Principios de POO

La Programación Orientada a Objetos se basa en un conjunto de conceptos y características fundamentales. Comencemos viendo los conceptos:

- **Clase**: Definiciones de las propiedades (atributos) y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
- **Herencia**: Es la facilidad mediante la cual la clase B hereda en ella cada uno de los atributos y operaciones de A, como si esos atributos y operaciones hubiesen sido definidos por la misma B. Por lo tanto, puede usar los mismos métodos y variables no privadas declaradas en A. Los componentes registrados como "privados" (private) también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos.
- **Objeto**: Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos) los mismos que consecuentemente reaccionan a eventos. Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa). Es una instancia a una clase.
- **Método**: Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- **Evento**: Es un suceso en el sistema (tal como una interacción del usuario con la máquina, un mensaje enviado por un objeto o sencillamente el paso del tiempo). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento la reacción que puede desencadenar un objeto, es decir la acción que genera.
- **Mensaje**: Comunicación dirigida a un objeto que le ordena que ejecute uno de sus métodos, normalmente el mensaje contiene ciertos parámetros asociados al evento que lo generó.
- **Propiedad o atributo**: Contenedor de un tipo de dato asociado a un objeto (o a una clase de objetos) que sirve para almacenar algún valor. Tal valor puede ser leído y modificado por la ejecución de algún método. El conjunto de atributos de un objeto, así como las relaciones que éste establece con otros objetos forman lo que se conoce como el 'estado del objeto'.

Las características son las siguientes:

- **Abstracción**: Proceso que toma como base un concepto del mundo real para obtener una entidad software que satisfaga las necesidades del problema a resolver. La abstracción es clave en el proceso de análisis y diseño orientado a objetos. Por ejemplo, mediante la abstracción, una empresa pueda identificar como una transacción clave en su negocio la entidad Factura y crear a partir de ella la clase software Factura, la cual contendrá sólo las características relevantes para el programa informático y no todas las propiedades del concepto empresarial Factura.
- **Encapsulación**: Reunir en una clase, y por tanto, en un mismo nivel de abstracción, aquellos elementos que pueden considerarse pertenecientes a una misma entidad. Encapsular correctamente permite aumentar la cohesión de los componentes del sistema. Por ejemplo, podríamos identificar los atributos que dieran lugar a una clase Vehiculo; sin embargo, supongamos que analizando un poco más, detectamos que es posible extraer ciertos atributos de Vehiculo para formar la clase Matricula. De esta manera, el resultado es que tenemos una mejor encapsulación que al principio. Por otro lado, no hay que confundir la encapsulación con la ocultación, ya que la encapsulación no implica ocultar los datos ni los métodos de ninguna manera.
- **Ocultación**: Cada objeto se mantiene aislado del exterior, aunque expone una interfaz para poder comunicarse con otros objetos. Este aislamiento protege las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de manera inesperada o inapropiada.
- **Polimorfismo**: Las referencias a objetos y las colecciones de objetos pueden contener objetos cuyo tipo sea diferente al definido inicialmente. El tipo de tal objeto será una subclase o una implementación del tipo definido inicialmente. La llamada de un método en una referencia invocará al método del objeto real apuntado por la referencia y no el de la clase definida por la referencia.
- **Herencia**: A veces interesa que las clases se relacionen formando una jerarquía de herencia. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y la encapsulación, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo.

Buenas prácticas de diseño

Hemos de tener en cuenta que los conceptos y características anteriores, a pesar de su importancia, no garantizan por si mismos que una aplicación orientada a objetos se diseñe adecuadamente. Es por ello que existen una serie de normas y buenas prácticas que se deben intentar seguir en el diseño de aplicaciones OO:

Normas de carácter general

- Encapsular lo que varía.
- Favorecer la composición frente a la herencia.
- Programar orientado a la interfaz y no a la implementación.
- Evitar el acoplamiento entre clases.
- Reducir las responsabilidades de cada clase.

Veamos estas normas con más detalle:

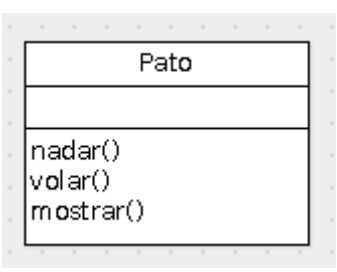
Encapsular lo que varía

El principio *Encapsular lo que varía*, se refiere a que siempre que se tiene un comportamiento en la aplicación que se considere que puede cambiar en un futuro, se debe extraer ese comportamiento de las partes de la aplicación que permanecen sin cambio.

¿Qué cosas pueden variar? Comportamiento, tecnología, configuraciones de entorno, etc.

¿Por qué es interesante separar lo que varía? Porque cuando se han encapsulado adecuadamente las partes variables la aplicación, ésta resulta más fácil de entender y de mantener, dado que la complejidad global se ha reducido. Por ejemplo, supongamos una aplicación en la que se necesitan distintas estrategias de facturación: por cantidad, fijo, escalonado, etc. Cada una de estas estrategias podría encapsularse en una clase y pasarse una instancia según convenga a una clase Factura.

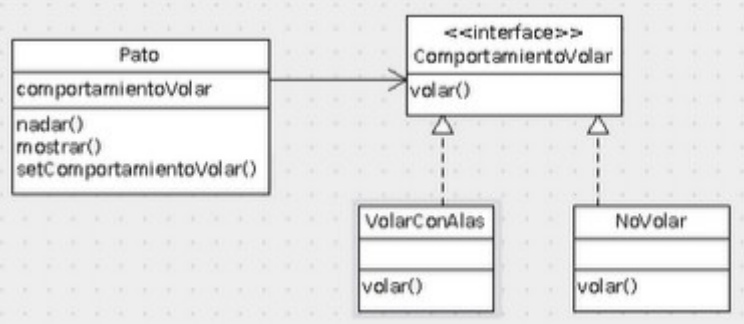
Veamos otros ejemplo:



Aquí tenemos una simple clase llamada Pato que tiene 3 métodos; nadar(), volar() y mostrar(). Los métodos nadar() y mostrar() permanecen constantes para todos los objetos Pato, pero el método

volar() no es constante ya que algunos objetos Pato no vuelan.

Por tanto, aquí tenemos una buena oportunidad para aplicar el principio de *encapsular lo que varía*. Podríamos hacer lo siguiente:



Como vemos en la figura anterior, se ha extraído de la clase Pato lo que varía, esto es, el comportamiento volar(). Si mañana necesitamos considerar patos que vuelan mediante prótesis electromecánicas, tan sólo tenemos que crear una nueva clase VolarProtesis que implemente la interfaz ComportamientoVolar. De esta manera, no se tiene que modificar ninguna de las clases existentes y, por tanto, se evita exponer la aplicación a posibles errores en la modificación.

Favorecer la composición frente a la herencia

La Herencia es una característica muy importante en la orientación a objetos, no obstante presenta ciertos inconvenientes frente a otras técnicas, como la Composición.

Inconvenientes de la Herencia

- Un cambio en una clase (cambio local) puede tener un impacto en otras clases de la jerarquía (cambio no local) .
- Facilita la reutilización de código pero es difícil de mantener, ya que hay cosas que no se pueden deducir a simple vista.
- La jerarquía de herencia queda definida estáticamente en tiempo de compilación, por lo que no se pueden hacer cambios dinámicamente, en tiempo de ejecución. Por ejemplo, mediante la Composición, en tiempo de ejecución, podemos pasar como parámetro a un método, un objeto cuya clase defina los estilos aplicables a un documento. Esto ofrece mucha flexibilidad y no es posible hacerlo mediante la Herencia.
- La relación "es-un" denota una jerarquía de Herencia. Por ejemplo, si tenemos la clase Alumno y la clase AlumnoJava, podemos crear una relación de herencia entre ambas clases. Sin embargo, esta relación es rígida y poco flexible, con un comportamiento difícil de cambiar en tiempo de ejecución. Así, un método que espere un Alumno como parámetro, o bien recibirá un Alumno o bien recibirá un AlumnoJava, pero no aceptará ningún otro tipo de objeto.

Programar orientado a la interfaz y no a la implementación

La idea es usar un contrato (la interfaz pública de una clase) sin conocer su implementación. Cuanto menos detalles de implementación conozca una clase cliente sobre una clase servidora, mucho mejor. A ser posible, la clase cliente no debería conocer nada sobre la implementación de una clase servidora. Esto evitará que la clase cliente deje de funcionar ante un cambio interno en la clase servidora.

Java no facilita este principio, ya que para crear objetos utilizamos el operador new() o métodos de factoría estáticos, que acaban utilizando también el operador new(). Sin embargo, utilizando algún contenedor IoC (Inversión de Control, inyección de dependencias) de terceros podemos programar obviando por completo las implementaciones (más sobre esto en breve).

Evitar el acoplamiento entre clases

Cuando una clase cliente conoce detalles de la implementación de una clase servidora, decimos que las dos clases están fuertemente acopladas. El acoplamiento muchas veces implica que al cambiar la implementación de una clase, las clases cliente fuertemente acopladas con ella dejan de funcionar. Esta coyuntura finalmente desemboca en una serie de modificaciones en cascada a lo largo y ancho del código de la aplicación. Por tanto, lo que hay que evitar, ante todo, es que una clase dependa de los detalles de implementación de otra para que pueda utilizarla. Es primordial la independencia de la implementación concreta entre clases.

Reducir las responsabilidades de cada clase

Una clase debe modelar sólo un concepto concreto. Por ejemplo, supongamos que tenemos una clase que genera un informe y permite imprimirlo. Ante un clase como esta, si necesitamos modificar algún aspecto relativo a la impresión podemos introducir errores en la generación del informe. Por otro lado, si tenemos que modificar algo relativo al informe podemos introducir errores en lo que atañe a la impresión. Esto sucede porque la clase lleva a cabo más de una responsabilidad. Sin embargo, la introducción de errores no es el único motivo por el que la implementación de varias responsabilidades es algo que debemos evitar. Aún más importante que esto: si una clase realiza varias cosas estamos ante un código que es difícil de entender, pues aborda cosas diferentes.

Los principios SOLID son otro grupo de principios a tener en cuenta en el diseño de software. Estos principios establecen:

- **Single responsability**: Una clase debe tener una única responsabilidad que justifique su existencia.
- **Open close principle**: La definición de una clase debe ser abierta para su extensión pero

cerrada para su modificación.

- **Liskov substitution**: Siempre debe ser posible sustituir una clase padre por otra hija sin que cambie el comportamiento de la aplicación.
- **Interface segregation**: Si una interfaz se ha vuelto demasiado compleja, es conveniente dividirla en varias interfaces más específicas. La idea es que una clase cliente nunca debería verse obligada a implementar métodos que no necesite.
- **Dependency inversion**: Las clases no deben crear instancias de otras clases con las que colaboren. La dependencia de una clase con respecto de otra debe inyectarse desde fuera de la clase, mediante un contenedor manejador de dependencias (como sería el caso de Spring Framework). Esto reduce enormemente el código de configuración de nuestra aplicación, facilitando los cambios en el código (refactorización). El código de configuración se pueda externalizar en ficheros XML o definir mediante metadatos (anotaciones Java) en el mismo código Java.

Estos principios generales pueden concretarse, a veces, es soluciones bien conocidas a problemas recurrentes en el diseño del software. Estas soluciones son lo que se conoce como Patrones de diseño.

Patrones de diseño

Es usual que durante el desarrollo de un programa nos encontremos de modo recurrente con el mismo tipo de problemas. Por ejemplo cómo garantizar la existencia de una única instancia para poder acceder a un determinado dispositivo. Otro ejemplo sería cómo estructurar una aplicación basada en un interfaz gráfico para que permita múltiples representaciones de los mismos datos.

Los patrones de diseño son soluciones bien conocidas y ampliamente utilizadas para resolver problemas recurrentes en el diseño de programas. Cada uno de los patrones de diseño tiene un nombre estandarizado, lo que define un vocabulario común que facilita el intercambio de ideas, y una plantilla de aplicación que muestra cuales son sus componentes y cómo se relacionan entre si.

Los patrones de diseño se agrupan por su cometido, así nos encontramos con patrones de diseño de creación (Singleton, Factory method, Abstract factory), de comportamiento (Strategy, Observer) y estructurales (Decorator) entre los más conocidos.