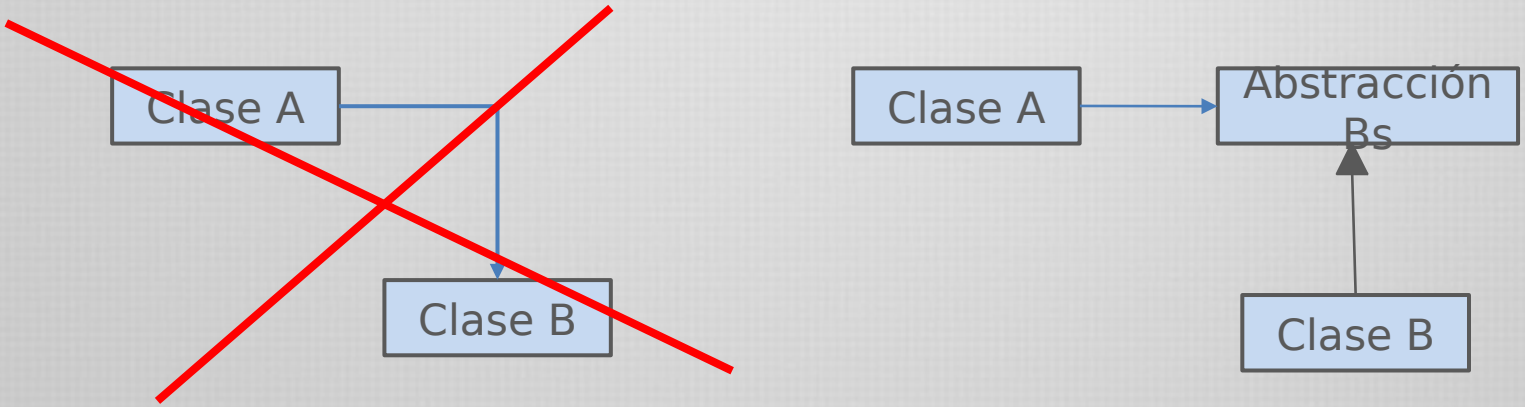
El principio de Segregación de Interfaces establece que
las interfaces deben ser lo más cohesivas posibles.
Una clase no debería "comprar" interfaces que no va a utilizar

Un sindrome habitual es la contaminación de interfaces.



Dependency Inversion Principle

El principio de Inversión de Dependencias establece reglas de desacoplamiento entre clases.
Sugiere depender de abstracciones prioritariamente.



Los módulos de alto nivel no deben depender de módulos de bajo nivel.
Ambos deben depender de abstracciones

Las abstracciones no deben depender de detalles.
Los detalles deben depender de abstracciones.