Principios SOLID

Análisis y Diseño Orientado a Objetos

SOLID

- Conjunto de Principios para escribir **software de calidad**
- Lenguajes de Programación Orientados a Objetos (**OOP**)
- **Patrones de Diseño** (alta cohesión / bajo acoplamiento)
- **Reutilización** del Diseño
- Proveer **flexibilidad**
- Código **más fácil de leer, testear y mantener**
- **Refactorización** más sencilla

Robert C. Martin

(Uncle Bob)

SOLID

S*ingle responsibility principle (SRP)*

O*pen/closed principle (OCP)*

L*iskov substitution principle (LSP)*

I*nterface segregation principle (ISP)*

D*ependency inversion principle (DIP)*

SOLID

- Principio de Responsabilidad única
- Principio de Abierto/Cerrado
- Principio de Sustitución de Liskov
- Principio de la Interfaz
- Principio de Inversión de la Dependencia

1/ Responsabilidad Única

Un objeto debe realizar una **única cosa.**

Indicadores para determinar su (in)cumplimiento:

- En una misma clase están involucradas dos capas de la arquitectura (presentación, lógica de negocio y persistencia).
- Nº de métodos públicos (agruparlos por funcionalidad).
- Métodos que usan cada uno de los campos de esa clase.
- Nº de imports.
- Nos cuesta testear la clase (test unitarios).
- En cada nueva funcionalidad la clase se ve afectada.
- Nº de líneas de código.

1/ Responsabilidad Única

Un ejemplo típico es el de un objeto que necesita ser renderizado de alguna forma, por ejemplo imprimiéndose por pantalla:

```
public class Vehicle {

    public int getWheelCount() {
        return 4;
    }

    public int getMaxSpeed() {
        return 200;
    }

    @Override public String toString() {
        return "wheelCount=" + getWheelCount() + ", maxSpeed=" + getMaxSpeed();
    }

    public void print() {
        System.out.println(toString());
    }
}
```

1/ Responsabilidad Única

```
public class PatientServiceImpl implements PatientService{
...

 public Patient findById(Integer patientId) {
   if(!hasPermission(SecurityContext.getPrincipal())) {
     throw new AccessDeniedException();
   }
   return patientDao.findById(patientId);
 }
 public List findByRoom(Integer roomId) {
   if(!hasPermission(SecurityContext.getPrincipal())) {
     throw new AccessDeniedException();
   }
   return patientDao.findByRoom(roomId);
 }

}
```

Responsabilidad Única

Conclusión

- Herramienta indispensable para proteger nuestro código frente a cambios, ya que implica que sólo debería haber un motivo por el que modificar una clase.

2/ Open/Closed

Una entidad de software debería estar **abierta a extensión** pero **cerrada a modificación.**

- Extender el comportamiento de nuestras clases sin necesidad de modificar su código.
- Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.
- Se suele lograr utilizando **herencia** y **poliformismo**.

* **Indicadores** para determinar su (in)cumplimiento:

- Cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas.

Open/Closed

Conclusión

- Tener código cerrado a modificación y abierto a extensión nos da la máxima flexibilidad con el mínimo impacto.
- Esta complejidad no siempre compensa, y como el resto de principios, sólo será aplicable si realmente es necesario.

3/ Sustitución de Liskov

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. .

- Si estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.
- Nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la padre.

* **Indicadores** para determinar su (in)cumplimiento:

- Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov.
- Los tests de la clase padre no funcionan para la clase hija.

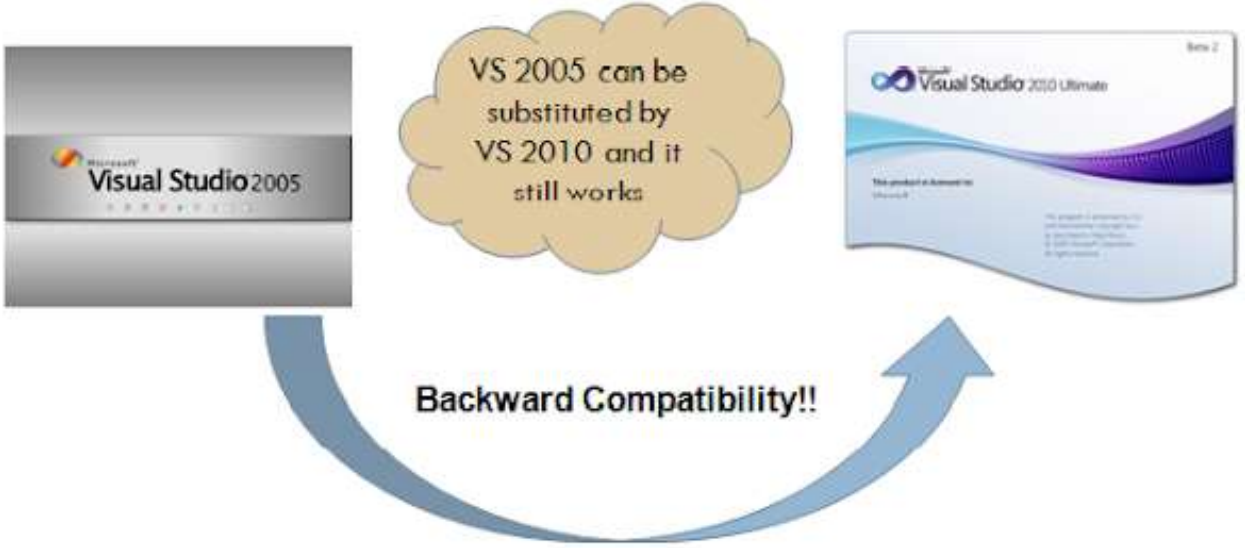
Liskov Substitution Principle

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Barbara Liskov
"Data Abstraction and Hierarchy"

Liskov Substitution Principle

"Derived classes should extend without replacing the functionality of old classes"



Sustitución de Liskov

Conclusión

- A la larga compensa por la flexibilidad que otorga a la arquitectura de nuestra aplicación.
- Nos ahorrará muchos errores derivados de nuestro afán por modelar lo que vemos en la vida real en clases siguiendo la misma lógica.
- No siempre hay una modelización exacta, por lo que este principio nos ayudará a descubrir la mejor forma de hacerlo.

4/ Segregación de Interfaces

Muchas interfaces específicas son mejores que una interfaz de propósito general.

- Asegurarnos de que todas las clases que implementen interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos.
- En caso contrario, es mejor tener varias interfaces más pequeñas.
- Las interfaces nos ayudan a **desacoplar** módulos entre sí.
- **Indicadores** para determinar su (in)cumplimiento:
- Al implementar una interfaz se observa que uno o varios de los métodos no tienen sentido y hace falta dejarlos vacíos o lanzar excepciones.

Segregación de Interfaces

Conclusión

- Reaprovechar los interfaces en otras clases.
- No obligar a ninguna clase a implementar métodos que no utiliza.
- Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas.

5/ Inversión de Dependencias

Depender de abstracciones, no depender de implementaciones.

Definición

- Las 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.

5/ Inversión de Dependencias

Depender de abstracciones, no depender de implementaciones.

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…problemas:

- La 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.

5/ Inversión de Dependencias

Depender de abstracciones, no depender de implementaciones.

Indicadores para determinar su (in)cumplimiento:

- Cualquier instanciación de clases complejas o módulos es una violación de este principio.
- No puedas probar (tests) esa clase con facilidad porque dependen del código de otra clase.

Inversión de Dependencias

Conclusión

- Este mecanismo nos obliga a organizar nuestro código de una manera muy distinta a como estamos acostumbrados, y en contra de lo que la lógica dicta inicialmente.
- A la larga compensa por la flexibilidad que otorga a la arquitectura de nuestra aplicación.