Principios de diseño

Introducción

Principios:

- Programar orientado a interfaces y no a implementaciones
- Preferencia de la composición sobre la herencia
- Bajo Acoplamiento (Low coupling)
- Alta cohesión (High cohesion)
- Responsabilidad única (SRP, Single-responsability principle)
- Abierto-Cerrado (OCP, Open-closed principle)
- Ley de Demeter (No hables con extraños)
- No repetición (DRY, Don’t repeat yourself)
- Sustitución de Liskov (LSP, Liskov substitution principle)
- Segregación de interfaces (ISP, Interface-segregation principle)
- Inversión de dependencias (DIP, Dependency inversion principle)

Introducción. ¿Qué son los principios de diseño del software?

Los principios de diseño de software son un conjunto de directrices cuyo objetivo es evitar el desarrollo de aplicaciones mal diseñadas. Los principios de diseño están asociadas a Robert Martin, quien los reunió en "Agile Software Development: Principles, Patterns, and Practices (Desarrollo de Software Ágil: principios, patrones y prácticas". De acuerdo con Robert Martin hay tres características importantes de un mal diseño que deben evitarse:

- **Rigidez**: Es difícil cambiar porque cada cambio afecta a muchas otras partes del sistema.
- **Fragilidad:** Cuando hacemos un cambio, dejan de funcionar correctamente partes inesperadas del sistema.
- **Inmovilidad:** Un componente (clase, paquete, etc) es difícil reutilizar en otra aplicación, ya que no se puede desvincular de la aplicación actual.

Por otro lado, los patrones de diseño promueven la reutilización de código y la independencia de los objetos, dos de los objetivos del diseño de software mediante el

paradigma de la programación orientada a objetos (POO). Sólo porque programemos utilizando herencia, encapsulación y polimorfismo no queda garantizado que vayamos a tener un código de fácil mantenimiento y con un alto grado de reutilización.

Como resultado, los expertos en la materia detallan una serie de directrices y principios que habría que procurar seguir a la hora de afrontar nuestros diseños.

Programar orientado a interfaces y no a implementaciones

Nuestras clases tienen que colaborar con interfaces bien definidas y no con clases de implementación.

Colaborar con implementaciones crea objetos altamente acoplados. Por tanto, al diseñar una API deberíamos crear interfaces y/o clases abstractas que definan los métodos necesarios y a continuación crear clases que proporcionen la implementación de los métodos.

Por otra parte, las clases clientes declararán variables del tipo de la interfaz o de la clase abstracta, en lugar del tipo de alguna de las clases de implementación. Lo mismo es aplicable para los parámetros de los métodos de las clases cliente.

> *Pero ¿Cómo evitamos hacer mención explícita de una clase de implementación? Si utilizamos el operador new para crear objetos entonces no podremos poner en práctica lo anterior. Tenemos que acudir a otras soluciones: implementar alguno de los patrones creaciones de la familia Factory o utilizar algún framework de inyección de dependencias, que ya incluyen una factoría abstracta de objetos.*

Con esto se consigue que el código cliente no sepa realmente con que implementación está colaborando, lo que es muy beneficioso: nos permite cambiar fácilmente de implementación –si esto es necesario, posibilita que el diseñador del API realice cambios en la implementación sin que afecte a las clases cliente, etc.

Preferencia de la composición sobre la herencia

Cuando sea posible, debemos dar prioridad a la composición antes que a la herencia.

La herencia tiene su aplicación. Es una técnica muy útil para promover la reutilización de código, que nos ahorra mucho trabajo al permitir que las subclases puedan utilizar los atributos y métodos definidos en la clase base. Además, mediante la herencia podemos hacer uso del polimorfismo.

No obstante, cuando utilizamos la herencia tenemos que asegurarnos que la interfaz definida en la clase base permanece estable, pues en caso contrario podemos romper la cadena de herencia. Por ejemplo, supongamos que definimos un método en la clase base que retorna un int. Al cabo de un tiempo, cambiamos este método (en la misma clase base) para devolver un long. Esta sencilla modificación altera la interfaz de la clase base y provoca que dejen de funcionar todas las clases clientes que confiaban en esta interfaz.

A causa de la dificultad para modificar la interfaz de la clase base, los expertos en la materia sugieren que usemos la técnica de la composición. Este concepto requiere que para crear nueva funcionalidad, en lugar de utilizar la herencia combinemos objetos existentes en otro objeto (objeto contenedor). Con esto, el efecto de alterar la interfaz de la clase base es mucho menor que en un modelo basado en la herencia.

La composición también promueve el encapsulamiento, ya que al ensamblar objetos para crear nueva funcionalidad cada uno tendrá un propósito bien definido, por lo que sólo nos tendremos que preocupar de sus interfaces y de los valores que retornen estos objetos, pero no de sus implementaciones.

De lo anterior surge una duda muy legítima: ¿Cuándo se debe usar herencia y cuando composición? La respuesta es que difícil decidir entre estas dos técnicas. Como regla general, debemos utilizar herencia cuando las clases, sin ningún género de dudas, definen una relación “es-un”. Por ejemplo: Manzana, Pera y Limón son Frutas; por tanto, aquí tenemos una jerarquía de herencia, siendo Fruta la clase base.

Ahora veamos una relación “es-un” dudosa. Modelando una competición deportiva tenemos que un Velocista “es-un” Corredor. Sin embargo, ¿qué ocurre cuando el Velocista participa en un decatlón? Ahora el Velocista pasa a ser, por ejemplo, un LanzadorDePeso, LanzadorDePertiga y/o un LanzadorDeJabalina. Es decir, el Velocista ha cambiado de rol. En este caso tendríamos que utilizar la técnica de la

composición para crear el tipo Atleta, compuesto de las diferentes clases que le permitan afrontar los eventos deportivos.

Bajo Acoplamiento (Low coupling)

Tenemos que minimizar el acoplamiento.

El acoplamiento mide la dependencia entre elementos (como clases, paquetes, etc), normalmente como un resultado de la colaboración entre los elementos para proporcionar un servicio.

Cuando una clase tiene un acoplamiento alto respecto a otras clases nos encontramos con lo siguiente:

- Un cambio en las clases relacionadas puede provocar un cambio en nuestra clase.
- Es difícil entender nuestra clase de manera aislada.
- Es difícil reutilizar nuestra clase, ya que su uso requiere la presencia de las clases relacionadas.

Alta cohesión (High cohesion)

Hemos de maximizar la cohesión.

La cohesión mide el grado de relación entre las diferentes responsabilidades de determinados elementos (una clase, las clases de un paquete, etc). Centrándonos en una clase, cuanto más relacionadas estén entre ellas las responsabilidades de la clase, más alta será la cohesión de la clase.

Una clase con cohesión baja tiene los siguientes problemas:

- Es difícil de entender.
- Es difícil de reutilizar.
- Es difícil de mantener.
- Es frágil respecto a los cambios que se produzcan en el sistema.

Una versión más restrictiva de este principio es el conocido como principio de responsabilidad única (SRP, Single-responsability priciple): “una clase sólo habría de tener un motivo para cambiar”.

Responsabilidad única (SRP, Single-responsability principle)

Una clase debe tener sólo una razón para cambiar.

Este principio afirma que si tenemos dos motivos para cambiar una clase, tendremos que dividir la funcionalidad en dos clases, encargándose cada clase de una sola responsabilidad. De esta manera, si en el futuro tenemos que modificar una responsabilidad sólo intervendremos sobre una clase. Cuando tenemos que hacer una modificación en una clase que tiene más de una responsabilidad, el cambio podría afectar a la otra funcionalidad de la clase.

El principio de responsabilidad única fue introducido por Tom DeMarco en su libro "Structured Analysis and Systems Specification (Análisis Estructurado y especificación del sistema)", en 1979. Robert Martin reinterpretó el concepto y definió la responsabilidad como una razón para la modificación.

Abierto-Cerrado (OCP, Open-closed principle)

Las entidades de software como clases, módulos y funciones deben estar abiertas para la extensión, pero cerradas para la modificación.

OCP es un principio general que hemos de tenerlo en mente para asegurarnos de que cuando necesitemos ampliar el comportamiento de una clase no la cambiemos directamente, sino que la extendamos. El mismo principio se puede aplicar para módulos, paquetes y bibliotecas. Si tenemos un conjunto de clases formando una librería, tendremos muchas razones para extender sin cambiar el código existente: compatibilidad hacia atrás, pruebas de regresión, etc. Es por esto que tenemos que asegurarnos de que nuestros módulos siguen el principio Abierto/Cerrado.

Centrándonos en las clases, el principio OCP se puede garantizar mediante el uso de clases abstractas y clases concretas que implementen el comportamiento definido en las abstractas. Así, en lugar de modificar clases existentes para obtener nueva funcionalidad, esta práctica fuerza a que las clases concretas tengan que implementar los nuevos comportamientos, quedando la funcionalidad previa intacta, luego asegurada. Algunos casos particulares de este principio son el patrón de diseño Template Method y el patrón Strategy.

Ley de Demeter (Law of Demeter)

La ley de Demeter, a veces conocida como “No hables con desconocidos”, es una heurística que nos ayuda a cumplir el principio abierto/cerrado.

Un método **m** de un objeto **o** sólo tendría que utilizar:

- Métodos del propio objeto **o**.
- Objetos que sean atributos de **o**.
- Objetos que reciba por parámetro el propio método **m**.
- Objetos que instancie el propio método **m**.

Si para el funcionamiento del sistema fuera necesario invocar un método más lejano, no se hará directamente, sino que cada método invocará a otro de su entorno cercano hasta llegar al que debe invocarse en último extremo.

Aunque a priori esto pudiera parecer poco eficiente, en realidad no lo es. Al seguir rígidamente esta ley, conseguimos un acoplamiento bastante menor, lo cual siempre redunda en una simplicidad de los métodos y una mejor adaptabilidad a nuevas situaciones.

No repetición (DRY, Don’t repeat yourself)

Cada pieza de conocimiento ha de tener una representación única en el sistema.

Este principio nos indica que hemos de evitar, siempre que podamos, la duplicación de la información y de responsabilidades. Si lo conseguimos, nuestro sistema será más sencillo y más fácil de mantener, ya que ante un cambio o error podremos identificar fácilmente el componente afectado.

Sustitución de Liskov (LSP, Liskov substitution principle)

Los tipos (clases) derivados deben ser completamente sustituibles por sus tipos base.

Este principio es una extensión del principio Abierto/Cerrado, en el sentido de que debemos asegurarnos de que las clases derivadas extienden las clases base sin alterar su comportamiento. Con esta premisa conseguimos que las clases derivadas

puedan sustituir a las clases base sin que se tenga que hacer ningún cambio en el código. Posibles puntos de sustitución son:

- El tipo de una referencia y el tipo del objeto instanciado. La referencia puede ser de un tipo y el objeto de un subtipo. Por ejemplo:

  ```
  Animal a = new Perro()
  ```

- El envío de argumentos a métodos, donde el argumento puede ser de un subtipo y el parámetro de un tipo. Por ejemplo:

  Clase A

  ```
  Perro p=new Perro();
  x = funcion(perro)
  ```

  Clase B

  ```
  void funcion(Animal a)
  ```

El principio de sustitución de Liskov fue presentado por Barbara Liskov en 1987 en su artículo "Data abstraction and hierarchy (Abstracción y jerarquías de datos)" , presentado en una conferencia en 1987 sobre lenguajes de programación y aplicaciones de sistemas orientados a objetos.

Segregación de interfaces (ISP, Interface-segregation principle)

Las clases no deberían depender de interfaces que no utilizan.

Este principio nos dice que a la hora de diseñar una interfaz tenemos que añadir sólo los métodos necesarios, ya que en caso contrario obligaremos a las clases que deban implementar la interfaz a tener que implementar métodos que no necesitan en absoluto. Por ejemplo, si creamos una interfaz llamada Trabajador y añadimos un método llamado almorzar(), todas las clases que implementen Trabajador tendrán que implementar el método almorzar(). ¿Qué pasa si una de estas clases es un robot?

Así, las interfaces que contienen métodos como el descrito en el ejemplo se conocen como interfaces contaminadas o grasientas. La cuestión es que debemos evitar este tipo de situaciones.

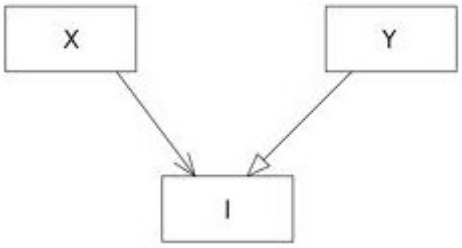
Inversión de dependencias (DIP, Dependency inversion principle)

También conocido como Principio de inversión de control (IoC) o inyección de dependencias (DI).

Este principio incide en que debemos desacoplar los módulos de alto nivel (un objeto de servicios, por ejemplo) de los módulos de bajo nivel (un objeto de acceso a datos, por ejemplo). Para ello, introducimos una capa de abstracción entre los dos tipos de módulos.

La inversión de dependencias trata sobre cómo se adquieren las dependencias entre objetos. En la forma clásica, cuando una clase necesita de otra para cumplir su cometido, inicializa una instancia de esa clase (posiblemente mediante una operación new) y retiene una referencia a tal objeto. Esto provoca que la clase que requiere la dependencia quede fuertemente acoplada a la otra clase. La siguiente figura muestra que la clase X necesita de la clase Y:

El principio de inversión de dependencias, con el objetivo de desvincular el acoplamiento entre clases (tipos), propone que exista un módulo externo -una capa de abstracción- que se encargue de gestionar las dependencias requeridas, instanciándolas e inyectándolas en un punto de enganche (un atributo, una variable local o un parámetro) que deberá proporcionar cada clase que requiera dependencias. La siguiente figura muestra que existe un componente llamado I, capaz de crear instancias tanto de la clase X como de la clase Y, así como de resolver cualquier dependencia que sea requerida. Hay que apreciar que ahora X no tiene conocimiento sobre cómo se obtiene una referencia de Y:



El gran beneficio viene cuando el tipo declarado en el punto de enganche es una interfaz (también si es una clase abstracta), ya que así tenemos la opción de intercambiar implementaciones de la interfaz que el módulo externo deberá inyectar.

Se pueden utilizar los patrones Factory y Abstract Factory para crear frameworks de inversión de dependencias. No obstante, podría ser un esfuerzo de trabajo muy considerable, en tanto que existen potentes y maduras soluciones open source (Spring, Google-guice, etc), conocidas como contenedores IoC.