

---

# INTRODUCCIÓN A LOS PRINCIPIOS SOLID

---



---

# Índice

|                                                           |           |
|-----------------------------------------------------------|-----------|
| <b>Índice</b>                                             | <b>2</b>  |
| <b>Introducción</b>                                       | <b>3</b>  |
| <b>Single Responsibility Principle (SRP)</b>              | <b>4</b>  |
| Definición de responsabilidad única o un motivo de cambio | 4         |
| Beneficios                                                | 5         |
| <b>Open-Close Principle (OCP)</b>                         | <b>6</b>  |
| Algunas técnicas para respetar OCP                        | 6         |
| Beneficios                                                | 7         |
| <b>Liskov Substitution Principle (LSP)</b>                | <b>8</b>  |
| Un ejemplo de LSP                                         | 8         |
| Beneficios                                                | 10        |
| <b>Interface Segregation Principle</b>                    | <b>11</b> |
| Beneficios                                                | 11        |
| <b>Dependency Inversion Principle (DIP)</b>               | <b>12</b> |
| Beneficios                                                | 12        |
| <b>Conclusión</b>                                         | <b>14</b> |
| <b>Sobre mí</b>                                           | <b>15</b> |

## Introducción

Los principios SOLID son 5 normas que distintos autores, entre ellos Robert C. Martin\*, crearon una serie de normas con el objetivo de ayudarnos a desarrollar un código más flexible y mantenible.

Creeme cuando te digo que realmente hay una diferencia abismal entre un proyecto grande que sigue SOLID y otro que no lo sigue. Tanto a nivel de costes de desarrollo, porque no es lo mismo tardar 1 semana que 3, como a nivel emocional de los trabajadores, ya que a nadie le gusta arreglar una cosa y romper dos, o tener que ingeniárselas para añadir un cambio dentro del sistema sujetándolo con pinzas para no tenerlo que cambiar todo.

Estos problemas acaban creciendo y desembocando en frustración y en pérdida del interés por el proyecto y por nuestra pasión por el desarrollo de videojuegos, cosa que he vivido personalmente.

\* Robert C. Martin, también conocido como el tío Bob, es un programador que lleva más de 50 años trabajando en el software, el cual empezó su carrera trabajando con tarjetas perforadas, así que algo podremos aprender de él y sus 50 años de experiencia.

## Single Responsibility Principle (SRP)

Quien no se ha encontrado con una clase de 500 líneas, o 1.000, o 2.000 líneas, yo he visto clases de 8.000 e incluso 12.000 líneas, ¡una locura! Es imposible trabajar agusto con un archivo tan grande, y lo que es peor, ¡se hace inmantenible!

Cuando una clase empieza a superar las 300 o 400 líneas y tiene varios puntos de entrada empieza a ser complicado de entender. Cuando llegas a esta clase primero te tienes que orientar, ver qué hace, porqué lo hace y entonces hacer tu cambio.

Normalmente eso se debe a que empezamos a meter cosas que creemos que se deben de gestionar en la misma clase pero no tiene porqué ser necesariamente así, esta clase puede contar con otras clases más pequeñas que le ayuden a hacer su cometido y delegar la responsabilidad.

Lo peor de los archivos grandes no solo es su tamaño si no los efectos colaterales que puedes tener al hacer cambios en este código. Al tener varias responsabilidades en una misma clase es fácil que acabemos compartiendo variables de estado y al realizar un cambio para un comportamiento rompamos otro que no hemos tenido en cuenta.

Esto es lo que intenta evitar el principio de responsabilidad única.

### Definición de responsabilidad única o un motivo de cambio

Los motivos de cambio son todo aquello que hará que tengamos que modificar un código ya existente para adaptarlo a los nuevos requerimientos.

Pongamos como ejemplo una clase Hero o Player, que es el típico ejemplo que está en todos los proyectos de videojuegos. Lo normal es que esta clase gestione lo que puede hacer el usuario con el personaje principal.

Donde gestionamos el movimiento, los ataques, la vida que le queda, ejecutamos las animaciones y muchas otras cosas relacionadas con el personaje.

Todo esto son distintos motivos de cambio, si mañana cambia como el personaje recibe daño tendremos que cambiar esta clase, o si ahora queremos añadir un ataque especial también tendremos que modificar esta clase.

Con esto corremos el riesgo de romper una cosa implementando otra porque no tenemos una división clara, empezamos a compartir variables de estados y el código empieza a ser un lío.

Esto mismo serían motivos de cambio o responsabilidades aplicados a una clase. Pero podemos extender el concepto a funciones y módulos de nuestro proyecto.

Si tenemos un módulo con la gestión de la batalla, no deberíamos de incluir cosas que no sean de la batalla, por ejemplo subir de nivel no forma parte de la batalla, aunque esta experiencia se obtenga en la batalla, debería de ser otro módulo el que haga el cálculo y aplique la experiencia.

Como venía diciendo, para evitar este tipo de cosas existe el principio de una sola responsabilidad, que dice que una clase, o módulo, solo debe de tener un único motivo de cambio, distintos cambios no pueden afectar a la misma clase.

Por ejemplo, si queremos modificar el movimiento de nuestro personaje la clase que gestiona los ataques no debería de verse afectada, o si cambiamos cómo gestionamos la vida del héroe nuestro gestor de movimientos tampoco debería de verse afectado.

## Beneficios

Seguir este principio nos aportará legibilidad, y nos facilitará el cambio, por no mencionar el enorme beneficio para el equipo que supone tener clases de menos de 300 líneas con una responsabilidad en lugar de clases de 1.000 líneas, o más, con muchas responsabilidades.

Ponte en la piel de una persona nueva que se incorpora al equipo, o tu mismo cuando entraste al proyecto. ¿Qué te resulta más fácil? Una clase de 1.000 líneas con varias responsabilidades, o tres clases de 300 líneas cada una con una responsabilidad y un buen nombre de clase?

Hay quienes defienden que tener un archivo grande y con la lógica centralizada les ayuda a encontrar las cosas de forma más rápida, pero esto realmente solo es un síntoma de otros problemas como:

- Pobreza de nombres.
- Los nombres de las clases no dicen qué responsabilidad tiene esa clase, se utilizan nombres vacíos como manager.
- Los nombres de las funciones no indican cual es la intención de esa función.
- La jerarquía de carpetas no deja claro que arquitectura seguimos y como están distribuidas las responsabilidades.

Si utilizamos unos nombres ricos que nos cuenten a la perfección la responsabilidad de las clases, la intención de las funciones, tenemos una jerarquía de carpetas fácil de seguir y siguiendo una arquitectura coherente, si tenemos todo esto no deberíamos de tener ningún problema a la hora de encontrar lo que estemos buscando.

## Open-Close Principle (OCP)

Por fin tenemos una versión jugable de nuestro juego pero “oh vaya” al Product Manager se le han ocurrido 3 nuevos tipos de enemigos y 2 clases nuevas de items que tenemos que incorporar....

Cada semana se le ocurren enemigos y power ups... vamos a tener que volver a editar los mismos archivos donde tenemos el switch para crear estos objetos, el enum de tipos de enemigos, el enum de tipos de power ups, y añadir la nueva lógica para cada tipo.

Bueno podría ser peor, ahora mismo solo tenemos 7 tipos de enemigos y 5 tipos de power ups, pero claro... solo llevamos 3 meses de desarrollo, ¿que pasará si sigue este ritmo de creatividad y desarrollo?

La clase que contiene el switch ahora mismo ya son 400 líneas en las que tenemos que buscar el tipo de enemigo y ver qué es lo que tenemos que cambiar.

La de power ups es algo más pequeña pero todavía no tenemos desarrollado qué efectos tendrán en los enemigos... sin contar con que tenemos que aplicar las nuevas características de los power ups al script donde tenemos el comportamiento del jugador...

No parece que vaya a mejorar la situación, y solo llevamos 3 meses...

¿Te suena de algo esa situación, verdad?

Enumeraciones interminables, switchs enormes que no hacen más que crecer y crecer, el diseñador no para de imaginar nuevas características que siempre hacen que tengamos que volver a visitar las mismas clases, etc.

Lo ideal sería que no tuviéramos que llegar a esta situación, pero ¿cómo? Pues en parte con el principio de abierto cerrado.

La idea fundamental de este principio es que nuestras clases y sistemas, deben estar abiertas a la extensión y cerradas a la modificación, pero, ¿esto qué quiere decir?

Pues ni más ni menos que debemos diseñar nuestros sistemas para que sean capaces de crecer, extender su funcionalidad, sin cambiar el código base que las hace funcionar, ¿y cómo hacemos esto?

## Algunas técnicas para respetar OCP

Nuestro aliado más poderoso serán las abstracciones, en lugar de trabajar con clases concretas trabajemos con clases abstractas o interfaces y recibamos la instancia de esta clase en el constructor.

Esto permitirá a la clase concreta evolucionar sin afectar a los consumidores de esta, solo tiene que respetar la interfaz acordada. También nos permitirá que podamos cambiar una clase concreta por otra de la misma interfaz sin que el consumidor tenga que realizar ningún cambio. Esto exactamente es el patrón Strategy\*.

Otra forma de respetar este principio será evitar las estructuras estáticas como la sentencia switch/case para preguntar por los tipos.

Si tenemos una enumeración de tipos y un switch para elegir qué tipo utilizar, o qué acción realizar en función del tipo, cada vez que añadamos un tipo nuevo tendremos que revisar este código para añadir un caso nuevo.

Para evitar esto podemos utilizar estructuras dinámicas como el diccionario, en el cual podemos crear de forma dinámica una asociación entre el tipo y la clase, o acción, que vamos a utilizar.

Una forma más de evitar este tipo de estructuras es tener una buena jerarquía de clases. En lugar de tener un switch preguntando por el tipo y en función de esto hacer una cosa u otra, podemos crear una clase común a estos tipos con un método para hacer la acción y por herencia sobrescribir la función para que cada tipo haga su lógica concreta.

\* Patrón Strategy: <https://thepowerups-learning.com/patrones-de-diseno-strategy/>

## Beneficios

Seguir este principio nos evitará cambios en cascada, con esto quiero decir que al cambiar una clase nos evitaremos cambiar a los consumidores de esta.

También nos permitirá ampliar el sistema sin tener que revisar clases que ya funcionan y están cerradas, lo cual nos evitará daños colaterales y posibles bugs al reducir el número de clases afectadas por el cambio.

Otro beneficio es que en algunos casos podremos hacer que nuestro sistema crezca sin tener que implementar nada nuevo, solo cambiando ficheros de configuración o la información que nos llega desde el servidor en una aplicación cliente-servidor.

## Liskov Substitution Principle (LSP)

El principio de substitución de Liskov dice que los objetos deberían de poder ser reemplazados por instancias de sus subclases sin que el programa deje de funcionar.

Por ejemplo, si tenemos una clase Cuadrado que hereda de Rectángulo, en cualquier momento deberíamos de poder trabajar con rectángulos directamente en lugar de con cuadrados, si no es así nos tenemos que replantear esa herencia.

Incumplir este principio nos puede llevar a tener errores silenciosos, ya que estaremos utilizando una clase que no estaba preparada para ser utilizada de esa forma y nadie está impidiendo que lo hagamos. Los errores que provoca no son fácilmente visibles, a no ser que hayamos tomado las precauciones de llenar todo nuestro código de asserts y throws.

### Un ejemplo de LSP

Sabemos que un cuadrado solo es una variación de un rectángulo donde ambos lados tienen el mismo tamaño, por lo tanto no sería descabellado pensar que podemos modelar estos objetos creando un rectángulo como padre y un cuadrado como hijo.

El cuadrado lo único que deberá hacer es asignar el mismo valor a los dos lados.

```
public class Rectangle {  
    private float width;  
    private float height;  
  
    public float Width  
    {  
        get { return width; }  
        set { width = value; }  
    }  
  
    public float Height  
    {  
        get { return height; }  
        set { height = value; }  
    }  
}
```

```
public class Square : Rectangle {
    public float Width
    {
        set {
            base.Width = value;
            base.Height = value;
        }
    }

    public float Height
    {
        set {
            base.Width = value;
            base.Height = value;
        }
    }
}
```

¿Qué ocurriría si pasamos un cuadrado a una función que trabaja con rectángulos y modifica uno de sus lados?

Lo que ocurriría es que tendríamos un cuadrado desvirtuado, con un lado de cada tamaño y por lo tanto no sería un cuadrado, pero nada nos impide hacer esto mismo y lo peor es que no recibiremos ningún error.

Consideremos el siguiente código:

```
Square square = new Square();
square.Width = 2; // ambos lados son seteados a 2
square.Height = 3; // ambos lados son seteados a 3

void SetWidthTo4(Rectangle rectangle) {
    rectangle.Width = 4;
}
```

Si ahora llamamos a la función con un cuadrado, la propiedad Width que se estará llamando es la del rectángulo y por lo tanto asignará 4 al ancho sin modificar la altura, con lo que nos estaremos saltando la restricción de que un cuadrado debe tener los dos lados iguales.

El principio de Liskov nos dice que una función debe de funcionar para todos los tipos padre de un objeto, con lo que la función anterior estaría incumpliendo este principio y por lo tanto esta herencia no está bien planteada.

## Beneficios

Los beneficios de no incumplir este principio se podrían resumir en que nuestro programa funcionará como esperamos.

Como hemos visto en el ejemplo, si incumplimos el principio no solo tendremos comportamientos inesperados si no que además serán muy difíciles de identificar ya que no tenemos ningún mecanismo que haga saltar un error.

Para evitar esto es importante plantearse muy bien la jerarquía de clases y preguntarnos si realmente es necesaria esta herencia o tal vez podamos conseguir el mismo resultado por composición y evitarnos esta problemática.

## Interface Segregation Principle

El principio de segregación de interfaces nos dice que los consumidores no deberían depender de métodos que no utilizan.

Para hacer esto lo que haremos es separar una interfaz en varias interfaces de forma que cada consumidor solo utilice la interfaz con los métodos que necesite. Estamos partiendo de la base de que utilizamos abstracciones (interfaces) y no concreciones, cosa que veremos en el último principio.

Además de separar las interfaces por funcionalidad, las podemos separar por privilegios, por ejemplo las funciones que modifiquen datos en una interfaz y las que sean de solo consulta en otra.

### Beneficios

El primer beneficio que obtendremos es que eliminaremos ruido para los consumidores de nuestro sistema, y les daremos solo la funcionalidad que necesitan. Con esto también nos estaremos protegiendo de un mal uso de nuestra clase.

Por ejemplo, si tenemos una clase que puede leer los datos del usuario pero también los puede escribir, no queremos que alguien por error haga uso de las funciones de guardado.

Para esto simplemente podemos extraer estas funciones de escritura en una interfaz y las funciones de lectura en otra. A los consumidores interesados en leer les damos solo la interfaz de lectura y a los interesados en escribir les daremos la de escritura, así nos aseguramos de que no escriben por error.

Con esto también les ayudamos a centrarse en las funciones que necesitan ya que solo tendrán esas accesibles.

Este principio también nos ayuda a cumplir con el principio de responsabilidad única (SRP), ya que cada interfaz tiene una única responsabilidad y perfectamente la podríamos implementar en archivos separados. Otro principio que nos ayuda a cumplir es el de Inversión de Dependencias (DIP) que lo veremos a continuación.

## Dependency Inversion Principle (DIP)

El principio de Inversión de Dependencias nos dice algo tan sencillo como que debemos de depender de abstracciones y no de concreciones. Esto quiere decir que es mejor consumir clases abstractas e interfaces antes que consumir clases concretas.

El motivo de esto es porque nos facilitará muchísimo el cambio. Al no depender de los detalles nos será muy fácil cambiar el comportamiento sin tener que modificar las clases que consumen estas interfaces o clases abstractas.

La gente suele confundir este principio con la Inyección de Dependencias y la Inversión de Control (IoC), y es que los nombres no ayudan. Si que es cierto que el principio de Inversión de Dependencias (DIP) ayuda a poder inyectar estas dependencias y a invertir el control.

Me explico:

La Inyección de Dependencias, también conocido como el patrón Strategy, es una técnica que se utiliza para que las dependencias de una clase en lugar de construirlas dentro de esta clase, ya sea con un new, instanciandolas o serializandolas en el editor de Unity, nos las pasamos en el constructor.

Esto nos aporta que esta clase no se acople a la implementación concreta y no le importen los detalles de construcción.

La Inversión de Control (IoC), es otra técnica para delegar la lógica de algo a otro sistema, como se suele decir: “no nos digas que hacer, nosotros te avisaremos”.

Por poner un ejemplo concreto, si tenemos un personaje que se mueve cuando recibe el input, para invertir el control, podríamos hacer una clase de movimiento y a esta pasarle el personaje.

Entonces sería esta clase de movimiento la encargada de mover al personaje cuando recibe un input, le hemos delegado esta responsabilidad. En lugar de moverse el personaje a sí mismo, invertimos el control y le decimos a otra clase “tú serás la encargada de mover a este personaje”.

## Beneficios

Como ya hemos dicho arriba, seguir este principio nos facilitará los posibles cambios que puedan llegar en el futuro al no acoplarnos a los detalles.

Utilizar abstracciones también nos ayudará a establecer contratos con los compañeros. Podemos definir la interfaz entre los dos equipos, uno se encargará de la implementación y el otro simplemente consumirá la interfaz como si ya existiera la implementación.

A este segundo no le preocupan los detalles de implementación ya que lo único que necesita es la interfaz. Esta técnica es muy utilizada entre cliente y servidor.

Como siempre con SOLID, este principio nos ayudará también a que nuestro código sea testeable. Dado que utilizamos interfaces, estas las podemos sustituir por una clase “ficticia” durante el test y decirle lo que tiene que hacer, esto es lo que llamamos un Mock o Sustituto.

## Conclusión

Los principios SOLID son 5 reglas que de seguirlas nos ayudarán muchísimo a la hora de crear un proyecto flexible y mantenible a largo plazo.

Gracias a ellos tendremos:

- Clases pequeñas con una única responsabilidad y por lo tanto más fáciles de leer y mantener.
- Módulos que no entran en las responsabilidades de otros módulos.
- Sistemas que podremos ampliar sin tener que visitar clases ya existentes para modificarlas.
- Reducción de los cambios en cascada al introducir nueva funcionalidad.
- Reduiremos los daños colaterales que podríamos producir por un mal diseño de clases al preguntarnos si los hijos de la clase padre pueden ser reemplazados por tipos de este padre.
- Reduiremos el ruido que tienen los consumidores al utilizar nuestro sistema y por lo tanto haremos que el sistema sea más sencillo e intuitivo de utilizar.
- No nos acoplaremos a los detalles de una implementación, lo cual repercutirá en la mantenibilidad del proyecto.
- Nos será más fácil cambiar una implementación por otra al utilizar interfaces en lugar de clases concretas.

Como puedes ver los beneficios son muchos y las desventajas son muy pocas o inexistentes, y desde luego los problemas de no aplicar SOLID son mucho mayores a los problemas de aplicarlos todos.

Dependiendo de la práctica que tengamos aplicando estos principios nos resultará más fácil pensar en SOLID y diseñar nuestras clases de forma que cumplan todos los principios desde el inicio, solo es cuestión de práctica y no rendirse.

## Sobre mí



Soy Daniel Parra, desarrollador de videojuegos desde hace 10 años y un apasionado de la programación.

Llevo toda mi carrera profesional desarrollando videojuegos para distintas plataformas y con distintos motores. He desarrollado juegos para WiiWare, PS4, Steam/PC y dispositivos móviles, y he trabajado con Unity, Unreal y motores propios de la empresa desarrollados en C++.

Soy un fanático del código limpio, los principios SOLID, patrones de diseño, TDD y arquitectura. Creo que uno de los problemas de que los proyectos se dilaten en el tiempo y acaben costando más dinero del estimado, es en gran medida por la falta de aplicar buenas prácticas y pensar en el largo plazo del proyecto.

En el caso de que te haya parecido interesante esta introducción, te animo a que sigas profundizando en los principios SOLID, por lo que te recomiendo mi curso “SOLID y Clean Code para VIDEOJUEGOS”.

Más información:

<https://thepowerups-learning.com/curso-de-solid/>