

**DESARROLLO DE APLICACIONES CLIENTE-SERVIDOR**

**2019**

***Patrones de Diseño***

***SOLID***

# 

# **GRUPO Nº7**

Integrantes:

* **AGUILERA**, Diego Leonardo
* **BENITEZ PERESSI**, Gonzalo
* **ESCALANTE**, Salvador Manuel
* **GAMARRA**, María Luciana
* **VIDAL**, Raul Antonio

Índice

[**PRINCIPIOS SOLID**](#_fk5xwh7go31e) **3**

[**S → SRP → Single Responsibility Principle**](#_3syu3lqycwjq) **4**

[**O→ OCP →Open / Closed Principle**](#_5p4tiplbo5y) **9**

[**L → LSP →Liskov Substitution Principle**](#_q4bst1k0whdz) **14**

[**I → ISP →Interface Segregation Principle**](#_b2s0brgkcxxx) **17**

[**D → DIP →Dependency Inversion Principle**](#_hvhue1dew0y) **22**

[**Bibliografía**](#_naaik9o7cojp) **26**

# 

# 

# 

# 

# PRINCIPIOS SOLID

Los principios SOLID son cinco principios enunciados por Robert C. Martin alrededor del año 2000 enfocados a la elaboración de software de calidad. Una clara definición de los principios SOLID la podemos encontrar aquí:

En ingeniería de software, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo mnemónico introducido por Robert C. Martin a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo. Los principio SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar código sucio provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Debe ser utilizado con el desarrollo guiado por pruebas o TDD, y forma parte de la estrategia global del desarrollo ágil de software y programación adaptativa. Ésta definición es perfectamente válida y se podría resumir diciendo que son cinco principios que hay que tener siempre presentes si queremos desarrollar un software de calidad, legible, entendible y fácilmente testeable. ¿Por qué fácilmente testeable? Principalmente porque nos va a obligar a sacar muchas interfaces, cosa que favorecerá la utilización de Mocking, y hará que el trabajo de testear una clase sea conciso, además favorece el TDD porque se prepara el código para los sucesivos cambios propios del TDD.

|  |  |  |
| --- | --- | --- |
| S | SRP: Single Responsibility Principle: | Un objeto debería tener una única responsabilidad. |
| O | OCP: Open/Closed Principle | Las entidades de software deberían estar abiertas para extensión pero cerradas para modificación. |
| L | LSP: Liskov Substitution Principle | Un objeto en un programa podría ser reemplazado con instancias de sus subtipos sin alterar la correctitud del programa. |
| I | ISP: Interface Segregation Principle | Muchas interfaces específicas son mejores que interfaces de propósitos generales. |
| D | DIP: Dependency Inversion Principle | Deberíamos depender de las abstracciones y no de las concreciones. |

En ésta sesión de trabajo veremos, uno por uno, cada principio, siguiendo este guión con cada uno de ellos

1. Definición original, en inglés.

2. Traducción literal.

3. Interpretación.

4. Un ejemplo que no cumple el principio y una explicación de por qué no lo cumple que incluye diagramas estáticos UML y código C# o java.

5. Trabajo en grupo para arreglar dicho ejemplo aplicando el principio en cuestión. Así que manos a la obra…

# S → SRP → Single Responsibility Principle

**Principio de única responsabilidad**

Enunciado original: "There should never be more than one reason for a class to change

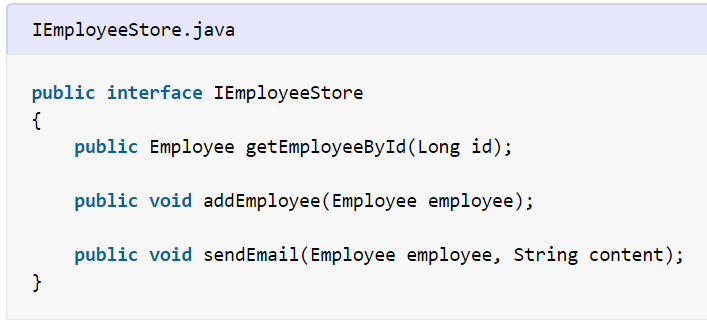
Traducción literal: "No debería haber nunca más de una razón para cambiar una clase." Interpretación: Una clase debería concentrarse sólo en hacer una cosa de tal forma que cuando cambie algún requisito en mayor o menor medida dicho cambio sólo afecte a dicha clase por una razón.

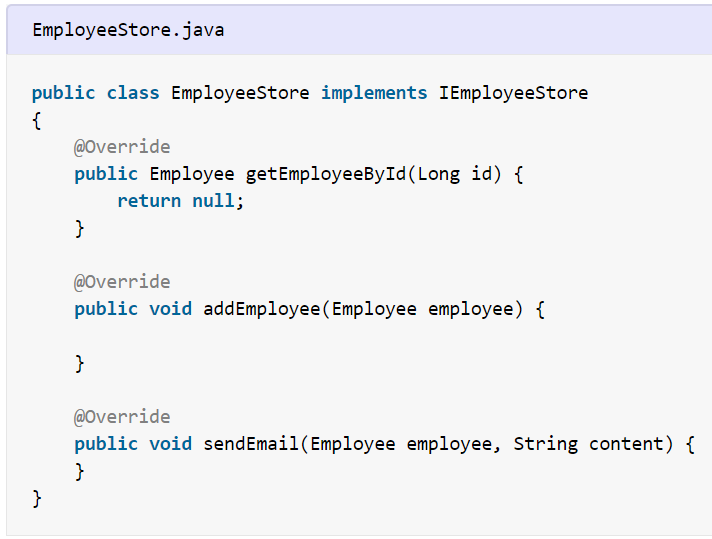
**Motivación**

Si tenemos que hacer modificaciones a una clase, **por diferentes motivos**, significa que la **abstracción es incorrecta** y que la **clase tiene demasiadas responsabilidades**. Básicamente, lo que queremos **evitar en todos los casos es tener objetos con múltiples responsabilidades** (a menudo llamados objetos Dios , porque saben demasiado o más de lo que deberían). Estos **objetos agrupan diferentes comportamientos (en su mayoría no relacionados), lo que los hace más difíciles de mantener.**

#### **Problema**

Para comprender el principio de SRP, supongamos que estamos trabajando en una aplicación que implica trabajar con los empleados. Tenemos una interfaz IEmployeeStore y su implementación EmployeeStore que tiene los siguientes métodos.





Mirando en general pareciera que la clase tiene un comportamiento para cualquier aplicación normal. Utilizando EmployeeStore, se puede obtener / agregar empleados y enviarles un correo electrónico.

Ahora, supongamos que después del lanzamiento del producto, tenemos el requisito de que el contenido del correo electrónico puede ser de dos tipos, es decir, HTML y texto.

Una forma de resolver este problema es crear otro método sendHtmlEmail () , pero qué sucede cuando se nos pide que admitamos diferentes protocolos para enviar correos electrónicos para ambos tipos de contenido. En general, la clase se verá muy fea, difícil de leer y mantener.

**¿Por qué incumple el principio de única responsabilidad?**

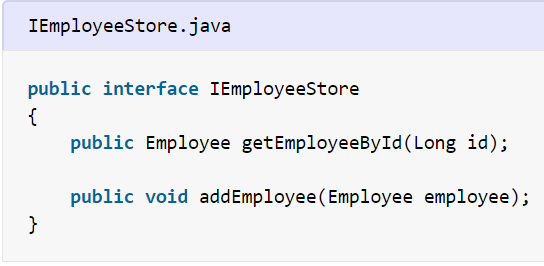
SRP nos dice que sólo debería haber un motivo para cambiar una clase o una interfaz y, por ende, las clases que la implementan. ¿Qué motivos veis que puedan hacer que la clase cambie? En éste caso la traducción literal es clara, si cambiará el contenido porque, por ejemplo, admitiera más tipos, habría que modificar el código cada vez que añadamos algún tipo de contenido o si, por ejemplo, cambiará el protocolo y quisiéramos añadir más campos también habría que modificar la clase email. Por tanto hay más de un motivo por el que tendríamos que modificar ésta clase.

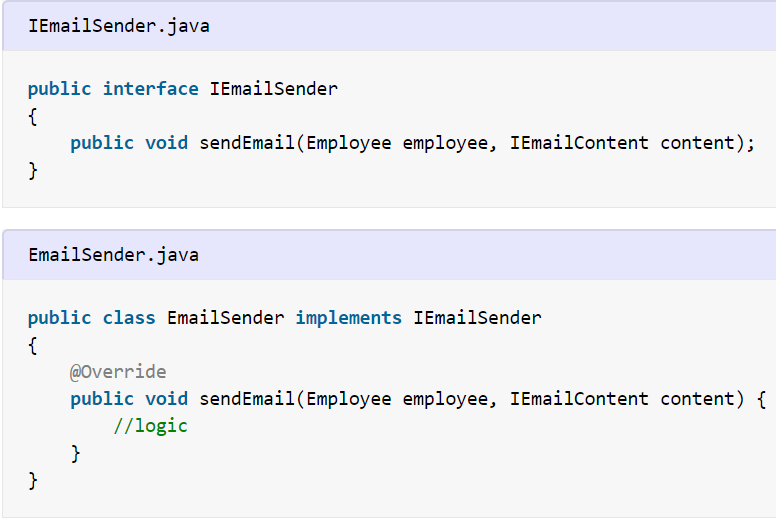
#### **La solución es el principio de SRP**

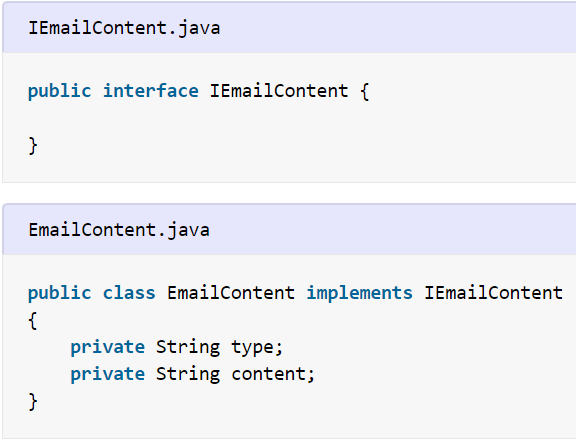
Para resolver este problema, debemos extraer la capacidad del correo electrónico mediante el uso de una interfaz y de clases que manejan específicamente solo la funcionalidad relacionada con el correo electrónico. De esta manera, estamos seguros de que otras características no se ven afectadas.

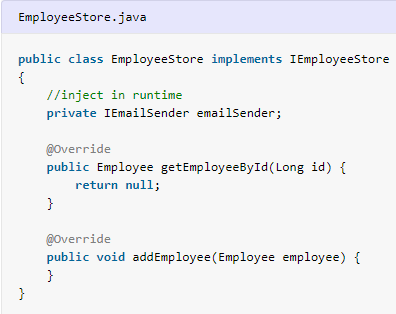
Sobre la base de nuestros supuestos, podemos abstraer dos interfaces IEmailSender y IEmailContent. La primera interfaz es responsable del proceso de envío de correo electrónico y la segunda interfaz es responsable de pasar el contenido del correo electrónico y su tipo.

Las clases refactorización se dan a continuación.









Ahora, si queremos cambiar la funcionalidad del correo electrónico, solo cambiaremos la clase EmailSender. Cualquier cambio en las operaciones de CRUD del empleado ocurrirá solo EmployeeStore . Cualquier cambio en una capacidad no cambiará a otra por error. De esta fácil todas las clases son más fáciles de leer y de mantener también.

**Beneficios**

#### **Fácil de entender y mantener**

Cuando la clase solo hace "una cosa", su interfaz generalmente tiene un número menor de métodos (y variables de miembros) que son bastante autoexplicativos. Hace que el código sea más fácil de leer y entender.

Cuando hay una necesidad de cambiar el comportamiento de la aplicación, los cambios relacionados con la responsabilidad de la clase son bastante aislados. Reduce la posibilidad de romper otras áreas no relacionadas del software. Hace que el código sea más fácil de mantener.

#### **Usabilidad mejorada**

Si una clase tiene múltiples responsabilidades y se requiere que se use en otras partes de la aplicación para usar una cierta responsabilidad, puede exponer negativamente otras responsabilidades que pueden no ser deseadas. Puede llevar a un comportamiento no deseado en la aplicación, por ejemplo, problemas de seguridad y privacidad de datos.

Si la clase sigue estrictamente el principio de SRP, entonces no se expondrá la funcionalidad innecesaria y hará que la clase sea más utilizable sin temer los efectos adversos

**Conclusión**

El patrón de diseño del principio de responsabilidad única tiene un impacto enormemente positivo en la adaptabilidad del código. En comparación con el código equivalente que no se adhiere al principio, el código compatible con SRP conduce a un mayor número de clases que son más pequeñas y tienen un alcance más directo.

El SRP se logra principalmente al abstraer el código detrás de las interfaces y delegar la responsabilidad de la funcionalidad no relacionada a cualquier implementación que se encuentre detrás de la interfaz en el tiempo de ejecución.

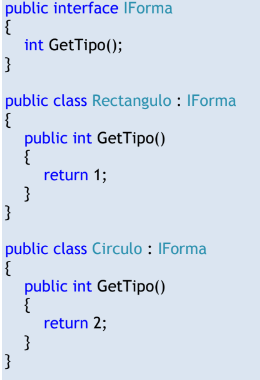
# O → OCP →Open / Closed Principle

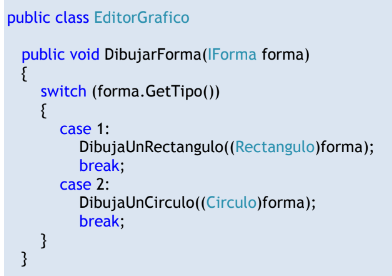
**Principio abierto/cerrado**

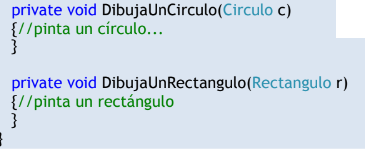
Las entidades de SW deberían estar abiertas a la extensión y cerradas a la modificación. Entendiendo por entidades de SW a las clases, funciones y módulos y demás. No se rechaza el cambio cuando hablamos de modificación, sino que su filosofía es anticiparse al cambio, que estos puedan implementarse en nuestras clases mediante herencia y composición.

Si tenemos que una herramienta para dibujar un círculo y un rectángulo.

Veamos el siguiente Código :







Este diseño no cumple con el criterio open/close, la clase editor gráfico no está abierta a la extensión. Si quisiéramos añadir una nueva figura geométrica para su pintado, habría que crear la nueva clase y habría que modificar el método público y añadir uno privado para el pintado de dicha nueva forma.

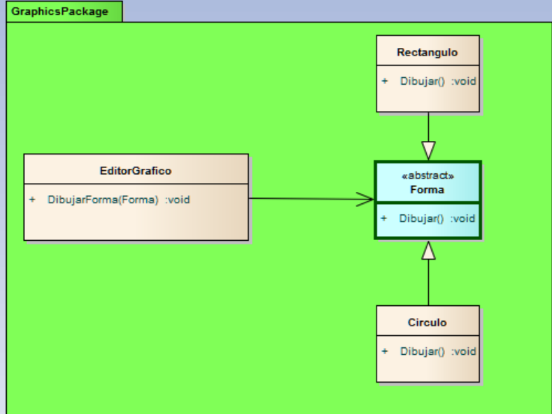
Las distintas formas geométricas que tenemos y que probablemente tengamos, si nos piden que añadamos alguna otra tendrán una responsabilidad, la de dibujarse a sí mismas. Si modificaramos todo éste código y en vez de una interfaz tuviéramos una clase abstracta “Forma” de la que heredarán las demás e implementaran un método abstracto “Dibujar” nos quitaremos de un plumazo el problema de la extensión y el problema del motivo de cambio.

Analicemos las siguientes cuestiones:

¿Cuál es la responsabilidad principal del Editor Gráfico? ¿Debería el Editor conocer

cómo se dibuja cada forma? Deberíamos abstraernos un poco.

Con el siguiente diagrama de clases uml solucionamos la question:

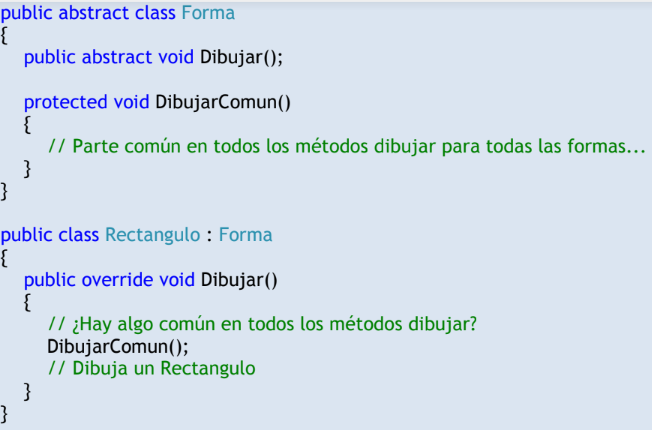


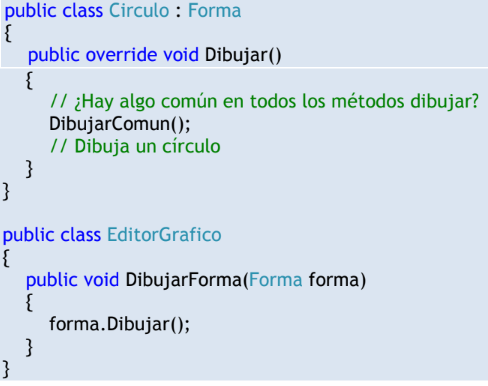
Obtenemos un diseño más sencillo, más robusto y sobre todo más cerrado a las modificaciones pues un círculo siempre se pinta igual y un rectángulo también, pero a la vez abiertos a la extensión porque podemos nuevas formas sin tener que modificar código anterior en el “ EditorGráfico”. Siempre que tengamos comportamientos que dependen del tipo pensemos en si se puede implementar con una clase abstracta.

Una de las ventajas de las clases abstractas es que podemos colocar en ellas todo el código que es común y nos evitamos tener que mantener bloques iguales de código en distintos sitios.

Si vas de cambiar una clase hazlo usando herencia y polimorfismo. No hay que temer el uso de clases abstractas ni interfaces sino al revés, potenciarlo, ya que son una ayuda contra el acoplamiento del software, te protegen frente a cambios.

El código quedaría de la siguiente forma:





# L → LSP → Liskov Substitution Principle

**Principio de Sustitución de Liskov**

**Enunciado:** *“Si por cada objeto O1 de tipo S hay un objeto O2 de tipo T tal que para todos los programas P que están definidos en términos de T, el comportamiento de P no cambia cuando O1 es sustituido por O2 , entonces S es un subtipo de T”*

En otras palabras, *“Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa”*

Se pueden encontrar este principio enunciado de diferentes maneras, como son:

* Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.
* Toda subclase debe soportar ser sustituida por su clase base lo que en términos de contratos significa que las precondiciones de los métodos de la subclase no serán más fuertes que las de la clase base, y sus pos condiciones no serán más débiles (coloquialmente: los métodos derivados no deben esperar más ni proveer menos que los originales).
* Las funciones que usan punteros o referencias a clases base deben ser capaces de utilizar objetos de las clases derivadas sin tener conocimiento de ello.
* Las clases derivadas deben ser utilizadas a través de la interfaz de la clase base, sin necesidad de que el usuario conozca la diferencia.

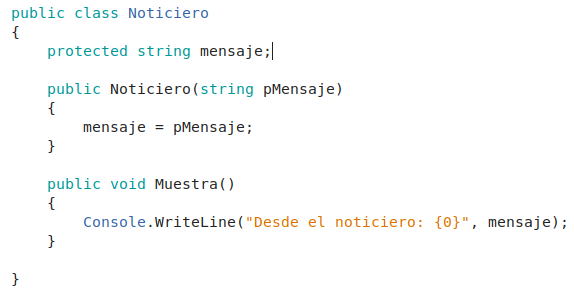
En lenguajes OO como C# o VB.NET, la clave para conseguir la abstraccion y polimorfismo de entidades es mediante la herencia, y es precisamente en esta característica en la que se basa el Principio de Sustitución de Liskov. El principio indica cuáles son los fundamentos básicos de diseño que debe seguir la herencia en un caso particular, o cuál es la mejor forma de crear jerarquías de herencia entre clases. Porque el hecho de que una clase herede de otra, no asegura que se cumpla el principio LSP, si no se respetan también sus “contratos de diseños”.

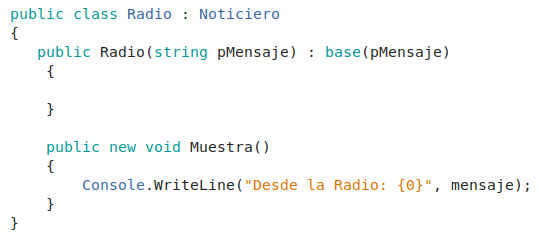
Ejemplo:

Enunciado: Partimos del caso que una radio es un noticiero, y ambos nos dan un mensaje.

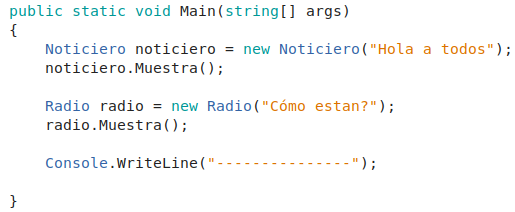
Código:

Creamos la clase Noticiero y la subclase Radio



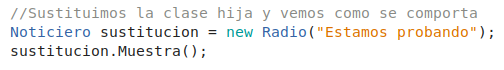


Programa principal



**¿Por qué incumple el principio?**

El comportamiento del programa fue correcto, pero agreguemos la siguiente línea para probar el polimorfismo:

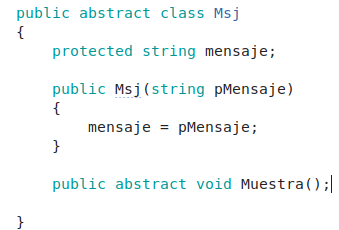


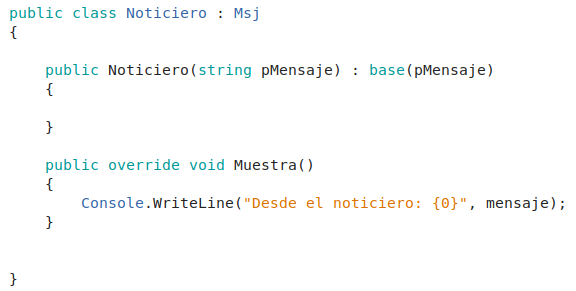
Una variable de tipo Noticiero está guardando una instancia de tipo Radio, así que esta definición es correcta. Pero al momento de ejecutar, nos devuelve el método de la clase base.

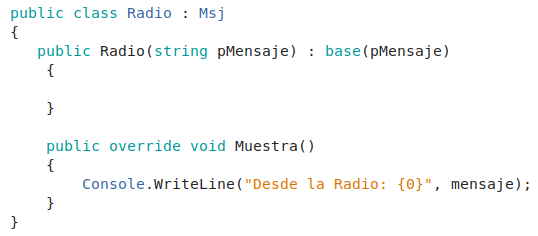
**Solución:**

Para evitar la pérdida de funcionalidad, debemos crear una nueva clase abstracta que la llamaremos Msj. En ella colocaremos todo aquello que queremos que sea constante y no sea modificado por las clases hijas, los métodos abstracto que se van a implementar de acuerdo a las clases hijas.

Así, en vez de que Radio herede de Noticiero, estas clases deben heredar de Msj.







**Conclusión:**

El principio de Liskov nos ayuda a utilizar la herencia de forma correcta, y a tener mucho más cuidado a la hora de extender clases.

Además, se deben tener dos criterios para usar clases bases, uno es dar a una aplicación un comportamiento común, y el otro usar métodos de ayuda para las clases derivadas

# I → ISP → Interface Segregation Principle

**Principio de segregación de la interfaz**

Varias interfaces específicas a los clientes son mejores que una interfaz de propósito general.

Ataca las desventajas de interfaces gordas (no cohesivas / contaminadas).

La interfaz gorda es dividida en grupos de métodos cohesivos.

Cada grupo sirve a un tipo de cliente específico.

Clientes no deben ser forzados a depender de interfaces que no usan.

Cuando un cambio en un componente afecta a otros componentes que no están relacionados, el costo de repercusión de cambios se torna impredecible, y el riesgo de romper algo se incrementa dramáticamente.

A continuación se muestra un ejemplo del principio de segregación de interfaces. Suponemos una aplicación ficticia que debe enviar correos electrónicos o mensajes de texto. En una primera aproximación, podríamos pensar en algo así:

interface IMessageSender

{

void sendToMail(string mail, string subject, string body);

void sendSMS(string recipient, string message);

}

class MailMessageSender : IMessageSender

{

public void sendSMS(SMSMessageDTO info)

{

/\* No implementado \*/

}

public void sendSMS(MailMessageDTO info)

{

// Enviar correo

}

}

De tal forma que el cliente utilizará MailMessageSender para enviar un correo electrónico sería algo así:

void sendCustomMail( MailInfoDTO info )

{

IMessageSender ms = new MailMessageSender();

ms.sendToMail(info.mail, info.subject, info.body);

}

Con esta implementación, aunque funcione, estamos violando el principio ISP, ya que MailMessageSender está usando la interfaz ISendMessage pero sólo está implementado uno de sus dos métodos.

Lo correcto y evidente en este ejemplo de juguete, sería crear una interfaz específica:

interface IMailMessageSender

{

void sendToMail(string mail, string subject, string body);

}

interface ISMSMessageSender

{

void sendSMS(string recipient, string message);

}

class MailMessageSender : IMailMessageSender

{

public void sendToMail(string mail, string subject, string body)

{

// Enviar correo

}

}

class SMSMessageSender : ISMSMessageSender

{

public void sendSMS(string recipient, string message)

{

// Enviar mensaje

}

}

De este modo, cada interfaz define la funcionalidad exacta de su propósito y las clases que derivan de ellas implementan exactamente lo que necesitan, generando clases pequeñas, acotadas y, por tanto, más testeables.

Por último, siguiendo con el ejemplo, estamos haciendo sendCustomMail() dependiente de una implementación concreta de IMailSendMessage, violando así el principio de inversión de dependencias (DIP). Una mejor implementación sería así:

void sendCustomMail( MailInfoDTO info, IMailMessageSender ms )

{

ms.sendToMail(info.mail, info.subject, info.body);

}

O bien de este otro modo utilizando una factoría de clases

void sendCustomMail( MailInfoDTO info )

{

IMailMessageSender ms = \_factory.getInstance();

ms.sendToMail(info.mail, info.subject, info.body);

}

# 

# D → DIP → Dependency Inversion Principle

**Principio de Inversión de Dependencias**

Enunciado original: "A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions."

Traducción literal: "A. Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.

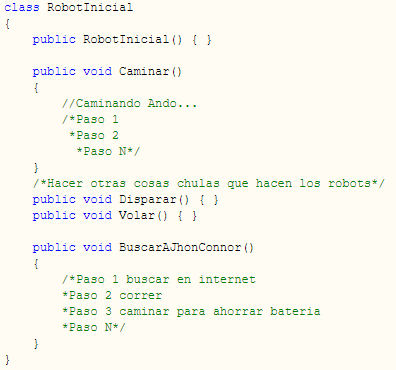
B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones."

Interpretación:para conseguir robustez y flexibilidad y para posibilitar la reutilización haz que tu código dependa de abstracciones y no de concreciones, esto es, utiliza muchas interfaces y muchas clases abstractas y, sobretodo, expón, por constructor o por parámetros, las dependencias que una clase pueda tener.

Este principio está principalmente basado en reducir las dependencias entre los módulos del código. Básicamente, este principio será de gran ayuda para entender cómo atar correctamente sistemas juntos. Si los detalles de tu implementación dependen de los altos niveles de abstracciones, te ayudará a conseguir un sistema bien acoplado. También, influirá en la encapsulación y cohesión de ese sistema.

Vamos con un ejemplo: Supongamos que estamos creando un videojuego y nuestro personaje es un Robot, este robot podría realizar diferentes acciones de diferentes maneras, por lo cual su cabeza puede ser conectada a infinidad de cuerpos distintos.

Planteamos una solución simple.

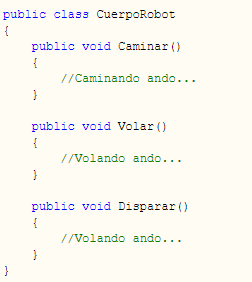


En este caso no se respeta la inversión de dependencias ni tampoco el principio de SRP (single responsibility) que señala: "Un objeto una responsabilidad”.

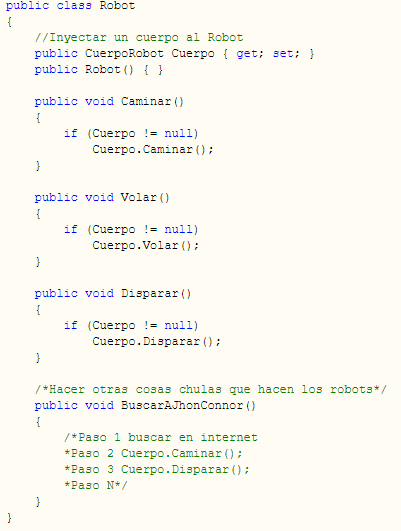
Así que para solucionar esto aplicamos el patrón de diseño “Inyección de Dependencias”, esto también podemos llamarlo Inyección de Comportamientos, que trata de colocar dentro de un objeto otros que puedan cambiar su comportamiento, sin que esto implique volver a crear el objeto.

La implementación habitual de Inyección de dependencias es crear un método para establecer el comportamiento, es decir capaz de cambiar el valor de un atributo asignándole una instancia de objeto diferente.

Ahora vamos a cambiar la implementación del robot por una que SI utilice la Inyección de dependencias. Creamos la clase [CuerpoRobot] que será la responsable de ejecutar las acciones del cuerpo:



Luego creamos un [Robot] que camina por medio de un [CuerpoRobot], delegando las responsabilidades a dicho objeto, además debemos establecer un mecanismo para asignar un [CuerpoRobot] a nuestro [Robot]



Como se puede observar la clase [Robot] tiene la acción caminar, pero la responsabilidad es de la clase [CuerpoRobot], también está claro que el valor de la propiedad [Cuerpo] se puede cambiar en tiempo de ejecución tan solo asignando una instancia diferente de la clase [CuerpoRobot ] a la propiedad [Cuerpo].

Esta es una primera aproximación a la inyección de dependencias, pero si nos ponemos más "formales", muchos comportamientos como Volar, Caminar, disparar, etc. podrían llegar a estar en clases distintas.

# 

# 

# 

# 

# 

# 

# 

# 

# Bibliografía

<https://jbravomontero.files.wordpress.com/2012/12/solid-y-grasp-buenas-practicas-hacia-el-exito-en-el-desarrollo-de-software.pdf>

<http://cs.uns.edu.ar/~atrellini/ayds/downloads/Clases/16.%20AyDS%20-%20Principios%20SOLID.pdf>

<https://www.rafablanes.com/codesample/cs04-isp-ejemplo>

<https://lbaigorria.com/2013/07/30/03-principio-de-sustitucion-de-liskov-solid/>

<http://joaquin.medina.name/web2008/documentos/informatica/documentacion/logica/OOP/Principios/Oop_Solid_LSP/2012_09_14_SOLID_ListkowSubstitutionPrincipe.html>

<https://desarrolloweb.com/articulos/principio-sustitucion-liskov-dotnet.html>

<https://howtodoinjava.com/design-patterns/single-responsibility-principle/>

<http://joaquin.medina.name/web2008/documentos/informatica/documentacion/logica/OOP/Principios/Oop_Solid_DIP/2012_09_14_DependencyInversionPrincipe.html>