

**FACULTAD DE CIENCIAS E INGENIERÍA**

**ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMATICA**

**TEMA**

**PRINCIPIOS SOLID**

**CURSO DE:**

**Programación I**

Coronel Castillo Erick Gustavo

Integrantes:

\*Taipe Jara José

\* Santiago Hermosilla Antony

\*Maclena Gerson

Los Olivos – Perú

2017 III

**QUE SON LOS PRINCIPIOS SOLID**

Son un preparativo del Code Camp de Tarragona ‘2009 cuando surgió la idea de escribir esta serie de artículos para describir cinco principios fundamentales de la Programación Orientada a Objetos (POO) cuyas iniciales conforman las siglas SOLID. La comprensión de dichos principios nos permitirá mejorar la percepción del no siempre fácil campo de la POO, evitando así malas prácticas que la gran flexibilidad que ofrece esta metodología otorga, fundamentalmente a través de los lenguajes y herramientas que la soportan. Las herramientas de desarrollo rápido (Rapid Application Development, RAD), como Visual Studio 2010, ofrecen al desarrollador un conjunto de funcionalidades que aumentan, ya sea a través de asistentes o mediante "arrastrar y soltar", la productividad en el desarrollo de aplicaciones, y le permiten focalizarse únicamente en la utilización de propiedades o eventos específicos, sin tener que preocuparse en muchas ocasiones del código generado "por debajo". Sin embargo, en ocasiones acabamos pagando un precio muy elevado, ya que estas herramientas dejan tras de sí "cajas negras lógicas" de código difíciles de modificar o reutilizar, en las que los conceptos clave de la orientación a objetos son sacrificados en aras de la productividad; pero hablar de productividad es hablar de facilidad de mantenimiento y calidad del software, con lo que dicho sacrificio, sencillamente, no tiene o debería tener cabida.

**S- RESPONSABILIDAD SIMPLE (SINGLE RESPONSIBILITY)**

Este principio trata de destinar cada clase a una **finalidad sencilla y concreta**. En muchas ocasiones estamos tentados a poner un método reutilizable que no tienen nada que ver con la clase simplemente porque lo utiliza y nos pilla más a mano. En ese momento pensamos "Ya que estamos aquí, para que voy a crear una clase para realizar esto. Directamente lo pongo aquí".

El problema surge cuando tenemos la necesidad de utilizar ese mismo método desde otra clase. Si no se refactoriza en ese momento y se crea una clase destinada para la **finalidad del método**, nos toparemos a largo plazo con que las clases realizan tareas que no deberían ser de su responsabilidad.

Con la anterior mentalidad nos encontraremos, por ejemplo, con un algoritmo de formateo de números en una clase destinada a leer de la base de datos porque fue el primer sitio donde se empezó a utilizar. Esto conlleva a tener métodos difíciles de detectar y encontrar de manera que el código hay que tenerlo memorizado en la cabeza.

**O-ABIERTO/CERRADO (OPEN/CLOSED)**

Principio atribuido a **Bertrand Meyer** que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo. Es decir, el diseño debe ser abierto para poderse extender pero cerrado para poderse modificar. Aunque dicho parece fácil, lo complicado es predecir por donde se debe extender y que no tengamos que modificarlo. Para conseguir este principio hay que tener muy claro cómo va a funcionar la aplicación, por donde se puede extender y cómo van a interactuar las clases.

El uso más común de extensión es mediante la **herencia**y la reimplementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una **interface**de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

Como ya he comentado llega un momento en que las necesidades pueden llegar a ser tan imprevisibles que nos topemos que con los métodos definidos en el interface o en los métodos extensibles, no sean suficientes para cubrir las necesidades. En este caso no habrá más remedio que romper este principio y refactorizar.

**L-SUSTITUCION LISOV (LISKOV SUSTITUTION)**

Este principio fue creado por **Barbara Liskov** y habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando creamos clases derivadas debemos asegurarnos de no reimplementar métodos que hagan que los métodos de la clase base no funcionases si se tratasen como un objeto de esa clase base.

**I.SEGREGACION DEL INTERFACE (INTERFACES SEGREGATION)**

Este principio fue formulado por **Robert C. Martin** y trata de algo parecido al primer principio. Cuando se definen interfaces estos deben ser específicos a una finalidad concreta. Por ello, si tenemos que definir una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definan pocos métodos que tener un interface con muchos métodos.

El objetivo de este principio es principalmente poder **reaprovechar los interfaces** en otras clases. Si tenemos un interface que **compara y clona** en el mismo interface, de manera más complicada se podrá utilizar en una clase que solo debe comparar o en otra que solo debe clonar.

**D.INVENCION DE DEPENDENCIAS (DEPENDENCY INVERSION)**

También fue definido por **Robert C. Martin**. El objetivo de este principio conseguir desacoplar las clases. En todo diseño siempre debe existir un acoplamiento pero hay que evitarlo en la medida de lo posible. Un sistema no acoplado no hace nada pero un sistema altamente acoplado es muy difícil de mantener.

El objetivo de este principio es el uso de abstracciones para conseguir que una clase interactue con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la**inyección de dependencias o service locator** que nos permiten invertir el control.

**PLANTEAMIENTO DEL PROBLEMA**

**¿Cuál es la importancia de los principios SOLID en la programación?**

**PLANTEAMIENTO DE SOLUCION**

* Los principios SOLID pretenden ser una guía a seguir durante la fase de desarrollo para facilitar el mantenimiento de las aplicaciones.
* Los principios SOLID tratan de eliminar el impacto de las inevitables modificaciones que éstas sufren durante su ciclo de vida, además de facilitar el uso de las unidades de testeo, entre otras ventajas.
* La correcta aplicación del SRP simplifica el código y se traduce en facilidad de mantenimiento, mayores posibilidades de reutilización de código y de crear unidades de testeo específicas para cada responsabilidad.

**COMO IMPLEMENTAR LA SOLUCION**

* Buscar información en libros, páginas web, foros y revistas sobre la los principios SOLID.
* Escuchando información de fuente de entrevistas para analizar como la programación ha ido mejorando durante el pasar de los años, y la ayuda que ha brindado a la humanidad.
* Utilización del programa Java netbeans.
* Investigando códigos para utilizar en nuestra programación con ayuda del profesor.
* Leer analizar y comprender el programa realizado.
* Analizar su funcionamiento de dicho programa viendo su estructura y la manera como ayudar a programar.

**JUSTIFICACION DE LA INVESTIGACION**

La programación es intentar resolver problemas a través de una determinada metodología.

Es el proceso de escribir programas. Un programa de computador es una serie de instrucciones órdenes dadas a la máquina para que se produzca la ejecución de una determinada tarea.

Hoy en día la programación en nuestra sociedad es crucial viendo la evolución que atenido la sistematización de tareas y el manejo de información ahora es un hecho, la misma que tiene como principal función el hecho de conseguir que innumerables trabajos que antes ejercíamos de forma manual y con alto costos sean ejecutados por un ordenador con ahorro significativo de tiempo, por otro lado es importante dada que maneja gran cantidad de información que hoy en día manejamos en distintas tareas, la programación es una herramienta con muchísimo valor porque permite navegar en la misma con muchísima facilidad.

Los programas son dirigidos a personal de todas las áreas y niveles de un industria tiene como propósito apoyar al personal que contribuya con su labor diaria, mejorando constantemente la imagen de sus empresas, los programas hace que los servicios que se proporcionan a los clientes sea más rápida y tengan más satisfacción.

**OBJETIVO GENERAL**

* Facilitar la programación y darle solución brindándole guías para mayor efectividad de una labor con menos esfuerzo y en tiempo real.

**OBJETIVOS ESPECIFICOS**

* Indagar la importancia de los principios SOLID.
* Analizara las ventajas y las posibles desventajas.
* Ir mostrando los beneficios que nos ofrece los principios SOLID.

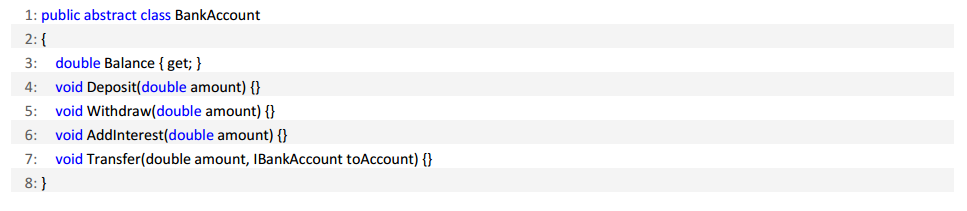
**MARCO TEORICO**

**Principio de responsabilidad única por Sean Chambers**

Después de Chad y Ray, también seguí el ejemplo y estoy haciendo el tema de Pablo del mes en el Principio de Responsabilidad Única o SRP para abreviar.

En SRP una razón para cambiar se define como una responsabilidad, por lo tanto SRP afirma, "Un objeto debe sólo tiene una razón para cambiar. "Si un objeto tiene más de una razón para cambiar, entonces tiene más de una responsabilidad y está en violación de SRP. Un objeto debe tener una y sólo una Razón para cambiar.

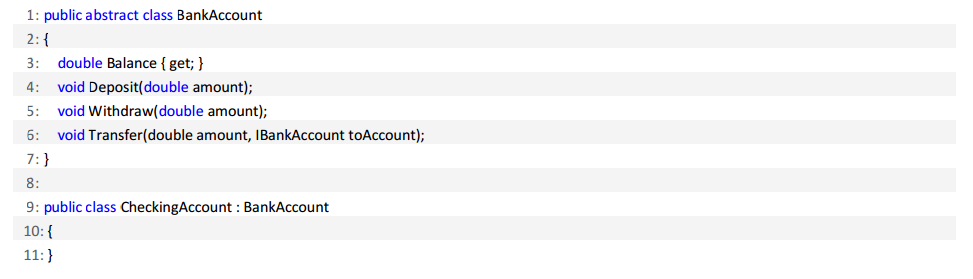
Veamos un ejemplo. En el ejemplo siguiente tengo una clase BankAccount que tiene un par de métodos:



Digamos que usamos esta clase BankAccount para la cuenta de Cheques y Ahorros de una persona. Ese haría que esta clase tuviera más de dos razones para cambiar. Esto es porque la comprobación cuentas no tienen interés añadido a ellos y sólo las cuentas de ahorro tienen interés añadido a ellos sobre una base mensual o sin embargo el banco lo calcula.

Algunas personas pueden decir que la clase tendría incluso 3 razones para cambiar debido a la depósito / retirar los métodos también, pero creo que definitivamente puede obtener un poco loco con SRP. Ese diciendo, creo que sólo depende del contexto.

Por lo tanto, vamos a refactorizar esto para ser más amigable SRP.

****

Así que lo que hemos hecho es simplemente crear una clase abstracta de BankAccount y luego crear una concreto CheckingAccount y SavingsAccount para poder aislar los métodos que están causando más de una razón para cambiar.

Cuando realmente piensas en ello, cada clase en el .Net Framework está violando SRP todo el tiempo. Los métodos GetHashCode () y ToString () están causando más de una razón para cambio, aunque se podría decir que estos métodos están exentos porque existen en el propio marco y fuera de nuestro alcance para el cambio.

Estoy seguro de que puede llegar a muchos más casos en los que han violado SRP, e incluso casos en los que sólo depende del contexto. Como se indica en Object Mentor: "El SRP es uno de el más simple del principio, y uno de los más difíciles de acertar”.

**Open Closed Principle by Joe Ocampo**

El principio abierto cerrado es uno de los principios más antiguos del Diseño Orientado a Objetos. No aburriré a usted con la historia ya que puede encontrar innumerables artículos en la red. Pero si quieres una comprensiva lea, por favor revise la excelente escritura de Robert Martin sobre el tema.

El principio abierto cerrado puede ser invocado en la siguiente declaración.

El principio abierto / cerrado indica que "las entidades de software (clases, módulos, funciones, etc.) deben ser abierto para la extensión, pero cerrado para la modificación "; [1] es decir, tal entidad puede permitir su comportamiento para ser modificado sin alterar su código fuente.

Suena bastante fácil pero muchos desarrolladores parecen faltar la marca en realmente la aplicación de esto enfoque extensible simple. No creo que sea una cuestión de habilidad, tanto como yo siento que ellos nunca se les ha enseñado cómo abordar la aplicación de OCP al diseño de clase.

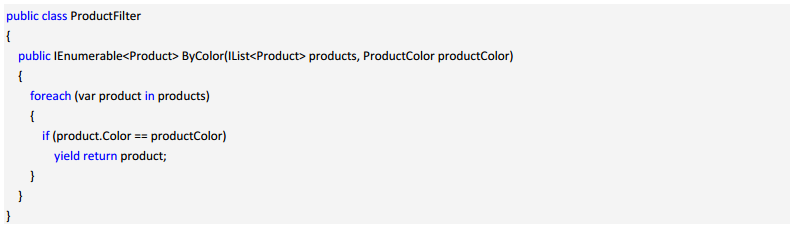
Escenario: Necesitamos una manera de filtrar productos basados en el color del producto.

Todas las entidades en un ecosistema de desarrollo de software se comportan de cierta manera que depende de un contexto gobernado. En el escenario anterior, te das cuenta de que vas a necesitar una clase de filtro que acepta un color y luego filtra todos los productos que se adhieren a ese color.

La responsabilidad de la clase de filtro es filtrar los productos (su trabajo), basándose en la acción del filtrado por color (su comportamiento). Así que su objetivo es escribir una clase que siempre será capaz de filtrar los productos.(Trabajar conmigo en esto estoy tratando de conseguir que en una mentalidad porque eso es todo OCP realmente está en su

Corazón.) Para hacer esto más fácil me gusta decir a los desarrolladores que llenen la siguiente plantilla.

La {clase} es responsable de {su trabajo} por {acción / comportamiento} el ProductFilter es responsable de filtrar los productos por color ahora, vamos a escribir nuestra clase simple para hacer esto:



Como se puede ver esto prácticamente hace el trabajo de filtrar un producto por color. Bastante simple, pero imagine si usted tuvo la siguiente conversación típica con uno de sus usuarios.

Usuario: "Necesitamos también ser capaces de filtrar por tamaño."

Desarrollador: "Sólo tamaño solo o color y tamaño”

Usuario: "Umm probablemente ambos."

Desarrollador: "Great!"

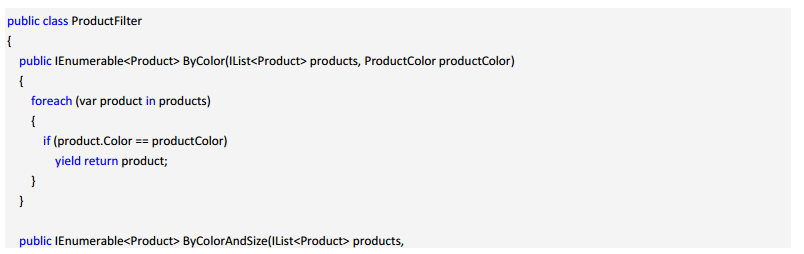
Así que vamos a usar nuestra plantilla de escenario OCP de nuevo.

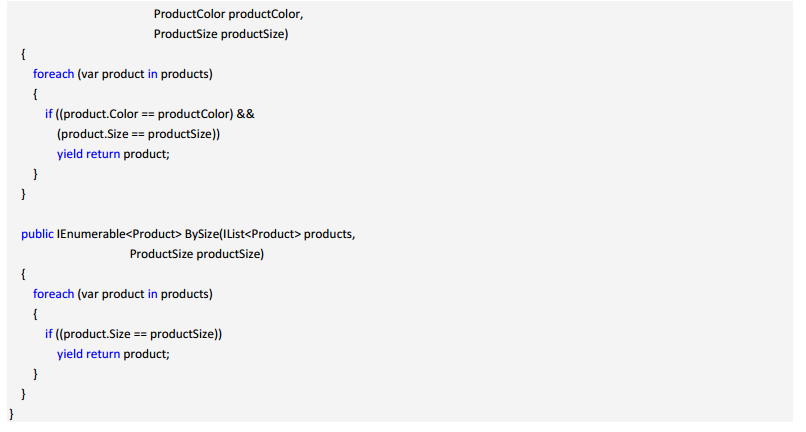
El ProductFilter es responsable de filtrar los productos por color

El ProductFilter es responsable de filtrar los productos por tamaño

El ProductFilter es responsable de filtrar los productos por color y tamaño

Ahora el código:





Esto es genial, pero esta implementación está violando OCP.

¿Dónde nos equivocamos?

Vamos a revisar de nuevo lo que Robert Martin tiene que decir sobre OCP.

Robert Martin dice que los módulos que se adhieren al Principio de Cierre Abierto tienen dos atributos principales:

1. "Open For Extension" - Es posible ampliar el comportamiento del módulo como el

Requisitos del cambio de aplicación (es decir, cambiar el comportamiento del módulo).

2. "Cerrado para modificación" - Ampliar el comportamiento del módulo no da lugar a la

El cambio del código fuente o código binario del propio módulo.

Hagamos la siguiente pregunta para asegurar que estamos violando la OCP.

Cada vez que un usuario solicita nuevos criterios para filtrar un producto, ¿tenemos que modificar el Clase ProductFilter?

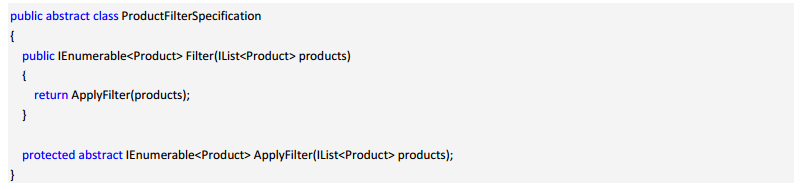
¡Sí! Esto significa que no está CERRADO para modificaciones.

Cada vez que un usuario solicita nuevos criterios para filtrar un producto, ¿podemos extender el ProductFilter para admitir los nuevos criterios, sin abrir de nuevo el archivo de clase y Modificándolo ¡No! Esto significa que no está ABIERTO para la extensión.

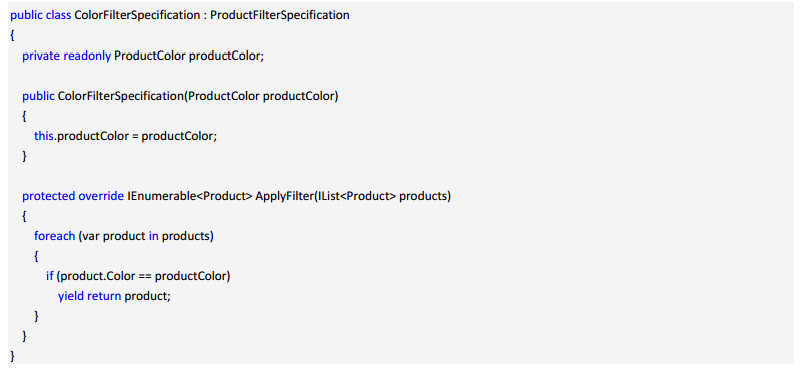
Soluciones

Una de las maneras más fáciles de implementar OCP es utilizar una plantilla o patrón de estrategia. Si todavía permitir que el filtro de producto para realizar su trabajo de invocar el proceso de filtrado, podemos poner el implementación de filtrado en otra clase. Esto se logra mezclando un poco de LSP.

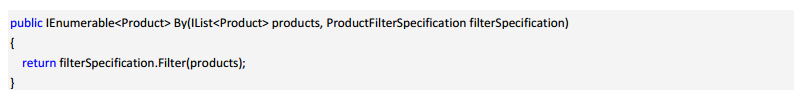
Aquí está la plantilla para el ProductFilterSpecification:



Vamos a seguir adelante y crear nuestro primer criterio, que es una especificación de color.



Ahora todo lo que tenemos que hacer es ampliar la clase ProductFilter actual para aceptar nuestra plantilla ProductFilterSpecification.



Así que asegúrese de que NO estamos violando OCP y hacer las mismas preguntas que hicimos antes.

Cada vez que un usuario solicita nuevos criterios para filtrar un producto, debemos modificar el Clase ProductFilter? ¡No! Porque hemos organizado el comportamiento de filtrado a la ProductFilterSpecification.

"Cerrado para modificación" cada vez que un usuario solicita nuevos criterios para filtrar un producto podemos extender el ProductFilter para admitir los nuevos criterios, sin abrir de nuevo el archivo de clase y modificándolo ¡Sí! Todo lo que simplemente tenemos que hacer es pasar una nueva ProductFilterSpecification. "Abrir para extensión" ahora vamos a asegurarnos de que no hemos modificado demasiado de nuestras intenciones de la ProductFilter. Todo lo que simplemente tenemos que hacer es validar que nuestro ProductFilter todavía tiene el mismo comportamiento como antes.

El ProductFilter es responsable de filtrar los productos por color: ¡Sí, todavía lo hace!

El ProductFilter es responsable de filtrar los productos por tamaño: ¡Sí, todavía lo hace!

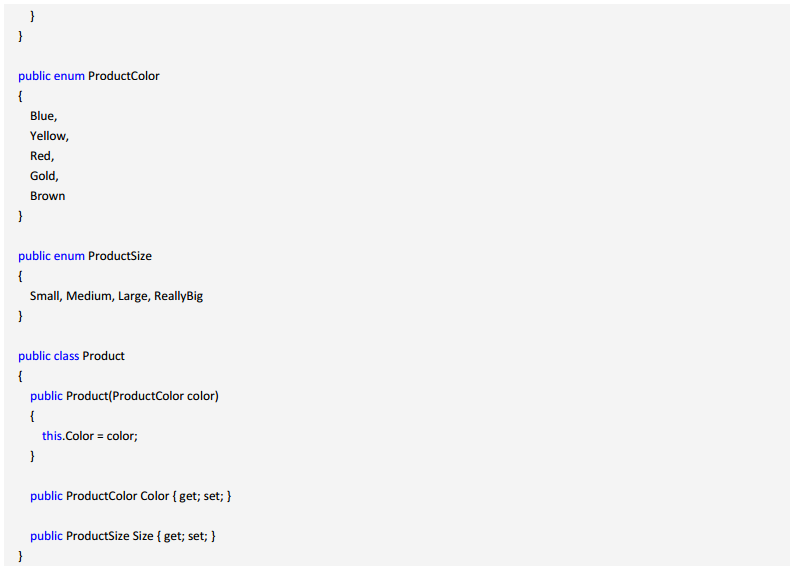
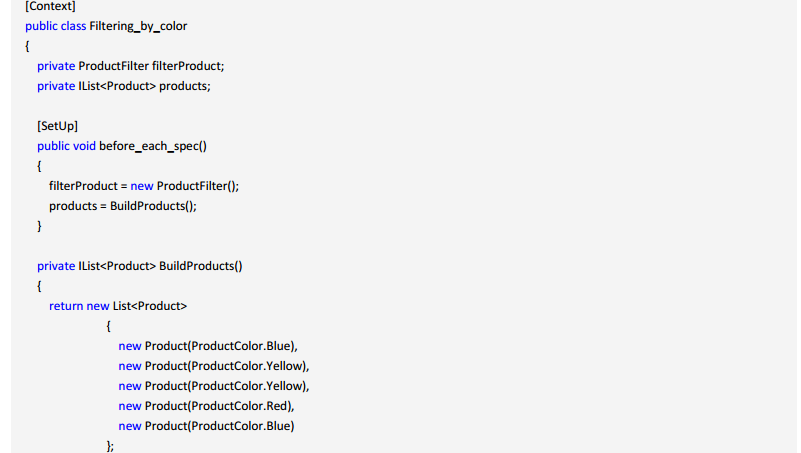
El ProductFilter es responsable de filtrar los productos por color y tamaño: ¡Sí, todavía lo hace!

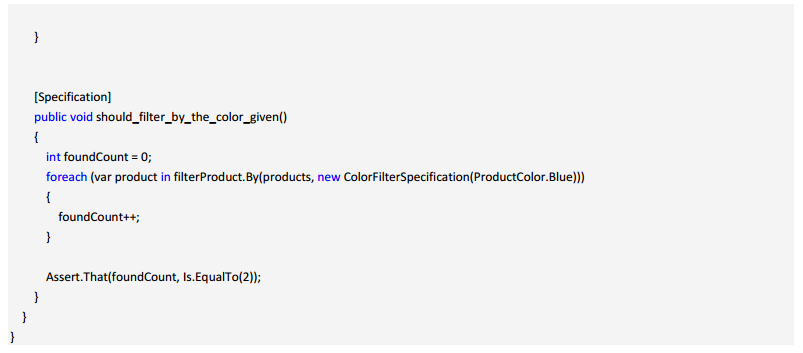
Si usted es un buen practicante de TDD / BDD ya debe tener todos estos escenarios cubiertos en su Suite de Prueba.

Aquí está el código final:









**Principio de sustitución de Liskov por Chad Myers**

En mi primer (de esperanzadamente más de un) post para The Los Techies El tema del mes de Pablo -Marzo: SOLID Principles effort, voy a hablar sobre el Principio de Substitución Liskov, como popularizado por Robert 'Uncle Bob' Martin en The C ++ Report.

Voy a intentar lo más posible no repetir todo lo que el tío Bob dijo, puede ir a leer las cosas importantes allí. Voy a tratar de dar algo real ejemplos y relacionar esto con el mundo .NET.

En caso de que sea demasiado perezoso para leer el enlace, permítanme comenzar con un breve resumen de lo que es LSP: Si usted tiene una clase base BASE y subclases SUB1 y SUB2, el resto de su código debe siempre consulte BASE y NO SUB1 y SUB2.

Un estudio de caso en la ignorancia de LSP los problemas que soluciona LSP casi siempre son fácilmente evitables. Hay algunos habituales tell-tale signos de que una violación LSP está apareciendo en su código. Aquí está un escenario que camina a través de cómo una violación LSP podría ocurrir. Estoy seguro de que todos hemos tenido situaciones similares. Ojalá caminando a través de esto, puede empezar a acostumbrarse a detectar la tendencia por adelantado y cortarla antes te pintas en una esquina.

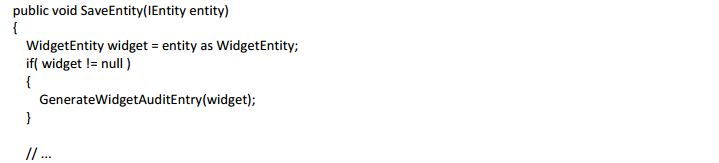
Digamos que en algún lugar de su código de acceso a datos tenía un método ingenioso a través del cual todos sus DAO / Entidades pasaron y hizo cosas comunes como establecer el CreatedDate / UpdatedDate, etc.



Inteligente, trabaja como un campeón. Muchos de ustedes esperanzadamente se han encogido con este código. Tuve un duro tiempo para escribirlo, pero es para ilustración. Hay un montón de código por ahí escrito de esta manera. Si tú no se encogió y usted no ve lo que está mal con ese código, por favor continúe leyendo. Ahora el las partes interesadas vienen a usted con una solicitud de característica:

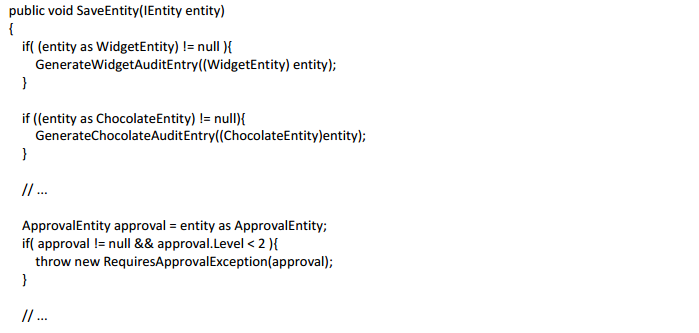
Cada vez que un usuario guarda un widget, necesitamos generar un registro de auditoría de widgets en la base de datos para seguimiento posterior.

Podrías sentir la tentación de añadirlo a tu rutina de SaveEntity práctico-dandy a través de la cual todos entidades pasan:



¡Estupendo! Eso también funciona como un campeón. Pero unas semanas más tarde, vienen a usted con una lista de 6 otras entidades que necesitan características similares de auditoría. Así que conecta esas 6 entidades. Unas pocas semanas más tarde, el venir a usted y pedirle algo como esto: cuando se guarda un registro de aprobación, necesitamos verificar que la aprobación sea del nivel correcto. Si no es el nivel correcto, tenemos que pedir al usuario una excusa, de lo contrario no pueden continuar ahorrando.

Oh chico, eso es complicado. Bueno, ahora nuestro SaveEntity se parece a esto:



Muy pronto su pequeño, inteligente método SaveEntity es de 1.500 líneas de largo y sabe todo acerca de cada entidad en todo el sistema.

¿Dónde nos equivocamos?

Bueno, hay varios lugares para empezar. Centralizar el ahorro de entidades no es la mejor idea.

Poner la lógica para la auditoría si las entradas deben ser creadas o no en la SaveEntity método era definitivamente la cosa incorrecta a hacer. Finalmente, debido a las complejidades de manejar diferente lógica de negocio para diferentes entidades, tiene un problema de flujo de control con la aprobación nivel que requiere el uso de una excepción lanzada para salir del flujo (que es similar a un 'Goto' declaración en días de antaño).

Las preocupaciones de la auditoría, el establecimiento de fechas creadas / actualizadas y los niveles de aprobación son ortogonales entre sí y no deben ser vistos juntos, pasando por el mismo método, generalmente haciendo un lío de cosas.

Más al punto de esta entrada del blog; SaveEntity viola el principio de sustitución de Liskov. Es decir SaveEntity toma una clase interfaz / base de IEntity pero trata con subclases específicas yImplementaciones de IEntity. Esto viola una regla fundamental del diseño orientado a objetos (Polimorfismo) ya que SaveEntity pretende trabajar con cualquier implementación particular de IEntity cuando, de hecho, no lo hace. Más exactamente, no trata a todos los IEntity exactamente igual, algunos obtienen más atención que otros.

¿Por qué es esto un problema? ¿Qué pasa si usted estaba reutilizando su método de SaveEntity terriblemente inteligente en otro proyecto y tienen docenas de implementaciones de IEntity y las partes interesadas para proyecto también quería la función de auditoría. Ahora tienes un problema.

Soluciones

Una aproximación fina a este problema de hacer las cosas al momento del ahorro sería usar el patrón del visitante según lo descrito por Matthew Cory en este poste. Aunque, yo diría en este ejemplo, hay un problema de diseño muy arraigado y sistémico que centralización del acceso a los datos.

Otro, en nuestro caso más preferible, es el uso del patrón de repositorio para gestión del acceso a los datos. En lugar de tener "Un método para regir a todos", usted podría tener los repositorios se preocupan por la hora de la fecha creada / actualizada y diseñan un sistema las implementaciones de repositorio comparten parte de la lógica de creación / configuración de entidades de fecha / creación actualizada / creada.

A medida que surjan problemas puntuales específicos (como auditoría, aprobación / verificación extra, etc.), pueden ser manipulados de manera similar de una sola vez. Esto se logra por medio de la Repositorio (quién sabe todo sobre ese tipo de entidad y eso es todo). Si observa que varios 52 SOLID Desarrollo de software de Pablo | LosTechies.com las entidades están haciendo el mismo tipo de cosas (es decir, la auditoría). Puede, crear una clase y un método para manejar la auditoría de una manera común y proporcionar los diferentes repositorios que necesitan auditoría con esa funcionalidad. Resistir, si es posible, la necesidad de crear un "AuditingRepositoryBase" que proporciona la funcionalidad de auditoría. Inevitablemente, una de esas entidades que requieren auditoría tendrá otra preocupación ortogonal para la que tendrá otra clase \* Base y, dado que usted no puede hacer la herencia múltiple en. NET, usted ahora se pega. Prefieren la composición de funcionalidad sobre herencia de funcionalidad siempre que sea posible.

Si usted tiene un modelo de dominio rico, tal vez el enfoque más elegante de todos sería hacer cosas como la auditoría de una característica de primera clase del modelo de dominio. Cada widget siempre tiene al menos un WidgetAuditEntry asociado con él y esta asociación se gestiona a través del dominio lógica misma. Asimismo, el nivel de aprobación se manejaría mejor en la cadena lógica hasta evitar las "trampas" de última hora en el ciclo de vida que requeriría algo menos que elegante como una excepción como un rescate de "goto" velado.

**Principio de Segregación de PInterface por Ray Houston**

En la siguiente suite con The Los Techies El tema del mes de Pablo - Marzo: SÓLIDO principios, escogí escribir un poco sobre el principio de la segregación de la interfaz (ISP). Como Chad señaló con LSP, el ISP es también uno de Robert 'Uncle Bob' Martin's S.O.L.I.D diseño principios.

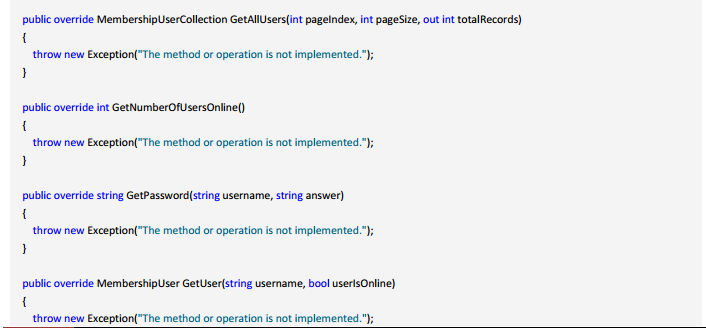
Básicamente, el ISP nos dice que los clientes no deben verse obligados a implementar interfaces que no utilizan. En otras palabras, si usted tiene una clase abstracta o una interfaz, los implementadores no deben ser obligados a implementar partes que no les importa.

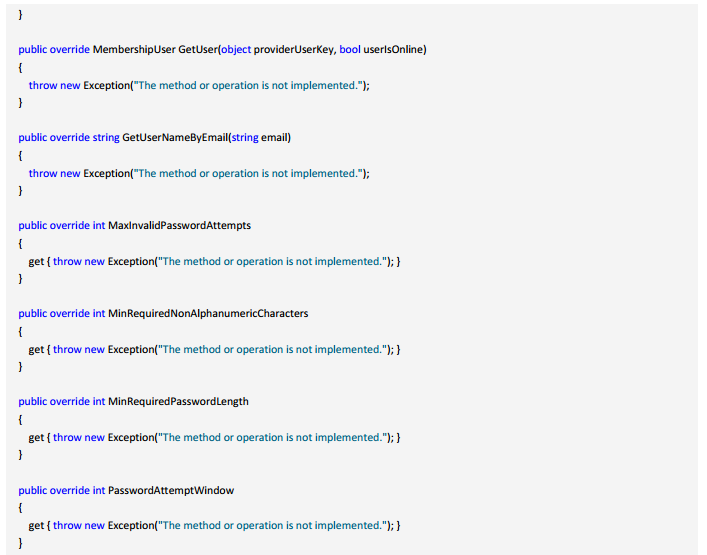
Tenía problemas para pensar en un ejemplo del mundo real para ISP, pero luego se me recordó implementación de un proveedor de membresía personalizado en ASP.NET 2.0. Había bloqueado completamente monstruosidad fuera de mi mente (por buenas razones).

El Proveedor de Membresía era una forma de integrarse con algunos de los ASP.NET construidos en gestión de usuarios y sus controles de servidor asociados. Para mí, terminó siendo mucho más problema que vale la pena, pero resulta ser un buen ejemplo de una interfaz de grasa. A fin de que implementar su propio Proveedor de Membresía usted "simplemente" implementa la clase abstracta MembershipProvider como así:









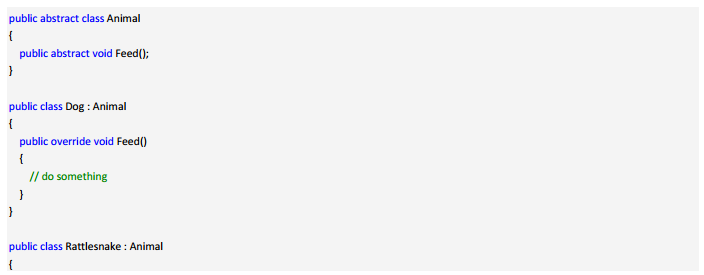


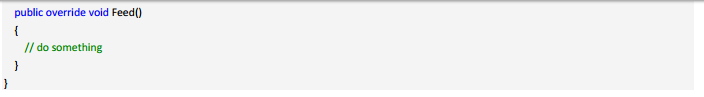


Resulta que usted no tiene que implementar las partes que no necesita, pero esto claramente viola el Principio de Segregación de Interfaz. Esta interfaz es extremadamente grasa y no cohesiva. Un mejor habría sido dividirlo en interfaces más pequeñas que permitan a los implementadores sólo se preocupan por las partes que necesitan. No voy a entrar en los detalles de dividir esto up, pero creo que tienes la idea.

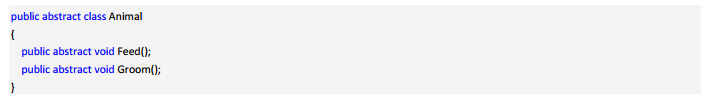
Como no puedo pensar en otro ejemplo del mundo real, veamos un ejemplo completamente falso.

Digamos que tiene el código siguiente:

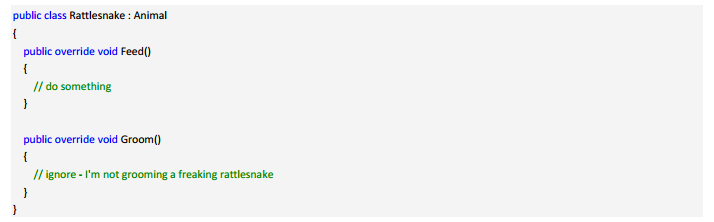




Pero entonces te das cuenta de que tienes una necesidad de algunos de los animales para ser tratados como mascotas y hacer que se arreglen. Usted puede ser tentado a hacer

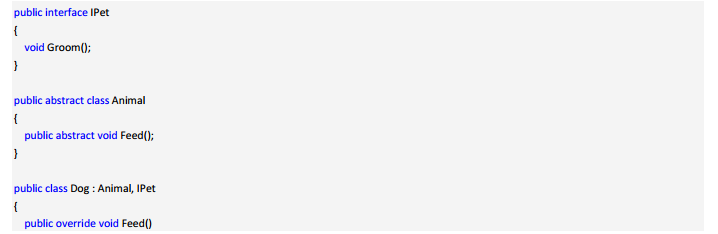


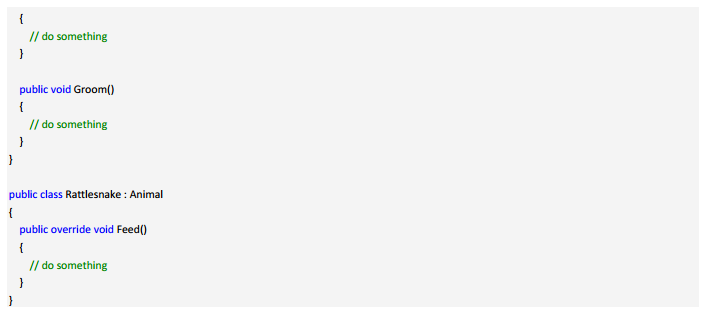
Que estaría bien para el perro, pero no puede ser fino para la serpiente de cascabel (aunque estoy seguro hay algunos freako por ahí que prepara su mascota serpiente de cascabel)



Aquí hemos violado el ISP contaminando nuestra interfaz Animal. Esto requiere que implementemos un método que no tiene sentido para el objeto de la Serpiente de cascabel. Una mejor opción sería implementar una interfaz IPet, que sólo Dog podría implementar, sin afectar a Rattlesnake.

Usted podría terminar con algo como esto:





Piensa que la clave es que si te encuentras creando interfaces que no se implementen completamente en sus clientes, entonces eso es una buena señal de que está violando el ISP.

**Dependency Inversion Principle de Jimmy Bogard**

El Principio de Inversión de Dependencia, el último del Tío Bob "SÓLIDO" diseño orientado a objetos principio natural, se puede pensar en la progresión natural del Principio de sustitución de Liskov, Principio Abierto Cerrado e incluso el Principio de Responsabilidad Única. Este post es el último en el Conjunto de mensajes SOLID:

 PTOM: El Principio de Responsabilidad Única

 PTOM: El principio abierto cerrado

 PTOM: El principio de sustitución de Liskov

 PTOM: El Principio de Segregación de Interfaz

El principio de inversión de dependencia, o DIP, se utiliza a menudo de forma intercambiable con la dependencia Inyección e Inversión de Control. Sin embargo, después de DIP no significa que debamos utilice automáticamente un contenedor IoC como Spring.NET, Windsor o StructureMap. Contenedores IoC son herramientas para ayudar en aplicaciones adheridas a DIP, pero podemos seguir DIP sin usar IoC contenedores el Principio de Inversión de Dependencia establece: o los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender sobre abstracciones.

O Las abstracciones no deben depender de detalles. Los detalles deben abstracciones el DIP puede ser un poco vago, ya que habla de "abstracciones", pero no describe lo que está siendo abstraído. Habla de "módulos", que no tienen mucho significado en .NET a menos que considere los "módulos" como conjuntos. Si está buscando en el diseño controlado por el dominio, los módulos significa algo completamente distinto.

El Principio de Inversión de Dependencia, junto con los otros principios SOLID, aliviar los problemas de los malos diseños. El software típico que encuentro en proyectos existentes código organizado en clases, pero todavía no es fácil de cambiar. Normalmente veo grandes bolas de barro con una telaraña loca de dependencias donde no se puede estornudar sin romper el código en el otro lado del planeta.

Telarañas y malos diseños, los malos diseños y el mal código no son buenos porque es difícil de cambiar. Los malos diseños son:

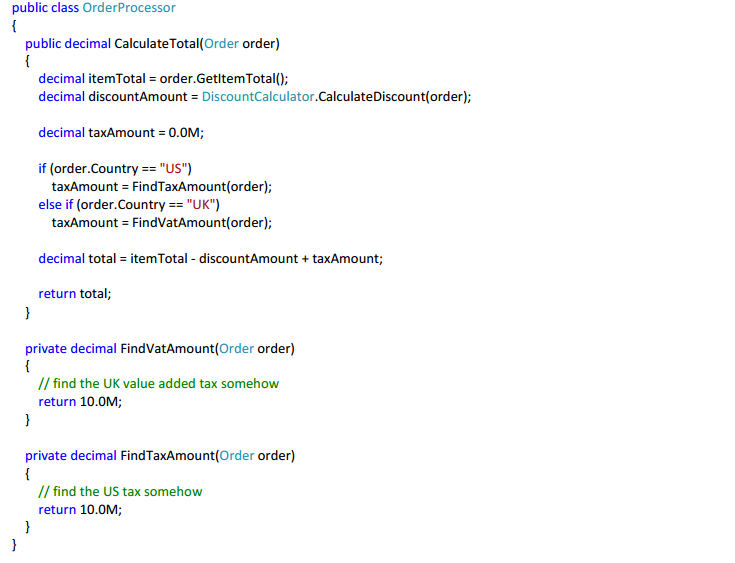
\*Rígido (el cambio afecta demasiadas partes del sistema)

\*Frágil (cada cambio rompe algo inesperado)

\*Inmovible (imposible reutilizar)

Algunas ideas de la gente de "mal diseño" sería algo como ver concatenación de cadena en lugar de StringBuilder. Aunque esto puede no ser la mejor opción, la cadena la concatenación no es necesariamente un mal diseño.

Es bastante fácil de detectar los malos diseños. Estas son secciones de código o aplicaciones completas que miedo de tocar. Un ejemplo típico de código rígido, frágil e inmóvil (malo) sería:



El OrderProcessor se propone hacer algo muy sencillo: calcular el total de una orden. A lo hace, debe saber el total del artículo de la orden, los descuentos aplicados, así como el impuesto importe (que depende del país de la Orden).

Demasiadas responsabilidades para ver por qué el DIP va de la mano con el principio de responsabilidad única, vamos a Responsabilidades del OrderProcessor:

 Saber calcular el total de artículos

 Encontrar la calculadora de descuento y encontrar el descuento

 Saber qué significan los códigos de país

 Encontrar el método impositivo correcto para cada código de país

 Saber calcular el impuesto para cada país (comentado por brevedad)

 Saber combinar todos los resultados en el total final correcto

Si una sola clase (o un solo método en este caso) responde a demasiadas preguntas (cómo, dónde, qué, por qué, etc.), es una buena indicación de que esta clase tiene demasiadas responsabilidades.

Para avanzar hacia un buen diseño, necesitamos eliminar las dependencias externas de la clase y reducirlo a su responsabilidad principal, es decir, encontrar el total de pedidos. De improviso, las dependencias I

Ver son:

 DiscountCalculator

 Decisiones tributarias

En el futuro, podríamos necesitar apoyar a más países, lo que significa más servicios de impuestos, y más responsabilidades. Para reducir la rigidez, fragilidad e inmovilidad de este diseño, necesitamos mover estas dependencias fuera de esta clase.

Hacia un mejor diseño al seguir el DIP, notará que el patrón de estrategia comienza a aparecer en muchos de sus diseños. La estrategia tiende a resolver los "detalles deben depender de las abstracciones" parte del DIP.

Factoring fuera del DiscountCalculator y las decisiones de impuestos, terminamos con dos nuevos

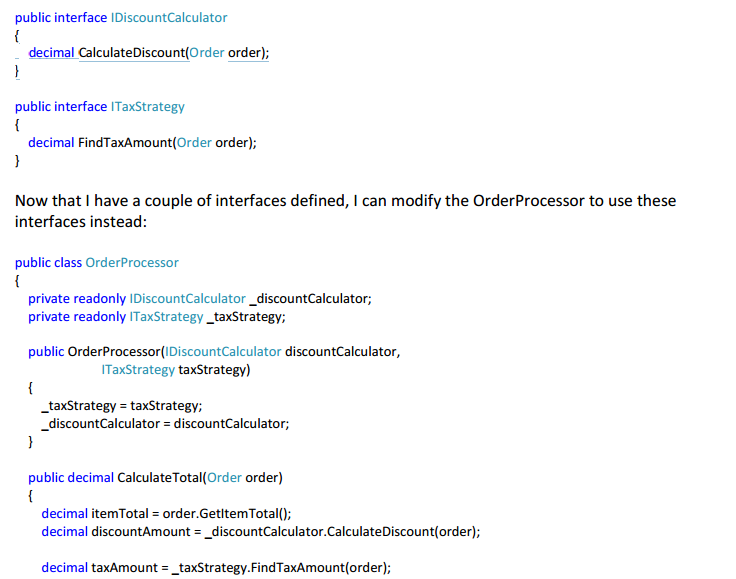
Interfaces:

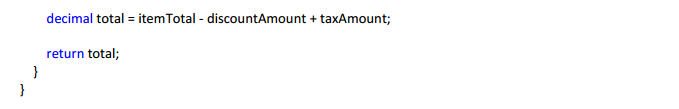
 IDiscountCalculator

 ITaxStrategy

No soy un gran fan del nombre "ITaxStrategy", pero será suficiente hasta que encontremos un mejor nombre de nuestro modelo.

Factoring de las dependencias para factorizar las dependencias, primero crearé un par de interfaces que coincidan con los existentes métodos de firmas:

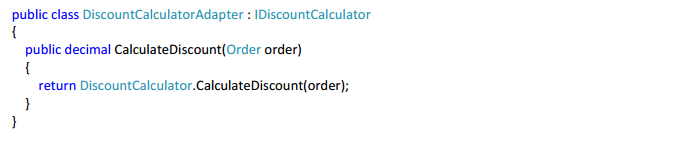




El método CalculateTotal parece mucho más limpio ahora, delegando los detalles de los descuentos y los impuestos a las abstracciones apropiadas. En lugar de OrderProcessor dependiendo directamente de los detalles, depende exclusivamente de las interfaces abstraídas que creamos anteriormente. Los detalles de cómo encontrar el el método fiscal correcto ahora está encapsulado en el OrderProcessor, así como la dependencia dura en un método estático en el Calculator de Descuento.

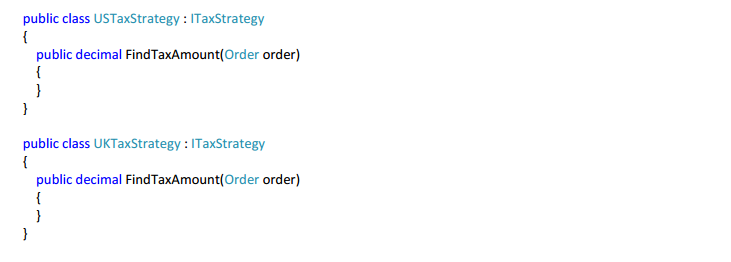
Completar las implementaciones ahora que tenemos las interfaces definidas, necesitamos implementaciones reales para estos dependencias. Mirando el Calculator de descuento, que es una clase estática, encuentro que no puedo Cambie inmediatamente a una clase no estática. Hay muchos otros lugares con referencias a este DiscountCalculator, y ya que es el mundo real, ninguno de estos otros lugares tienen pruebas.

En su lugar, sólo puedo utilizar el patrón de adaptador para adaptar la interfaz que necesito para una IDiscountCalculator:



Al aplicar el patrón de adaptador, sólo envuelvo el DiscountCalculator real en una clase diferente. En este caso, la ventaja del patrón de adaptador es el actual DiscountCalculator puede continuar cuando el mecanismo para calcular los descuentos cambia, mi OrderProcessor no necesidad de cambio.

Para las estrategias tributarias, puedo crear dos implementaciones para cada tipo de cálculo de impuestos hoy:



Dejé las implementaciones, pero básicamente trasladé los métodos del OrderProcessor a

Estas nuevas clases. Ninguno de los métodos originales usó ningún campo de instancia, así que podría copiar

Recto.

My OrderProcessor ahora tiene dependencias factorizadas, por lo que su única responsabilidad es fácilmente discernido de mirar el código. Además, las implementaciones de IDiscountCalculator y ITaxStrategy puede cambiar sin afectar al OrderProcessor.

66 Desarrollo de software SOLID de Pablo | LosTechies.com aislar las cosas feas para mí, el DIP es todo sobre el aislamiento de las cosas feas. Para calcular los totales de orden, no debería preocupado por dónde están los descuentos o cómo decidir qué estrategia tributaria debe ser utilizada.

Hicimos aumentar el número de clases significativamente, pero esto es lo que sucede cuando nos movemos lejos de una mentalidad procedimental a un verdadero diseño orientado a objetos.

Todavía tengo la complejidad de resolver de empujar las dependencias en el OrderProcessor.

Los clientes del OrderProcessor ahora tienen la carga de crear las dependencias correctas y

Dándoles a OrderProcessor. Ese problema ya está resuelto con Inversión de Control (IoC) contenedores como Spring.NET, Windsor, StructureMap, Unity y otros.

Estos contenedores IoC me permiten configurar el "qué" al inyectar dependencias, por lo que incluso la decisión se elimina del cliente. Si no quería ir con un contenedor de IoC, incluso un simple método de creación o clase de fábrica podría abstraer la construcción del OrderProcessor con las dependencias correctas.

Al adherirme al Principio de Inversión de Dependencia, puedo crear diseños que estén limpios, con responsabilidades claramente definidas. Con las dependencias extraídas, la implementación los detalles de cada dependencia pueden cambiar sin afectar a la clase original.

Ese es mi objetivo final: el código que es fácil de cambiar. Más fácil de cambiar significa un menor costo total de propiedad y mayor facilidad de mantenimiento. Dado que sabemos que los requisitos cambio, es nuestro mejor interés promover un diseño que facilite el cambio a través del principio de inversión de dependencia.

**CONCLUCIONES Y BENEFICIOS SOLID**

Acoplamiento bajo al resumir muchas de nuestras necesidades de implementación en varias interfaces e introducir los conceptos en OCP y DIP, hemos creado un sistema que tiene acoplamiento muy bajo. Muchas de estas piezas individuales ser sacado del sistema con poco o ningún desorden del espagueti que arrastra después de él. Separando los distintos nos ha ayudado a asegurar que podemos cambiar el comportamiento como sea necesario, con poca modificación al sistema general, sólo actualizar la pieza única que contiene el comportamiento en cuestión.

**ALTA COHESION**

Esto realmente es un resultado directo del acoplamiento bajo y SRP; Tenemos un montón de piezas pequeñas que se pueden apilar juntos como bloques de construcción para crear algo más grande y más complejo. Cualquiera de estos piezas no pueden representar mucha funcionalidad o comportamiento, pero entonces, una pieza individual no es muy divertido para uso sin un montón de otras piezas. DIP también nos ha permitido atar los varios bloques juntos por dependiendo de una abstracción y permitiendo que esa abstracción se cumpla con diferentes implementaciones. Esto crea un sistema que es mucho mayor que la mera suma de sus partes.

**ENCAPSULACION**

La verdadera encapsulación no es sólo hacer que los campos sean privados y ocultar datos de objetos externos, esconderse detalles de la implementación de otros objetos, dependiendo únicamente de las abstracciones y comportamientos esperados de esas abstracciones. LSP, DI y SRP trabajan de la mano para crear encapsulación verdadera en el nuevo estructura del proyecto. Hemos encapsulado nuestras implementaciones de comportamiento en muchos objetos individuales, impidiéndoles que se escapen unos a otros mientras se asegura que la dependencia de esos comportamientos es encapsulado detrás de una interfaz conocida. Hemos ocultado los detalles de implementación y hemos permitido implementación para la definición de la interfaz a través de DIP. Al mismo tiempo, hemos la debida diligencia necesaria para garantizar que no estamos violando ninguna de la semántica o el propósito (LSP), asegurando que podemos reemplazar adecuadamente la implementación según sea necesario.

**REFERENCIAS BIBLIOGRAFICAS**

Rubira, J. (2011), *Solid cinco principios básicos de diseño de clases,* Recuperado de <https://www.genbetadev.com/paradigmas-de-programacion/solid-cinco-principios-basicos-de-diseno-de-clases>

Torres, J.M y Hariri, H (2016), *Principio de responsabilidad única,* Recuperado de <http://www.lawebdelprogramador.com/temas/archivos/dotnetmania_71_pag32-35.pdf>

Pablos. (2008), *Pablo's SOLID Software Development,* Recuperado de

<https://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf>