



Tecnología de Programación



Principios de diseño **SOLID**



Dr. Federico Joaquín 
federico.joaquin@cs.uns.edu.ar

Algunos derechos reservados

Principios de diseño SOLID.



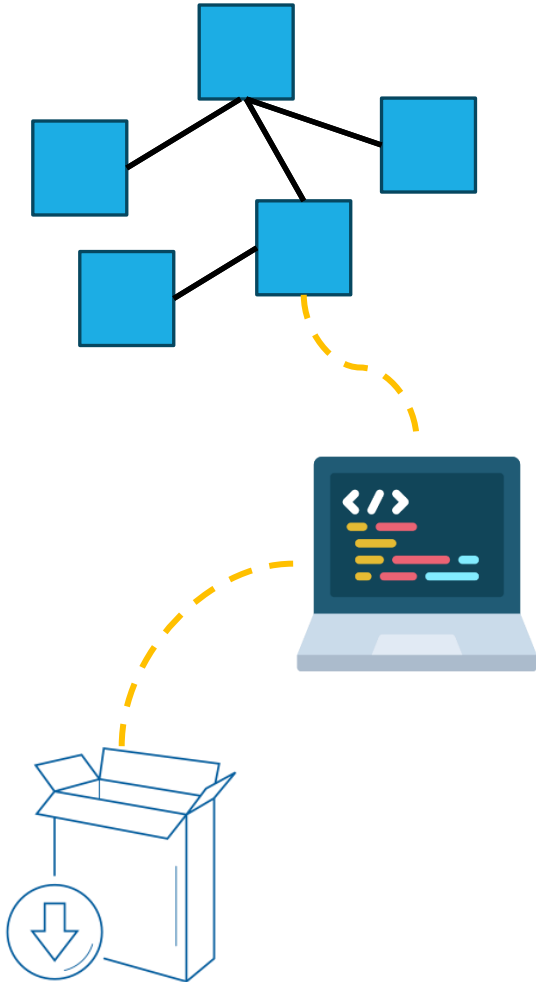
© Octubre 2025 por Federico Joaquín

Esta obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional

Repaso

¿QUÉ HERRAMIENTAS ANALIZAMOS HASTA AQUÍ?

Repaso



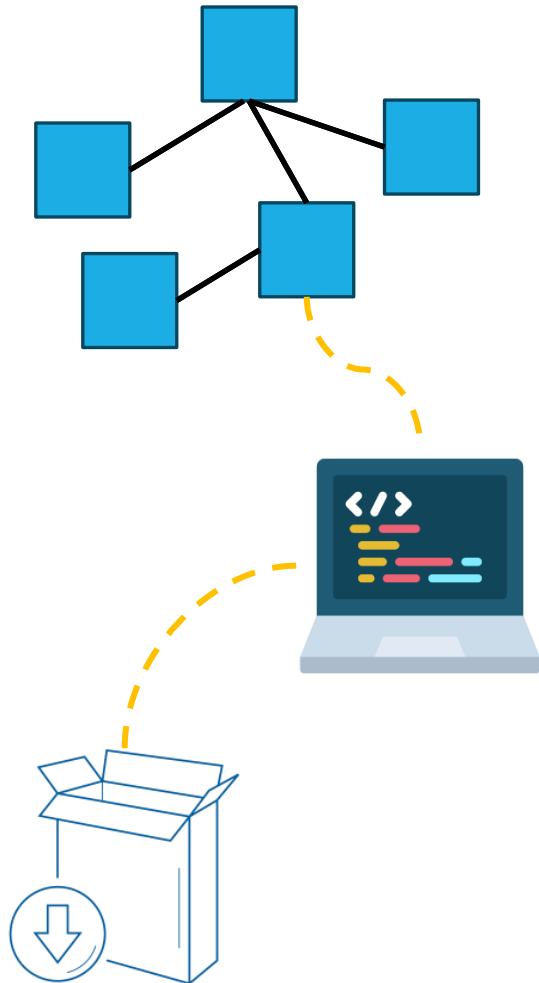
viernes, 3 de octubre de 2025

Paradigma Orientado a Objetos

- La programación orientada a objetos es un **paradigma** de programación que procura **favorecer** un **buen diseño modular**.

Una forma de pensar un problema y su correspondiente solución

Repaso



viernes, 3 de octubre de 2025

Paradigma Orientado a Objetos

- La programación orientada a objetos es un **paradigma** de programación que procura **favorecer** un **buen diseño modular**.
- Recordemos que los **principales objetivos** del POO son:

Confiabilidad

Minimizar la cantidad de errores.

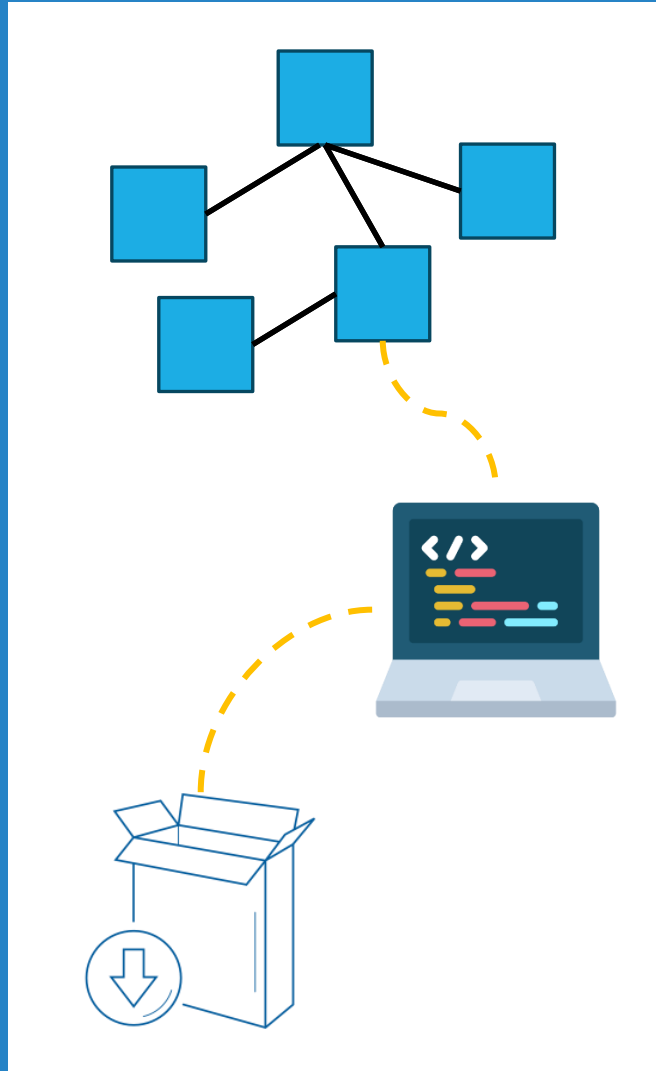
Extensibilidad

Facilidad para introducir nuevos cambios sin mayores esfuerzos.

Reusabilidad

Posibilidad y facilidad de reutilizar cosas.

Repaso



viernes, 3 de octubre de 2025

Paradigma Orientado a Objetos

- La programación orientada a objetos es un **paradigma** de programación que procura **favorecer** un **buen diseño modular**.
- El POO define **tres mecanismo claves**:

Encapsulado

+

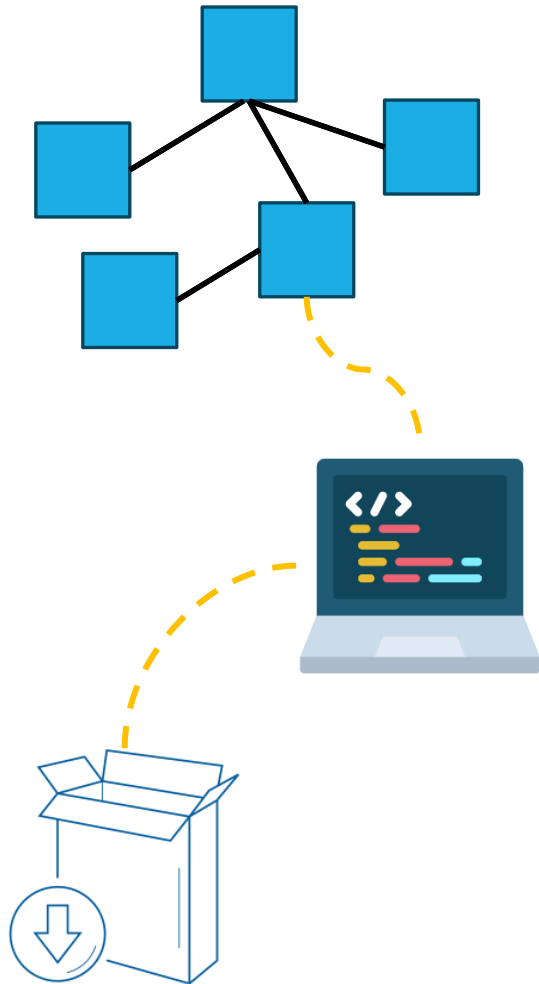
Polimorfismo

+

Herencia

- Estos **tres mecanismos** son generalmente aceptados como los ingredientes **necesarios** de la orientación a objetos (no así los conceptos).

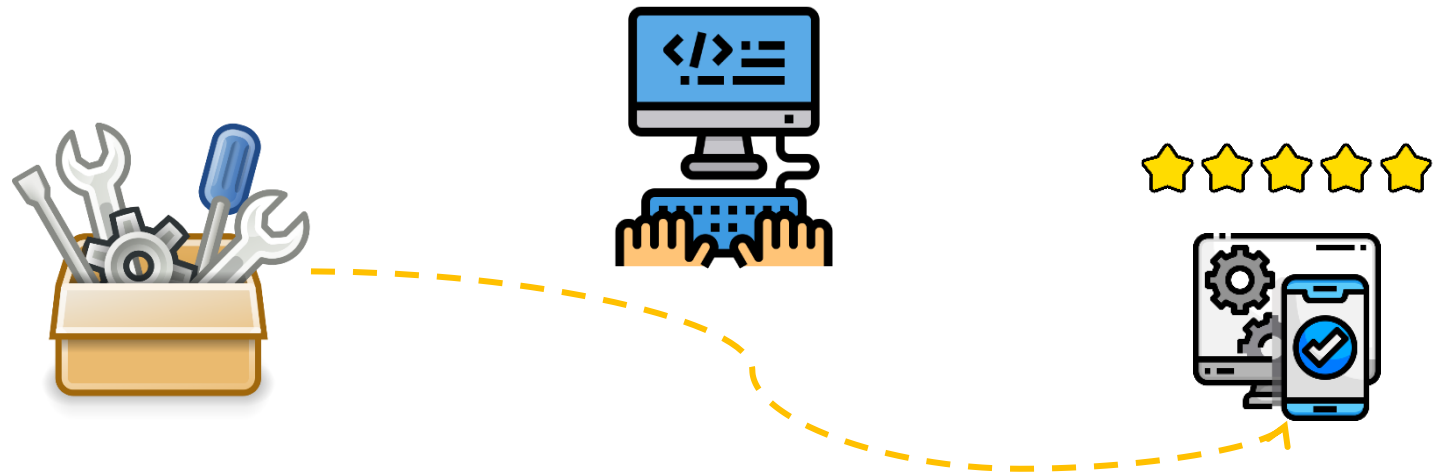
Repaso



viernes, 3 de octubre de 2025

Paradigma Orientado a Objetos

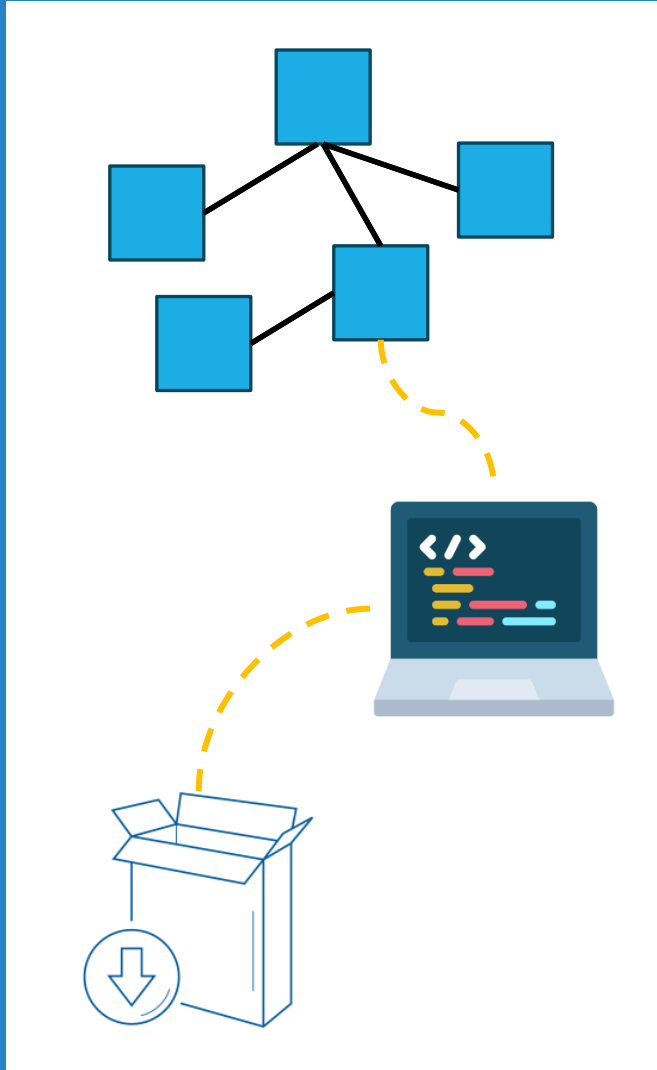
- La programación orientada a objetos es un **paradigma** de programación que procura **favorecer** un **buen diseño modular**.
- El **POO brinda herramientas**, pero no son las herramientas **las que hacen** a la **calidad del SW**, sino **lo que hacemos** con esas herramientas.



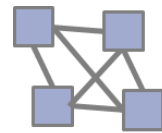
Principios de diseño SOLID

Repaso

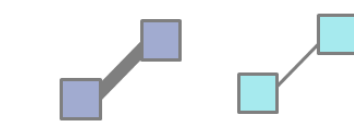
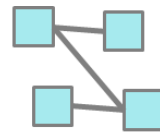
Reglas propuestas por Bertrand Meyer



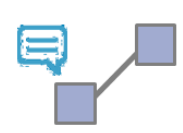
viernes, 3 de octubre de 2025



Pocas Interfaces



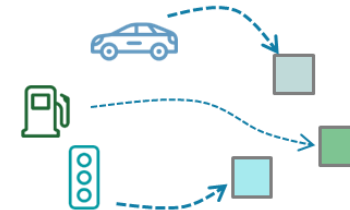
Interfaces Pequeñas



Interfaces explícitas

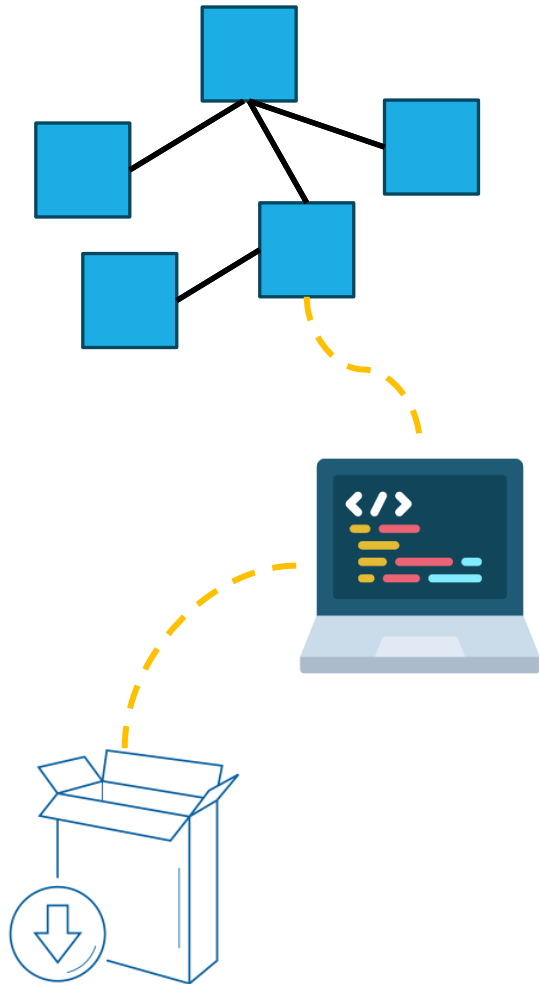


Ocultamiento de
Información



Mapeo Directo

Repaso



viernes, 3 de octubre de 2025

Criterios propuestos Bertrand Meyer

Descomposición modular

Ayuda a descomponer el problema en subproblemas simples y suficientemente independientes.

Comprensión modular

Ayuda a producir software en el cual el humano puede comprender cada módulo sin tener que comprender mucho de los restantes.

Protección modular

Favorece la contención de situaciones anormales en ejecución a uno o pocos módulos.

Composición modular

Favorece la producción de elementos de software que pueden combinarse para producir nuevos sistemas.

Continuidad modular

Favorece que un cambio pequeño en la especificación impacte en uno o pocos módulos.

A continuación



Temas de esta clase

- El paradigma OO procura **favorecer** un **buen diseño modular**.
- El POO **brinda herramientas**, pero no son las herramientas **las que hacen** a la **calidad del SW**, sino **lo que hacemos** con esas herramientas.

*Principios
de diseño
SOLID*

Principio de Responsabilidad Única

Principio de Abierto Cerrado

Principio de Sustitución de Liskov

Principio de Segregación de Interfaces

Principio de Inversión de Dependencia

Introducción

¿QUÉ SON? ¿PARA QUÉ?

¿Qué son los principios SOLID?

- Son un conjunto de **directrices** que procuran facilitar la creación de software de **calidad**.
- Representan **fundamentos relevantes** de la **arquitectura** y **desarrollo** de software.
- Establecen una **guía** para el **buen diseño de software**.

“

Estos principios **no definen** un **procedimiento** a seguir, sino que **expresan recomendaciones** que **se deben tener en cuenta** al momento de **diseñar software** para que este sea de **calidad**.



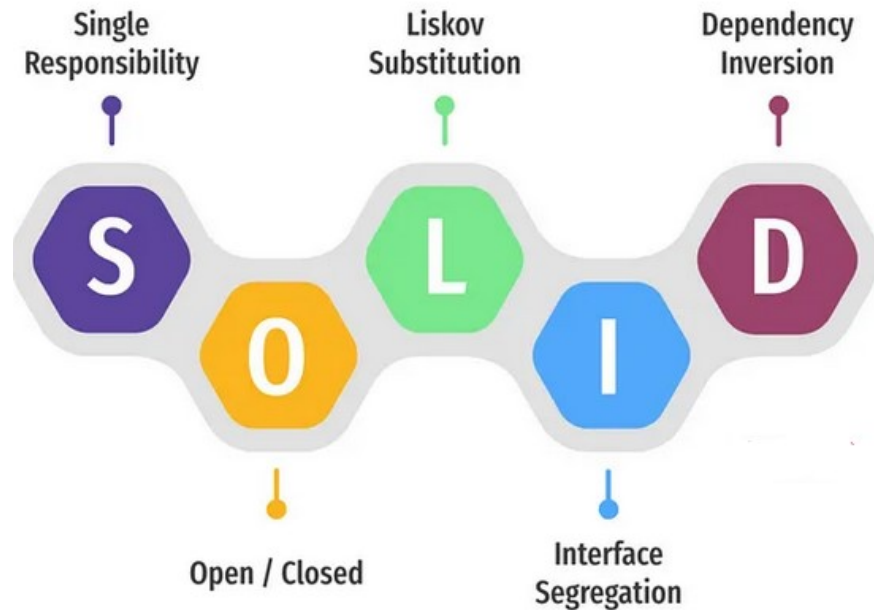
¿Qué son los principios SOLID?

- Son un conjunto de **directrices** que procuran facilitar la creación de software de **calidad**.
- Representan **fundamentos relevantes** de la **arquitectura** y **desarrollo** de software.
- Establecen una **guía** para el **buen diseño de software**.



¿Qué son los principios SOLID?

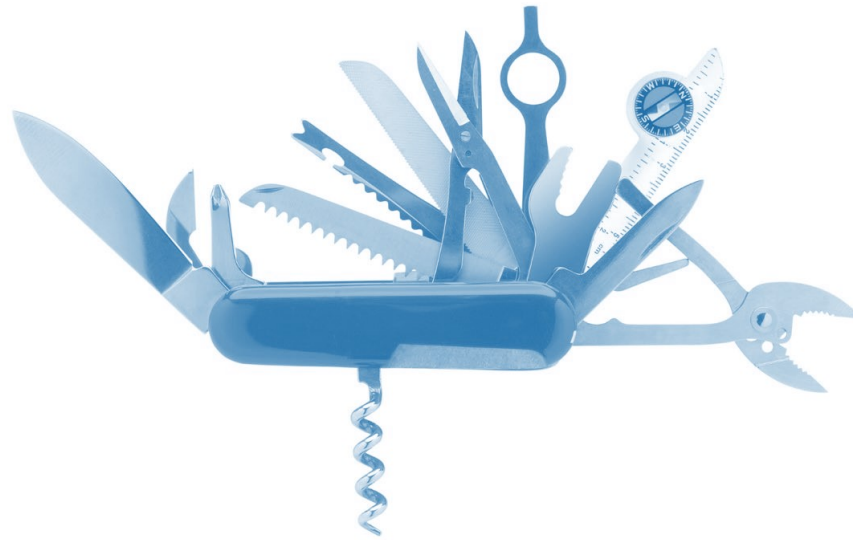
- Son un conjunto de **directrices** que procuran facilitar la creación de software de **calidad**.
- Representan **fundamentos relevantes** de la **arquitectura** y **desarrollo** de software.
- Establecen una **guía** para el **buen diseño de software**.



Son **objetivos** de **SOLID** favorecer

Flexibilidad
Reusabilidad
Extensibilidad

SRP: The Single-Responsability Principle



Principio de Responsabilidad Única

Principio de Responsabilidad Única (SRP)

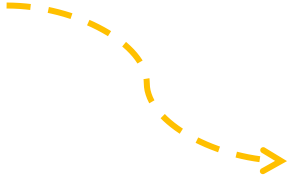
“*Una clase debe tener **una**, y **solo una**, razón para **cambiar**.*



Razón para **cambiar** = Responsabilidad

SRP: Separación de responsabilidades

- ¿Por qué **separar** responsabilidades?

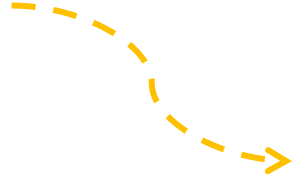


*Cada **responsabilidad** define un **eje de cambio**.*

- Indicios ante el **incumplimiento** de SRP:
 - Cuando **cambien** los **requerimientos**, estos cambios se **manifestarán** en **cambios** en las **responsabilidades** de clases.
 - Si una clase asume **más de una responsabilidad**, habrá **más de una razón** para que esta clase **cambie**.

SRP: Separación de responsabilidades

- ¿Por qué **separar** responsabilidades?



*Cada **responsabilidad** define un **eje de cambio**.*

- Indicis ante el **incumplimiento** de SRP:
 - Si una clase tiene **más de una responsabilidad**, las responsabilidades **se combinan**.
 - Los **cambios** de una responsabilidad **pueden afectar o inhibir** la capacidad de una clase para cumplir con los demás.

SRP: Escenario que **no cumple** el principio

- La clase **UserRegistry** denota dos responsabilidades generales
 - **Mantener** el **registro** de usuarios en una base de datos.
 - Hacerse cargo de **encriptar** los *passwords* de los usuarios.



```
1  // Propuesta que NO respeta SRP.
2  public class UserRegistry{
3      private DataBase db;
4
5      public void addNewUser(String email, String password){
6          int salt = BCrypt.generateSaltSync(10);
7          String passwordEncrypted = BCrypt.hashSync(password, salt);
8          User newUser = new User(email, passwordEncrypted);
9          db.saveUser(newUser);
10     }
11 }
```

SRP: Escenario que cumple el principio

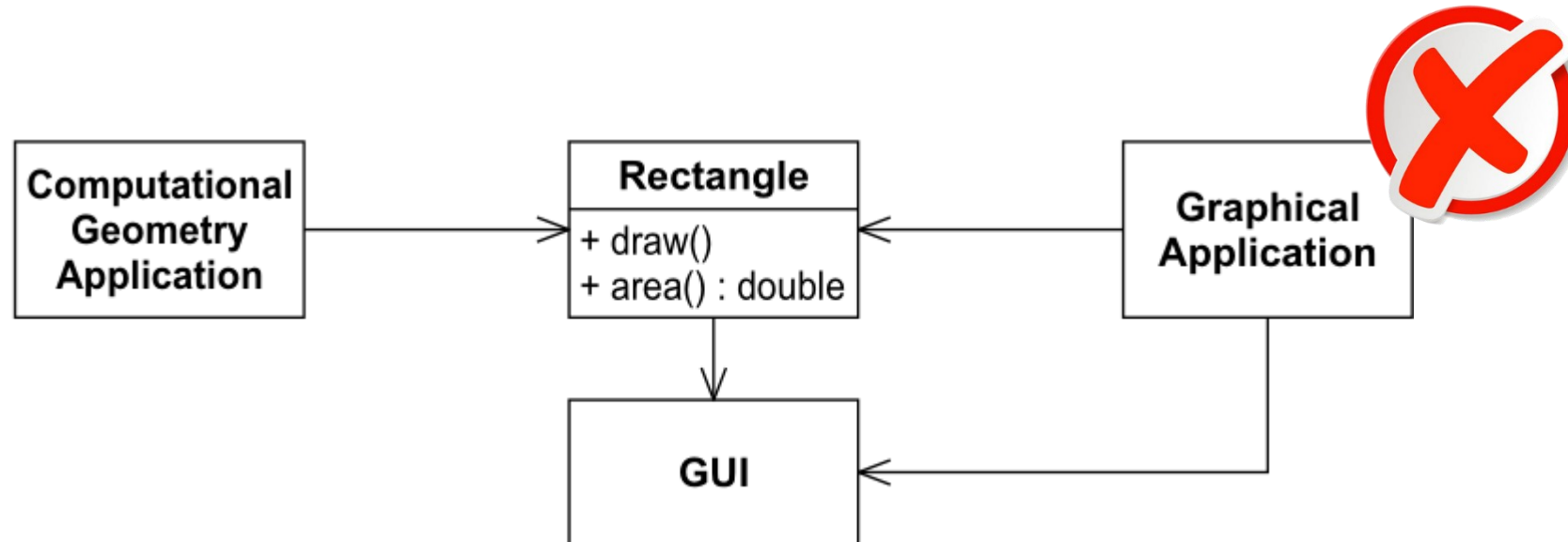
- Las clases **UserRegistry** y **Encrypter** poseen una única responsabilidad.



```
1 // Propuesta que SI respeta SRP.
2 public class UserRegistry{
3     private DataBase db;
4     private Encrypter encrypter;
5
6     public void addNewUser(String email, String password){
7         String passwordEncrypted = encrypter.encrypt(password);
8         User newUser = new User(email, passwordEncrypted);
9         db.saveUser(newUser);
10    }
11 }
12
13 public class Encrypter{
14     public String encrypt(String text){
15         int salt = BCrypt.generateSaltSync(10);
16         String textEncrypted = BCrypt.hashSync(text, salt);
17         return textEncrypted;
18     }
19 }
```

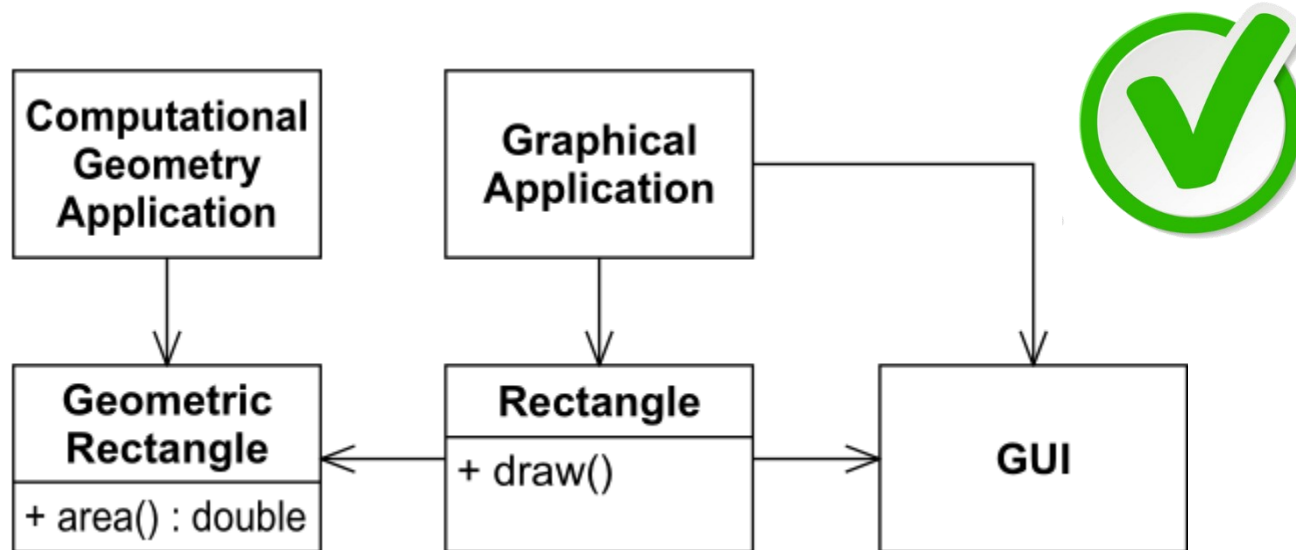
SRP: Escenario que **no cumple** el principio

- La clase **Rectangle** denota dos responsabilidades generales, al ser utilizada desde dos clases diferentes:
 - Estima el área como funcionalidad para **ComputationalGeometryApplication**.
 - Se dibuja sobre la GUI, como funcionalidad para **GraphicalApplication**.



SRP: Escenario que cumple el principio

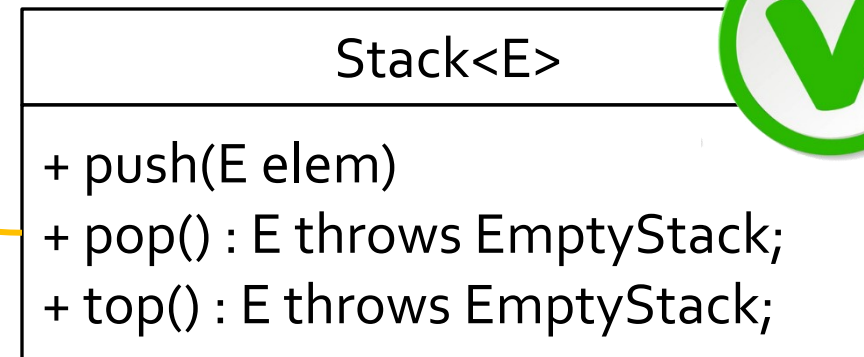
- Un mejor diseño, respetando SRP, separa las responsabilidades del rectángulo en dos:
 - La funcionalidad lógica del cálculo geométrico, a la clase **GeometryRectangle**.
 - La funcionalidad gráfica, ligada a la clase **Rectangle**.



SRP: Definiendo una responsabilidad

- Es complejo definir **formalmente** el concepto de **responsabilidad única**.
- Uno podría pensar que cada operación de **Stack** define una **responsabilidad diferente**.
- Sin embargo, cada una de estas operaciones corresponde con la **responsabilidad general única** de manipular la pila.

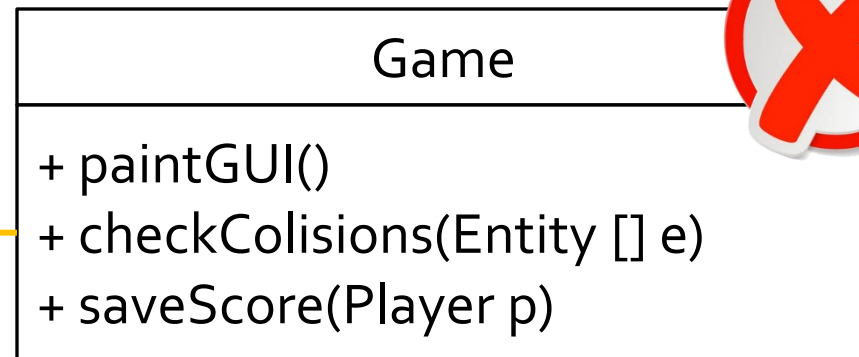
Cada una de las operaciones,
¿define una **responsabilidad**, esto
es, **una razón** para cambiar?



SRP: Definiendo una responsabilidad

- Es complejo definir **formalmente** el concepto de **responsabilidad única**.
- **Modificar** los **requerimientos** para **visualizar la GUI**, o para **detectar las colisiones** entre entidades, o la forma en la que se **almacenan los puntajes**, implican modificar Game.
- Cada una de estas operaciones corresponde con una **responsabilidad única** y **deberían** proveerse en clases **diferentes**.

Cada una de estas operaciones define una **responsabilidad única**, y motivarán cambios por cada **uno de los cambios de requerimientos** que existan en ellas



OCP: The Open-Close Principle



Principio de Abierto Cerrado

Principio de Abierto Cerrado (OCP)

“ *Las entidades de software (clases, módulos, funciones) **deben** estar **abiertas** a **extensiones**, pero **cerradas** a **modificaciones**.*



*Debe ser posible **extender** el **comportamiento** de las entidades, y al mismo tiempo **se debe mantener intacto el comportamiento original**.*

OCP: Indicios de cumplimiento del principio

- Las entidades que se **ajustan** al OCP evidencian:
 - El comportamiento de la entidad es **ampliable**, mediante la **extensión** de la misma.
 - La ampliación del comportamiento de la entidad **no resulta en cambios** en el **código original** de la entidad extendida.



*Para lograr esto, la **abstracción** será **clave**.*

- Toda **entidad** manejada desde una **abstracción**, puede **cerrarse** a **modificaciones**.
- Toda **entidad** manejada desde una **abstracción**, puede **extenderse** creando nuevas **entidades derivadas** de la abstracción.

OCP: Escenario que no cumple el principio

- La clase **AreaCalculator** no se encuentra cerrada a modificaciones
 - Si se **incorpora** una **nueva figura (Rectangle)**, debe recompilarse para que funcione.



```
1 // Propuesta que NO respeta OCP.
2 public class AreaCalculator{
3     public float calculateArea(Shape [] shapes){
4         float totalArea = 0;
5         for (Shape shape : shapes){
6             if (shape.type == "circle")
7                 // totalArea + = Logica para área de un círculo;
8             else if (shape.type == "triangle")
9                 // totalArea + = Logica para área de un triángulo;
10        }
11        return totalArea;
12    }
```

OCP: Escenario que cumple el principio

- La abstracción **Shape** permite que **AreaCalculator** esté cerrada a modificaciones.
- La abstracción **Shape** permiten que las figuras puedan extenderse en **Triangle**, **Circle**, y cualquier otra entidad nueva.



```
1 // Propuesta que SI respeta OCP.
2 public interface Shape{
3     public float area();
4 }
```



```
1 // Propuesta que SI respeta OCP.
2 public class AreaCalculator{
3     public float calculateArea(Shape [] shapes){
4         float totalArea = 0;
5         for (Shape shape : shapes){
6             totalArea += shape.area();
7         }
8         return totalArea;
9     }
10 }
```

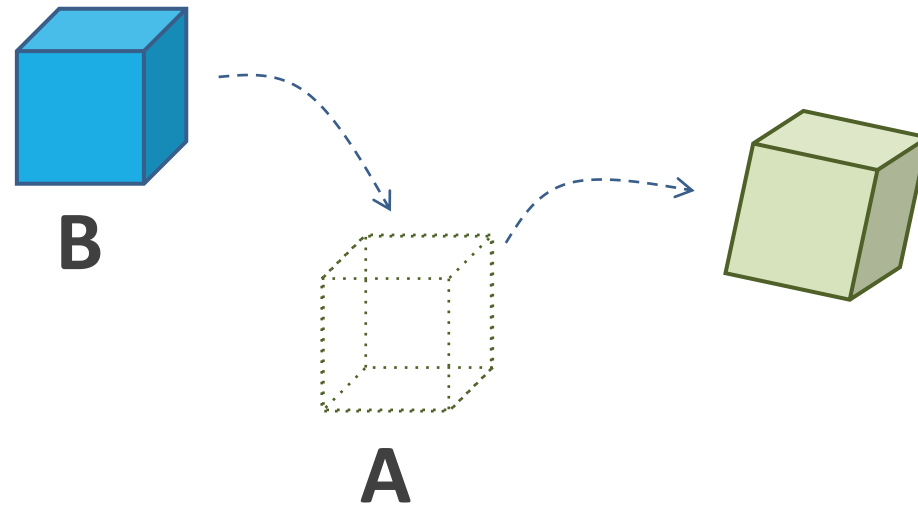


```
1 // Propuesta que SI respeta OCP.
2 public class Triangle implements Shape{
3     public float area(){
4         return (base * heighth) / 2;
5     }
6 }
```



```
1 // Propuesta que SI respeta OCP.
2 public class Circle implements Shape{
3     public float area(){
4         return Math.PI * radius * radius;
5     }
6 }
```

LSP: The Liskov Substitution Principle



Principio de Sustitución de Liskov

Principio de Sustitución de Liskov (LSP)

“ A partir de una **jerarquía de herencia**, las **clases derivadas** deben **poder sustituir cualquier aparición** de sus **clases bases** sin alterar el comportamiento del programa.



Si **B** es una **subclase** de **A**, se **debe** poder **usar B** en **lugar de A** sin que se produzcan inconsistencias.

Principio de Sustitución de Liskov (LSP)



Barbara Liskov
Institute Professor - MIT

Primera mujer con un
doctorado en Universidad
de Stanford
Turing Award – 2008

Subtipo de dato

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 .

Si B es una **subclase** de A , se **debe** poder
usar (**instancias de**) B en **lugar de** (**instancias de**) A
sin que se produzcan inconsistencias.



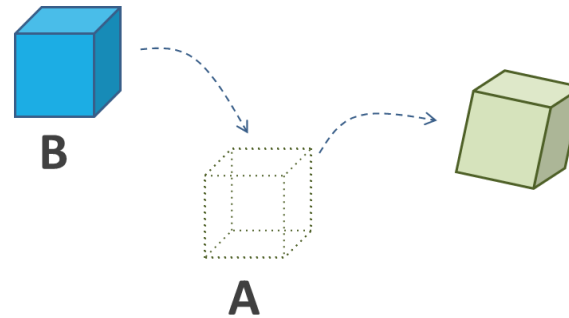
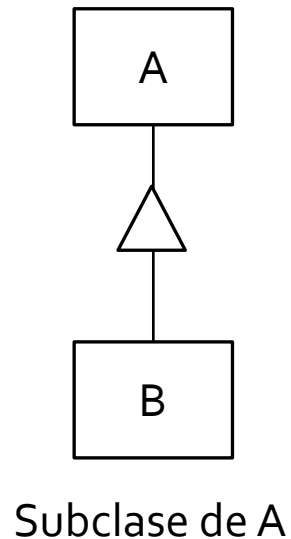
Esto **sólo sucederá** si B **define un subtipo** de A .

Principio de Sustitución de Liskov (LSP)

*Si **B** es una **subclase** de **A**, se **debe** poder
usar (instancias de) B en **lugar de (instancias de) A**
sin que se produzcan inconsistencias.*

Principio de Sustitución de Liskov (LSP)

*Si **B** es una **subclase** de **A**, se **debe** poder usar (**instancias de**) **B** en lugar de (**instancias de**) **A** sin que se produzcan inconsistencias.*



LSP

Sintácticamente, siempre es posible usar instancias de B **en lugar de las** instancias de A, dado que las instancias de B **"responden a los mismos mensajes que A"**.

Observa la **sustitución** desde un punto de vista **semántico**, donde **no alcanza** con solo **"responder a los mismos mensajes"**

LSP: Escenario que no cumple el principio

```
1 public class Rectangle{
2     public void setWidth(float w){
3         width = w;
4     }
5
6     public void setHeight(float h){
7         height = h;
8     }
9
10    public float area(){
11        return width * height;
12    }
13
14    public void do3x10(){
15        width = 3;
16        height = 10;
17    }
18 }
```

```
1 public class Square extends Rectangle{
2     public void setWidth(float w){
3         width = w;
4         height = w;
5     }
6
7     public void setHeight(float h){
8         height = h;
9         width = h;
10    }
11
12    public void do3x10(){
13        setWidth(3);
14        // setWidth(10); Otra opción
15    }
16 }
```

```
1 public do3x10AndPrint(Rectangle r){
2     r.do3x10();
3     System.out.println("Area of r is: " + r.area());
4 }
5
6 public static void main(String [] args){
7     Rectangle r = new Rectangle();
8     do3x10AndPrint(r);
9 }
```

Se *espera* área igual a 30

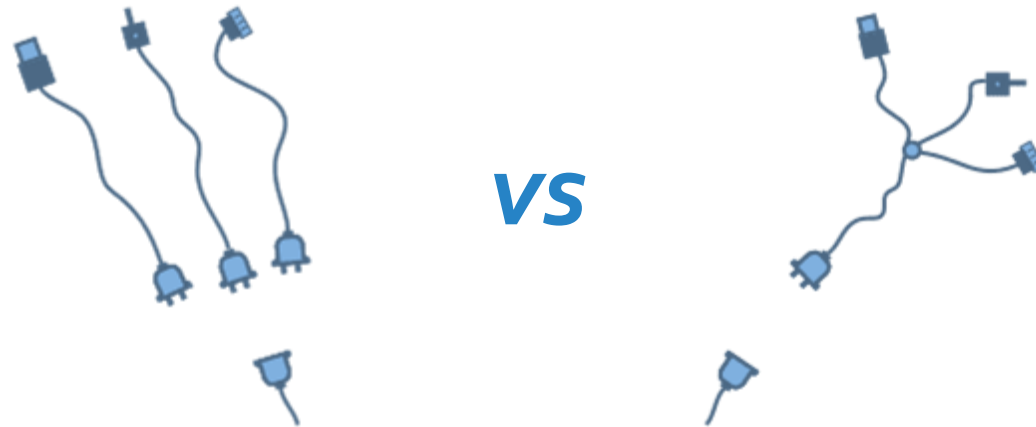
```
1 public do3x10AndPrint(Rectangle r){
2     r.do3x10();
3     System.out.println("Area of r is: " + r.area());
4 }
5
6 public static void main(String [] args){
7     Rectangle r = new Square();
8     do3x10AndPrint(r);
9 }
```



La jerarquía **Rectangle-Square**, con la definición tal y como está,
no respeta LSP.

En el sentido de **esta aplicación**, un **cuadrado no es un rectángulo**

ISP: The Interface-Segregation Principle



Principio de Segregación de Interfaces

Principio de Segregación de Interfaces (ISP)

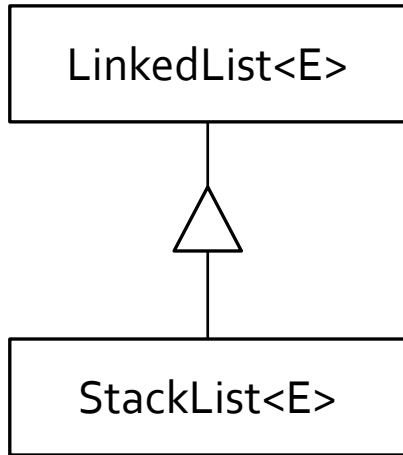
“

*Los clientes **no deben** verse **obligados** a depender de **interfaces** que **no utilizan**.*



*Las interfaces **deben permanecer lo más pequeñas posible**, no se deben adoptar interfaces que **no se van a usar**.*

ISP: Escenarios que no cumplen el principio



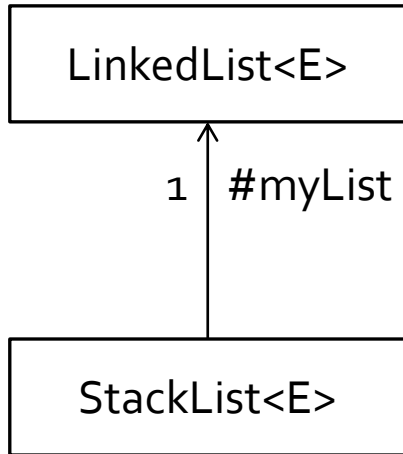
Para implementar una pila con una lista, **StackList** hereda de **LinkedList**



```
1 // Propuesta que NO respeta ISP.
2 public interface Vehicle{
3     public void drive();
4     public void sail();
5 }
6
7 public class Car implements Vehicle{
8     public void drive(){
9         // Lógica para conducir
10    }
11
12    public void sail(){
13        throw new RuntimeException("Car does not sail");
14    }
15 }
```

Car no debería implementar **Vehicle**, o la definición de esta interfaz debería ser diferente.

ISP: Escenarios que cumplen el principio



Para implementar una pila con una lista, **StackList** mantiene una instancia de **LinkedList**



```
1 // Propuesta que SI respeta ISP.
2 public interface LandVehicle{
3     public void drive();
4 }
5
6 public interface WaterVehicle{
7     public void sail();
8 }
9
10 public class Car implements LandVehicle{
11     public void drive(){
12         // Lógica para conducir
13     }
14 }
```

Car únicamente implementa **LandVehicle**.

DIP: The Dependency-Inversion Principle

```
f() {  
    HashSet() LinkedList list = new LinkedList();  
    //...  
    g( list );  
}  
  
    HashSet()  
g( LinkedList list ) {  
    list.add( ... );  
    g2( list )  
}
```

```
f() {  
    Collection list = new LinkedList();  
    //...  
    g( list );  
}  
  
g( Collection list ) {  
    list.add( ... );  
    g2( list )  
}
```

Principio de Inversión de Dependencia

Principio de Inversión de Dependencia (DIP)

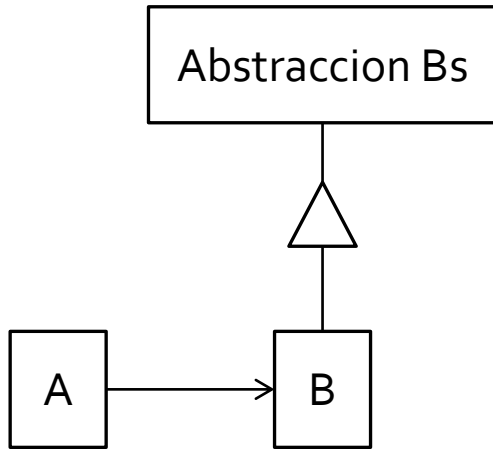
“

*Los módulos, si **dependen** de otros,
deben hacerlo de aquellos
del **mayor nivel de abstracción** posible.*



*El principio **fomenta** una regla de **desacoplamiento**
entre **clases**, permitiendo diseños más flexibles.*

DIP: Escenario que no cumplen el principio



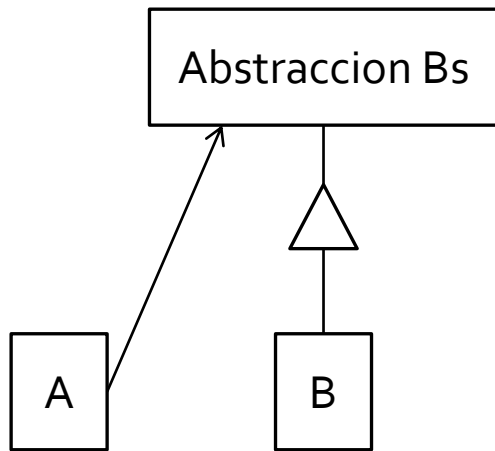
Para implementar la **dependencia** entre A y B, lo ideal es que A dependa **de la mayor abstracción de B** posible.



```
1 // Propuesta que NO respeta DIP.
2 public interface LightBulb{
3     public void turnOn();
4     public void turnOff();
5 }
6
7 public class LightSwitch{
8     public LightBulb light;
9
10    public LightSwitch(LightBulb lighBulb){
11        light = lighBulb;
12    }
13 }
```

LightSwitch solo funciona para **un tipo específico** de lámparas. Una mejor solución, es que acepte **Lights** en general.

DIP: Escenarios que cumplen el principio



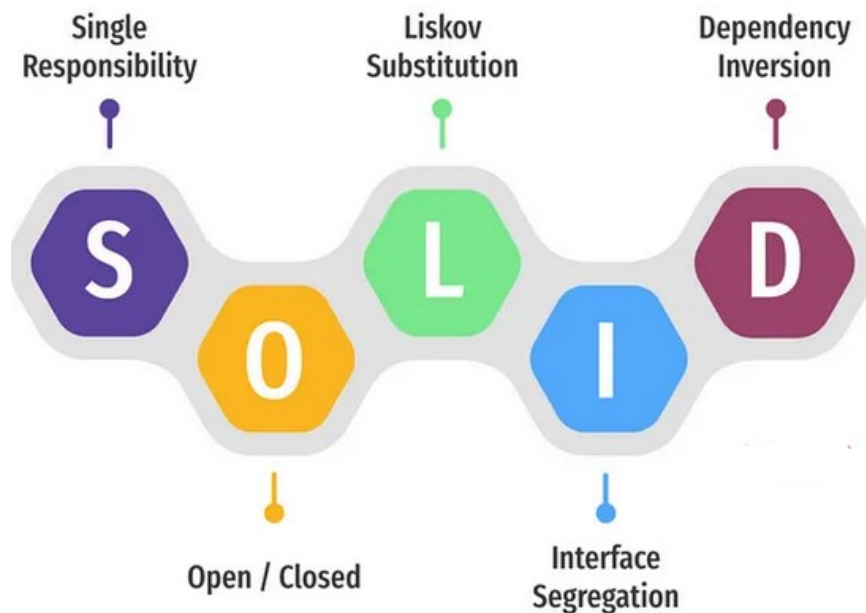
```
1 // Propuesta que SI respeta DIP.
2 public interface SwitchableDevice{
3     public void turnOn();
4     public void turnOff();
5 }
6
7 public interface Light extends SwitchableDevice{}
8
9 public class LightBulb implements Light{
10     public void turnOn(){
11         // Logica turn on
12     }
13
14     public void turnOff(){
15         // Logica turn off
16     }
17 }
18
19 public class LightSwitch{
20     public Light light;
21
22     public LightSwich(Light l){
23         light = l;
24     }
25 }
```

Resumen

Si una clase B es una subclase de A, se debe poder usar B en lugar de A sin que se produzcan errores

Cada clase debe tener una única razón para cambiar, esto es, una sola responsabilidad que puede motivar este cambio.

Permite que el código existente se mantenga intacto mientras se permite extender el mismo agregando nuevas funcionalidades.



Las abstracciones no deben depender de detalles; los detalles deben depender de las abstracciones.

Sugiere que es mejor tener múltiples interfaces específicas que una única interfaz general

Resumen

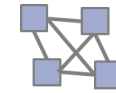
Descomposición modular

Comprensión modular

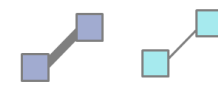
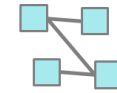
Continuidad modular

Composición modular

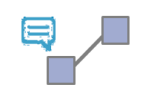
Protección modular



Pocas Interfaces



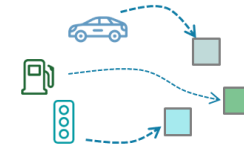
Interfaces Pequeñas



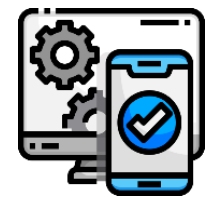
Interfaces explícitas



Ocultamiento de Información



Mapeo Directo



Analizando los principios SOLID sobre un proyecto



<https://www.youtube.com/watch?v=oi8ytxD2Zos>



Fin de la presentación.