

ESCUELA ACADEMICA PROFESIONAL DE INGENIERIA DE SISTEMAS E INFORMATICA

CICLO VII

PRINCIPIOS S.O.L.I.D

CURSO: **INGENIERIA DE SOFTWARE III**

DOCENTE: **ERIC GUSTAVO CORONEL CASTILLO**

INTEGRANTES:

* HUALLPA JUAREZ CARLOS
* LLAVE HERRERA JANFRANCO
* MANCO ARIAS AYLIN
* MENDOZA DIAZ AXEL

**LOS OLIVOS –2016**

Contenido

[**RESUMEN** 3](#_Toc463450483)

[**ANTECEDENTES** 4](#_Toc463450484)

[**DESARROLLO DEL TEMA** 5](#_Toc463450485)

[**CAPITULO 1** 5](#_Toc463450486)

[**PRINCIPIO DE RESPONSABILIDAD UNICA** 5](#_Toc463450487)

[**CAPITULO 2** 6](#_Toc463450488)

[**PRINCIPIO DE ABIERTO/CERRADO** 6](#_Toc463450489)

[**CAPITULO 3** 6](#_Toc463450490)

[**PRINCIPIO DE SUSTITUCION DE LISKOV** 7](#_Toc463450491)

[**CAPITULO 4** 9](#_Toc463450493)

[**PRINCIPIO DE SEGReGACIÓN DE INTERFACES** 9](#_Toc463450494)

[**CAPITULO 5** 11](#_Toc463450495)

[**PRINCIPIO DE INVERSION DE DEPENDENCIAS** 11](#_Toc463450496)

[**CASOS** 12](#_Toc463450497)

[**1 .PRINCIPIO DE RESPONSABILIDAD UNICA** 12](#_Toc463450498)

[**2.- EL PRINCIPIO DE ABIERTO/CERRADO DE MEYER** 15](#_Toc463450499)

[**3. PRINCIPIO DE SUSTITUCION DE LISKOV** 17](#_Toc463450500)

[**4 .PRINCIPIO DE SEGREGACION DE INTERFACES** 20](#_Toc463450501)

[**5. PRINCIPIO DE INVERSION DE DEPENDENCIAS** 24](#_Toc463450502)

[**CONCLUSIONES** 27](#_Toc463450503)

[**RECOMENDACIONES** 28](#_Toc463450504)

[**BILBIOGRAFIAS** 28](#_Toc463450505)

# **RESUMEN**

SOLID (Single responsibility, Open - closed, Liskov substitution, Interface segregation and Dependency inversion). Es un acrónimo inventado por Robert C.Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los costes de mantenimiento pueden abarcar el 80% de un proyecto de software por lo que hay que valorar un buen diseño.

# **ANTECEDENTES**

Quién lleva años programando sabe que en el 99,99% de los casos los requisitos iniciales se ven alterados una vez ha comenzado el desarrollo e incluso cuando ha terminado. Es más, seguramente la aplicación desarrollada evolucione constantemente.

Esto es una realidad que tienen que asumir los programadores y tienen que tenerlo muy en cuenta en sus desarrollos. Si un jefe de proyecto asume el tiempo malgastado por no recoger bien los requisitos o si un cliente asume ese error nosotros no tenemos por qué preparar ‘la tercera guerra mundial’ contra nuestro responsable. Esta serenidad ante los cambios es algo que se va ganando con los años de experiencia y aflora en aquellas personas que comienzan a desarrollar porque tienen (he tenido) el concepto de la ‘aplicación perfecta’ donde todas las funcionalidades se encuentran en un documento inalterable.

Estos cambios hay que verlos como un nuevo reto, como una nueva pieza a encajar en el puzzle que estamos desarrollando, el problema es que en la mayoría de las ocasiones, la encajamos a ‘puñetazos’.

Para que estos cambios constantes cuesten lo menos posible, Robert C. Martín nos propuso en el año 2000 unos principios para que nuestro código orientado a objeto sea más flexible. Estos principios están recogidos sobre el acrónimo SOLID.

Seguramente se habrá escuchado muchas veces la expresión ‘aplicaciones sólidas”, pues bien, lo de sólidas se refiere a que en el desarrollo se aplican estos principios.

SOLID describe cinco principios fundamentales, uno por cada letra, sobre el diseño orientado a objetos y a continuación veremos que significan.

# **DESARROLLO DEL TEMA**

# **CAPITULO 1**

## **PRINCIPIO DE RESPONSABILIDAD UNICA**

El principio de responsabilidad única establece que cada módulo o clase debe tener responsabilidad sobre una sola parte de la funcionalidad proporcionada por el software y esta responsabilidad debe estar encapsulada en su totalidad por la clase. Todos sus servicios deben estar estrechamente alineados con esa responsabilidad. Robert C. Martin expresa el principio de la siguiente forma:

Una clase debe tener solo una razón para cambiar.

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 sólo 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.

# **CAPITULO 2**

## **PRINCIPIO DE ABIERTO/CERRADO**

Dentro de la programación orientada a objetos, el principio de abierto/cerrado establece que una entidad de software (clase, módulo, función, etc.) debe quedarse abierta para su extensión, pero cerrada para su modificación. Es decir, se debe poder extender el comportamiento de tal entidad, pero sin modificar su código fuente.

La denominación abierta/cerrado ha sido utilizada de dos maneras: ambas se basan en la herencia para resolver el aparente dilema, pero sus objetivos, técnicas y resultados son diferentes.

**PRINCIPIO DE ABIERTO/CERRADO DE MEYER**

Se considera que Bertrand Meyer es el instaurador del término principio de abierto/cerrado, el cual apareció originalmente en su obra Object-Oriented Software Construction, de 1988.

* Se dice que un módulo está abierto si se puede extender. Por ejemplo: Se deberían poder añadir campos a la estructura de datos que contiene dicho módulo o nuevas funcionalidades a su comportamiento.
* Se dice que un módulo queda cerrado si queda utilizable para otros módulos. Esto implica que dicho módulo goza de una descripción estable y bien definida (implicando a su interfaz pública, en el sentido de protección de la información).

En la época en que Meyer escribió esto, añadir campos o funciones a una librería implicaba, inevitablemente, tener que modificar todos los programas clientes de esa librería. La solución propuesta por Meyer descansaba en el concepto de herencia de la orientación a objetos.

# **CAPITULO 3**

## **PRINCIPIO DE SUSTITUCION DE LISKOV**

En ingeniería de software, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo mnemónico introducido por Robert C. Martin1 2a comienzos de la década del 2003 que representa cinco principios básicos de la programación orientada a objetos y el diseño en especial, con la alta cohesión y el bajo acoplamiento. 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. Debe 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.

“objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa”.

**Barbara Liskov**

Actualmente está trabajando en el departamento de [Ingeniería eléctrica](https://es.wikipedia.org/wiki/Ingenier%C3%ADa_el%C3%A9ctrica) y [Ciencias de la computación](https://es.wikipedia.org/wiki/Ciencias_de_la_computaci%C3%B3n) del [MIT](https://es.wikipedia.org/wiki/MIT), como profesora de ingeniería de [Ford](https://es.wikipedia.org/wiki/Ford). Consiguió su graduación en [Matemáticas](https://es.wikipedia.org/wiki/Matem%C3%A1ticas) en la [Universidad de California, Berkeley](https://es.wikipedia.org/wiki/Universidad_de_California,_Berkeley) en [1961](https://es.wikipedia.org/wiki/1961) y años más tarde, en [1968](https://es.wikipedia.org/wiki/1968), se convirtió en la primera mujer de los [Estados Unidos](https://es.wikipedia.org/wiki/Estados_Unidos) en conseguir un [Doctorado de Filosofía](https://es.wikipedia.org/wiki/PhD)  en Ciencias de la computación por la [Universidad de Stanford](https://es.wikipedia.org/wiki/Universidad_de_Stanford).

En 1970, se casó con Nathan Liskov, y su hijo, Moses Liskov, nació en 1975.

Barbara Liskov ha dirigido varios proyectos significativos, como el diseño e implementación del [lenguaje de programación CLU](https://es.wikipedia.org/wiki/CLU), el primer [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) que soportaba la abstracción de datos, [Argus](https://es.wikipedia.org/wiki/Argus_(lenguaje_de_programaci%C3%B3n)" \o "Argus (lenguaje de programación)), que fue el primer lenguaje de alto nivel en soportar la implementación de programas [distribuidos](https://es.wikipedia.org/wiki/Computaci%C3%B3n_distribuida) y [Thor](https://es.wikipedia.org/w/index.php?title=Thor_(base_de_datos)&action=edit&redlink=1), un sistema de [base de datos](https://es.wikipedia.org/wiki/Base_de_datos) [orientado a objetos](https://es.wikipedia.org/wiki/Orientado_a_objetos). Junto con [Jeannette Wing](https://es.wikipedia.org/w/index.php?title=Jeannette_Wing&action=edit&redlink=1), desarrolló una particular definición de [subtipo](https://es.wikipedia.org/w/index.php?title=Subtipo&action=edit&redlink=1), comúnmente conocido como el [Principio de sustitución de Liskov](https://es.wikipedia.org/wiki/Principio_de_sustituci%C3%B3n_de_Liskov).

La profesora Liskov es miembro de la Academia Nacional de Ingeniería (National Academy of Engineering) de los [Estados Unidos](https://es.wikipedia.org/wiki/Estados_Unidos). En [2004](https://es.wikipedia.org/wiki/2004) ganó la [Medalla John von Neumann](https://es.wikipedia.org/wiki/Medalla_John_von_Neumann) por "su fundamental contribución a los lenguajes de programación, metodologías de programación y sistemas distribuidos". En [2008](https://es.wikipedia.org/wiki/2008) ganó el [premio Turing](https://es.wikipedia.org/wiki/Premio_Turing) por "su contribución a los fundamentos teóricos y prácticos en el diseño de lenguajes de programación y sistemas, especialmente relacionados con la abstracción de datos, tolerancia a fallos y computación distribuida".

Además, Liskov es autora de 3 libros y cientos de informes técnicos.

**CLU**

Es un [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) creado por el instituto de tecnología de Massachusetts ([MIT](https://es.wikipedia.org/wiki/MIT)) por [Barbara Liskov](https://es.wikipedia.org/wiki/Barbara_Liskov" \o "Barbara Liskov) y sus estudiantes entre [1974](https://es.wikipedia.org/wiki/1974) y [1975](https://es.wikipedia.org/wiki/1975). Usaba constructores para los tipos de datos abstractos que se incluyeron en el código, un paso adelante en la [programación orientada a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos) (POO). No obstante, muchas otras de las características de POO estaban incompletas o necesitaban un mayor desarrollo. Por otra parte, el lenguaje se obstaculiza por una sintaxis que en ocasiones puede resultar engorrosa. CLU y Alphard parecen ser ambos lenguajes orientados a objetos completos, sin serlo en realidad.

**Definición**

El Principio de Sustitución de Liskov es un principio de la [programación orientada a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos). y puede definirse como: Cada [clase](https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)) 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 Principio de Sustitución de Liskov (LSP) es una definición particular de una relación de subtipificación, llamada tipificación (fuerte) del comportamiento, que fue introducido inicialmente por [Barbara Liskov](https://es.wikipedia.org/wiki/Barbara_Liskov" \o "Barbara Liskov) en una conferencia magistral en 1987 titulada La Abstracción de Datos y Jerarquía.[1](https://es.wikipedia.org/wiki/Principio_de_sustituci%C3%B3n_de_Liskov#cite_note-1) 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](https://es.wikipedia.org/w/index.php?title=Jeannette_Wing&action=edit&redlink=1) formularon el principio de manera conjunta en un artículo en el año 1994 de la siguiente manera:

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

En el mismo artículo, Liskov y Wing detallan que su concepto de comportamiento de subtipos es una extensión de la [lógica de Hoare](https://es.wikipedia.org/wiki/L%C3%B3gica_de_Hoare), que tiene una cierta semejanza con el diseño por contrato de [Bertrand Meyer](https://es.wikipedia.org/wiki/Bertrand_Meyer), ya que considera la interacción de subtipos con pre y post condiciones.

# **CAPITULO 4**

## **PRINCIPIO DE SEGReGACIÓN DE INTERFACES**

El Principio de Segregación de Interfaces fue utilizado por primera vez por Robert C. Martin durante unas sesiones de consultoría en Xerox. Por aquella época, Xerox estaba diseñando una impresora multifuncional. El software diseñado para la impresora funcionaba y se adaptaba perfectamente a las necesidades iniciales de la impresora; sin embargo, conforme fue evolucionando, y por lo tanto cambiando, se hizo cada vez más difícil de mantener. Cualquier modificación tenía un gran impacto global sobre el sistema. La utilización de ISP permitió reducir los riesgos de las modificaciones y otorgó una mayor facilidad al mantenimiento. El ISP declara que: Los clientes no deben ser forzosamente dependientes de las interfaces que no utilizan.

El principio de segregación de interfaces es el cuarto principio del acrónimo SOLID. Aunque algunos lo confunden con el principio de responsabilidad única tiene matices diferentes que se verán a continuación.

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por lo 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.

**El problema**

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan *fat interfaces*. Probablemente ocurrirá que las clases hijas acabarán por no usar muchos de esos métodos, y habrá que darles una implementación. Muy habitual es lanzar una excepción, o simplemente no hacer nada.

Pero, al igual que vimos en algún ejemplo en el principio de sustitución de Liskov, esto es peligroso. Si lanzamos una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar nuestro programa. El resto de implementaciones “por defecto” que podamos dar pueden generar efectos secundarios que no esperemos, y a los que sólo podemos responder conociendo el código fuente del módulo en cuestión, cosa que no nos interesa.

**¿Cómo detectar que estamos violando el Principio de Segregación de Interfaces?**

Como se comentaba en los párrafos anteriores, si al implementar una interfaz se ve que uno o varios de los métodos no tienen sentido y te hacen falta dejarlos vacíos o lanzar excepciones, es muy probable que estés violando este principio. Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos.

Hay que recordar que no pasa nada cuando una clase ahora necesite implementar varias interfaces. El punto importante es que use todos los métodos definidos por esas interfaces.

# **CAPITULO 5**

## **PRINCIPIO DE INVERSION DE DEPENDENCIAS**

Este principio es una técnica básica, y será el que más presente tengas en tu día a día si quieres hacer que tu código sea estable y mantenible. Gracias al principio de inversión de dependencias, podemos hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que utilices, la base de datos, cómo te conectes a tu servidor… Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

Es la base teórica sobre la que se basa la inyección de dependencias usada por muchos frameworks, como Spring.

La definición que se suele dar es:

**A.** Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.

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

Un bosque contiene árboles, que a su vez contienen hojas, que contienen células…

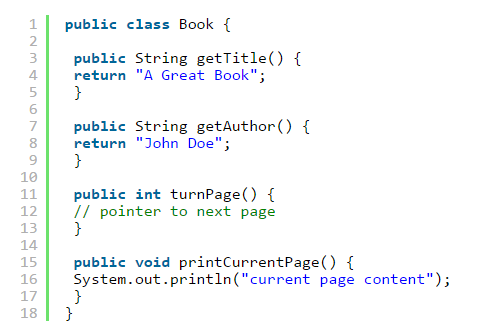
Por eso se eligió la palabra “inversión”, porque rompe con esta dinámica.

Lo que se pretende es que no exista la necesidad de que los módulos dependan unos de otros, sino que dependan de abstracciones. De esta forma, nuestros módulos pueden ser más fácilmente reutilizables.

# **CASOS**

## **1 .PRINCIPIO DE RESPONSABILIDAD UNICA**

Vamos a suponer que tenemos una entidad libro que además de proveer diferentes acciones relacionadas:

****

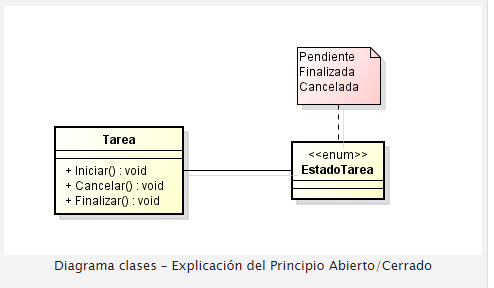
Como se puede apreciar claramente, esta clase tiene al menos dos responsabilidades, la relacionada con la gestión del  libro como son obtener el título, autor, siguiente página; como la relacionada con la representación. Aquí se están mezclando lógicas de negocio relacionadas con la gestión y la representación en pantalla. Esto se considera una violación del principio SRP. Una solución para solventar este problema sería por ejemplo la siguiente:

****

Aunque esto no deja de ser un ejemplo muy sencillo, se puede apreciar claramente como se ha separado la lógica de negocio relacionada con la representación gráfica de la lógica de negocio relacionada con la gestión del libro. Por tanto se cumple el principio SRP.

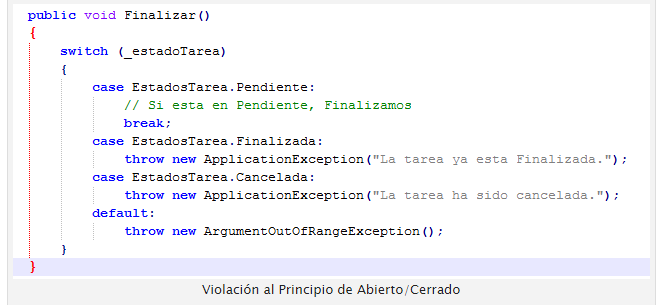
## **2.- EL PRINCIPIO DE ABIERTO/CERRADO DE MEYER**

Para comprender mejor este principio vamos a ejemplificarlo por medio de un diseño.

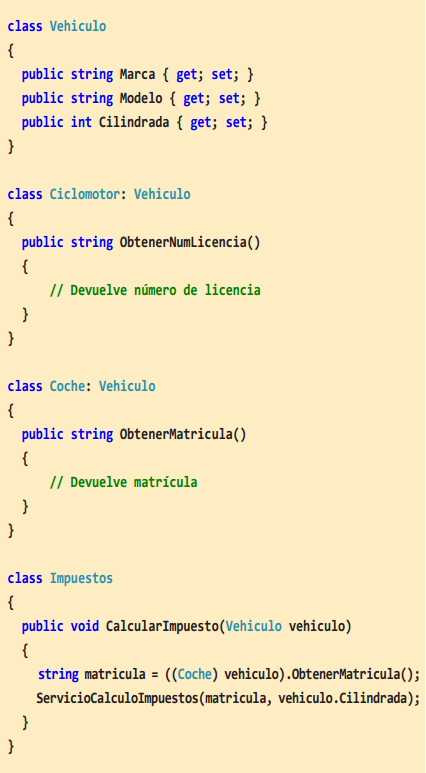
****

El diagrama anterior es un ejemplo de una clase que rompe con el principio abierto/cerrado (OCP). Vamos a suponer un sistema de gestión de proyectos, de tal manera nos vamos a centrar en la entidad Tarea y vamos a descubrir cómo rompe con el principio abierto/cerrado.

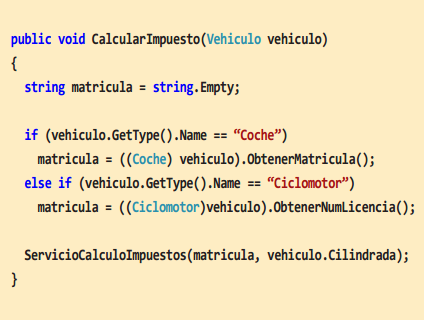
Como se puede observar en las relaciones dicha clase viene determinando por uno de los estados de la enumeración EstadoTarea (Pendiente, Finalizada, Cancelada), además se puede observar que la clase implementa 3 métodos, Iniciar, Cancelar y Finalizar que cambian si es posible el estado de la tarea. Veamos cómo sería una posible implementación del método Finalizar.

****

## **3. PRINCIPIO DE SUSTITUCION DE LISKOV**



Básicamente, LSP afirma que si tenemos dos objetos de tipos diferentes –**Coche** y **Ciclomotor**– que derivan de una misma clase base –**Vehículo**–, deberíamos poder reemplazar cada uno de los tipos –**Coche/Ciclomotor** y viceversa– allí dónde el tipo base –**Vehículo**– esté implementado. En el ejemplo anterior tenemos un claro caso de violación del LSP, ya que la ejecución del método **CalcularImpuesto** generará una excepción de conversión de tipo si el objeto pasado por parámetro es de tipo **Ciclomotor** en lugar de **Coche**, pese a que ambas clases derivan de la misma clase base **Vehículo**.



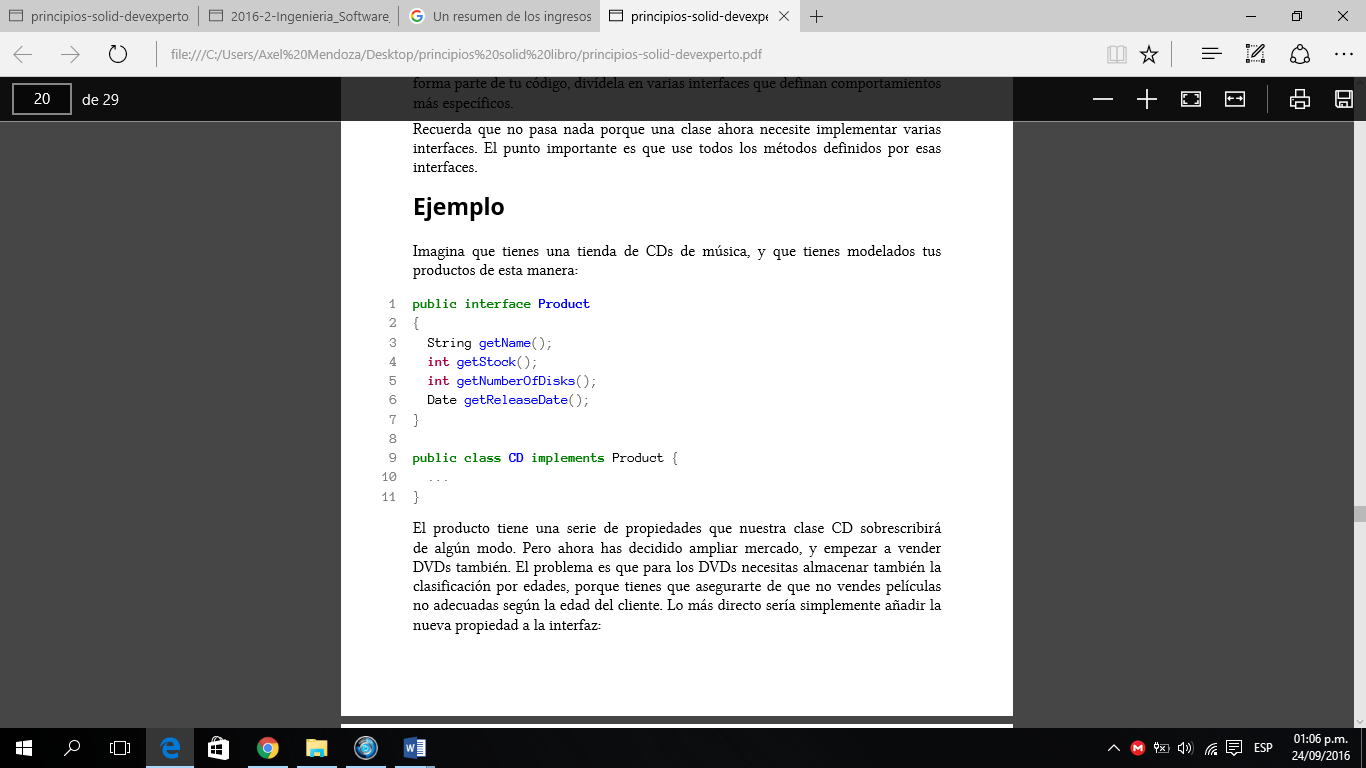
Podríamos pensar en solucionar el problema de la forma que la siguiente imagen. Pese a que el compilador no genere ninguna excepción de conversión de tipo, esta clase aún viola el LSP. Esto es debido a que estamos forzando a un objeto **Vehículo** pasado como parámetro a comportarse como **Ciclomotor** o **Coche**. Además, esta aproximación vulnera el Principio Open/Closed que es el principio anterior, ya que ante cualquier nueva entidad que derive de **Vehículo** deberemos modificar el método **CalcularImpuesto**.

El Principio de Substitución de Liskov no dicta tanto el qué sino el cómo debe heredarse tipos.

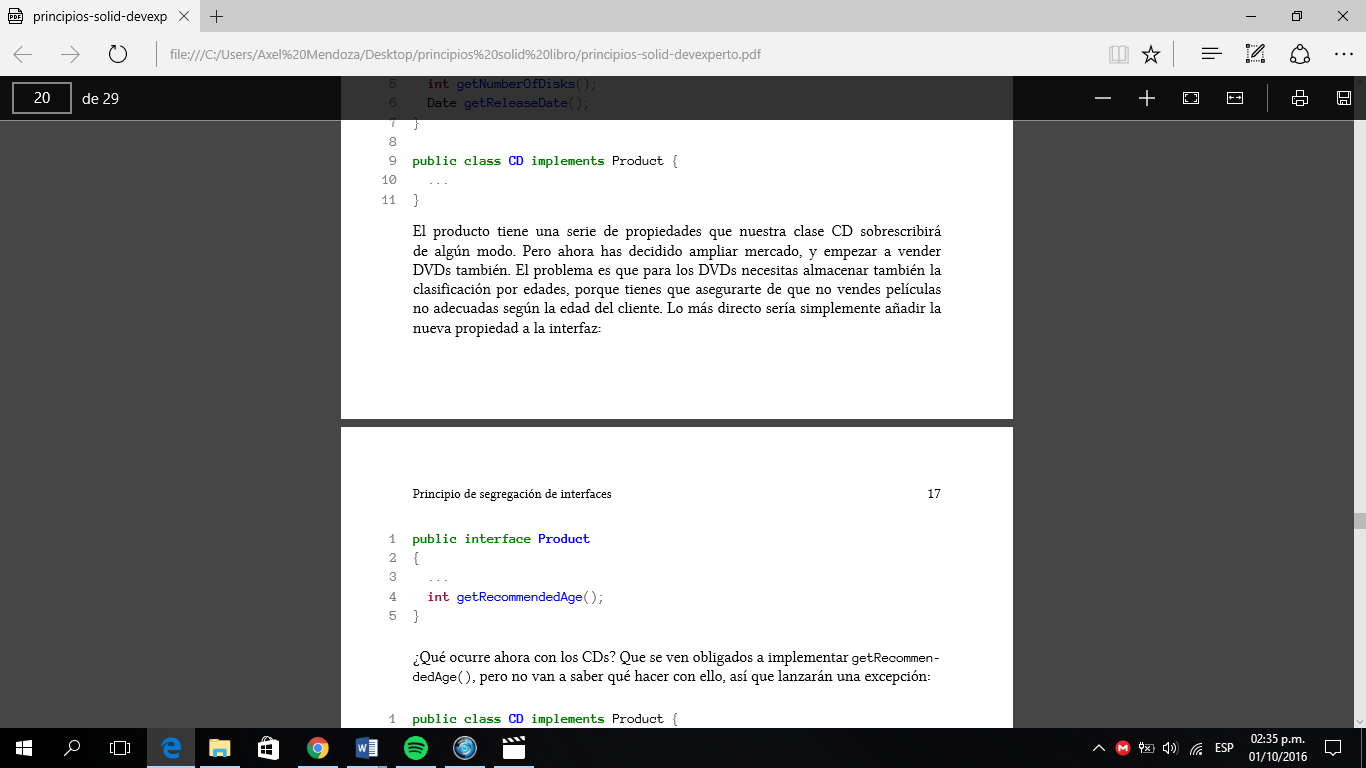
No olvidemos que una clase son datos más comportamiento, y que dicho comportamiento no debe ser sacrificado entre herencias. Por tanto, minimizaremos el impacto de una incorrecta implementación y por tanto de la modificación del comportamiento aplicando el Principio de Sustitución de Liskov.

## **4 .PRINCIPIO DE SEGREGACION DE INTERFACES**

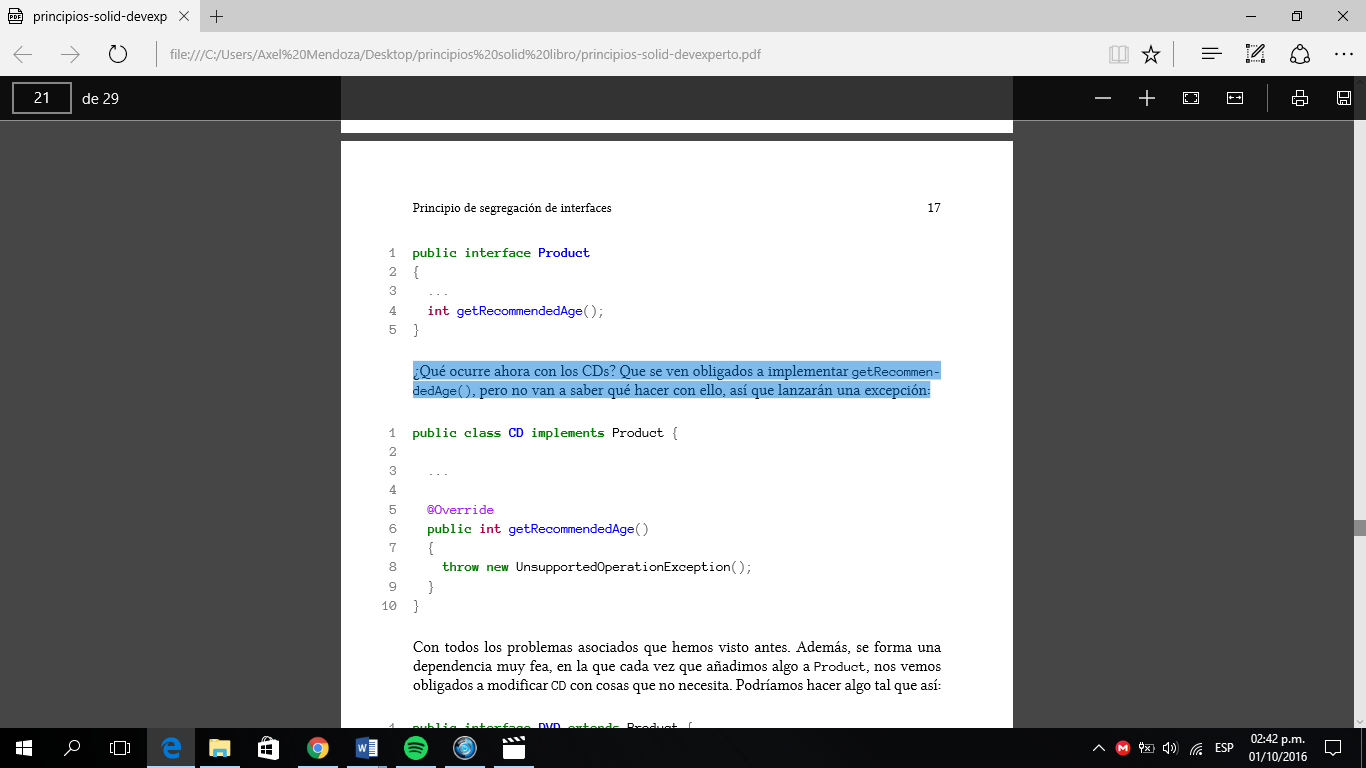
Imaginemos que tenemos una tienda de CDs de música, y que tenemos modelados los productos de esta manera:



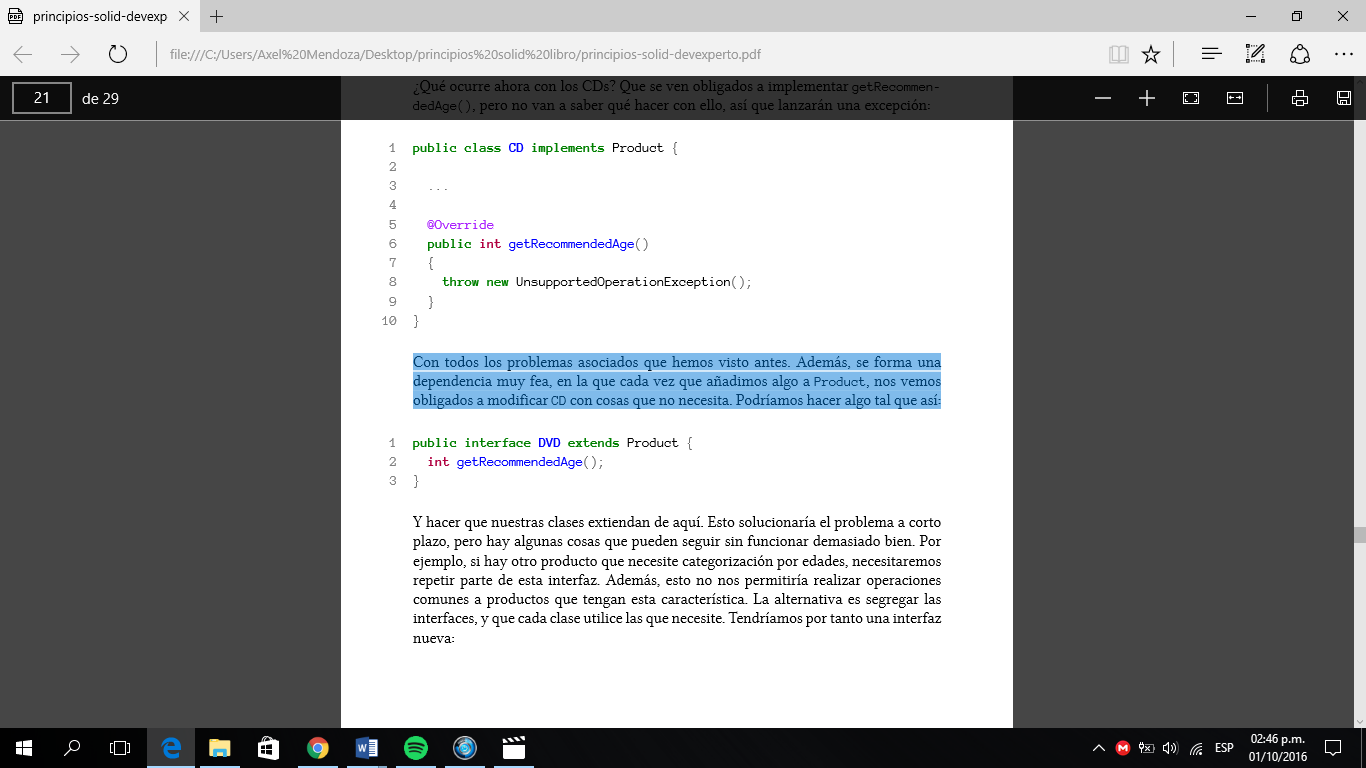
El producto tiene una serie de propiedades que la clase CD sobrescribirá de algún modo. Pero ahora hemos decidido ampliar mercado, y empezar a vender DVDs también. El problema es que para los DVDs necesitamos almacenar también la clasificación por edades, porque tenemos que asegurarnos de no vender películas no adecuadas según la edad del cliente. Lo más directo sería simplemente añadir la nueva propiedad a la interfaz:



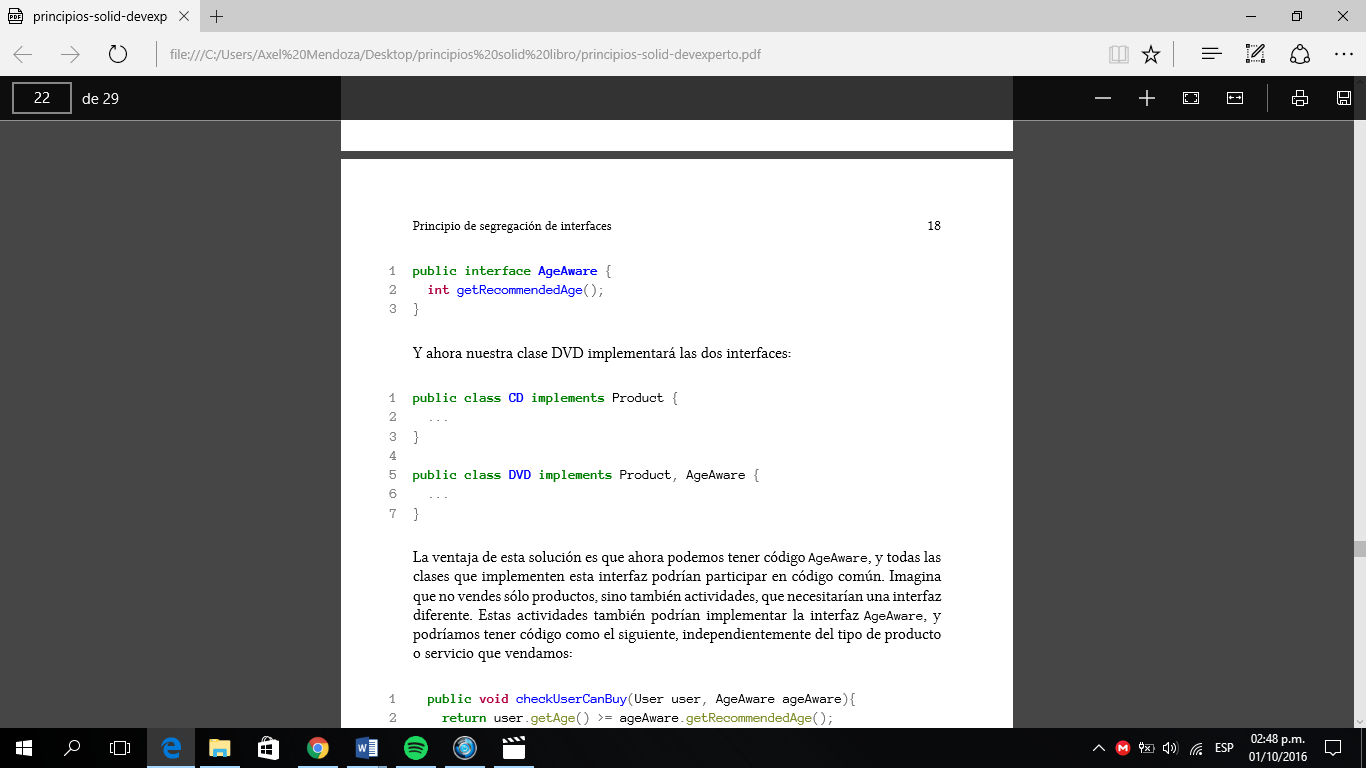
¿Qué ocurre ahora con los CDs? Que se ven obligados a implementar getRecommendedAge( ), pero no van a saber qué hacer con ello, así que lanzarán una excepción:



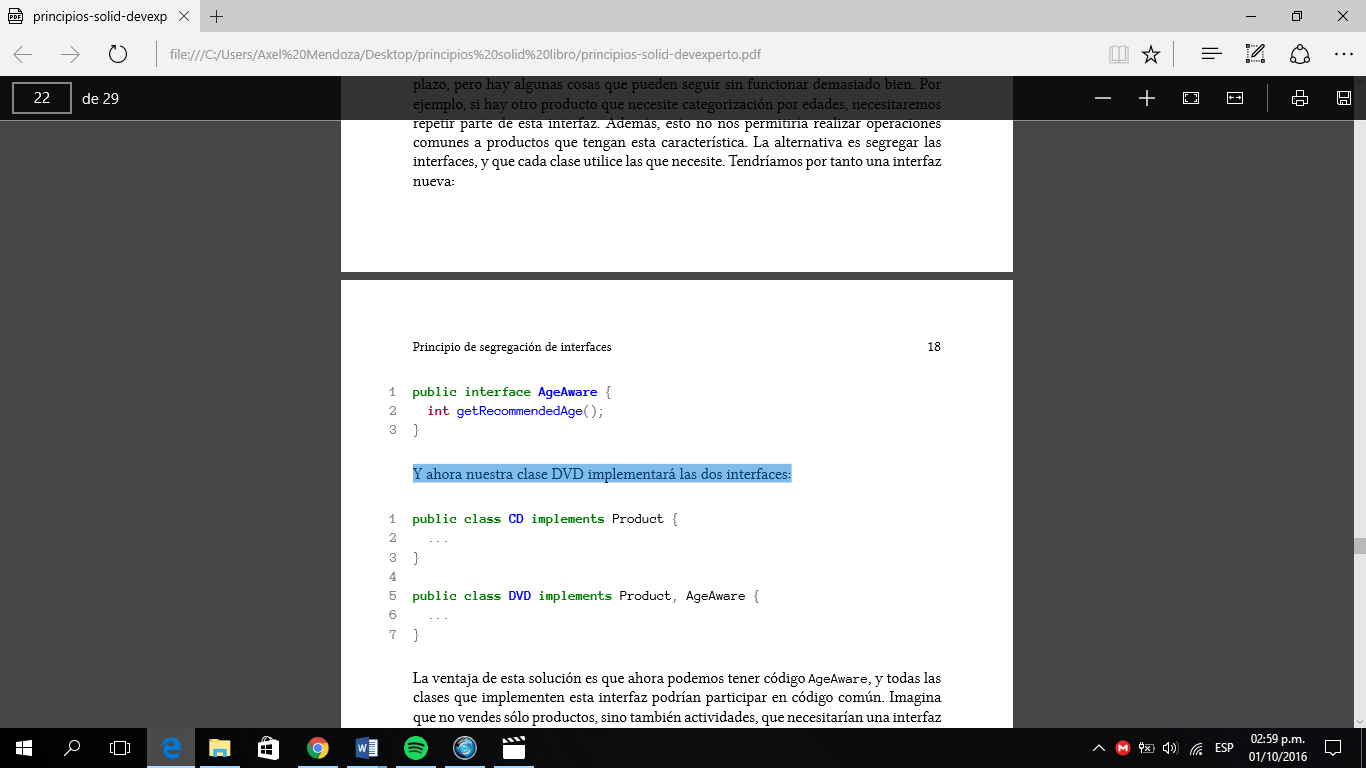
Con todos los problemas asociados que hemos visto antes. Además, se forma una dependencia muy fea, en la que cada vez que añadimos algo a Product, nos vemos obligados a modificar CD con cosas que no necesita. Podríamos hacer algo tal que así:



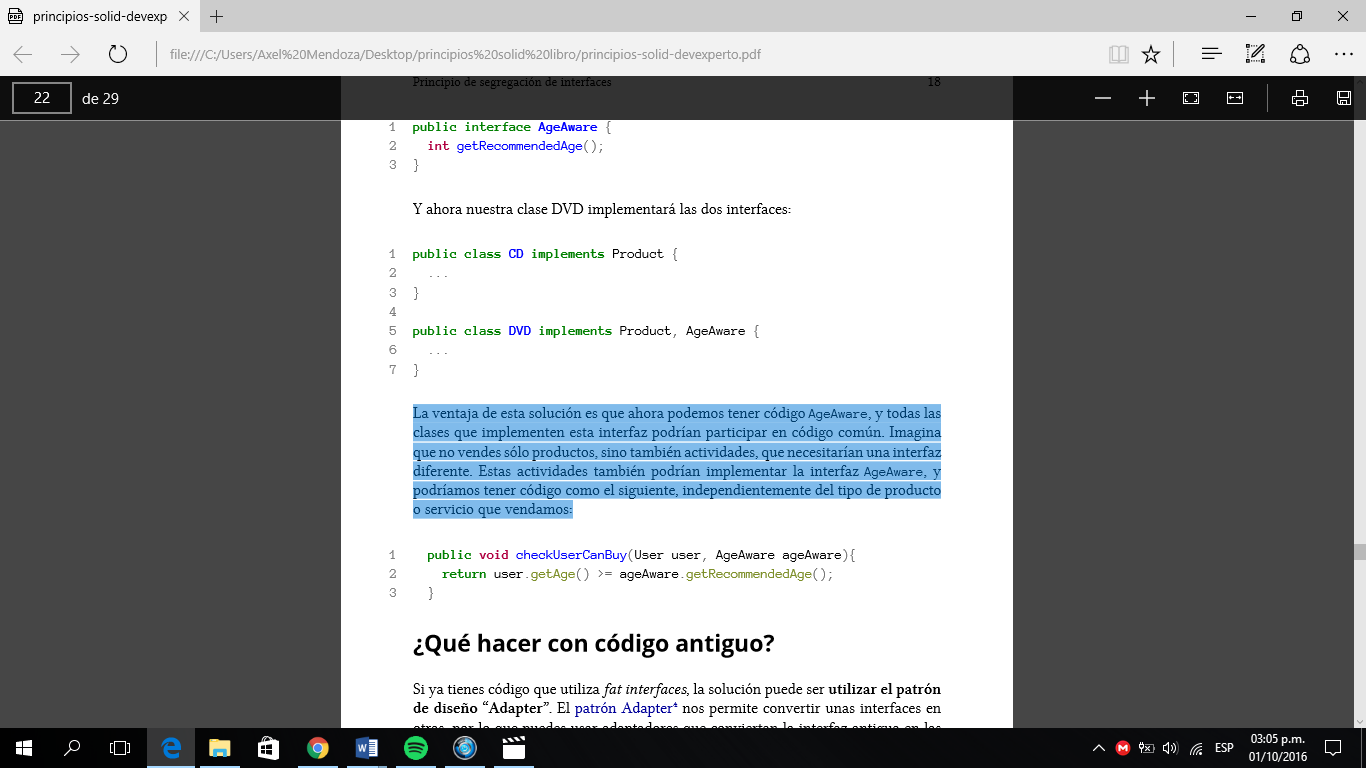
Y hacer que nuestras clases extiendan de aquí. Esto solucionaría el problema a corto plazo, pero hay algunas cosas que pueden seguir sin funcionar demasiado bien. Por ejemplo, si hay otro producto que necesite categorización por edades, necesitaremos repetir parte de esta interfaz. Además, esto no nos permitiría realizar operaciones comunes a productos que tengan esta característica. La alternativa es segregar las interfaces, y que cada clase utilice las que necesite. Tendríamos por tanto una interfaz nueva:



Y ahora nuestra clase DVD implementará las dos interfaces:



La ventaja de esta solución es que ahora podemos tener código AgeAware*,* y todas las clases que implementen esta interfaz podrían participar en código común. Imaginemos que no vendemos sólo productos, sino también actividades, que necesitarían una interfaz diferente. Estas actividades también podrían implementar la interfaz AgeAware, y podríamos tener código como el siguiente, independientemente del tipo de producto o servicio que vendamos:

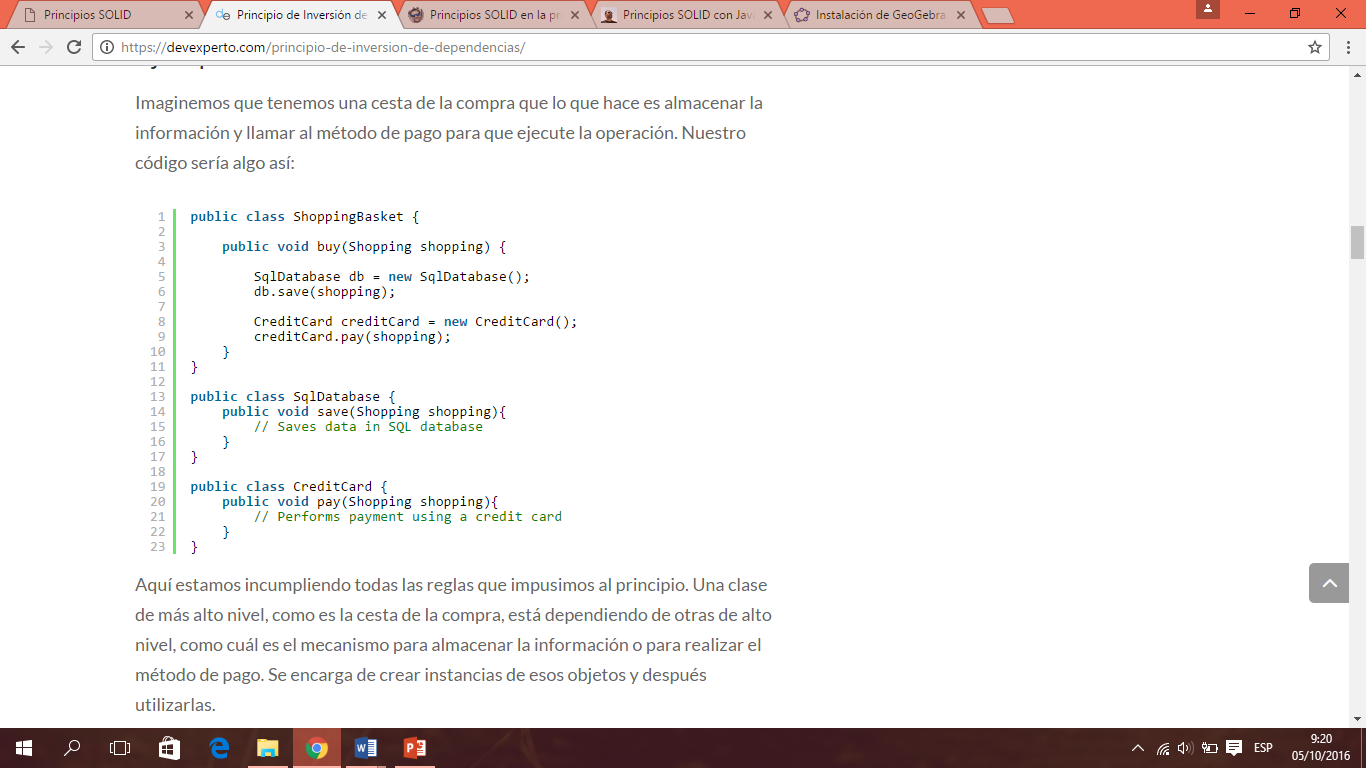


**¿Qué hacer con el código antiguo?**

Si ya tenemos código que utiliza *fat* interfaces, la solución puede ser utilizar el patrón de diseño “Adapter”. El patrón Adapter nos permitirá convertir unas interfaces en otras, por lo que podemos usar adaptadores que conviertan la interfaz antigua en las nuevas.

## **5. PRINCIPIO DE INVERSION DE DEPENDENCIAS**

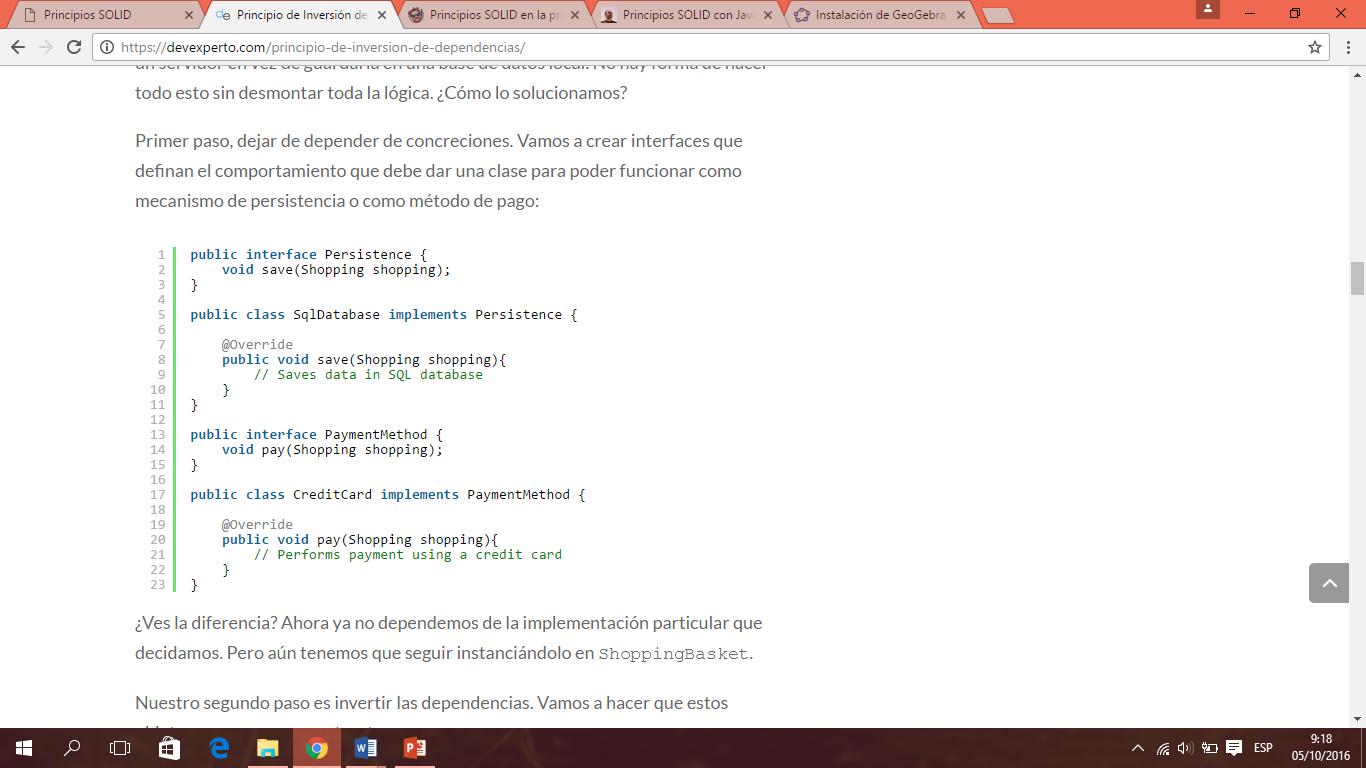
Tenemos una cesta de la compra que lo que hace es almacenar la información y llamar al método de pago para que ejecute la operación.



Aquí estamos incumpliendo todas las reglas que impusimos al principio. Una clase de más alto nivel, como es la cesta de la compra, está dependiendo de otras de alto nivel, como cuál es el mecanismo para almacenar la información o para realizar el método de pago. Se encarga de crear instancias de esos objetos y después utilizarlas.

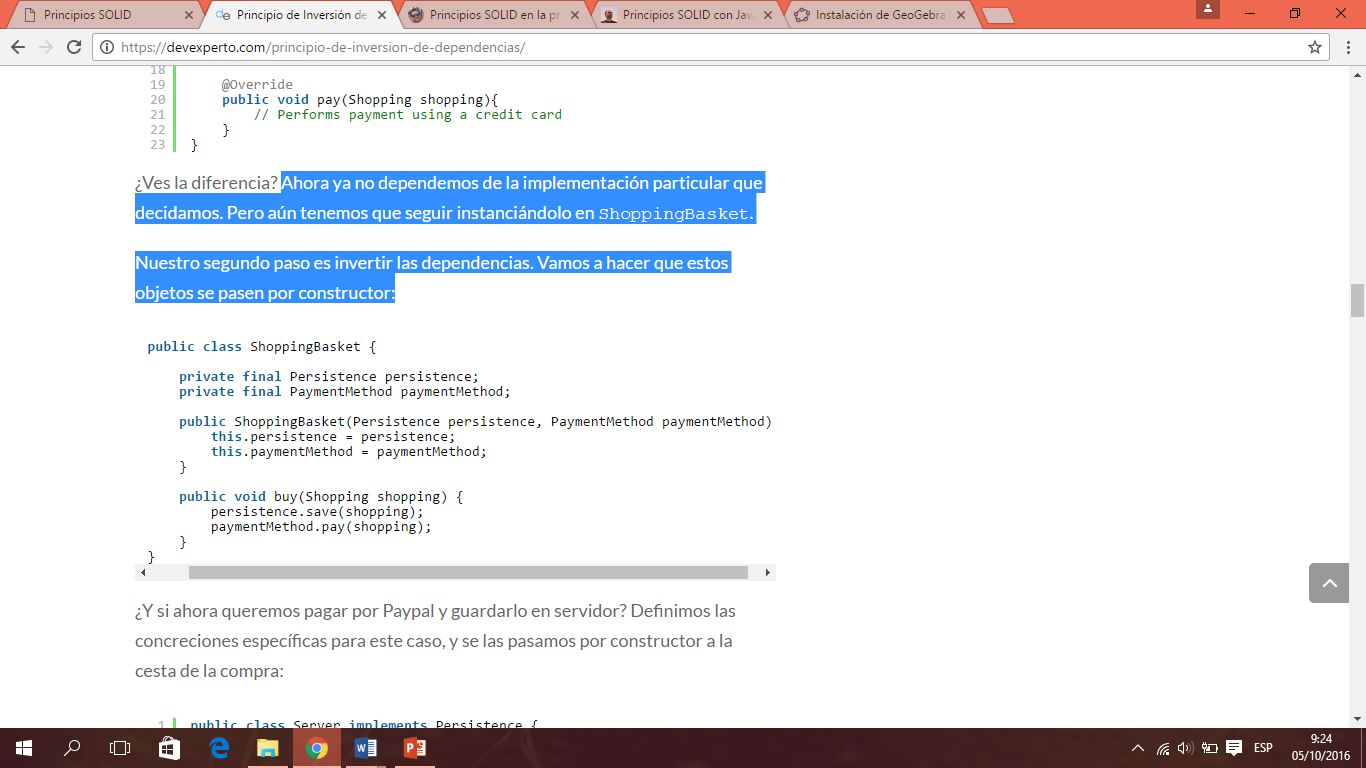
Piensa ahora qué pasa si quieres añadir métodos de pago, o enviar la información a un servidor en vez de guardarla en una base de datos local. No hay forma de hacer todo esto sin desmontar toda la lógica. ¿Cómo lo solucionamos?

Primer paso, dejar de depender de concreciones. Vamos a crear interfaces que definan el comportamiento que debe dar una clase para poder funcionar como mecanismo de persistencia o como método de pago:

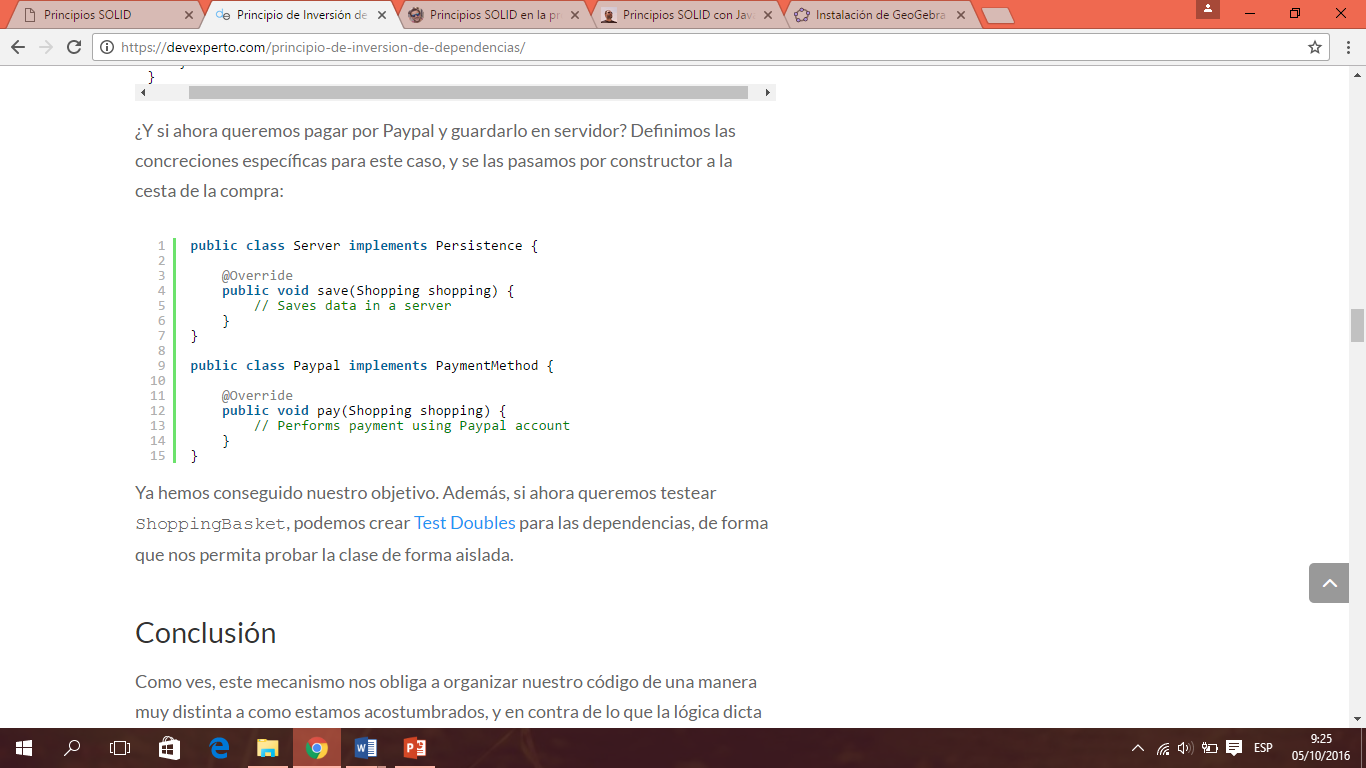


Ahora ya no dependemos de la implementación particular que decidamos. Pero aún tenemos que seguir instanciándolo en ShoppingBasket.

Nuestro segundo paso es invertir las dependencias. Vamos a hacer que estos objetos se pasen por constructor:



¿Y si ahora queremos pagar por Paypal y guardarlo en servidor? Definimos las concreciones específicas para este caso, y se las pasamos por constructor a la cesta de la compra:



## **CONCLUSIONES**

* El Principio de Responsabilidad Única es una herramienta indispensable para proteger nuestro código frente a cambios, ya que implica que sólo debería haber un motivo por el que modificar una clase.

En la práctica, muchas veces nos encontraremos con que estos límites tendrán más que ver con lo que realmente necesitemos que con complicadas técnicas de disección. Tu código te irá dando pistas según el software evolucione.

* El principio Open/Closed es una herramienta indispensable para protegernos frente a cambios en módulos o partes de código en los que esas modificaciones son frecuentes. Tener código cerrado a modificación y abierto a extensión nos da la máxima flexibilidad con el mínimo impacto.
* El Principio de Liskov nos ayudara a utilizar la herencia de forma correcta, y a tener mucho más cuidado a la hora de extender clases. En la práctica nos ahorrará muchos errores derivados de nuestro afán por modelar lo que vemos en la vida real en clases siguiendo la misma lógica. No siempre hay una modelización exacta, por lo que este principio nos ayudará a descubrir la mejor forma de hacerlo.
* El principio de segregación de interfaces nos ayudara a no obligar a ninguna clase a implementar métodos que no utiliza. Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas. Además nos ayudara a reutilizar código de forma más inteligente.
* Como se observó, El Principio de inversión de Dependencias nos obligara a organizar nuestro código de una manera muy distinta a como estamos acostumbrados, y en contra de lo que la lógica dicta inicialmente, pero a la larga se compensara por la flexibilidad que otorga a la arquitectura de nuestra aplicación.

## **RECOMENDACIONES**

Como dice Robert C. Martin en su libro *Clean Code*, lo más probable es que el siguiente que va a tener que mantener el código que tú has creado, vas a ser tú mismo. Aunque sólo sea por esto, debemos mantener un código de calidad para poder decir con orgullo, que somos informáticos y programadores.

# **BILBIOGRAFIAS**

<https://www.adictosaltrabajo.com/tutoriales/solid-5/>

<https://devexperto.com/principios-solid-guia-gratis/>