# Cinco maneras de mejorar tu código: SOLID

Resumen de la charla/clase *Cinco maneras de mejorar tu código: SOLID* impartida por Sergio Gómez (@sgomez) en la asignatura de Ingeniería del Software de segundo curso del Grado en Ing. Informática.

Existen varios principios de diseño software para hacer código reutilizable y extensible:

- *Don't Repeat Yourself* (**DRY**), no te repitas
- *You ain't gonna need it* (**YAGNI**), no lo hagas hasta que no lo necesites
- *Keep it simple, stupid* (**KISS**), mantenlo simple, estúpido
- **SOLID**:
    - **S**ingle responsibility principle, responsabilidad única
    - **O**pen/closed principle
    - **L**iskov substitution principle
    - **I**nterface segregation principle
    - **D**ependency inversion principle

# Principio de Única Responsabilidad

Principio de Única Responsabilidad (*Single responsibility principle*) la noción de que un objeto solo debería tener una única responsabilidad.

En el ejemplo tenemos una clase *Book* cuya responsabilidad es representar un libro. Ubicar en esta clase métodos para obtener un localizador de un libro o almacenarlo en la base de datos sería erróneo. Lo adecuado sería crear dos clases que se encargasen de esto.

Ejemplos:

- srp-book-01.php
- srp-book-02.php
- srp-book-03.php

# Principio Abierto/Cerrado

Las "entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación". Esto significa que el código debe estar abierto a incorporar nueva funcionalidad o mejoras, pero que estas mejoras no deberían alterar el código existen (salvo corrección de errores, obviamente).

En el ejemplo la extensión de funcionalidad se realiza incrementando el número de casos en una estructura de control *swicth*. Una solución más adecuada sería utilizar una clase abstracta y que la extensión de funcionalidad se haga con nuevas clases que implementen la clase abstracta, de esta forma la extensión se realiza sin modificar el código anterior.

Ejemplos:

- ocp-rules-01.php
- ocp-rules-02.php

# Principio de sustitución de Liskov

Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Esto se traduce en que las clases hijas no deben modificar el comportamiento de las clases madre.

En el ejemplo tenemos la clase *Rectangulo*, de la cuál hereda *Cuadrado* modificando los métodos *set* para que el ancho y alto sean iguales. El problema surge con la función que calcula el área, ya que que el comportamiento esperado de esta función es distinto para las dos clases. La clase hija debería poder utilizarse como la clase madre de manera idéntica sin tener que conocer la diferencia entre las clases. En el ejemplo el test de unidad fallará cuando no debería hacerlo.

Ejemplos:

- lsp-rectangulo-01.php

# Principio de segregación de la interfaz

Al plantear qué métodos debe definir una interfaz (en el sentido de clase abstracta o similares) no se debería establecer un conjunto único, sino que los métodos deberían agruparse por funcionalidad resultando en distintas interfaces.

Por ejemplo, una interfaz para manejar una impresora no debería asumir que todas las impresoras tendrán un escáner o un fax integrado, de otra forma la interfaz obligaría a definir los métodos relacionados con escanear documentos a todas las clases que implementen la interfaz, aunque la impresora para la que se desarrolla el controlador no disponga de interfaz. Como alternativa las funcionalidades relacionadas con imprimir, escanear o enviar faxes deberían agruparse por separado.

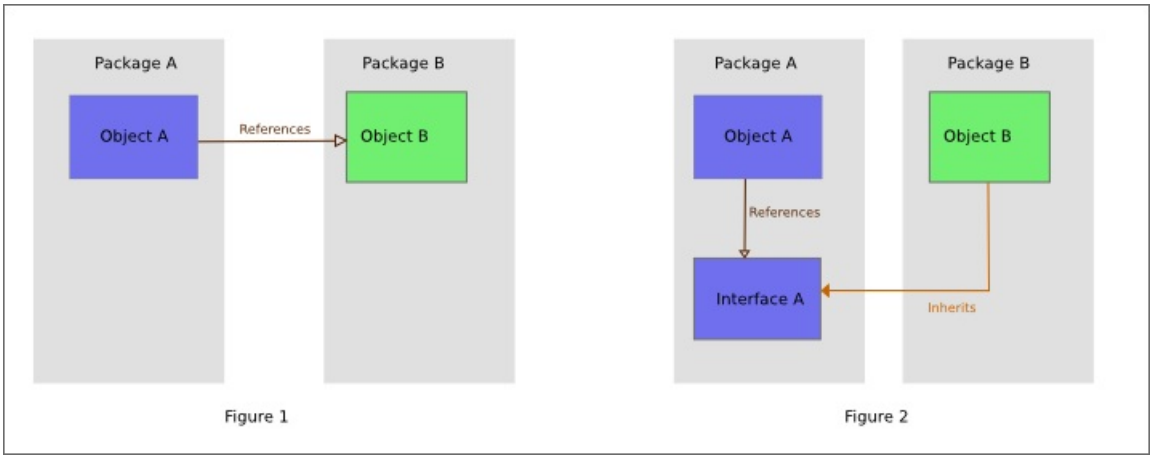
Ejemplos:

- isp-xerox-01.php
- isp-xerox-02.php

# Principio de inversión de dependencia

Este principio se refiere a una forma de desacoplar módulos software en el sentido de que la tradicional dependencia de los módulos de alto nivel respecto a los inferiores debe *invertirse* para romper esta dependencia de los detalles de implementación de bajo nivel. El principio dice:

1. Un módulo de alto nivel no debe depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
2. Las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.




La figura muestra un código con la misma funcionalidad, sin embargo, en la figura de la derecha se ha utilizado una interfaz para invertir la dependencia. El patrón de diseño *injección de dependencia* ayuda a cumplir este principio.

Ejemplos:

- dip-usuario-01.php
- dip-usuario-02.php
- dip-usuario-03.php
- dip-usuario-04.php