**SOLID: los 5 principios que te ayudarán a desarrollar software de calidad**

Si hablamos de diseño y desarrollo de aplicaciones, **Principios SOLID** son unas palabras que debes conocer como uno de los fundamentos de la [arquitectura y desarrollo de software](https://profile.es/creamos-soluciones-digitales/custom-development/).

**SOLID** es el acrónimo que acuñó [Michael Feathers](https://twitter.com/mfeathers), basándose en los principios de la [programación orientada a objetos](https://profile.es/blog/que-es-la-programacion-orientada-a-objetos/) que Robert C. Martin había recopilado en el año 2000 en su paper “[Design Principles and Design Patterns](http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/materials_adicionals/principles_and_patterns.pdf)”.

Ocho años más tarde, [*el tío Bob*](https://twitter.com/unclebobmartin) siguió compendiando **consejos y buenas prácticas de desarrollo** y se convirtió en el padre del código limpio con su célebre libro *Clean Code*, del que ya hablamos en la recopilación de [libros esenciales para desarrolladores](https://profile.es/blog/libros-esenciales-para-desarrolladores-y-programadores/).

Los **5 principios SOLID** de diseño de aplicaciones de software son:

* S – Single Responsibility Principle (SRP)
* O – Open/Closed Principle (OCP)
* L – Liskov Substitution Principle (LSP)
* I – Interface Segregation Principle (ISP)
* D – Dependency Inversion Principle (DIP)

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

* Crear un **software eficaz**: que cumpla con su cometido y que sea **robusto y estable**.
* Escribir un **código limpio y flexible** ante los cambios: que se pueda modificar fácilmente según necesidad, que sea **reutilizable** y **mantenible**.
* Permitir **escalabilidad**: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

En definitiva, desarrollar un **software de calidad**.

En este sentido la aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de [**patrones de diseño**](https://profile.es/blog/patrones-de-diseno-de-software/), que nos permitirán mantener una **alta cohesión** y, por tanto, un **bajo acoplamiento** de software.

¿Qué son la cohesión y el acoplamiento?

Son dos conceptos muy relevantes a la hora de diseñar y desarrollar software. Veamos en qué consisten.

Acoplamiento

El acoplamiento se refiere al **grado de interdependencia que tienen dos unidades de software entre sí**, entendiendo por unidades de software: clases, subtipos, métodos, módulos, funciones, bibliotecas, etc.

Si dos unidades de software son completamente independientes la una de la otra, decimos que están desacopladas.

Cohesión

La cohesión de software es el **grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado** que si trabajaran por separado. Se refiere a la forma en que podemos agrupar diversas unidades de software para crear una unidad mayor.

1. Principio de Responsabilidad Única

***“A class should have one, and only one, reason to change.”***

La S del acrónimo del que hablamos hoy se refiere a **Single Responsibility Principle (SRP)**. Según este principio “una clase debería tener **una, y solo una, razón para cambiar**”. Es esto, precisamente, “razón para cambiar”, lo que Robert C. Martin identifica como “responsabilidad”.

El principio de Responsabilidad Única es **el más importante y fundamental de SOLID**, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

El propio Bob resume cómo hacerlo: *“Gather together the things that change for the same reasons. Separate those things that change for different reasons”*, es decir: “Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes”.

2. Principio de Abierto/Cerrado

***“You should be able to extend a classes behavior, without modifying it.”***

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “[Object Oriented Software Construction](https://sophia.javeriana.edu.co/~cbustaca/docencia/POO-2016-01/documentos/Object%20Oriented%20Software%20Construction-Meyer.pdf)” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar **abiertas para poder extenderse y cerradas para modificarse**.

En su blog Robert C. Martin defendió este principio que [a priori puede parecer una paradoja](https://blog.cleancoder.com/uncle-bob/2014/05/12/TheOpenClosedPrinciple.html). Es importante tener en cuenta el **Open/Closed Principle (OCP)** a la hora de desarrollar **clases,**[**librerías**](https://profile.es/blog/librerias-javascript/)**o frameworks.**

3. Principio de Sustitución de Liskov

***“Derived classes must be substitutable for their base classes.”***

La L de SOLID alude al apellido de quien lo creó, [Barbara Liskov](https://www.eldiario.es/sociedad/Barbara-Liskov-medidas-mujeres-ingenierias_0_780422139.html), y dice que **“las clases derivadas deben poder sustituirse por sus clases base”.**

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, **deberíamos poder usar cualquiera de sus subclases** sin interferir en la funcionalidad del programa.

Según Robert C. Martin incumplir el **Liskov Substitution Principle (LSP)** implica violar también el principio de Abierto/Cerrado.

4. Principio de Segregación de la Interfaz

***“Make fine grained interfaces that are client specific.”***

En el cuarto principio de SOLID, *el tío Bob*sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para **una finalidad concreta**.

En este sentido, según el **Interface Segregation Principle (ISP),**es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.

5. Principio de Inversión de Dependencias

***“Depend on abstractions, not on concretions.”***

Llegamos al último principio: “**Depende de abstracciones**, no de clases concretas”.

Así, Robert C. Martin recomienda:

1. Los módulos de alto nivel **no deberían depender de módulos de bajo nivel**. Ambos deberían depender de abstracciones.
2. **Las abstracciones no deberían depender de los detalles**. Los detalles deberían depender de las abstracciones.

El objetivo del **Dependency Inversion Principle (DIP)** consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

Críticas a SOLID

El ámbito del desarrollo de software es un terreno de **continuo debate** y SOLID no se queda fuera de la controversia.

Aunque estos cinco principios son considerados por muchos como **una base fundamental de un buen desarrollo** o al menos como una guía a tener en cuenta, no son pocos los profesionales que critican los principios SOLID.

Los acusan de **ambiguos, confusos, de complicar el código, de demorar el proceso de desarrollo** y los tildan incluso de totalmente equivocados e innecesarios.

Aquí tienes un par de**ejemplos de estas críticas**:

* [The SOLID Design Principles – Absolute Nonsense](https://jamesmccaffrey.wordpress.com/2016/08/24/the-solid-design-principles-absolute-nonsense/)
* [Why Every Element of SOLID is Wrong](https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong)

Y como a toda acción le corresponde una reacción, las **respuestas en defensa de SOLID**:

* [In Defense of the SOLID Principles](https://blog.ndepend.com/defense-solid-principles/)
* [Why Every Single Argument of Dan North is Wrong](https://www.entropywins.wtf/blog/2017/02/17/why-every-single-argument-of-dan-north-is-wrong/)

Conclusión

Los principios SOLID son eso: principios, es decir, **buenas prácticas** que pueden ayudar a escribir un mejor código: más limpio, mantenible y escalable.

Como indica el propio Robert C. Martin en su artículo [“Getting a SOLID start”](https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start) **no se trata de reglas, ni leyes, ni verdades absolutas**, sino más bien soluciones de sentido común a problemas comunes. **Son heurísticos, basados en la experiencia**: “se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir.”

Dice *el tío Bob*, que SOLID nos ayuda a categorizar lo que es un buen o mal código y es innegable que un código limpio tenderá más a salir airoso del “control de calidad de código” **WTFs/Minute**. Consejo: cuando estés revisando un código, lleva la cuenta de cuántas veces por minuto sale de tu boca un *WTF?*

