PATRONES CREACIONALES DE DISEÑO

Cristhian Mauricio Yara Pardo
20181020081
Universidad Distrital Francisco José de Caldas
Bogotá, Colombia
cmyarap@correo.udistrital.edu.co

***Abstract*— This paper seeks to develop a brief overview of the concepts of "creational design patterns" for software development.**
With which the software extensibility and easy maintenance thereof is not guaranteed.

INTRODUCCIÓN

Este documento busca desarrollar un breve resumen sobre los conceptos de los "patrones creacionales de diseño", para el desarrollo de software.
Con los cuales se garantiza la extensibilidad de dicho software y el fácil mantenimiento del mismo.

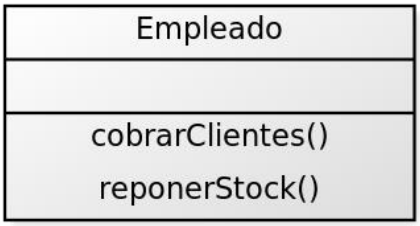
SOLID

Es un conjunto de 5 principiosque si como programadores aplicamos siempre, estaremos desarrollando software más robusto y mucho más fácil de mantener. Los principios que se incluyen, son los siguientes:

SINGLE RESPONSIBILITY PRINCIPLE

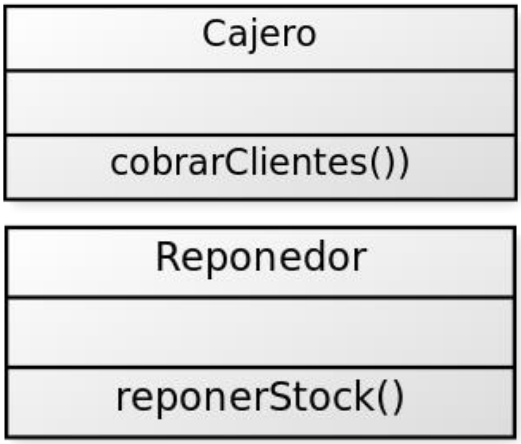
Este principio determina que las clases o módulos deben tener una única responsabilidad y una sola razón para cambiar. Además, esa clase o modulo debe ser la única con esa responsabilidad. Es decir, si tenemos que cambiar una clase, que sea por una única razón.

Ejemplo:
En el momento de diseñar un software para un supermercado, una primera aproximación que generamos es la siguiente:



Donde la clase empleado tiene más de una responsabilidad, en la cual se encarga de cobrar y mantener el stock.

Pero puede ser que los requisitos deban cambiarse y ahora se asigne a otro tipo de personal, con ciertas características la función de mantener el stock, lo cual implica hacer cambios en dicha clase (empleado), y esta después nos pueda generar una serie de problemas, para lo cual, determinamos que la manera más conveniente es convertir esta clase en dos, asignando responsabilidades diferentes.



OPEN-CLOSED PRINCIPLE

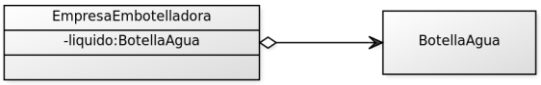
La idea de este principio establece que las clases o módulos estén abiertas a la extensión, pero cerradas a la modificación, lo que significa que, ante peticiones de cambio en nuestro programa, hay que ser capaces de añadir funcionalidad sin modificar la existente (siempre que sea posible).

En un mal diseño, modificar una funcionalidad durante el ciclo de vida del software o aplicación, suele desencadenar una serie de cambios adicionales, y este efecto puede propagarse aún más, si el código está muy acoplado.

Una de las formas más recomendadas para seguir el principio, es hacer uso de interfaces o clases abstractas de las que dependen implementaciones concretas.
De esta forma puede cambiarse la implementación de la clase concreta manteniéndose la interfaz intacta.

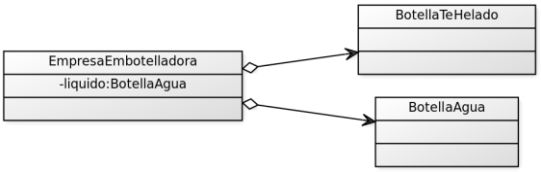
Ejemplo:

Tenemos una empresa que desde sus comienzos ha estado vendiendo agua embotellada,
y su programa informático tenía esta forma.



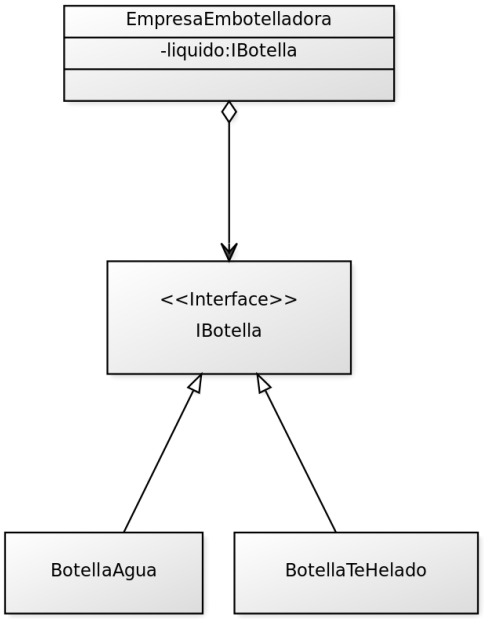
Ahora ha surgido una oportunidad de negocio y quiere empezar a vender botellas de té helado, por lo que necesita un cambio en su modelo.

Una primera aproximación sería algo similar a esto:



Pero implicaría realizar cambios en la empresa, que tiene que aprender a comunicarse con el nuevo tipo de botella, que probablemente sea muy parecida a la anterior.

Si hubieran seguido el principio Abierto-Cerrado su diagrama de clases habrían utilizado una interfaz que comunicaría el resto del sistema con las botellas, por lo que, si no se cambia el api, el resto del sistema puede trabajar ajeno al cambio.
Quedaría de esta forma.



LISKOV SUBSTITUTION

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

INTERFACE SEGREGATION PRINCIPLE

Para referirse a este principio se suelen usar las frases:

"Muchas interfaces específicas son mejores que una única más general"

"Los clientes no deberían verse forzados a depender de interfaces que no usan"

Cuando los clientes son forzados a utilizar interfaces que no usan por completo, están sujetos a cambios de esa interfaz. Esto al final resulta en un acoplamiento innecesario entre los clientes.

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 si usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en la clase en cuestión.

Esto se explica más abajo con ejemplos.

Debemos intentar evitar este tipo de acoplamiento cuando sea posible, y esto se consigue separando las interfaces en otras más pequeñas y específicas.

Ejemplo:

Hay casos de proyectos que se han vuelto inmantenibles debido a la violación de este principio al crearse una clase en la que se va metiendo casi toda la funcionalidad en lugar de ir extrayéndola en diferentes clases. En ocasiones esta clase suele ser un singleton que está accesible siempre para el resto del programa.

Esto va provocando que casi todos los cambios y bugs se encuentren siempre en la misma clase, y normalmente arreglar o modificar algo en ella conlleva consecuencias inesperadas en el resto de esta.

Por si fuera poco, normalmente este tipo de proyectos tienen un testing con muy poca cobertura o ni siquiera cuentan con él, por lo que encontrar los fallos provocados por arreglar otra parte de nuestra clase se vuelve muy costoso.

Vamos a ver un ejemplo de violación de este principio.

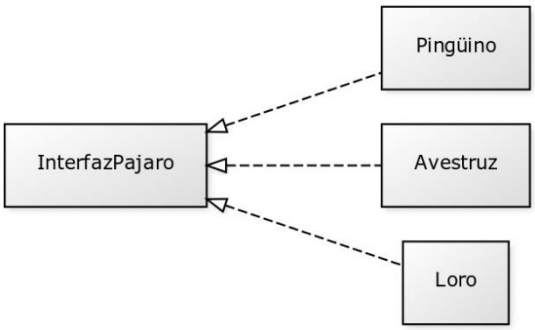
Estamos implementando un zoo, y queremos crear una interfaz que sirva para todas las aves.

Pensamos en loros, flamencos, gaviotas, aves rapaces y gorriones, por lo que implementamos los métodos de comer y volar.

Posteriormente el zoo consigue presupuesto extra y compra una pareja de avestruces, por lo que definimos también el método correr.

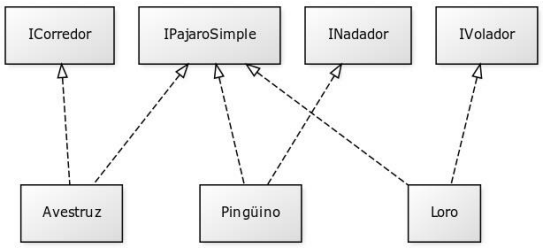
No nos podemos olvidar tampoco de los pingüinos, necesitamos un método para nadar.

Como no hemos ido refactorizando entre estos pasos, ahora nuestro sistema tiene esta pinta.



El problema viene con que, por ejemplo, el avestruz tiene que implementar métodos que no usa, y con la llegada del pingüino tuvo que cambiar innecesariamente para implementar el método de nadar.

Una forma correcta de haber modelizado el problema sería haber dividido la interfaz en otras más pequeñas de esta manera.



De esta manera cada pájaro concreto solo tiene lo que realmente necesita y se pueden añadir nuevas clases sin modificar otras zonas que no estén afectadas.

DEPENDENCY INVERSION PRINCIPLE

Estas premisas definen el principio.

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

B: Las abstracciones no deberían depender de los detalles. Son los detalles los que deberían depender de abstracciones.

En la orientación a objetos, lo normal es tener una jerarquía de objetos que se unen porque los de más alto nivel suelen incluir una instancia de los de más bajo nivel.

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.

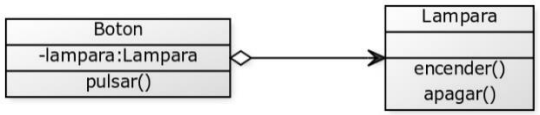
Ejemplo:

En esta ocasión voy a utilizar el ejemplo original del artículo de Robert C. Martin en el que se expuso este principio.

Consideremos un objeto botón y un objeto lámpara. El objeto botón reacciona a un estímulo cambiando su estado entre encendido y apagado. Puede ser un botón físico, uno de una interfaz gráfica en nuestro teléfono, o incluso algún tipo de sensor.

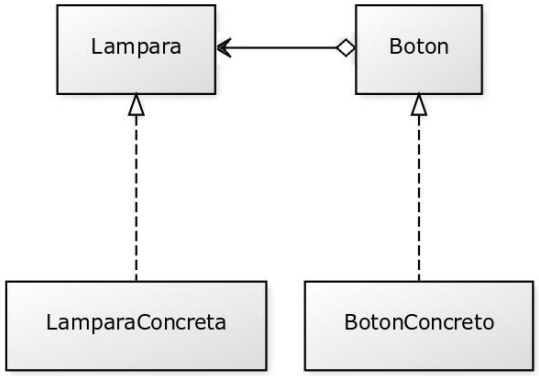
La lámpara si recibe la orden de encenderse o de apagarse actuará en consecuencia, no importa qué tipo de lámpara sea.

Una modelización clásica de este sistema sería una clase botón que contiene una instancia de la clase lámpara, a la que cambia su estado entre encendida y apagada.



El problema de este modelo es que el botón está demasiado acoplado a la lámpara innecesariamente, a este botón le costaría encender y apagar un motor, por ejemplo, porque ya tiene la lámpara dentro de su estructura.

Vamos a aplicar la inversión de dependencias a este sistema a ver qué conseguimos.

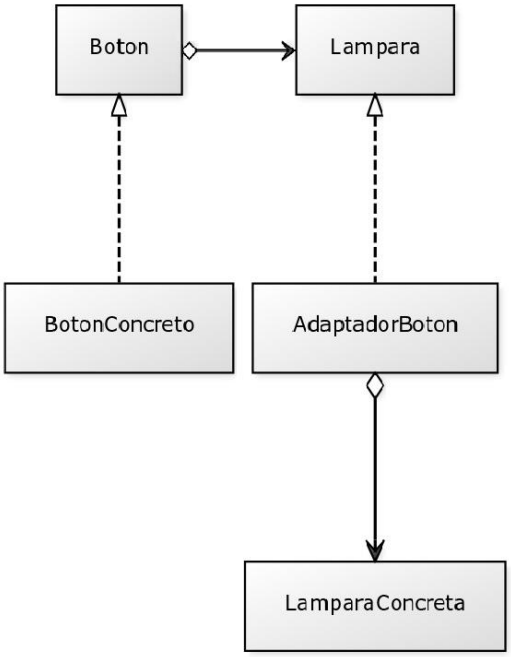


Ahora ya no se conocen entre sí el botón y la lámpara, pero pueden funcionar juntos. De hecho, podríamos cambiar fácilmente el objeto que interactúa con el botón para que fuera un motor de un coche, por ejemplo.

La interfaz botón no tiene ni que conocer cómo funciona el mecanismo del botón concreto ni saber nada sobre la lámpara.

Algunos estaréis pensando que seguramente nuestro botón esté pensado para utilizarse sólo en lámparas, por genéricas que estas sean.

Pero esto se puede solucionar con la ayuda de un patrón adaptador que adapte la información que recibe el objeto con la que espera recibir. Es una manera fácil de trabajar con software de terceros que no podemos cambiar, por ejemplo.



PRINCIPIO SOC (SEPARATION OF CONCERNS)

Los asuntos, son los diferentes aspectos de la funcionalidad de nuestra aplicación. Por ejemplo, la capa de negocio, la capa de datos etc. Un buen ejemplo de separación es el patrón MVC.

Ejemplo:

- Separación de una aplicación informática en front-end y back-end, o en las capas de capa de presentación y de acceso a datos.

- Patrones de arquitectura tales como modelo-vista-controlador, que separan una aplicación en varias partes interconectadas pero independientes.

EL PRINCIPIO YAGNI

Muchas veces, para evitar problemas posteriores, los programadores tendemos a desarrollar funcionalidades que no estamos seguros de necesitar. Simplemente lo hacemos "por si acaso". El principio YAGNI (You ain't gonna need it), viene a decir que no debemos implementar algo si no estamos seguros de necesitarlo. Así ahorramos tiempo y esfuerzo.

LA REGLA DEL BOY SCOUT

Como programadores, debemos hacer como los Boy Scout, que dejan el campo más limpio que cuándo llegaron. El código siempre puede mejorar. Si podemos, debemos refactorizar el código para dejarlo todavía más limpio y simple que antes.

LA LEY DE DEMETER

Según este principio, una unidad solo debe tener conocimiento limitado de otras unidades, y solo conocer aquellas que están relacionadas. La una unidad solo debe hablar con amigos y nunca con extraños. Además, la unidad solo debe hablar con amigos inmediatos.

Simplificando mucho, tenemos que tratar de evitar utilizar métodos de un objeto que ha sido devuelto por otro método. En este caso es útil seguir la regla de nunca usar más de un punto cuando accedemos a métodos de un objeto. Por ejemplo no usar clienteActual.ObtenerDireccion.calle.CambiarNombreDeCalle. Recuerda, no hay que hablar con extraños.

EL PRINCIPIO DE HOLLYWOOD

Basado en la típica respuesta que se les da a los actores que hacen una prueba para una película: "No nos llame, nosotros le llamaremos". Este principio está relacionado con el principio de inversión de dependencias de SOLID. Un ejemplo del principio de Hollywood es la inversión de control (IoC), que hace que una clase obtenga las referencias a objetos que necesita para funcionar, a través de una entidad externa.

DISEÑO POR CONTRATO

A grandes rasgos podríamos decir que el Diseño por Contrato consiste en definir de manera formal el contrato expuesto por cada parte de un sistema. Y digo «parte» porque realmente el diseño por contrato lo podemos aplicar a distintos niveles: clases, módulos, métodos, etc.

El «contrato» marca varias cosas, incluyendo los tipos de datos sobre los que se trabaja, el tipo de resultado que se devuelve, las condiciones que deben cumplirse antes de poder ejecutar una operación (precondiciones), las condiciones que se cumplirán después (postcondiciones) y las condiciones que se cumplirán tanto antes como después (invariantes).

Ejemplo:

Si tenemos un método que elimina un elemento de una lista que no contiene repeticiones, las precondiciones serían que la lista no es null, que el elemento no es null y que la lista contiene el elemento, y la postcondición sería que después de ejecutar el método la lista ya no contiene el elemento.

Las condiciones (ya sean pre, post o invariantes) no tienen por qué limitarse a los parámetros de entrada y salida, sino que pueden referirse al estado de otras variables. Por ejemplo, si tenemos un método que confirma un pedido de una tienda online, podríamos comprobar que cuando se llama a ese método al menos se ha añadido una línea al pedido.

Normalmente suelo poner más enfásis en la validación de precondiciones e invariantes que en la validación de postcondiciones, principalmente porque muchas veces la postcondiciones de un método se convierte en la precondición del siguiente (el valor de retorno de un método lo usas como valor de entrada de otro), y al validar la precondición del siguiente método se suelen detectar posibles errores. Además, las postcondiciones son precisamente lo que se validan con los tests unitarios (a fin de cuentas, son el resultado de la ejecución), por lo que quedan cubiertas a través de ellos.

Independientemente del mecanismo que utilices para implementar Diseño por Contrato, es importante que tengas en cuenta que las excepciones que se lanzan por violaciones de contrato nunca deben ser capturadas. Una violación de contrato es un bug en la aplicación, no una circunstancia anómala de la que uno debe intentar recuperarse.

CONCLUSIONES

Como verán, logró hablar un poco sobre las distintas categorías y tipos de patrones de diseño, no debemos “reinventar la rueda” en varias de nuestras aplicaciones. Muchos ya trabajaron sobre este tema, y desarrollaron una solución satisfactoria. “¿Para qué voy a inventar un ladrillo si ya otro lo hizo y el mismo ya fue usado en la edificación de millones de estructuras con éxito?”

Nuestro compromiso es el de comprender la teoría (patrones creacionales y patrones de diseño), y aplicarlo de manera efectiva en nuestros proyectos ya sea web, o aplicaciones de software, entre otros campos que respecta a la ingeniería del software.

REFERENCIAS

[1] https://www.adictosaltrabajo.com/2014/10/28/solid-5/

[2] http://java.ciberaula.com/articulo/diseno_patrones_j2ee/

[3] https://www.genbeta.com/desarrollo/doce-principios-de-diseno-que-todo-desarrollador-deberia-conocer

[4] https://enmilocalfunciona.io/principios-solid/

[5] /https://www.adictosaltrabajo.com/2014/10/23/solid-2/

[6] https://www.adictosaltrabajo.com/2014/10/24/solid-3

[7] https://www.adictosaltrabajo.com/2014/10/22/solid-1/

[8] http://blog.koalite.com/2014/01/diseno-por-contrato/