|  |  |
| --- | --- |
|  | **Universidad Nacional de Ingeniería.**  **Facultad de Ingeniería Industrial y de Sistemas.** |



“Principios SOLID”

**Profesor:** Coronel Castillo, Eric Gustavo.

**Curso:** Programador Java.

**Integrantes:**

* Hinostroza Yañez, Evans del Jair
* Barrios Alarcón, Fernando Luis

Lima, Febrero de 2020.

Dedicatoria

*Este proyecto está dedicado a Dios en primer lugar por darme la fuerza, la fe y la entereza de seguir adelante cada día.*

*A mis padres por su educación, compromiso y lucha en mi crianza y formación como ser humano, por lo bueno y lo malo, por su amor y sus valores, los amo y los extraño.*

*A mi hermano Héctor Barrios, para que le sirva como ejemplo, que siga transitando el camino del bien, avance y nunca se rinda, siempre ha sido un buen hermano, un ejemplo de fuerza y un guerrero de la vida.*

*Y muy especialmente a mi novia Mary Margaret por su compañía y su apoyo incondicional en mis momentos más duros, por ser un ejemplo de lucha y de constancia, por ser una hija, una nieta, una amiga y una novia leal y firme, por sus principios y valores, te amo mucho.*

***Fernando Barrios.***

Contenido

[Dedicatoria 5](#_Toc33004872)

[Resumen 6](#_Toc33004873)

[1. Principios Solid 7](#_Toc33004874)

[1.1. Principio de responsabilidad única (Single responsibility principle) 7](#_Toc33004875)

[1.2. Principio de abierto / cerrado (Open/Closed Principle) 8](#_Toc33004876)

[1.3. Principio de sustitución de Liskov (Liskov substitution principle) 8](#_Toc33004877)

[1.4. Principio de Segregación de la Interfaz (Interface segregation principle) 9](#_Toc33004878)

[1.5. Principio de inversión de la dependencia (Dependency inversion principle) 9](#_Toc33004879)

[2. Casos desarrollados 9](#_Toc33004880)

[2.1 Aplicando el Principio de Responsabilidad Única 9](#_Toc33004881)

[2.2 Aplicando el Principio de Abierto/Cerrado 10](#_Toc33004882)

[2.3 Aplicando el Principio de Sustitución de Liskov 12](#_Toc33004883)

[2.4 Aplicando el Principio de Segregación de Interfaz 14](#_Toc33004884)

[2.5 Aplicando el Principio de inversión de dependencias 16](#_Toc33004885)

[Conclusión 19](#_Toc33004886)

[Recomendaciones 20](#_Toc33004887)

[Glosario de Nomenclaturas. 21](#_Toc33004888)

Resumen

El presente trabajo de investigación está enfocado en los Principios Solid, definirá sus conceptos de una manera sencilla y concisa, implementando una serie de ejemplos y definiciones con un lenguaje sencillo.

Se hace de esta manera ya que una gran parte de la comunidad de programadores no aplica estos principios al momento de desarrollar un software debido a su complejidad, haciendo que muchos de estos programas sean difíciles de entender, mantener y extender, incluso para su mismo programador o para otro que intente trabajar en su código.

Se mostrarán ejemplos detallados y explicados paso a paso de una manera sutil y sencilla.

**Definición:**

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 2003 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 con el tiempo. ​ Los principios 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. Puede ser utilizado con el desarrollo guiado por pruebas o TDD, y forma parte de la estrategia global del desarrollo ágil de software y desarrollo adaptativo de software.

1. Principios Solid
   1. Principio de responsabilidad única (Single responsibility principle)

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”.

En programación orientada a objetos, se suele definir como principio de diseño que cada clase debe tener una única responsabilidad, y que esta debe estar contenida únicamente en la clase. Así:

* Una clase debería tener solo una razón para cambiar
* Cada responsabilidad es el eje del cambio
* Para contener la propagación del cambio, debemos separar las responsabilidades.
* Si una clase asume más de una responsabilidad, será más sensible al cambio.
* Si una clase asume más de una responsabilidad, las responsabilidades se acoplan.
  1. Principio de abierto / cerrado (Open/Closed Principle)

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” 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.

Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías o frameworks.

* 1. Principio de sustitución de Liskov (Liskov substitution principle)

Es un principio de la programación orientada a objetos. Y puede definirse como: Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. En lenguaje más formal: si S es un subtipo de T, entonces los objetos de tipo T en un programa de computadora pueden ser sustituidos por objetos de tipo S (es decir, los objetos de tipo S pueden sustituir objetos de tipo T), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.). Más formalmente, el PSL es una definición particular de una relación de subtipificación, llamada tipificación (fuerte) del comportamiento, que fue introducido inicialmente por Barbara Liskov en una conferencia magistral en 1987 titulada La Abstracción de Datos y Jerarquía. Esto se refiere más a una relación semántica que a una relación sintáctica, ya que sólo tiene la intención de garantizar la interoperabilidad semántica de tipos en una jerarquía, los tipos de objeto en particular. Liskov y Jeannette Wing formularon el principio de manera conjunta en un artículo en el año 1994 de la siguiente manera:

Sea ϕ(x) una propiedad comprobable acerca de los objetos x de tipo T. Entonces ϕ(y) debe ser verdad para los objetos y del tipo S donde S, es un subtipo de T.

* 1. Principio de Segregación de la Interfaz (Interface segregation principle)

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas 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. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones. El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

* 1. Principio de inversión de la dependencia (Dependency inversion principle)

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.

1. Casos desarrollados
   1. Aplicando el Principio de Responsabilidad Única

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término “decoupled” en inglés). Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

Considera este ejemplo:



Como podemos observar, la clase Coche permite tanto el acceso a las propiedades de la clase como a realizar operaciones sobre la BBDD, por lo que la clase ya tiene más de una responsabilidad.

Supongamos que debemos realizar cambios en los métodos que realizan las operaciones a la BBDD. En este caso, además de estos cambios, probablemente tendríamos que tocar los nombres o tipos de las propiedades, métodos, etc. Cosa que no parece muy eficiente porque solo estamos modificando cosas que tienen que ver con la BBDD.

Para evitar esto, debemos separar las responsabilidades de la clase, por lo que podemos crear otra clase que se encargue de las operaciones a la BBDD.

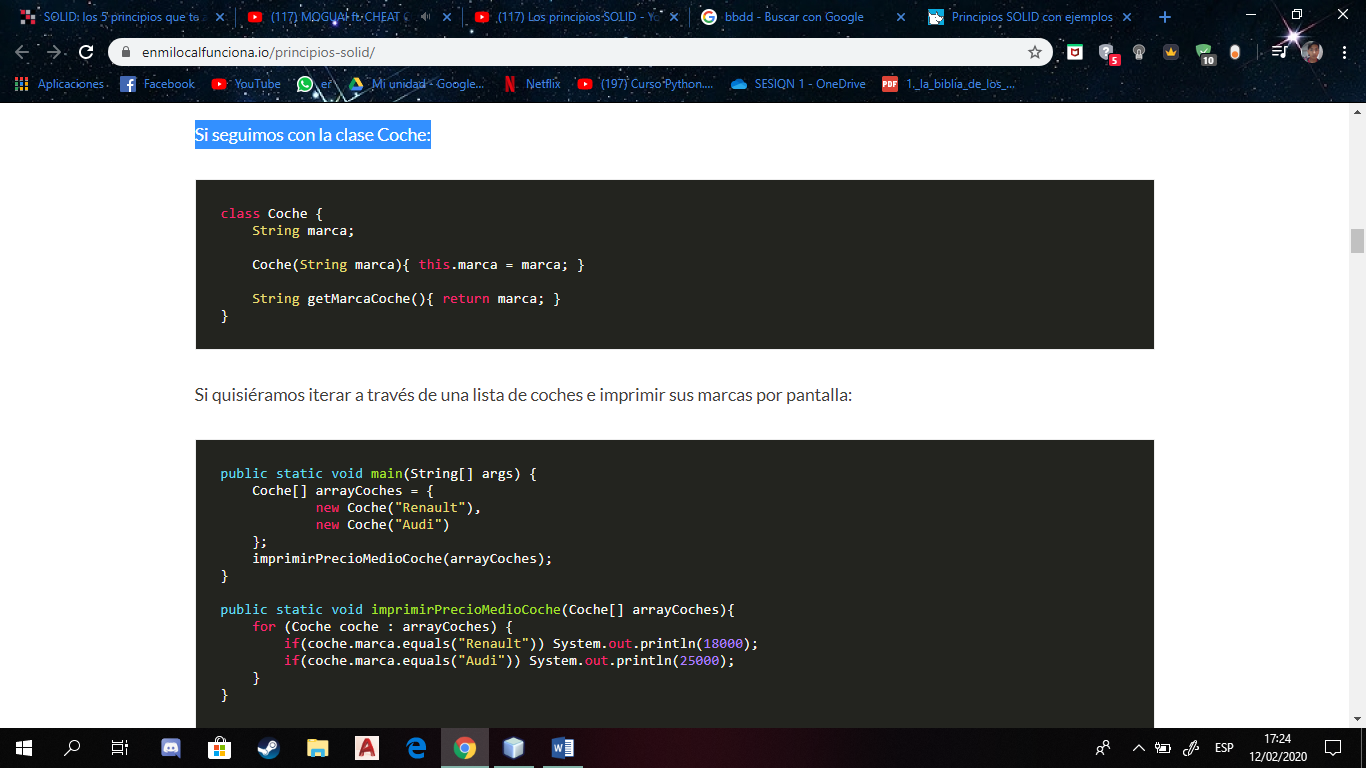


El programa será mucho más cohesivo y estará más encapsulado aplicando este principio.

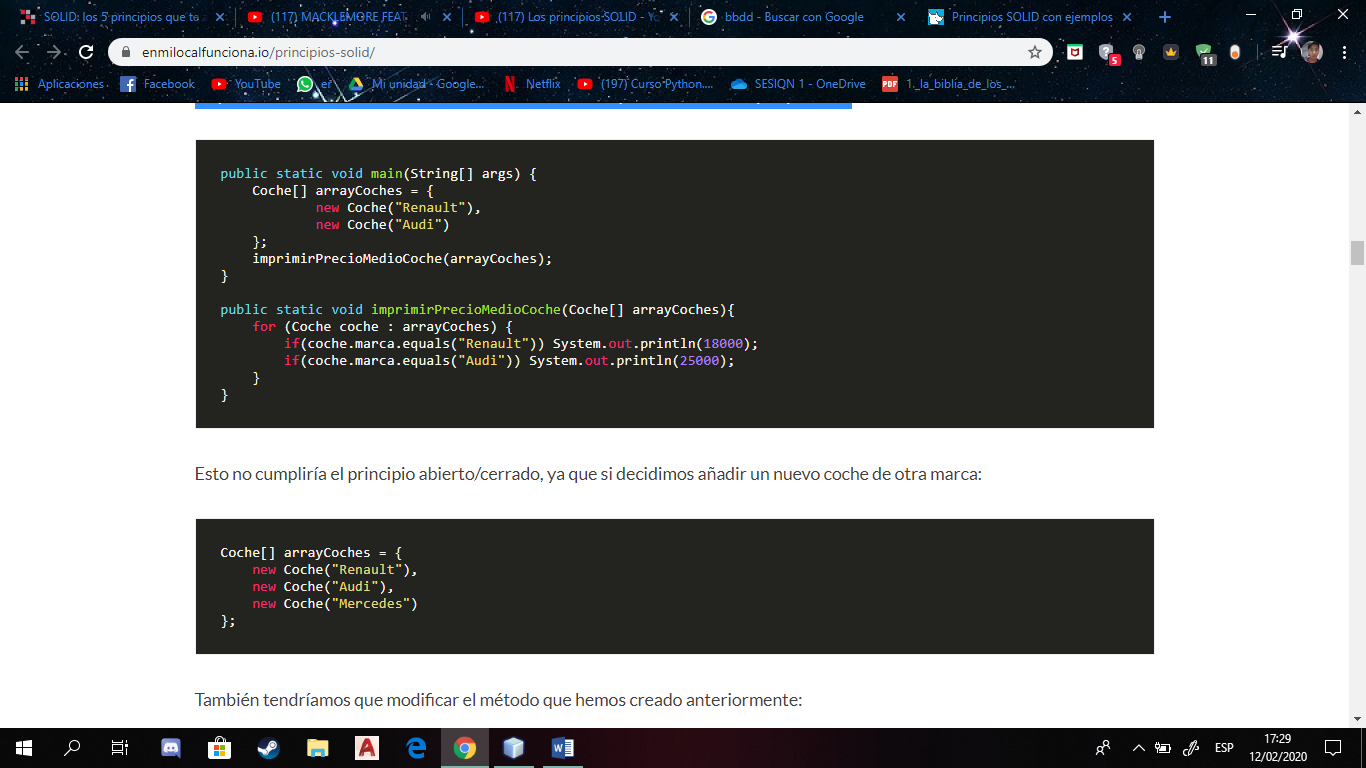
* 1. Aplicando el Principio de Abierto/Cerrado

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.

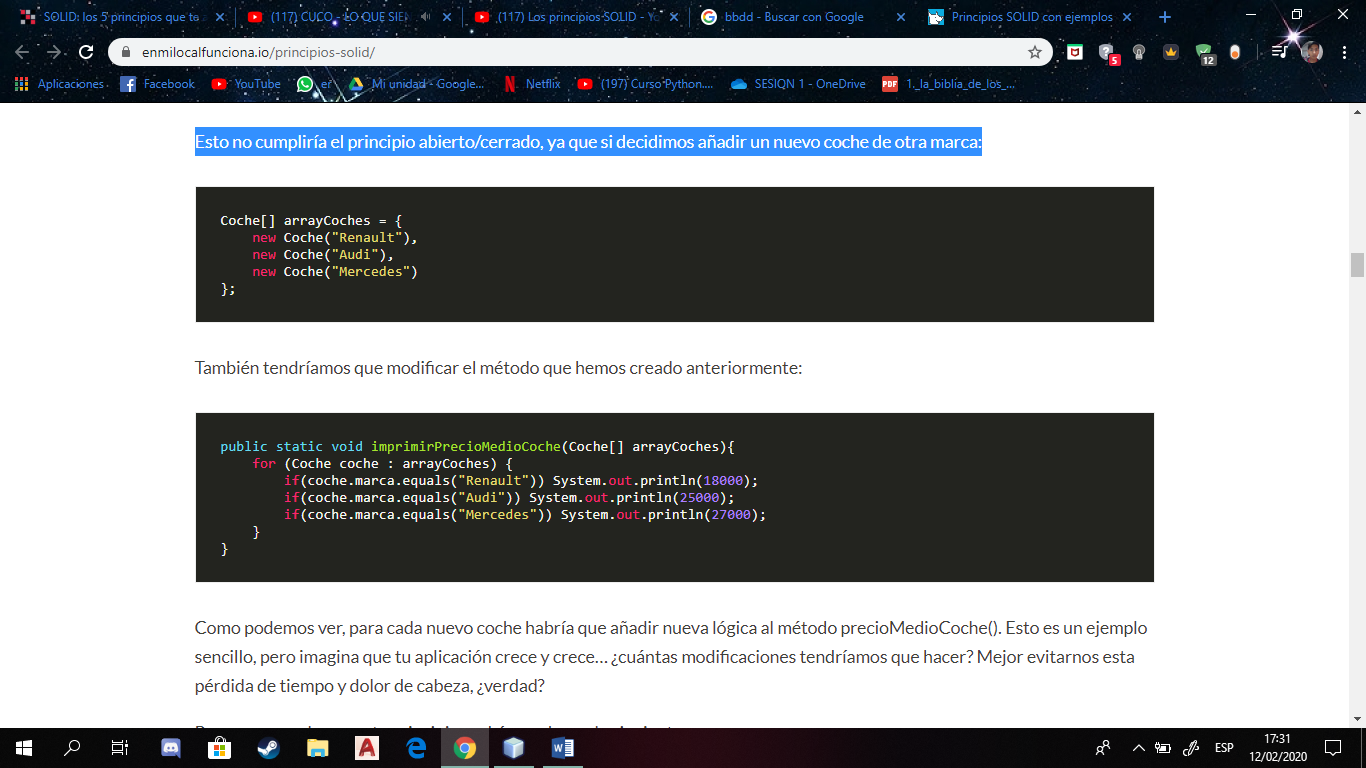
Si seguimos con la clase Coche:



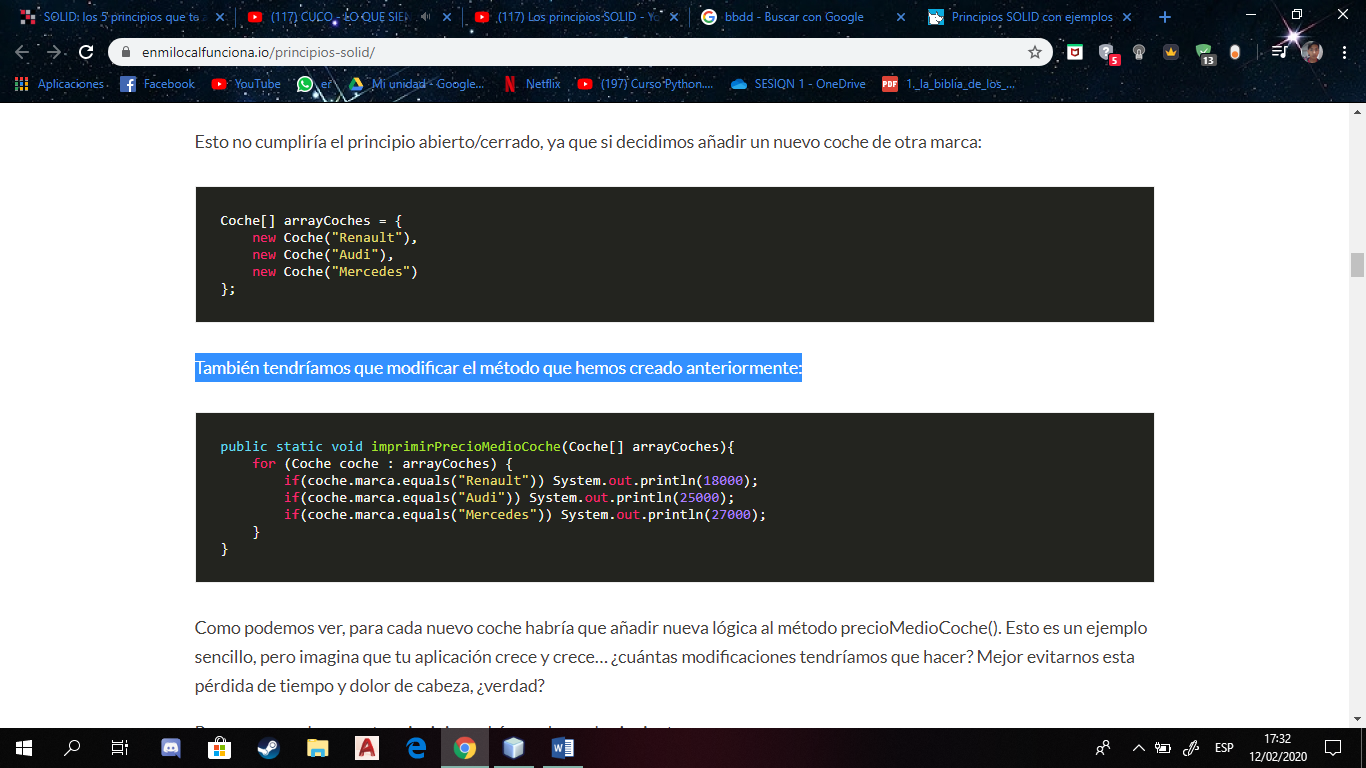
Si quisiéramos iterar a través de una lista de coches e imprimir sus marcas por pantalla:



Esto no cumpliría el principio abierto/cerrado, ya que si decidimos añadir un nuevo coche de otra marca:

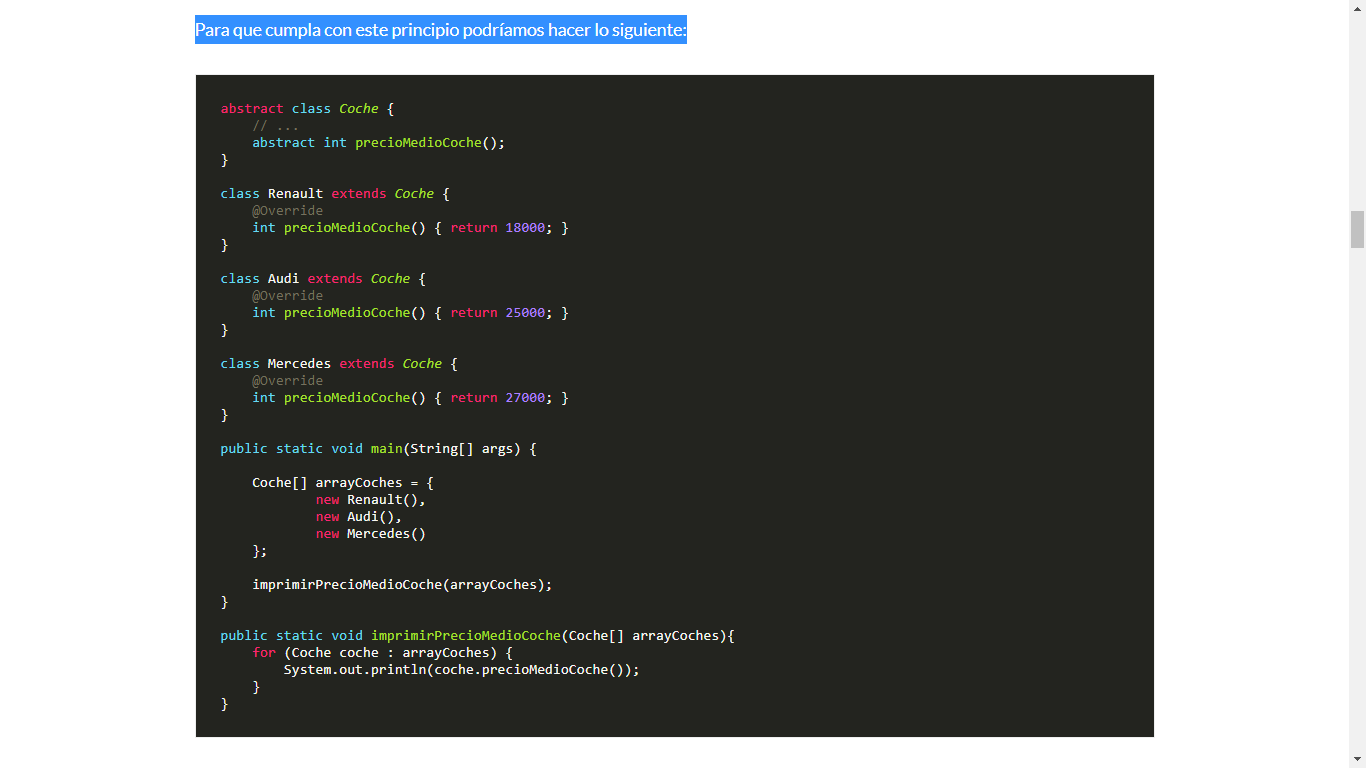


También tendríamos que modificar el método que hemos creado anteriormente:



Como podemos ver, para cada nuevo coche habría que añadir nueva lógica al método precioMedioCoche(). Esto es un ejemplo sencillo, pero imagina que tu aplicación crece y crece… ¿cuántas modificaciones tendríamos que hacer? Mejor evitarnos esta pérdida de tiempo y dolor de cabeza, ¿verdad?

Para que cumpla con este principio podríamos hacer lo siguiente:



Cada coche extiende la clase abstracta Coche e implementa el método abstracto precioMedioCoche().

Así, cada coche tiene su propia implementación del método precioMedioCoche(), por lo que el método imprimirPrecioMedioCoche() itera el array de coches y solo llama al método precioMedioCoche().

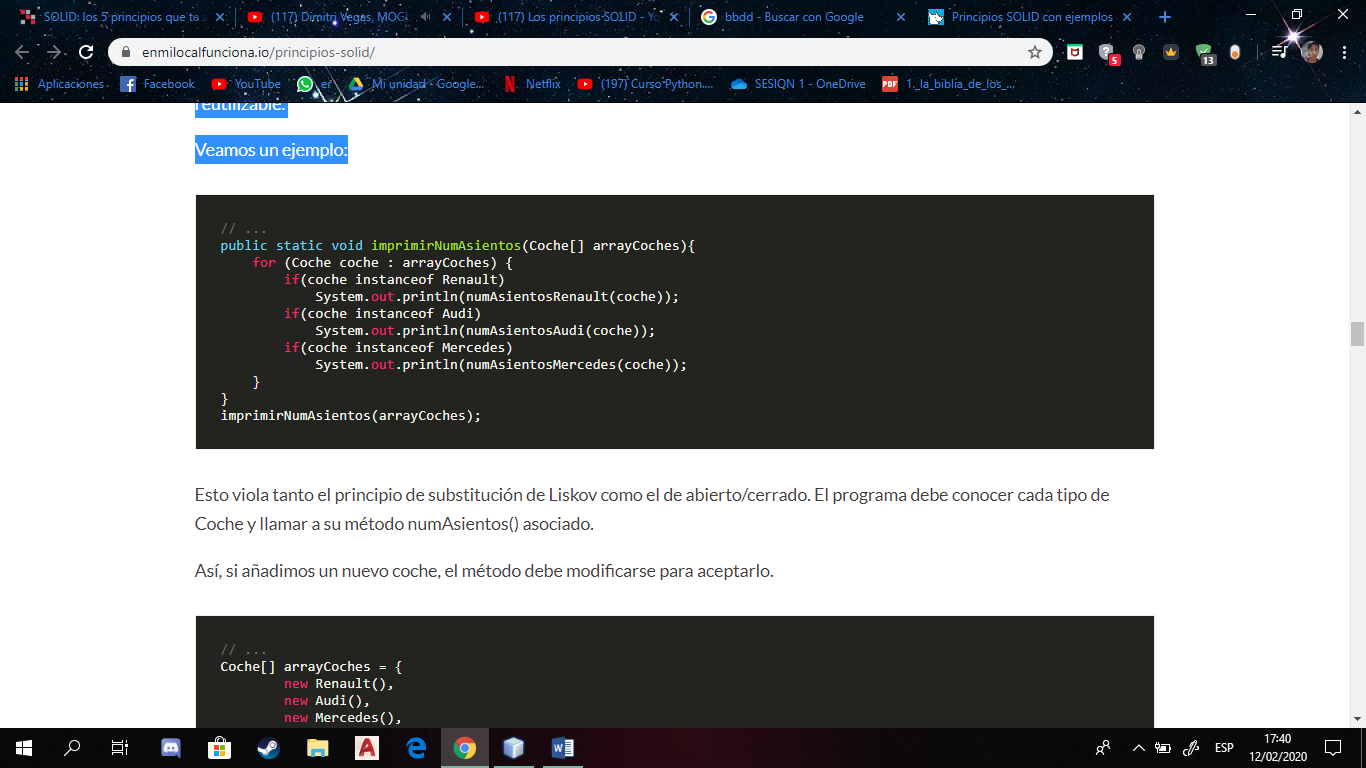
Ahora, si añadimos un nuevo coche, precioMedioCoche() no tendrá que ser modificado. Solo tendremos que añadir el nuevo coche al array, cumpliendo así el principio abierto/cerrado.

* 1. Aplicando el Principio de Sustitución de Liskov

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio.

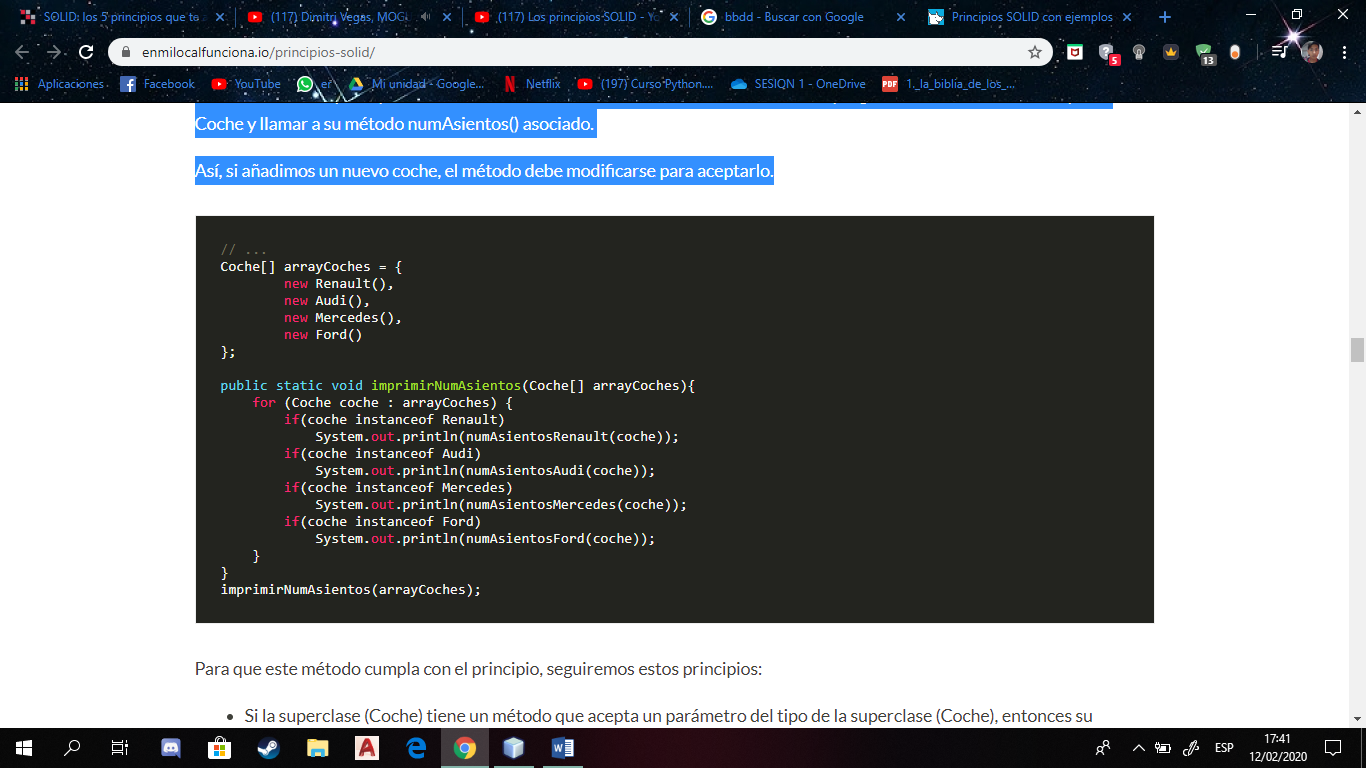
Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

Veamos un ejemplo:



Esto viola tanto el principio de substitución de Liskov como el de abierto/cerrado. El programa debe conocer cada tipo de Coche y llamar a su método numAsientos() asociado.

Así, si añadimos un nuevo coche, el método debe modificarse para aceptarlo.



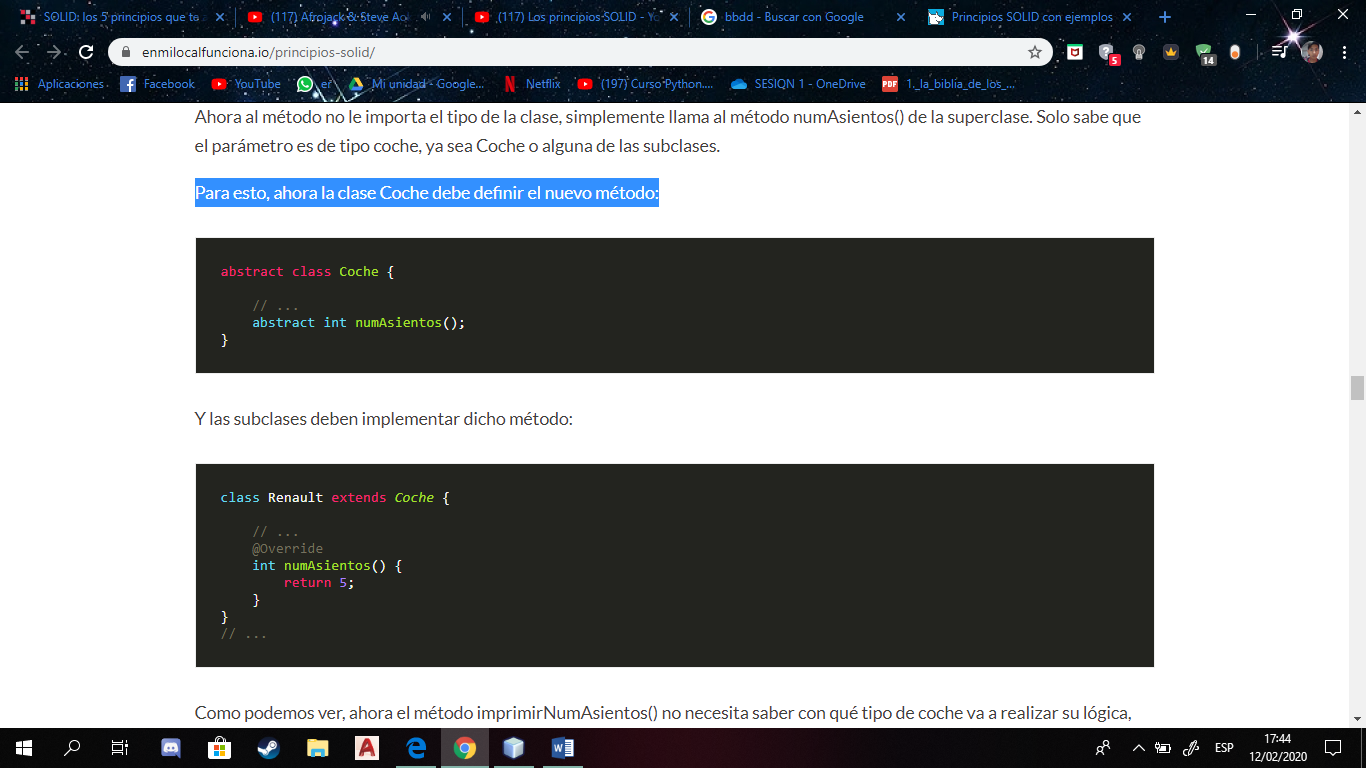
Para que este método cumpla con el principio, seguiremos estos principios:

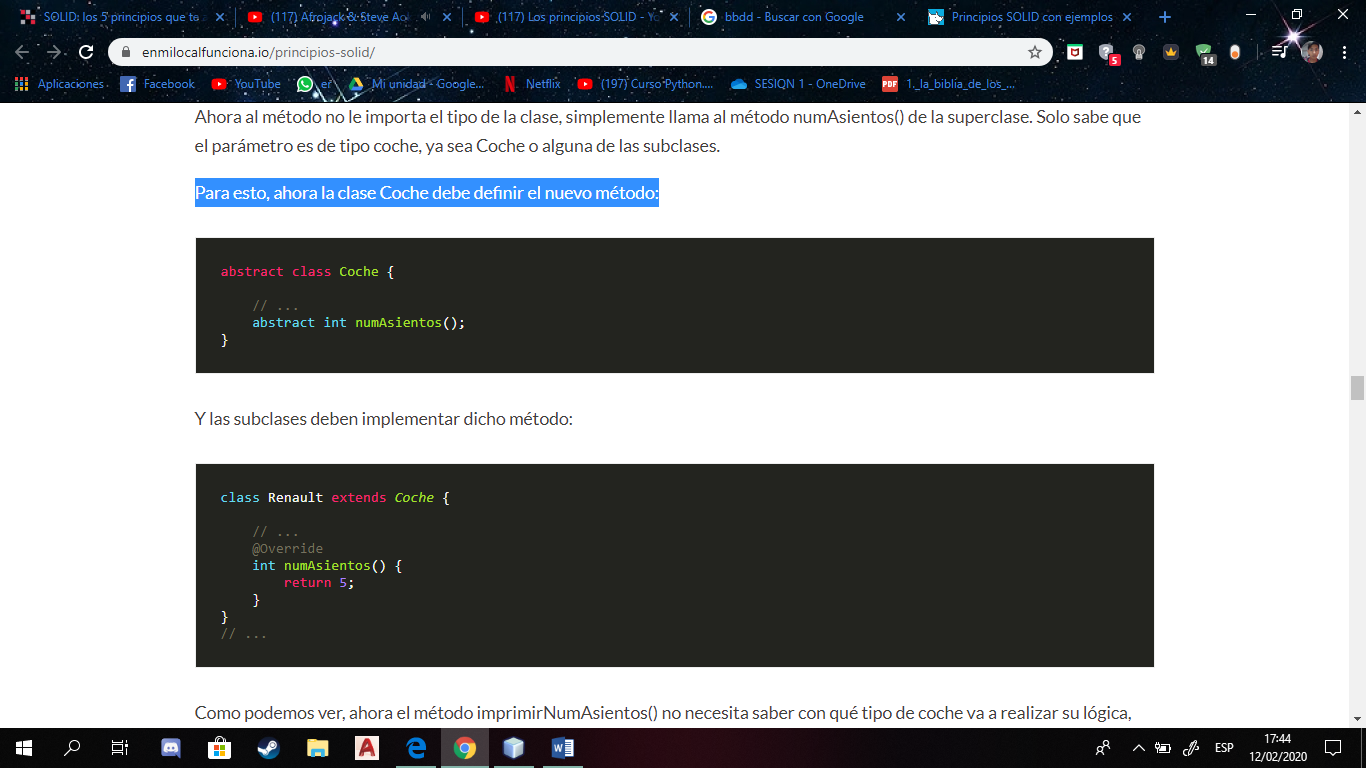
* Si la superclase (Coche) tiene un método que acepta un parámetro del tipo de la superclase (Coche), entonces su subclase (Renault) debería aceptar como argumento un tipo de la superclase (Coche) o un tipo de la subclase (Renault).
* Si la superclase devuelve un tipo de ella misma (Coche), entonces su subclase (Renault) debería devolver un tipo de la superclase (Coche) o un tipo de la subclase (Renault).

Si volvemos a implementar el método anterior:



Ahora al método no le importa el tipo de la clase, simplemente llama al método numAsientos() de la superclase. Solo sabe que el parámetro es de tipo coche, ya sea Coche o alguna de las subclases.

Para esto, ahora la clase Coche debe definir el nuevo método:

Y las subclases deben implementar dicho método:

Como se puede observar , ahora el método imprimirNumAsientos() no necesita saber con qué tipo de coche va a realizar su lógica, simplemente llama al método numAsientos() del tipo Coche, ya que por contrato, una subclase de Coche debe implementar dicho método.

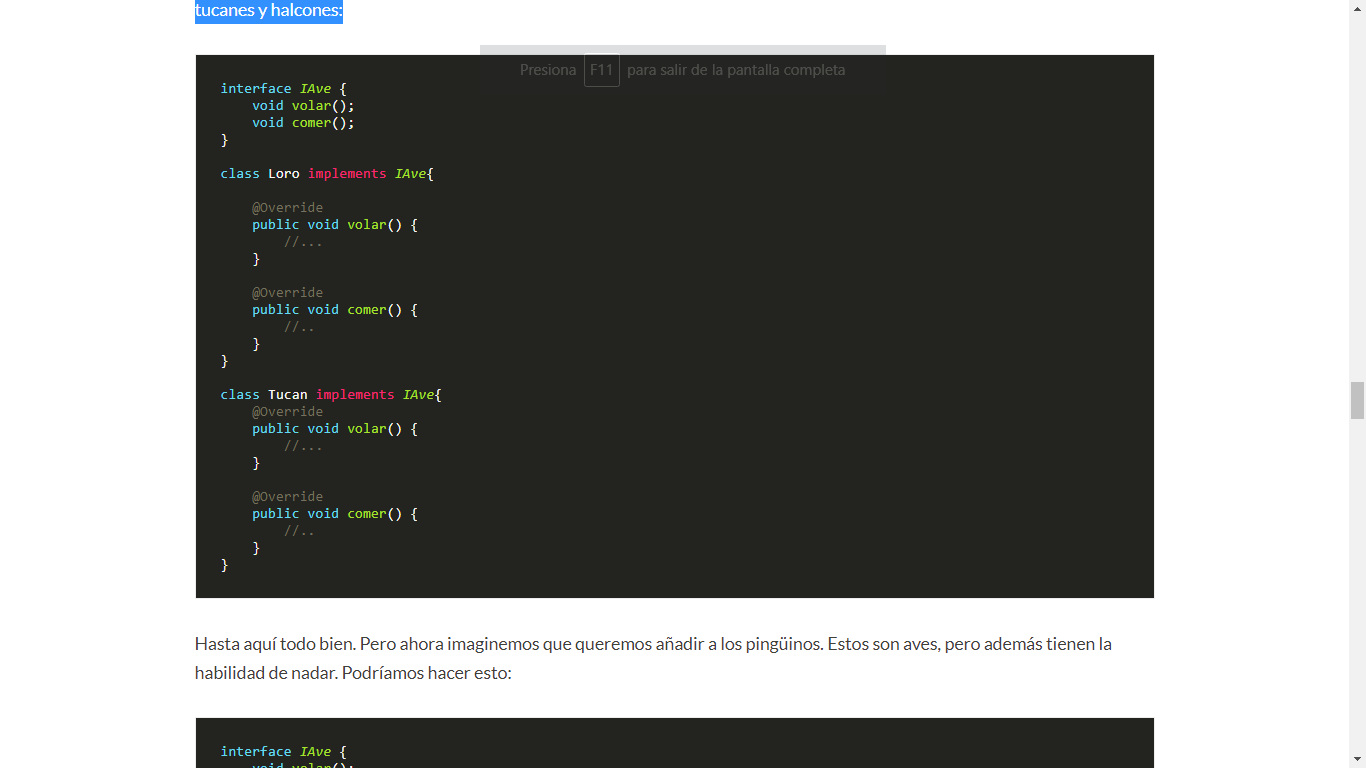
* 1. Aplicando el Principio de Segregación de Interfaz

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.

Imaginemos que queremos definir las clases necesarias para albergar algunos tipos de aves.

Por ejemplo, tendríamos loros, tucanes y halcones:

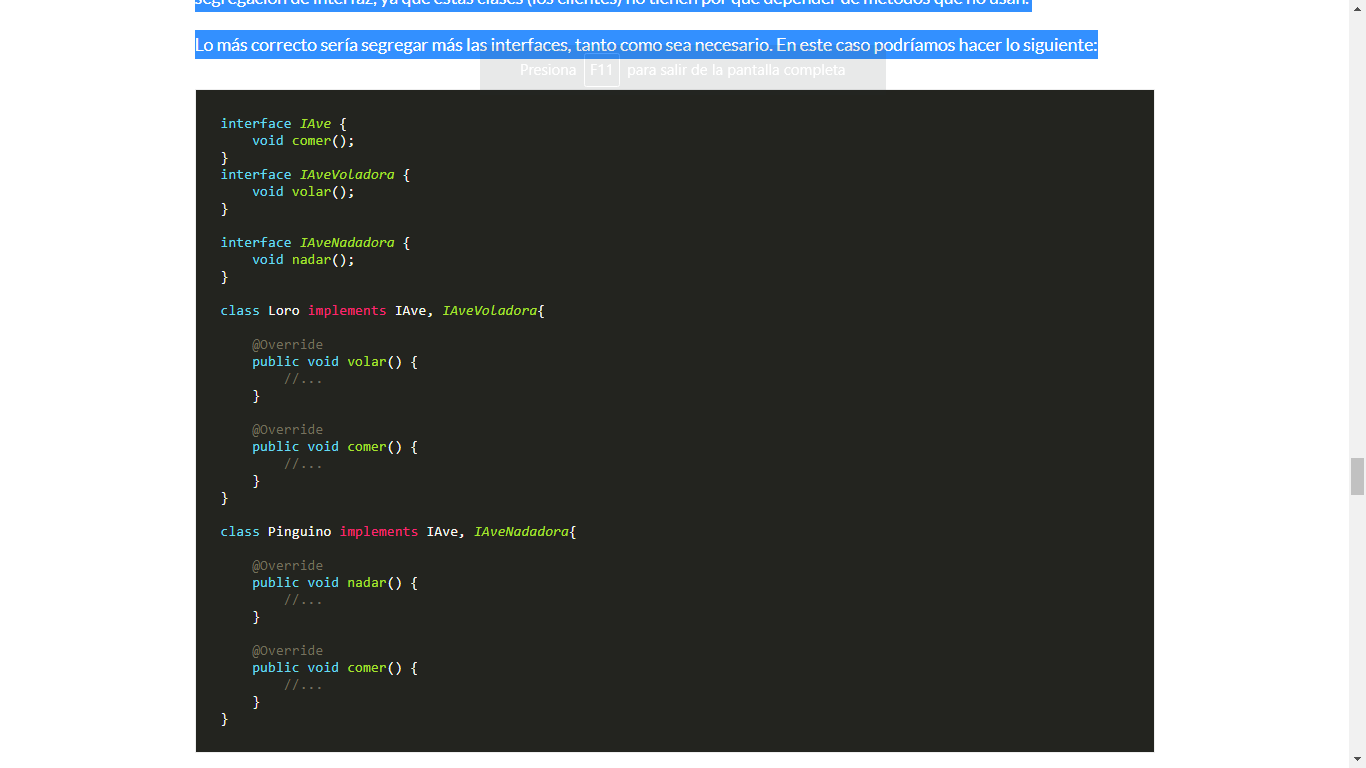


Hasta aquí todo bien. Pero ahora imaginemos que queremos añadir a los pingüinos. Estos son aves, pero además tienen la habilidad de nadar. Podríamos hacer esto:



El problema es que el loro no nada, y el pingüino no vuela, por lo que tendríamos que añadir una excepción o aviso si se intenta llamar a estos métodos. Además, si quisiéramos añadir otro método a la interfaz IAve, tendríamos que recorrer cada una de las clases que la implementa e ir añadiendo la implementación de dicho método en todas ellas. Esto viola el principio de segregación de interfaz, ya que estas clases (los clientes) no tienen por qué depender de métodos que no usan.

Lo más correcto sería segregar más las interfaces, tanto como sea necesario. En este caso podríamos hacer lo siguiente:



Así, cada clase implementa las interfaces de la que realmente necesita implementar sus métodos. A la hora de añadir nuevas funcionalidades, esto nos ahorrará bastante tiempo, y además, cumplimos con el primer principio (Responsabilidad Única).

* 1. Aplicando el Principio de inversión de dependencias

Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.

Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.

En algún momento nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase, es cuando debemos usar inyección de dependencias, lo que nos permitirá controlar las funcionalidades desde un sitio concreto en vez de tenerlas esparcidas por todo el programa. Además, este aislamiento nos permitirá realizar testing mucho más fácilmente.

Supongamos que tenemos una clase para realizar el acceso a datos, y lo hacemos a través de una BBDD:



Imaginemos que en el futuro queremos cambiar el servicio de BBDD por un servicio que conecta con una API. Para un minuto a pensar qué habría que hacer... ¿Ves el problema? Tendríamos que ir modificando todas las instancias de la clase AccesoADatos, una por una.

Esto es debido a que nuestro módulo de alto nivel (AccesoADatos) depende de un módulo de más bajo nivel (DatabaseService), violando así el principio de inversión de dependencias. El módulo de alto nivel debería depender de abstracciones.Para arreglar esto, podemos hacer que el módulo AccesoADatos dependa de una abstracción más genérica:



Así, sin importar el tipo de conexión que se le pase al módulo AccesoADatos, ni este ni sus instancias tendrán que cambiar, por lo que nos ahorraremos mucho trabajo.

Ahora, cada servicio que queramos pasar a AccesoADatos deberá implementar la interfaz Conexion:



Así, tanto el módulo de alto nivel como el de bajo nivel dependen de abstracciones, por lo que cumplimos el principio de inversión de dependencias. Además, esto nos forzará a cumplir el principio de Liskov, ya que los tipos derivados de Conexion (DatabaseService y APIService) son sustituibles por su abstracción (interfaz Conexion).

Conclusión

Aplicar estos cinco principios puede parecer algo tedioso, pero a la larga, mediante la práctica y echándoles un vistazo de vez en cuando, se volverán parte de nuestra forma de programar.

El programa será más sencillo de mantener, pero no solo para nosotros, si no más aún para los desarrolladores que vengan después, ya que verán un programa con una estructura bien definida y clara.

Como indica el propio Robert C. Martin en su artículo “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.”

Recomendaciones

Es ampliamente recomendable aplicar estos principios como buena practica, son una gran ayuda a la hora de guiar nuestro trabajo de desarrollo, sin embargo no debemos dejar atrás nuestra capacidad analítica como programadores y arquitectos, ya que estos principios no son estrictamente infalibles ni obligatorios en la práctica. Solo nos amplían las posibilidades de crear software cuyo código sea más entendible, extensible y cómodo de trabajr en el futuro, ahorrándonos tiempo a la hora de realizar cambios y haciéndonos más productivos.

Glosario de Nomenclaturas.

* 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.

* Refactorizar

La refactorización (del inglés refactoring) es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

* Desarrollo guiado de pruebas o TDD

Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una práctica de ingeniería de software que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring).

* Librería

En [informática](https://es.wikipedia.org/wiki/Inform%C3%A1tica), una librería o biblioteca (del [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s) library) es un conjunto de implementaciones funcionales, codificadas en un [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n), que ofrece una interfaz bien definida para la funcionalidad que se invoca.

* Framework

Un framework, entorno de trabajo1​ o marco de trabajo2​ es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

* Abstracción

La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n), el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?". El común denominador en la evolución de los [lenguajes de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n), desde los clásicos o [imperativos](https://es.wikipedia.org/w/index.php?title=Programaci%C3%B3n_imperativa&action=edit&redlink=1) hasta los [orientados a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos), ha sido el nivel de abstracción del que cada uno de ellos hace uso.

* Microservicios

Los microservicios representan un estilo de arquitectura y un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus componentes más pequeños, y son independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los microservicios son independientes y funcionan en conjunto para llevar a cabo las mismas tareas.

* BBDD (Base de datos)

Una base de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.

* Iteraración (Iterar)

Iteración significa repetir varias veces un proceso con la intención de alcanzar una meta deseada, objetivo o resultado. Cada repetición del proceso también se le denomina una "iteración", y los resultados de una iteración se utilizan como punto de partida para la siguiente iteración.

* Clase Abstracta

Una clase que declara la existencia de métodos pero no la implementación de dichos métodos (o sea, las llaves { } y las sentencias entre ellas), se considera una clase abstracta.  
  
Una clase abstracta puede contener métodos no-abstractos pero al menos uno de los métodos debe ser declarado abstracto. Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

* Método

En la [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n), un método es una [subrutina](https://es.wikipedia.org/wiki/Subrutina) cuyo código es definido en una [clase](https://es.wikipedia.org/wiki/Clases_(programaci%C3%B3n_orientada_a_objetos)) y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un [objeto](https://es.wikipedia.org/wiki/Objetos_(programaci%C3%B3n_orientada_a_objetos)), como es el caso de los métodos de instancia. Análogamente a los procedimientos en [lenguajes imperativos](https://es.wikipedia.org/wiki/Lenguajes_imperativos), un método consiste generalmente de una serie de sentencias para llevar a cabo una acción, un juego de parámetros de entrada que regularán dicha acción o, posiblemente, un valor de salida (o valor de retorno) de algún tipo.

* Interfaces (Interfaz)

**Interfaz** es lo que conocemos en inglés como *interface* (“superficie de contacto”).

En informática, se utiliza para nombrar a la conexión funcional entre dos sistemas, programas, dispositivos o componentes de cualquier tipo, que proporciona una comunicación de distintos niveles permitiendo el intercambio de información. Su plural es interfaces.