# PATRONES SOLID

Los principios SOLID son recomendaciones para escribir un mejor código, que sea más mantenible, reusable con alta cohesión y bajo acoplamiento.

En la práctica es una forma de reorganizar el código utilizando interfaces para que más adelante cuando el sistema crezca no haya tantos problemas y no se tenga que cambiar todo lo que se ha escrito, solo para agregar un cambio solicitado por el usuario(cliente).

## SINGLE RESPONSABILITY PRINCIPLE (Principio de responsabilidad única)

Una clase debe tener una y solamente una razón para cambiar. Esto lo que quiere decir es que la clase debe tener una sola responsabilidad.

Ejemplo:

### Problema

Clase ‘Order Service’

Metodo ‘Save Order’: Se inserta la orden, se crea un invoice a partir de la orden, envía un email al usuario (la persona que compró o realizó la orden) y además va a escribir un log.

Si se necesita modificar el cuerpo del método ‘CreateInvoice’ se hace necesario modificar la clase ‘OrderService’ o cambiar algo en la forma en que se accede a los logs, también se debe modificar la clase ‘OrderService’ o el envío de email. Son demasiadas responsabilidades para una misma clase. Que, aunque se ve sencilla puede crecer.

### Solución

La clase ‘OrderRepository’ tiene claramente una sola responsabilidad, “Crear una orden”.

Lo mismo para las demás, por ejemplo ‘InvoiceService’, crear el Invoice.

“LoggerService” se encarga únicamente del servicio de Log y de nada más.

La clase “OrderService” sigue teniendo las mismas líneas de código, pero ahora llama a la dependencia que tiene que llamar. Y cada servicio se encarga de lo que debe hacer bajo su responsabilidad.

Con esto se cumple el principio de responsabilidad única. Así la clase “OrderService” se encargará de conectar los diferentes servicios. Esa es su única responsabilidad.

## OPEN-CLOSE PRINCIPLE (Principio de abierto y cerrado)

Se debería ser capaz de extender el comportamiento de una clase ‘ReportingService’’ sin modificar la misma clase.

Ejemplo:

### Problema

La clase ‘ReportingService’

Si se necesita que en la clase ‘ReportingService’ ahora exporte reportes en Json o XML, habría que construir estos 2 métodos adicionales. O si se necesita modificar la generación de alguno de estos tipos de reportes ya existentes. Entonces esta clase va a sufrir el paso del tiempo.

### Solución

La clase ‘ReportingService’ no se tiene que tocar nunca más.

En su constructor recibe una interfaz que es el generador de reportes. A esta clase no le importa como se generan los reportes ni de qué tipo es. Lo único que le importa es que a partir de una lista de ordenes se genere un reporte.

El ‘cómo’ lo van a definir las extensiones que están en IReportGenerator (ReportGeneratorExcel, ReportGeneratorJSON, ReportGeneratorPDF)

## LISKOV SUBSTITUTION PRINCIPLE (Principio de sustitución de liskov)

Las clases derivadas deben poder sustituirse por su clase base. Los objetos tienen que poder ser reemplazados por las instancias o sus propias instancias (los objetos que heredan de ese mismo objeto) sin que el sistema se rompa, sin alterar el funcionamiento del sistema.

Ejemplo:

### Problema

La class ‘Bike’ hereda de vehiculo pero no va a poder tener ninguna definición e implementación para los métodos aterrizar y despegar. Este diseño rompe con el principio de substitución.

Lo mismo ocurre con la clase ‘Car’

### Solución

En lugar de tener una clase abstracta, se deben utilizar interfaces con el comportamiento puntual.

Ahora las clases, pueden implementar la cantidad de interfaces que necesite, y de esa manera ya no se rompe el principio de sustitución. Así cada clase va a implementar solamente el comportamiento que realmente puede realizar.

## INTERFACE SEGREGATION PRINCIPLE (Principio de segregación de interface)

Muchas interfaces específicas son mejores que una gran interface que trate de abarcar todo.

La idea es que cuando se crea una interface, esta interface tenga una finalidad muy concreta.

Ejemplo:

### Problema

La interface ‘’ exponer 3 métodos. Uno para ordenar hamburguesas, carne o ensaladas.

Si se tiene un servicio dedicado a las hamburguesas, la clase ‘BurgerOrderService’, que implementa a interface ‘’ solo va a poder implementar el método para ordenar la hamburguesa. Los otros métodos no se pueden implementar.

### Solución

Se necesita una interface para cada pedido puntual. Una para la orden de la hamburguesa, otra para la ensalada y otra para la carne.

Luego cada uno de los servicios puntuales va a implementar esa interface específica.

De esta manera se ha segregado las interfaces.

## DEPENDENCY INVERSION PRINCIPLE (Principio de inversión de dependencias)

Hay que depender de abstracciones y no de clases concretas.

‘DataDog’ es un servicio de monitoreo de aplicaciones. Me permite loguear eventos.

Ejemplo:

### Problema

El problema es que Datadog no es una abstracción sino una clase concreta. Y si más adelante se cambia el sistema de generación de log por otro, entonces se debe cambiar el servicio de ordenes ‘OrderService’

### Solución

Se debe abstraer el servicio de notificaciones.

Se crea una interface, que expone un método para crear un log a partir de un evento, sin importar hacia donde se haga.

Se crea un servicio que va a implementar esa interface para loguear el evento. Se crea otro servicio ‘AnyOtherMonitoringService’ que también va a implementar esa interface y va a tener el mismo método.

Ahora ‘Order Service’ ya no depende de ‘DataDog’, ahora depende de la abstracción. Si se cambia Datadog, nunca se va a tener que modificar ‘OrderService’, porque su responsabilidad no tiene que ver con el Loguin de eventos.