Teoría de la Programación Orientada a Objetos

Conceptos relacionados con la Programación Orientada a Objetos (Siglas POO o OOP en inglés) que nos sirvan para entender este paradigma de la programación. Estos artículos tratan la orientación a objetos desde un punto teórico, sin querer entrar en ningún lenguaje de programación particular, de ese modo todos los conocimientos explicados en este manual te servirán para aplicarlos a la práctica de la POO en cualquier lenguaje que puedas llegar a usar.

Autores del manual

Este texto "Teoría de la Programación Orientada a Objetos" es obra de los siguientes autores.

Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.

http://www.desarrolloweb.com
8 artículos publicados

Luis Fernández Muñoz

Profesor de la Escuela Superior de Informática de la UPM

2 artículos publicados

Ana Alvarez Sanchez

Directora de contenidos de EstiloyModa.com

http://www.estiloymoda.com
1 artículo publicado

Parte 1:

Bases de la Programación Orientada a Objetos

En los primeros artículos de este manual conocerás las bases esenciales del paradigma orientado a objetos de la programación. Son conceptos fundamentales que te ayudarán a entender clases, objetos, métodos, mensajes, estados y propiedades, para comenzar a pensar en objetos.

1.1.- POO: Fundamentos de la Programación Orientada a Objetos

Conceptos fundamentales de la Programación Orientada a Objetos, desde un punto de vista académico, que nos hacen entender la importancia de este paradigma de la programación.

En este artículo te ofrecemos un vídeo excelente para comenzar con la Programación Orientada a Objetos, desde un punto de vista muy académico, que te ofrecerá una serie de conocimientos importantes y necesarios para orientar tu formación y saber el por qué de la importancia de este paradigma de la programación.

En el texto que leerás a continuación encontrarás un resumen de la clase, emitida en directo y al final encontrarás el vídeo de la grabación de esta sesión. Te la ofrecimos de manera gratuita en DesarrolloWeb.com, el pasado 6 de mayo, junto al profesor de la UCM Luis Fernández. En esta pequeña charla introductoria podrás conocer a Luis y entender por qué es el mejor profesor para conocer los detalles de la Programación Orientada a Objetos. Durante ésta y las próximas dos semanas estará además con nosotros en el Curso de Programación Orientada a Objetos que estamos impartiendo actualmente en EscuelaIT.

Luis nos habló entre otras cosas de la definición de software, de la denominada Crisis del Software, y de las bases de la Programación Orientada a Objetos. Por aquí te dejamos un resumen de la charla.

Qué es el software

Las personas que nos dedicamos a la programación no deberíamos tener dudas a la hora de definir un concepto tan importante como el "software". La mayoría de nosotros, si tratamos de definirlo, lo haremos de mala manera. Una buena definición de lo que es el software sería “La información, de todo tipo, que das al ordenador para manipular la entrada de datos de un usuario”. **Es decir, el software es información**.

Cuando estamos escribiendo un código estamos dando una información al ordenador. Si la manera en la que ponemos la información no es la adecuada, nos encontraremos con un código desordenado, que hará que el trabajo sea poco eficiente a corto y largo plazo. El código tiene que estar bien escrito, para que a la hora de realizar algún cambio no tengamos que perder nuestro tiempo... y alterar nuestros nervios.

La Crisis del Software

El grupo Standish realizó un estudio en el año 1994 en el que se analizaban 50.000 proyectos. De esos 50.000 proyectos sólo el 16% fueron exitosos, y de ese 16% tan sólo el 61% cumplió toda la funcionalidad prometida. Los proyectos que fueron problemáticos sufrieron retrasos, costaron más de lo previsto y algunos fueron incluso cancelados al cabo de un tiempo.

Esta situación se detectó en los años 70, y fue denominada la Crisis del Software. Actualmente seguimos en ella.

De este estudio podemos llegar a la conclusión de que el software es información voluminosa y compleja que no podemos tratar de manera desordenada, pues esto marcará la diferencia entre que un proyecto salga adelante o fracase.

La historia del software

Al principio se empezaron a hacer lenguajes de alto nivel, orientados a procesos. Esa forma de trabajar daba más importancia a los procesos que a los datos.

Entonces muchas personas se dieron cuenta de que los datos merecían mayor atención y así aparecieron los lenguajes orientados a datos. Estos lenguajes tenían mucho cuidado con los datos, pero los procesos eran muy sencillos o casi inexistentes. El problema era que cuando había que cambiar la forma de los datos era necesario revisar todas las estructuras anteriores.

De la síntesis entre la programación orientada a procesos y de la programación orientada a datos nace la Programación Orientada a Objetos, un equilibrio entre las dos anteriores, que cuida tanto datos como procesos y evita los problemas de acoplamiento de los otros paradigmas.

Bases de la Programación Orientada a Objetos

Existen cuatro conceptos fundamentales dentro de la Programación Orientada a Objetos que se relacionan entre sí y que

nos permitirán tener las riendas de nuestro código:

Abstracción: proceso mental de extracción de las características esenciales de algo, ignorando los detalles superfluos.

Encapsulación: proceso por el que se ocultan los detalles del soporte de las características esenciales de una abstracción.

Modularización: proceso de descomposición de un sistema en un conjunto de módulos o piezas independientes y cohesivos (con significado propio). Lo adecuado es conseguir los mínimos acoplamientos.

Jerarquización: proceso de estructuración por el que se produce una organización (jerarquía) de un conjunto de elementos en grados o niveles de responsabilidad, incumbencia o composición entre otros.

Si entiendes esos cuatro conceptos entenderás el por qué de la programación orientada a objetos y, consecuentemente, serás un mejor programador. Obviamente, en el vídeo Luis los explica mucho mejor, con ejemplos interesantes para ser capaces de entenderlo perfectamente. Además lo hace de una manera muy divertida, como comprobarás y nos anima a todos a tomarnos en serio nuestra profesión.

> **Nota:** Para ahondar en los 4 conceptos anteriores queremos recomendarte la lectura del artículo Beneficios de la Programación Orientada a Objetos.

A continuación el vídeo de la clase, que os recomendamos mucho que veáis, pues además de resultaros interesante podréis entender mucho mejor los conceptos de los que hemos apuntado en este texto.

Por Ana Alvarez Sanchez

1.2.- Beneficios de la Programación Orientada a Objetos

Cuáles son los beneficios que nos aporta la Programación Orientada a Objetos: Abstracción, Encapsulación, Modularidad, Jerarquización.

Hace poco hicimos un vídeo de una clase de Programación Orientada a Objetos, que nos ofrece las bases y fundamentos de este paradigma de la programación. La clase la impartió Luis Fernández y nos aclaró muchos detalles de los cuales queremos ahora hablar. Se trata de ahondar en torno de varios conceptos que son básicos para entender los beneficios de la Programación Orientada a Objetos.

Son varios conceptos que encontramos presentes en los lenguajes de programación más modernos, por lo que ya no solo se trata de "objetos" sino de programación en general. Nuevos lenguajes con nuevos paradigmas no hacen más que proporcionarnos herramientas para que podamos los programadores potenciar al máximo la presencia de los beneficios que vamos a explicar en este artículo. Son además la base de la evolución de los lenguajes, que han marcado la historia de la programación.

> **Nota:** Este es un artículo un tanto teórico sobre programación. Es un conocimiento básico para asentar los conocimientos de Programación Orientada a Objetos sobre unos cimientos más sólidos, pero si lo que estás buscando es una explicación directa sobre lo que son objetos, clases y esas cosas, puedes leer el artículo Qué es la Programación Orientada a Objetos.

Comenzamos enumerando los conceptos que vamos a tratar: **Abstracción, Encapsulación, Modularización y Jerarquía**.

Abstracción

Es un proceso mental por el que se ignoran las características de algo, quedándonos con lo que realmente nos importa. La clave para entenderlo es "proceso mental", así que nos tenemos que poner a pensar, extrayendo aquello que realmente nos importa e ignorando lo superfluo.

La abstracción es algo que hacemos constantemente los humanos en nuestro día a día. Si no lo realizáramos nuestro cerebro se colapsaría con toda la información que nos rodea. Por ejemplo, para ir a la universidad o al trabajo vas en autobús. ¿Cuál es la abstracción del autobús? Es aquello que nos interesa como pasajeros, el precio del billete, la línea que recorre, el horario de las salidas, etc. No nos importa si se mueve con gasolina, gas o electricidad. Tampoco nos importa su cilindrada, potencia, consumo, la empresa que lo fabricó o el año en que lo hizo, ni la edad del conductor, por poner varios ejemplos. Así pues nos estamos abstrayendo de muchos detalles y quedándonos con lo fundamental.

Sin embargo, la abstracción es un proceso subjetivo. El mecánico del autobús realizará otro tipo de abstracción quedándose con determinados detalles que por su profesión le interesan. Por ejemplo, a él sí le importará si es diesel o gasolina, el fabricante, año, tipo de motor, transmisión, suspensión, etc.

Una forma de manejar la complejidad del software es conseguir escribir código de tal manera que permita la abstracción. En ensamblador, la aplicación más grande que podían hacer era de 10.000 líneas de código. Luego, con los lenguajes de alto nivel llegaron a 100.000, 300.000, 600.000 y eso se consiguió gracias a la abstracción procedimental. Por tanto, en la programación tradicional, la incorporación de las funciones o procedimientos trajo consigo una abstracción que cambió el mundo. Algo tan simple como tener funciones que realizan procedimientos y a las que podemos invocar enviando parámetros, abstrayéndonos de su complejidad. El algoritmo de esa función puede estar hecho de mil maneras distintas, recursivo, iterativo, con un bucle for, while, etc. Pero todos esos detalles le importan absolutamente nada a quien va a llamar a esta función, sino que lo esencial para él es el nombre de la función, los parámetros y el tipo de retorno.

Encapsulación

Es el proceso por el cual se ocultan los detalles del soporte donde se almacenan las características de una abstracción. En este punto el detalle clave para entender está en la palabra "soporte". Cuando encapsulamos estamos guardando cómo se soporta algo, como se almacena, qué medio es, cuál es su nombre, etc. Lo vas a entender mejor con un ejemplo.

Es como cuando te dan un medicamente encapsulado, donde tú no ves lo que hay dentro. Solo puedes tocar la cápsula pero no ves si dentro hay polvo, un comprimido, el color que tiene, etc. Nadie quiere que manosees el medicamento y por ello te lo entregan encapsulado. Te dicen "tómate esto" pero no te permiten tocar ni ver lo que hay dentro. Probablemente porque la cápsula está preparada para disolverse en el lugar correcto y no interactuar con saliva de la boca, o quizás con los jugos intestinales. Te venden un ordenador y te lo dan encapsulado, para que no lo abras. De un

ordenador no tocas lo que hay dentro, solo tocas aquello que quieren que toques, como el teclado, ratón, botón de encendido y apagado, etc.

En la informática la encapsulación es algo también habitual. Piensa en una fecha. Primero tenemos una abstracción que sería "día / mes / año" (de todo lo que nos puede decir una fecha quedarnos con tres datos funamentales). Ahora, una fecha la puedo guardar de muchas maneras, con tres variables que mantienen esos valores, llamadas dia, mes y año. Aunque las variables podían estar en inglés, day, month, year. También podríamos usar un array de tres elementos donde tengo esos tres valores, o con el número de segundos transcurridos desde una fecha dada (conocido como "tiemstamp" así guardan la fecha tradicionalmente sistemas operativos como Linux). O quizás quieras guardar un simple string. Si un programador al escribir un programa te oculta deliberadamente el cómo ha almacenado esa fecha, entonces está encapsulando.

Además hay que añadir que, si un programador da a conocer cómo guardaba esa fecha (por ejemplo usó tal y tal variable) y deja que los demás toquen esas variables libremente, estará provocando serios problemas. Uno programa un tratamiento para la fecha y potencialmente infinitas personas lo pueden usar. Pero si todos acceden a las características fundamentales de la abstracción de la fecha (esas variables) y las manipulan se produce el problema: El día que el programador de la fecha se arrepiente y quiere empezar a tratar la fecha de otro modo, por ejemplo con el "timestamp" (número de segundos desde 1970), porque observa que hay una sensible mejoría en los procesos internos del programa al usar ese soporte, obligaría a todos los que usan fechas a cambiar su código. Es decir, si infinitas personas mencionan las variables con las que aquel programador guardaba la fecha (las características esenciales de la abstracción), todos los programadores que usaban la fecha van a tener que actualizar su código para acceder ahora a otra variable y tratar la fecha de una manera diferente, produciéndose un acoplamiento feroz que deriva en oleadas de mantenimiento y pérdidas de dinero en horas de trabajo.

El primer momento donde apareció encapsulación en el mundo de la programación fue con las variables locales de las funciones. Las variables locales son propias de una función y no pueden mencionarse desde fuera, luego están encapsuladas. Y ojo que no siempre los lenguajes permitían encapsulación, los ingenieros tardaron en darse cuenta que era beneficioso no tocar las variables que habían dentro de una función.

Para acabar hay que insistir en que **no se oculta la información**, aquel programador que programó la fecha te tiene que informar en todo momento sobre el día el mes y el año, si se lo pides, lo que no te deja saber ni tocar son las variables o cualquier otro soporte mediante el cual está almacenando la fecha.

Modularización

Es la descomposición de un sistema, creando una serie de piezas que colaboran entre si, poco acoplados y cohesivos. Modularidad es tomar un sistema y tener la capacidad de segmentarlo en diversas partes independientes, que tengan sentido.

Como sistema entendemos el concepto amplio que todos debemos de tener en la cabeza. Por ejemplo, un ser vivo (una planta, el hombre...), o parte de un ser vivo (el sistema nervioso, circulatorio, etc.), o cualquier tipo de producto u organismo (desde un coche, una bicicleta, a una universidad o empresa, el sistema solar, etc). En cualquiera de estos sistemas debemos de reconocer diversas partes y podremos observar que todas colaboran entre si para llevar a cabo un objetivo o conjunto de tareas.

Como acoplamiento entendemos los cambios en piezas y el grado de repercusión en otros elementos del sistema. Es decir, tengo una pieza del sistema y la cambio. Inmediatamente eso repercute a otras piezas del sistema, de modo que a ese grado de repercusión le denominamos acoplamiento. Poco acoplado quiere decir que los cambios en elementos del sistema repercuten poco en otros.

Por ejemplo, en una universidad, si le ocurre algo al rector no es motivo para que la universidad deje de funcionar. Probablemente los profesores no se estén enterando que el rector ha fallecido y sigan dando la clase sin ningún tipo de interrupción. Eso es porque el rector está poco acoplado con otras piezas de una universidad.

Los mejores sistemas son poco acoplados, porque me permiten hacer cambios en las piezas sin que éstos repercutan en

realizar cambios en otras partes del sistema. Pero podrá haber sistemas más acoplados y menos acoplados, porque acoplamiento siempre va a existir. Si tenemos una pieza que no tiene ningún acoplamiento con ninguna otra del sistema probablemente es que forma parte de otro sistema. Por ejemplo, la uña del dedo gordo del pie tiene cero acoplamiento con otras piezas del sistema respiratorio y eso es porque no forma parte de ese sistema. En resumen, si una pieza no colabora con nadie, simplemente se debe a que no forma parte de ese sistema y por tanto, hay que reducir los acoplamientos, no anularlos.

Cohesión en un sistema bien modularizado implica que sus módulos deben ser entendibles por si mismos. El término viene de coherencia y quiere decir que las partes de un sistema deben funcionar de una manera dada en todas las ocasiones y además ese funcionamiento debe entenderse bien, sin que para ello necesites explicar otra cantidad indeterminada de datos adicionales que quizás no forman parte de esa pieza.

En términos de programación, un módulo cohesivo se ve cuando consigues explicarlo atendiendo a su propio código. Si al explicarlo tienes que hablar de código que está disperso por otros lugares, otras estructuras guardadas en variables ajenas al módulo, entonces no es cohesivo. Si te dan un código de un módulo (una función, un procedimiento, clase...) y lo entendes sin recurrir a otras partes del proyecto, entonces es cohesivo.

La modularización va de la mano de la abstracción y la encapsulación, puesto que estos conceptos representan las herramientas necesarias para hacer módulos cohesivos y poco acoplados. **Encapsular reduce los acoplamentos**, porque los cambios en las cosas de dentro de un módulo no afectan al resto de los módulos del sistema.

La modularidad va de la mano de la abstracción. Si yo entiendo una pieza por si misma normalmente es que es una buena abstracción. Nos centramos en las características esenciales de algo, no es un conglomerado de cosas sin conexión o sin sentido. Por tanto, **la abstracción correcta permite entender las piezas por si mismas y por tanto disponer de módulos cohesivos.**

Jerarquización

Es la estructuración por niveles de los módulos o elementos que forman parte de un sistema. Es la forma de organizar los módulos, existiendo jerarquías de todo tipo y con diversos grados de dependencia, responsabilidad, incumbencia, composición, entre otros.

Jerarquías típicas podría ser algo como una factura, de la que dependen clientes, productos, total y otras informaciones. Podríamos tener jerarquías como los animales, en las que podemos clasificar vertebrados e invertebrados. Los vertebrados a su vez se dividen en mamíferos, aves, reptiles...

En programación jerarquías hay por todos lados. Algo tan simple como una expresión cualquiera "5 * 2 + 3 * 8" es una jerarquía, en la que tenemos el "+" arriba y luego abajo las expresiones "5 * 2" y "3 * 8", las cuales se pueden descomponer en otras jerarquías. Las funciones que llaman a otras y éstas que llaman a otras, forman en si una jerarquía. Los programadores usamos esa jerarquía de funciones para descomponer nuestro código y dominarlo, en lugar de nadar entre cientos de líneas de código que se hacen difíciles de manejar. En programación orientada a objetos, las jerarquías las forman las clases, las cuales unas se apoyan en otras para llevar a cabo las tareas, ya sea mediante herencia o composición, que veremos más adelante.

Conclusión

Los cuatro conceptos que hemos visto en este artículo son fundamentales y están relacionados. La abstracción y la encapsulación están relacionados porque la encapsulación oculta los detalles de una abstracción. Están relacionados con la modularidad, porque para ser poco acoplados tienen que encapsular y para ser cohesivos deben ser buenas abstracciones. La jerarquía está relacionada con los módulos ya que éstos deben estar organizarlos por niveles, con algún tipo de jerarquía, ya sea de composición, clasificación, o alguna otra.

Los cuatro conceptos son fundamentales y por ello volveremos muchas otras veces sobre ellos al explicar la programación orientada a objetos. Son esenciales para que tu código se entienda bien, se pueda cambiar, se pueda ampliar, se pueda adaptar a otra plataforma, para aumentar la reutilización.

Tenéis que agradecer el conocimiento a Luis Fernandez y la transcripción a Miguel Angel Alvarez. Recordar que tenéis el vídeo completo de la clase de los fundamentos de la programación orientada a objetos, con estas descripciones y muchas otras en nuestro canal de Youtube. Otros artículos sobre objetos los encuentras en el Manual de Programación Orientada a Objetos.

Por Luis Fernández Muñoz

1.3.- Qué es la programacion orientada a objetos

Introducimos para los más profanos las bases sobre las que se asienta la Programación Orientada a Objetos.

La programación Orientada a objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación.

Con la POO tenemos que aprender a pensar las cosas de una manera distinta, para escribir nuestros programas en términos de objetos, propiedades, métodos y otras cosas que veremos rápidamente para aclarar conceptos y dar una pequeña base que permita soltarnos un poco con este tipo de programación.

Motivación

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser utilizados por otras personas se creó la POO. Que es una serie de normas de realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, de manera que consigamos que el código se pueda reutilizar.

La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque podamos hacer los programas de formas distintas, no todas ellas son correctas, lo difícil no es programar orientado a objetos sino programar bien. Programar bien es importante porque así nos podemos aprovechar de todas las ventajas de la POO.

Cómo se piensa en objetos

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

Pues en un esquema POO el coche sería el objeto, las propiedades serían las características como el color o el modelo y los métodos serían las funcionalidades asociadas como ponerse en marcha o parar.

Por poner otro ejemplo vamos a ver cómo modelizaríamos en un esquema POO una fracción, es decir, esa estructura matemática que tiene un numerador y un denominador que divide al numerador, por ejemplo 3/2.

La fracción será el objeto y tendrá dos propiedades, el numerador y el denominador. Luego podría tener varios métodos como simplificarse, sumarse con otra fracción o número, restarse con otra fracción, etc.

Estos objetos se podrán utilizar en los programas, por ejemplo en un programa de matemáticas harás uso de objetos

fracción y en un programa que gestione un taller de coches utilizarás objetos coche. Los programas Orientados a objetos utilizan muchos objetos para realizar las acciones que se desean realizar y ellos mismos también son objetos. Es decir, el taller de coches será un objeto que utilizará objetos coche, herramienta, mecánico, recambios, etc.

Clases en POO

Las clases son declaraciones de objetos, también se podrían definir como abstracciones de objetos. Esto quiere decir que la definición de un objeto es la clase. Cuando programamos un objeto y definimos sus características y funcionalidades en realidad lo que estamos haciendo es programar una clase. En los ejemplos anteriores en realidad hablábamos de las clases coche o fracción porque sólo estuvimos definiendo, aunque por encima, sus formas.

Propiedades en clases
Las propiedades o atributos son las características de los objetos. Cuando definimos una propiedad normalmente especificamos su nombre y su tipo. Nos podemos hacer a la idea de que las propiedades son algo así como variables donde almacenamos datos relacionados con los objetos.

Métodos en las clases
Son las funcionalidades asociadas a los objetos. Cuando estamos programando las clases las llamamos métodos. Los métodos son como funciones que están asociadas a un objeto.

Objetos en POO

Los objetos son ejemplares de una clase cualquiera. Cuando creamos un ejemplar tenemos que especificar la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama instanciar (que viene de una mala traducción de la palabra instace que en inglés significa ejemplar). Por ejemplo, un objeto de la clase fracción es por ejemplo 3/5. El concepto o definición de fracción sería la clase, pero cuando ya estamos hablando de una fracción en concreto 4/7, 8/1000 o cualquier otra, la llamamos objeto.

Para crear un objeto se tiene que escribir una instrucción especial que puede ser distinta dependiendo el lenguaje de programación que se emplee, pero será algo parecido a esto.

```
miCoche = new Coche()
```

Con la palabra new especificamos que se tiene que crear una instancia de la clase que sigue a continuación. Dentro de los paréntesis podríamos colocar parámetros con los que inicializar el objeto de la clase coche.

Estados en objetos
Cuando tenemos un objeto sus propiedades toman valores. Por ejemplo, cuando tenemos un coche la propiedad color tomará un valor en concreto, como por ejemplo rojo o gris metalizado. El valor concreto de una propiedad de un objeto se llama estado.

Para acceder a un estado de un objeto para ver su valor o cambiarlo se utiliza el operador punto.

```
miCoche.color = rojo
```

El objeto es miCoche, luego colocamos el operador punto y por último el nombre e la propiedad a la que deseamos acceder. En este ejemplo estamos cambiando el valor del estado de la propiedad del objeto a rojo con una simple asignación.

Mensajes en objetos
Un mensaje en un objeto es la acción de efectuar una llamada a un método. Por ejemplo, cuando le decimos a un objeto coche que se ponga en marcha estamos pasándole el mensaje ponte en marcha.

Para mandar mensajes a los objetos utilizamos el operador punto, seguido del método que deseamos invocar.

miCoche.ponerseEnMarcha()

En este ejemplo pasamos el mensaje ponerseEnMarcha(). Hay que colocar paréntesis igual que cualquier llamada a una función, dentro irían los parámetros.

Otras cosas

Hay mucho todavía que conocer de la POO ya que sólo hemos hecho referencia a las cosas más básicas. También existen mecanismos como la herencia y el polimorfismo que son unas de las posibilidades más potentes de la POO.

La herencia sirve para crear objetos que incorporen propiedades y métodos de otros objetos. Así podremos construir unos objetos a partir de otros sin tener que reescribirlo todo. Puedes encontrar en DesarrolloWeb.com un artículo completo dedicado a la Herencia.

El polimorfismo sirve para que no tengamos que preocuparnos sobre lo que estamos trabajando, y abstraernos para definir un código que sea compatible con objetos de varios tipos. Puedes acceder a otro artículo para saber más sobre Polimorfismo.

Son conceptos avanzados que cuesta explicar en las líneas de ese informe. No hay que olvidar que existen libros enteros dedicados a la POO y aquí solo pretendemos dar un repaso a algunas cosas para que os suenen cuando tengáis que poneros delante de ellas en los lenguajes de programación que debe conocer un desarrollador del web. Sin embargo, si quieres saber más también puedes continuar leyendo en DesarrolloWeb.com, en el manual de la teoría de la Programación orientada a objetos.

Ejemplo concreto de programación orientada a objetos

Para conseguir un ejemplo concreto de lo que es la programación orientada a objetos, podemos entrar en el Manual de PHP 5. Realmente este manual explica las características de orientación a objetos de PHP 5 y ofrece ejemplos concretos de creación de clases con características como herencia, polimorfismo, etc.

Por Miguel Angel Alvarez

1.4.- Métodos y atributos static en Programación Orientada a Objetos

Definición y ejemplos de elementos o miembros static, los métodos y atributos de clase o estáticos en la programación orientada a objetos.

En el Manual de la Teoría de la programación orientada a objetos estamos revisando diversos conceptos y prácticas comúnmente usadas en este paradigma de la programación. Ahora le toca el turno a "static".

Seguramente lo has visto en alguna ocasión en algún código, pues se trata de una práctica bastante común. Con "static", un modificador que podemos aplicar en la definición de métodos y atributos de las clases, podemos definir estos elementos como pertenecientes a la clase, en lugar de pertenecientes a la instancia. A lo largo de este artículo queremos explicar bien qué es esto de los elementos estáticos "static" o también llamados elementos "de clase".

Qué es static

La definición formal de los elementos estáticos (o miembros de clase) nos dice que son aquellos que pertenecen a la clase, en lugar de pertenecer a un objeto en particular. Recuperando concetos básicos de orientación a objetos, sabemos que tenemos:

Clases: definiciones de elementos de un tipo homogéneo.
Objetos: concreción de un ejemplar de una clase.

En las clases defines que tal obteto tendrá tales atributos y tales métodos, sin embargo, para acceder a ellos o darles valores necesitas construir objetos de esa clase. Una casa tendrá un número de puertas para entrar, en la clase tendrás definida que una de las características de la casa es el número de puertas, pero solo concretarás ese número cuando construyas objetos de la clase casa. Un coche tiene un color, pero en la clase solo dices que existirá un color y hasta que no construyas coches no les asignarás un color en concreto. En la clase cuadrado definirás que el cálculo del área es el "lado elevado a dos", pero para calcular el área de un cuadrado necesitas tener un objeto de esa clase y pedirle que te devuelva su área.

Ese es el comportamiento normal de los miembros de clase. Sin embargo, los elementos estáticos o miembros de clase son un poco distintos. Son elementos que existen dentro de la propia clase y para acceder los cuales no necesitamos haber creado ningún objeto de esa clase. Osea, en vez de acceder a través de un objeto, accedemos a través del nombre de la clase.

Ejemplos de situaciones de la vida real donde tendríamos miembros estáticos

De momento así dicho queda un tanto abstracto. Pero antes de ponerse con ejemplos concretos de programación donde hablemos de la utilidad práctica de los miembros estáticos sea bueno tratar de explicar estos conceptos un con situaciones de la vida real.

Por ejemplo, pensemos en los autobuses de tu ciudad. No sé si es el caso, pero generalmente en España todos los autobuses metropolitanos tienen la misma tarifa. Yo podría definir como un atributo de la clase AutobúsMetropolitano su precio. En condiciones normales, para acceder al precio de un autobús necesitaría instanciar un objeto autobús y luego consultar su precio. ¿Es esto práctico? quizás solo quiero saber su precio para salir de casa con dinero suficiente para pagarlo, pero en el caso de un atributo normal necesariamente debería tener instanciado un autobús para preguntar su precio.

Pensemos en el número "Pi". Sabemos que necesitamos ese número para realizar cálculos con circunferencias. Podría tener la clase Circunferencia y definir como atributo el número Pi. Sin embargo, igual necesito ese número para otra cosa, como pasar ángulos de valores de grados a radianes. En ese caso, en condiciones normales sin atributos de clase, necesitaría instanciar cualquier círculo para luego preguntarle por el valor de "Pi". De nuevo, no parece muy práctico.

> **Nota:** Ojo, porque en el caso del número Pi, su valor será siempre constante. Podríamos en esa caso usar constantes si nuestro lenguaje las tiene, pero los atributos estáticos no tienen por qué ser siempre un valor invariable, como es el caso del precio de los AutobusesMetropolitanos, que sube cada año.

Con esos tenemos dos ejemplos de situaciones en las que me pueden venir bien tener atributos "static", o de clase, porque me permitirían consultar esos datos sin necesidad de tener una instancia de un objeto de esa clase.

En cuanto a métodos, pensemos por ejemplo en la clase Fecha. Puedo intentar construir fechas con un día, un mes y un año, pero puede que no necesite una fecha en un momento dado y solo quiera saber si una fecha podría ser válida. En situaciones normales debería intentar construir esa fecha y esperar a ver si el constructor me arroja un error o si la fecha que construye es válida. Quizás sería más cómodo tener un método vinculado a la clase, en el que podría pasarle un mes, un día y un año y que me diga si son válidos o no.

Cómo definir y usar miembros estáticos

Esto ya depende de tu lenguaje de programación. Generalmente se usa el modificador "static" para definir los miembros de clase, al menos así se hace en Java o PHP, C#, pero podrian existir otros métodos en otros lenguajes.

> **Nota:** El código que verás a continuación no es de uno u otro lenguaje. No es mi objetivo explicar Java, PHP, C#, etc. sino ver en pseudo-codigo rápidamente una posible implementación de el uso de static. Es parecido a Java, pero le he quitado algunas cosas que dificultan el entendimiento de los ejemplos. Tampoco es PHP, porque no he metido los $ de las variables y el uso del acceso a variables estáticas se hace con el operador :: que me parece muy feo y que complica la lectura y comprensión de los ejemplos. Tendrás que consultar la documentación de tu propio lenguaje para ver cómo se hace en él, aquí lo que quiero que queden claros son los conceptos nada más.

```
class AutobusMetropolitano {
 public static precio = 1.20}
```

Podrás observar que en la definición de la clase hemos asignado un valor al precio a la hora de declarar ese atributo de clase. Esto es condición indispensable en muchos lenguajes, puesto que si existe ese atributo debería tener un valor antes de instanciar el primer objeto. Osea, no es como los atributos normales, que les podemos asignar valores a través del constructor.

Luego podríamos acceder a esos elementos a través del nombre de la clase, como se puede ver a continuación:

```
if(1.5 >= AutobusMetropolitado.precio){
 //todo bien}
```

Ese código sería en el caso que estés accediendo al atributo estático desde fuera de la clase, pero en muchos lenguajes puedes acceder a esos atributos (o métodos) desde la vista privada de tus clases (código de implantación de la clase) a través de la palabra "self".

```
class AutobusMetropolitano {
 static precio = 1.20;
 //...
 public function aceptarPago(dinero){
  if (dinero < self.precio){
   return false;
  }
  // ...
 }
```

En el caso de los métodos estáticos la forma de crearlos o usarlos no varía mucho, utilizando el mismo modificador "static" al declarar el método.

```
class Fecha{
 //...
public static function valida(ano, mes, dia){
  if(dia >31)
```

```
   return false;
  // ...
 }
 //..}
```

Ahora para saber si un conjunto de año mes y día es válido podrías invocar al método estático a través del nombre de la clase.

```
if(Fecha.valida(2014, 2, 29)){
 // es valida
}else{
 // no es valida}
```

Como en el caso de los atributos de clase, también podrías acceder a métodos de clase con la palabra "self" si estás dentro del código de tu clase.

Miembros estáticos son susceptibles de hacer "marranadas"

No quiero dejar este artículo sin aprovechar la oportunidad de explicar que los miembros estáticos deben estár sujetos a un cuidado adicional para no cometer errores de diseño.

En ocasiones se tiende a usar los métodos estáticos para agregar datos a modo de variables globales, que estén presentes en cualquier lugar de la aplicación a partir del nombre de una clase. Esto nos puede acarrear exactamente los mismos problemas que conocemos por el uso de variables globales en general.

En cuanto a métodos estáticos a veces se usan las clases como simples contenedores de unión de diferentes métodos estáticos. Por ejemplo la clase Math en Javascript, o incluso Java, que tiene simplemente una serie de métodos de clase. Usarlos así, creando métodos estáticos en la clase, sin agregar más valor a la clase a través de métodos o atributos "normales" es caer en prácticas similares a las que se vienen usando en la programación estructurada. Dicho de otra manera, es usar la Programación Orientada a Objetos sin aprovechar los beneficios que nos aporta.

Por Miguel Angel Alvarez

1.5.- Herencia en Programación Orientada a Objetos

Concepto de herencia en la programación orientada a objetos: un mecanismo básico por el que las clases hijas heredan el código de las clases padre.

Este artículo viene a completar el texto Qué es Programación Orientada a Objetos publicado en DesarrolloWeb.com ya hace más de doce años. Aunque ya tiene su tiempo, según lo releo puedo decir que sigue perfecto, aunque le faltan algunas partes fundamentales que hacen de la programación orientada a objetos (POO) un paradigma extremadamente útil y potente.

Si ya entendiste lo que son clases y objetos, atributos y estados, métodos y mensajes, ahora puedes ampliar la información en el presente texto para conocer acerca de la herencia. Pero antes de ello, centrémonos en entender algunas de las prácticas más útiles y deseables de la programación en general.

Jerarquización

Es un proceso por el cual se crean organizaciones de elementos en distintos niveles. No es un concepto específicamente de POO, sino que es algo que vemos en la vida real en muchos ámbitos, algo inherente a cualquier tipo de sistema. Puedo tener diversos tipos de jerarquías, como clasificación o composición.

Composición: Es cuando unos elementos podemos decir que están compuestos de otros, o que unos elementos están presentes en otros. Por ejemplo, el sistema respiratorio y los pulmones, la nariz, etc. Podemos decir que los pulmones están dentro del sistema respiratorio, así como dentro de los pulmones encontramos bronquios y alvéolos. En esta jerarquía de elementos tenemos composición porque donde unos forman parte de otros. En una factura también podemos decir que puede haber una jerarquía de composición. La factura tiene un cliente, varios conceptos facturables, un impuesto, etc.

Clasificación: Este tipo de jerarquización indica que unos elementos son una especialización de otros. Por ejemplo, los animales, donde tenemos vertebrados e invertebrados. Luego, dentro de los vertebrados encontramos aves, reptiles, mamíferos, etc. En los mamíferos encontramos perros, vacas, conejos... Éste es el tipo de jerarquización en que quiero que te fijes.

Los lenguajes de programación orientados a objetos son capaces de crear jerarquizaciones basadas en composición con lo que ya sabemos de clases y objetos. Eso es porque podemos tener como propiedades de objetos, otros objetos. Por ejemplo, en el caso de la factura, podríamos tener como propiedades el cliente, el impuesto, la lista de conceptos facturables, etc. Sin embargo, para hacer jerarquías de clasificación nos hace falta conocer la herencia.

Reutilización del código

Por otra parte, otro de los mecanismos que cualquier lenguaje de programación debe proveer es la posibilidad de reutilizar el código. En la programación estructurada tenemos las funciones, así que ya hemos podido reutilizar código de alguna manera. Así pues, el equivalente a las funciones, los métodos, ya nos da un grado de reutilización, pero no llegan al nivel de potencia de las que encontraremos en la herencia.

No necesitamos decirte mucho más para entender las bondades de la reutilización: en inglés lo resume el término "DRY", Don't Repeat Yourself (no te repitas) y es uno de los enunciados que debes tener más presente cuando programas. "No es mejor programador quien más líneas de código hace, sino quien mejor las reutiliza".

Quizás está de más decirlo, porque seguro que ya sabes que debemos evitar escribir dos veces el mismo código, evitar los copia/pega y pensar que la reutilización nos ayuda seriamente en el mantenimiento del software. Enseguida verás cómo la herencia es un mecanismo fundamental para reutilizar código.

Herencia el la POO

Ahora que ya conoces dos beneficios que nos proporciona la herencia y por qué es algo tan deseable en la programación, creo que te sentirás motivado para profundizar en las bases de este mecanismo, herencia, clave de la Orientación a Objetos.

La herencia es la transmisión del código entre unas clases y otras. Para soportar un mecanismo de herencia tenemos dos clases: la clase padre y la/s clase/s hija/s. La clase padre es la que transmite su código a las clases hijas. En muchos lenguajes de programación se declara la herencia con la palabra "extends".

```
class Hija extends Padre{ }
```

Eso quiere decir que todo el código de la clase padre se transmite, tal cual, a la clase hija. Si lo quieres ver así, es como si tuvieras escrito, línea a línea, todo el código de la class "Padre" dentrode las llaves de la class "Hija". Por eso, la herencia es fundamental para reutilizar código, porque no necesitas volver a incorporar el código de Padre en Hija, sino que realmente al hacer el "extends" es como si ya estuviera ahí.

Ejemplo de herencia

Volvamos a los animales, pensemos en los mamíferos. Todos tienen una serie de características, como meses de gestación en la barriga de la madre, pechos en las hembras para amamantar y luego funcionalidades como dar a luz, mamar, etc. Eso quiere decir que cuando realices la clase perro vas a tener que implementar esos atributos y métodos, igual que la clase vaca, cerdo, humano, etc.

¿Te parecería bien reescribir todo ese código común en todos los tipos de mamíferos, o prefieres heredarlo? en este esquema tendríamos una clase mamífero que nos define atributos como numero_mamas, meses_gestacion y métodos como dar_a_luz(), mamar(). Luego tendrías la clase perro que extiende (hereda) el código del mamífero, así como las vacas, que también heredan de mamífero y cualquiera de los otros animales de esta clasificación.

Otro ejemplo, tenemos alumnos universitarios. Algunos son alumnos normales, otros Erasmus y otros becarios. Probablemente tendremos una clase Alumno con una serie de métodos como asistir_a_clase(), hacer_examen() etc., que son comunes a todos los alumnos, pero hay operaciones que son diferentes en cada tipo de alumno como pagar_mensualidad() (los becarios no pagan) o matricularse() (los Erasmus que son estudiantes de intercambio, se matriculan en su universidad de origen).

Lo que debes observar es que con la herencia siempre consigues clases hijas que son una especialización de la clase padre. Para saber si está correcto emplear herencia entre unas clases y otras, plantéate la pregunta ¿CLASE HIJA es un CLASE PADRE? (por ejemplo, ¿un perro es un mamífero? ¿Un becario es un alumno de universidad?)

> **Nota:** Existen otros modos de decir clases hija, como clase heredada, clase derivada, etc.

Otras cosas que tienes que saber sobre herencia

En este artículo nos hemos limitado a hablar sobre el concepto de herencia, pero no sobre una serie de mecanismos asociados que resultan clave para entender todavía mejor las posibilidades de esta capacidad de la POO. Nos referimos a la visibilidad de propiedades y métodos entre clases padre e hija, la posibilidad de hacer clases abstractas, que son las que contienen métodos abstractos o incluso propiedades abstractas. Hemos dejado de lado asuntos como la herencia múltiple, que te proporciona la posibilidad de heredar de varias clases a la vez (los ejemplos mencionados son de herencia simple).

Todo eso es algo que tendrás que aprender en otros textos, futuros artículos o en la referencia de tu propio lenguaje de

programación. Nosotros esperamos que el presente texto te haya aclarado el concepto de una forma amena, que es lo más fundamental para que a partir de aquí tengas la base suficiente para profundizar en las características de tu propio lenguaje de programación con orientación a objetos.

Por Miguel Angel Alvarez

1.6.- Polimorfismo en Programación Orientada a Objetos

Qué es el polimorfismo en la Programación Orientada a Objetos, el motivo de su existencia y cómo implementar polimorfismo en clases y objetos.

El concepto de polimorfismo es en realidad algo muy básico. Realmente, cuando estamos aprendiendo Programación Orientada a Objetos (también conocida por sus siglas POO / OOP) muchos estudiantes nos hacemos un embolado tremendo al tratar de entender el concepto, pero en su base es algo extremadamente sencillo.

Trataremos de explicarlo en este artículo con palabras sencillas, pero para los valientes, aquí va una primera definición que no es mía y que carece de la prometida sencillez. Pero no te preocupes, pues la entiendas o no, luego lo explicaré todo de manera más llana.

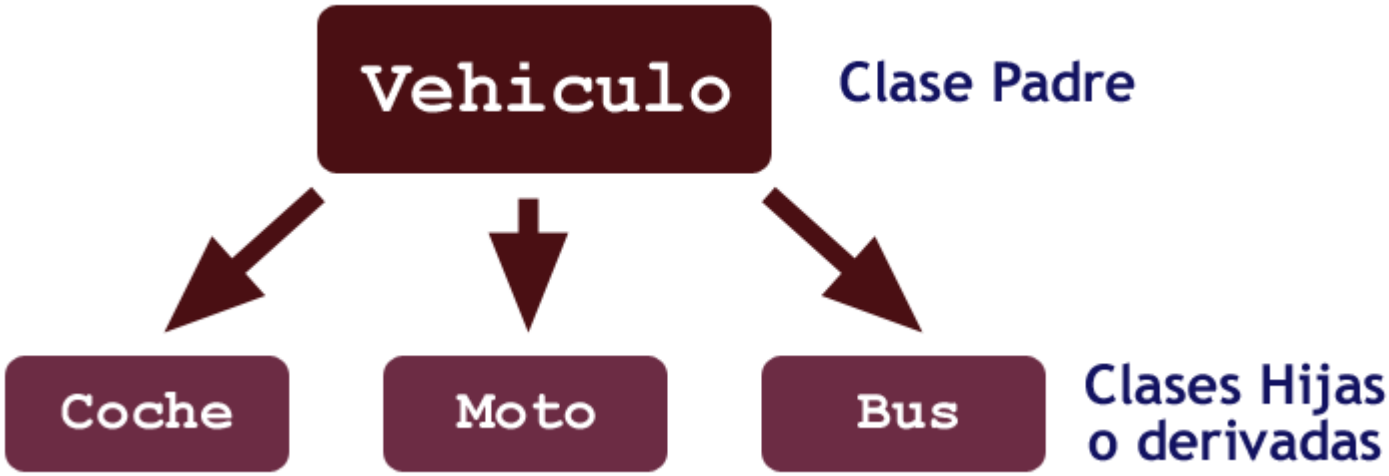
Definición: El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, ...).

> **Nota:** Esta es la definición académica que nos ofrece el profesor de la UPM Luis Fernández, del que fui alumno en la universidad y en EscuelaIT.

Herencia y las clasificaciones en Programación Orientada a Objetos

Para poder entender este concepto de OOP necesitas entender otras cosas previas, como es el caso de la herencia. Esto lo hemos explicado en un artículo anterior en DesarrolloWeb.com: Herencia en la Programación Orientada a Objetos.

Veremos que el polimorfismo y la herencia son dos conceptos estrechamente ligados. Conseguimos implementar polimorfismo en jerarquías de clasificación que se dan a través de la herencia. Por ejemplo, tenemos una clase vehículo y de ella dependen varias clases hijas como coche, moto, autobús, etc.



Pero antes de entender todo esto, queremos ir un poco más hacia atrás, entendiendo lo que es un sistema de tipos.

Por qué el sistema de tipos es importante en Polimorfismo

Muchos de los lectores que asumo se introducen en el concepto de polimorfismo a través de este artículo han aprendido a programar en lenguajes **débilmente tipados**, como es el caso de PHP y Javascript. Por ello es conveniente entender cómo es un lenguaje **fuertemente tipado**, como es el caso de Java o C.

En estos lenguajes, cuando defino una variable, siempre tengo que decir el tipo de datos que va a contener esta variable. Por ejemplo:

```
int miNumero;
```

Así le indicamos que la variable declarada "miNumero" va a contener siempre un entero. Podrás asignarle diversos valores, pero siempre deben de ser números enteros. De lo contrario el compilador te lanzará un mensaje de error y no te permitirá compilar el programa que has realizado.

Esto incluso pasa con los objetos. Por ejemplo, si en Java defino la clase "Largometraje" (una cinta que se puede exhibir en la televisión o el cine), cuando creo objetos de la clase "Largometraje" debo declarar variables en las que indique el tipo de objeto que va a contener.

```
Largometraje miLargo = new Largometraje("Lo que el viento se llevó");
```

Esa variable "miLargo", por declaración tendrá una referencia a un objeto de la clase "Largometraje". Pues bien, durante toda su vida, deberá tener siempre una referencia a cualquier objeto de la misma clase. O sea, mañana no podremos guardar un entero en la variable, ni una cadena u otro objeto de otra clase.

Volviendo al ejemplo de los vehículos, si defino una variable que apunta a un objeto de clase "Coche", durante toda la vida de esa variable tendrá que contener un objeto de la clase Coche, no pudiendo más adelante apuntar a un objeto de la clase Moto o de la clase Bus. Esta rigidez, como decimos, no existe en los lenguajes débilmente tipados como es el caso de Javascript o PHP, sin embargo es una característica habitual de lenguajes como Java, que son fuertemente tipados.

```
Coche miCoche = new Coche("Ford Focus 2.0");
//la variable miCoche apunta a un objeto de la clase coche
//si lo deseo, mañana podrá apuntar a otro objeto diferente, pero siempre tendrá que ser de la clase Coche
```

```
miCoche = new Coche("Renault Megane 1.6");
```

Lo que nunca podré hacer es guardar en esa variable, declarada como tipo Coche, otra cosa que no sea un objeto de la clase Coche.

```
//si miCoche fue declarada como tipo Coche, no puedo guardar un objeto de la clase Moto
miCoche = new Moto("Yamaha YBR");
//la línea anterior nos daría un error en tiempo de compilación
```

Fíjate que en este punto no te estoy hablando todavía de polimorfismo, sino de algo de la **programación en general como es el sistema de tipos**. Sin embargo, tienes que amoldar la cabeza a esta **restricción de lenguajes fuertemente tipados** para que luego puedas entender por qué el polimorfismo es importante y clave en la programación orientada a objetos. Y ojo, insisto que esto es algo relacionado con lenguajes fuertemente tipados (también llamados de tipado estático), en PHP no habría problema en cambiar el tipo de una variable, asignando cualquier otra cosa, dado que no se declaran los tipos al crear las variables.

Entendida esa premisa, pensemos en el concepto de función y su uso en lenguajes de tipado estático.

> **Nota:** A veces, a los lenguajes fuertemente tipados se les llama de "tipado estático" y a los débilmente tipados se les llama "tipado dinámico". Si quieres saber más sobre lenguajes tipados y no tipados, te recomiendo ver el #programadorIO tipados Vs no tipados.

Cuando en un lenguaje fuertemente tipado declaramos una función, siempre tenemos que informar el tipo de los parámetros que va a recibir. Por ejemplo, la función "sumaDosNumeros()" recibirá dos parámetros, que podrán ser de tipo entero.

```
function sumaDosNumeros(int num1, int num2)
```

A esta función, tal como está declarada, no le podremos pasar como parámetros otra cosa que no sean variables -o literales- con valores de número entero. En caso de pasar otros datos con otros tipos, el compilador te alertará. Osea, si intentas invocar sumaDosNumeros("algo", "otro"), el compilador no te dejará compilar el programa porque no ha encontrado los tipos esperados en los parámetros de la función.

Esto mismo de los parámetros en las funciones te ocurre también con los atributos de las clases, cuyos tipos también se declaran, con los datos que se insertan en un *array*, etc. Como ves, en estos lenguajes como Java el tipado se lleva a todas partes.

Polimorfismo en objetos

Ahora párate a pensar en clases y objetos. Quédate con esto: **Tal como funcionan los lenguajes fuertemente tipados, una variable siempre deberá apuntar a un objeto de la clase que se indicó en el momento de su declaración**. Una función cuyo parámetro se haya declarado de una clase, sólo te aceptará recibir objetos de esa clase. Un array que se ha declarado que es de elementos de una clase determinada, solo aceptará que rellenemos sus casillas con objetos de esa clase declarada.

```
Vehiculo[] misVehiculos = new Vehiculo[3];
```

Esa variable misVehiculos es un array y en ella he declarado que el contenido de las casillas serán objetos de la clase "Vehiculo". Como se ha explicado, en lenguajes fuertemente tipados sólo podría contener objetos de la clase Vehiculo. Pues bien, polimorfismo es el mecanismo por el cual podemos **"relajar el sistema de tipos"**, de modo que nos acepte también objetos de las clases hijas o derivadas.

Por tanto, la "relajación" del sistema de tipos no es total, sino que tiene que ver con las clasificaciones de herencia que tengas en tus sistemas de clases. Si defines un array con casillas de una determinada clase, el compilador también te aceptará que metas en esas casillas objetos de una **clase hija** de la que fue declarada. Si declaras que una función recibe como parámetros objetos de una determinada clase, el compilador también te aceptará que le envíes en la invocación objetos de una clase derivada de aquella que fue declarada.

En concreto, en nuestro array de vehículos, gracias al polimorfismo podrás contener en los elementos del array no solo vehículos genéricos, sino también todos los objetos de clases hijas o derivadas de la clase "Vehiculo", osea objetos de la clase "Coche", "Moto", "Bus" o cualquier hija que se haya definido.

Para qué nos sirve en la práctica el polimorfismo

Volvamos a la clase "Largometraje" y ahora pensemos en la clase "Cine". En un cine se reproducen largometrajes. Puedes, no obstante, tener varios tipos de largometrajes, como películas o documentales, etc. Quizás las películas y documentales tienen diferentes características, distintos horarios de audiencia, distintos precios para los espectadores y por ello has decidido que tu clase "Largometraje" tenga clases hijas o derivadas como "Película" y "Documental".

Imagina que en tu clase "Cine" creas un método que se llama "reproducir()". Este método podrá recibir como parámetro aquello que quieres emitir en una sala de cine y podrán llegarte a veces objetos de la clase "Película" y otras veces objetos de la clase "Documental". Si has entendido el sistema de tipos, y sin entrar todavía en polimorfismo, debido a que los métodos declaran los tipos de los parámetros que recibes, tendrás que hacer algo como esto:

```
reproducir(Pelicula peliculaParaReproducir)
```

Pero si luego tienes que reproducir documentales, tendrás que declarar:

```
reproducir(Documental documentaParaReproducir)
```

Probablemente el código de ambos métodos sea exactamente el mismo. Poner la película en el proyector, darle al *play*, crear un registro con el número de entradas vendidas, parar la cinta cuando llega al final, etc. ¿Realmente es necesario hacer dos métodos? De acuerdo, igual no te supone tanto problema, ¿pero si mañana te mandan otro tipo de cinta a reproducir, como la grabación de la final del mundial de fútbol en 3D? ¿Tendrás que crear un nuevo método reproducir() sobre la clase "Cine" que te acepte ese tipo de emisión? ¿es posible ahorrarnos todo ese mantenimiento?

Aquí es donde el polimorfismo nos ayuda. Podrías crear perfectamente un método "reproducir()" que recibe un largometraje y donde podrás recibir todo tipo de elementos, películas, documentales y cualquier otra cosa similar que sea creada en el futuro.

Entonces lo que te permiten hacer los lenguajes es declarar el método "reproducir()" indicando que el parámetro que vas a recibir es un objeto de la clase padre "Largometraje", pero donde realmente el lenguaje y compilador te aceptan cualquier objeto de la clase hija o derivada, "Película", "Documental", etc.

```
reproducir(Largometraje elementoParaReproducir)
```

Podremos crear películas y reproducirlas, también crear documentales para luego reproducir y lo bonito de la historia es que todos estos objetos son aceptados por el método "reproducir()", gracias a la relajación del sistema de tipos. Incluso, si mañana quieres reproducir otro tipo de cinta, no tendrás que tocar la clase "Cine" y el método "reproducir()". Siempre que aquello que quieras reproducir sea de la clase "Largometraje" o una clase hija, el método te lo aceptará.

Pongamos otro ejemplo por si acaso no ha quedado claro con lo visto hasta el momento, volviendo de nuevo a la clase Vehiculo. Además nos centramos en la utilidad del polimorfismo y sus posibilidades para reducir el mantenimiento de los programas informáticos, que es lo que realmente me gustaría que se entienda.

Tenemos la clase Parking. Dentro de ésta tenemos un método estacionar(). Puede que en un parking tenga que estacionar coches, motos o autobuses. Sin polimorfismo tendría que crear un método que permitiese estacionar objetos de la clase "Coche", otro método que acepte objetos de la clase "Moto" para estacionarlos, etc. Pero todos estaremos de

acuerdo que estacionar un coche, una moto o un bus es bastante similar: "entrar en el parking, recoger el ticket de entrara, buscar una plaza, situar el vehículo dentro de esa plaza...".

Lo ideal sería que nuestro método me permita permita recibir todo tipo de vehículos para estacionarlos, primero por reutilización del código, ya que es muy parecido estacionar uno u otro vehículo, pero además porque así si mañana el mercado trae otro tipo de vehículos, como una van, todoterreno hibrido, o una nave espacial, mi software sea capaz de aceptarlos sin tener que modificar la clase Parking.

Gracias al polimorfismo, cuando declaro la función estacionar() puedo decir que recibe como parámetro un objeto de la clase "Vehiculo" y el compilador me aceptará no solamente vehículos genéricos, sino todos aquellos objetos que hayamos creado que hereden de la clase Vehículo, osea, coches, motos, buses, etc. Esa relajación del sistema de tipos para aceptar una gama de objetos diferente es lo que llamamos polimorfismo.

En fin, esto es lo que significa polimorfismo. A partir de aquí puede haber otra serie de consideraciones y recomendaciones, así como características implementadas en otros lenguajes, pero explicar todo eso no es el objetivo de este artículo. Esperamos que con lo que has aprendido puedas orientar mejor tus estudios de Programación Orientada a

Objetos. Si quieres más información sobre el tema lee el artículo Qué es Programación Orientada a Objetos, que seguro te será de gran utilidad.

Por Miguel Angel Alvarez

1.7.- Abstracción en Programación Orientada a Objetos

Concepto de abstracción en el paradigma de la Programación Orientada a Objetos y situaciones en las que se puede y se debe aplicar.

Abstracción es un término del mundo real que podemos aplicar tal cual lo entendemos en el mundo de la Programación Orientada a Objetos. Algo abstracto es algo que está en el universo de las ideas, los pensamientos, pero que no se puede concretar en algo material, que se pueda tocar.

Pues bien, una clase abstracta es aquella sobre la que no podemos crear especímenes concretos, en la jerga de POO es aquella sobre la que no podemos instanciar objetos. Ahora bien, ¿cuál es el motivo de la existencia de clases abstractas? o dicho de otra manera, ¿por qué voy a necesitar alguna vez declarar una clase como abstracta?, ¿en qué casos debemos aplicarlas? Esto es todo lo que pretendemos explicar en este artículo.

Abstracción en el mundo real

La programación orientada a objetos sabemos que, de alguna manera, trata de "modelizar" los elementos del mundo real. En el mundo en el que vivimos existe un universo de objetos que colaboran entre sí para realizar tareas de los sistemas. Llevado al entorno de la programación, también debemos programar una serie de clases a partir de las cuales se puedan instanciar objetos que colaboran entre sí para la resolución de problemas. Si asumimos esto, a la vista de las situaciones que ocurren en el mundo real, podremos entender la abstracción.

Cuando estudiamos en el concepto de Herencia en Programación Orientada a Objetos vimos que con ella se podían definir jerarquías de clasificación: los animales y dependiendo de éstos tenemos mamíferos, vertebrados, invertebrados. Dentro de los mamíferos tenemos vacas, perros...

Animal puede ser desde una hormiga a un delfín o un humano. En nuestro cerebro el concepto de animal es algo genérico que abarca a todos los animales: "seres vivos de un "reino" de la existencia". Si defines animal tienes que usar palabras muy genéricas, que abarquen a todos los animales posibles que puedan existir en el mundo. Por ello no puedes decir que animales son aquellos que nacen de huevos, o después de un periodo de gestación en la placenta.

Adonde quiero llegar es que el animal te implica una abstracción de ciertos aspectos. Si lo definimos con palabras no podemos llegar a mucho detalle, porque hay muchos animales distintos con características muy diferentes. Hay características que se quedan en el aire y no se pueden definir por completo cuando pensamos en el concepto de animal "genérico".

Para acabar ¿en el mundo real hay un "animal" como tal? No, ni tan siquiera hay un "mamífero". Lo que tenemos son especímenes de "perro" o "vaca", "hormiga", "cocodrilo", "gorrión" pero no "animal" en plan general. Es cierto que un perro es un animal, pero el concepto final, el ejemplar, es de perro y no animal.

Por tanto "animal", en términos del lenguaje común, podemos decir que es un concepto genérico, pero no una concreción. En términos de POO decimos que es un concepto abstracto, que implementaremos por medio de una clase abstracta. No instanciaremos animales como tal en el mundo, sino que instanciaremos especímenes de un tipo de animal concreto.

En los animales existen propiedades y métodos que pueden ser comunes a todos los animales en general. Los animales podrán tener un nombre o una edad, determinadas dimensiones o podrán desempeñar acciones como morir. Lo que nos debe quedar claro es que no deberíamos poder instanciar un animal como tal. ¿Cómo nace un animal en concreto?, ¿cómo se alimenta? Para responder a esas preguntas necesitamos tener especímenes más concretos. Sí que sé cómo nace o cómo se alimenta una hormiga, o un gorrión, pero no lo puedo saber de un animal genérico, porque puede hacerlo de muchas maneras distintas.

Seguiremos trabajando para explicar estos conceptos, pero de momento entendemos que "animal" es una clase abstracta, pero "hormiga", "perro" o "gorrión" no serían clases abstractas, que sí podríamos instanciar.

Herencia y abstracción

Si entendemos el concepto de herencia podremos entender mejor la abstracción y cómo se implementa.

Recuerda nuestro ejemplo: Tengo animales. Hemos acordado que no puedo tener un animal concreto instanciado en un sistema. Si acaso tendré instancias de perros, saltamontes o lagartijas. Pues bien, en los esquemas de herencia este caso nos puede surgir muy habitualmente.

En la clase "animal" puedo tener determinadas propiedades y acciones implementadas. Por ejemplo, todos los animales pueden tener un nombre, o una edad (ya sean segundos, días o años de edad). También es posible que pueda definir diversas acciones de una vez para todos los animales de una jerarquía de herencia, por ejemplo, la acción de morir, pues todos morimos igual (simplemente dejamos de existir aunque aquí dependiendo de las creencias de cada uno esto pueda ser discutible).

Aunque mi sistema no pueda crear animales como tal, tener definidas esas cuestiones comunes a todos los animales me resulta útil para no tener que programarlas de nuevo en todos los tipos de animales que puedan existir. Simplemente las heredaré en las clases hijas, de modo que estarán presentes sin tener que volver a programar todas esas cosas comunes.

Sin embargo hay cosas de los animales que no podré implementar todavía. Atributos como el número de patas, el alcance de la visión, se implementarán a futuro en los tipos de animales que las necesiten, pero fijémonos en las acciones o métodos. Por ejemplo nacer, alimentarse, etc. No sé cómo va a nacer un animal, pero sé que todos los animales del mundo nacen de algún modo (unos nacen de huevos, otros estaban en la barriga de las hembras y nacen a consecuencia de un parto, etc.)

En estos casos nos puede ser útil definir como métodos abstractos en la clase "animal" esos métodos que van a estar presentes en todos los animales, aunque no seamos capaces de implementarlos todavía.

```
public abstract function nacer();
```

Esto quiere decir que todos los animales del mundo heredarán un método abstracto llamado nacer. En las clases concretas que hereden de animal y donde ya sepamos cómo nace tal animal, por ejemplo, la gallina, podemos implementar ese método, para que deje de ser abstracto.

```
public function nacer(){
 //se rompe el huevo y nace el pollito que más adelante será una hermosa gallina
}
```

Hasta ahora sabemos que hay clases que tienen métodos abstractos, que no somos capaces de implementar todavía y clases en las que se heredan métodos abstractos y en las que seremos capaces de implementarlos.

La utilidad de esto la entenderemos mejor en unos instantes, al tratar el polimorfismo, pero de momento debemos ser capaces de asimilar estas definiciones más formales:

"Una clase abstracta es aquella en la que hay definidos métodos abstractos, sobre la que no podremos instanciar objetos" Además, en un esquema de herencia, **"Si heredamos de una clase abstracta métodos abstractos, tampoco se podrán instanciar objetos de las clases hijas y tendrán que definirse como abstractas, a no ser que implementemos todos y cada uno de los métodos que se habían declarado como abstractos en la clase padre"**.

Polimorfismo y abstracción

Creo que si no se examina de cerca la abstracción bajo el prisma del polimorfismo no se puede entender bien la verdadera utilidad de hacer clases abstractas. Pero antes de seguir, recuerda qué es el Polimorfismo en Programación Orientada a Objetos.

Cuando hablamos de polimorfismo explicamos que es una relajación del sistema de tipos por la cual éramos capaces de aceptar objetos de un tipo y de todas las clases hijas. Por ejemplo, tengo la clase "PoligonoRegular". Sé que los polígonos regulares voy a querer conocer su área, pero para saber su área necesito conocer el número de lados que tiene. Entonces la clase "PoligonoRegular" tendrá un método abstracto "dameArea()". Luego, al definir la clase "cuadrado", o el "pentágono", etc. podremos implementar tal método, con lo que dejará de ser abstracto. Tenemos "Alumnos de una Universidad", los alumnos los vas a querer matricular en las universidades, pero dependiendo del tipo de alumno la matrícula se hace diferente, pues no es lo mismo matricular un alumno becario, o de familia numerosa, que uno normal. Entonces, en la clase "alumno" tendré un método abstracto que sea "matriculate()" que podré definir del todo cuando implemente las clases hijas.

Ahora piensa en esto. Gracias a que fueron definidos los métodos abstractos "dameArea()" y "matriculate()" en las clases padres, tengo clara una cosa: cuando trabajo con elementos de la clase "poligonoRegular", sé que a todos los polígonos regulares que pueda recibir les puedo pedir que me devuelvan su área. También sé que a todos los alumnos les puedo pedir que se matriculen en una universidad.

Ahí está la potencia del polimorfismo, recibir un objeto que pertenece a una jerarquía de clasificación y saber que puedo pedirle determinadas cosas. Quizás en la clase padre no pudieron implementarse esos comportamientos, porque no sabíamos el código necesario para ello, pero al menos se declararon que iban a poder realizarse en el futuro en clases hijas. Eso me permite, en un esquema de polimorfismo, que pueda estar seguro que todos los objetos que reciba puedan responder a acciones determinadas, pues en las clases hijas habrán sido definidas necesariamente (si no se definen deberían declararse las clases como abstractas y en ese caso es imposible que me manden objetos de esa clase).

Es cierto que el concepto se puede quedar un poco en "la abstracción" pero cuando practiques un poco te darás cuenta de la esencia de las clases abstractas y entenderás lo útil y necesario que es declarar métodos abstractos para poder implementar el polimorfismo. De momento es todo en cuanto a este concepto, esperamos que este texto te haya servido

de algo. Por supuesto, se agradecen los comentarios.

Por Miguel Angel Alvarez

Parte 2:

Patrones de diseño de Programación Orientada a Objetos

Los patrones de diseño resuelven de manera práctica y elegante una infinidad de problemas habituales y recurrentes a los que se enfrenta todo programador en el desarrollo de las aplicaciones, ya sean simples o complejas. Ofrecen diversas soluciones pensadas, estudiadas e implementadas en cualquier lenguaje orientado a objetos. Conocer patrones de diseño te ayudará a programar de con mejores prácticas y te aportará versatilidad, reutilización de código, desacoplamiento y cohesividad, características ya de por si presentes cuando se realiza una buena programación con objetos.

2.1.- Qué es MVC

Te explicamos de manera general MVC, Model - View - Controller o Modelo - Vista - Controlador un patrón de diseño de software para programación que propone separar el código de los programas por sus diferentes responsabilidades.

En líneas generales, MVC es una propuesta de diseño de software utilizada para implementar **sistemas donde se requiere el uso de interfaces de usuario**. Surge de la necesidad de crear software más robusto con un ciclo de vida más adecuado, donde se potencie la facilidad de mantenimiento, reutilización del código y la separación de conceptos.

Su fundamento es la **separación del código en tres capas diferentes**, acotadas por su responsabilidad, en lo que se llaman **Modelos, Vistas y Controladores**, o lo que es lo mismo, *Model, Views & Controllers*, si lo prefieres en inglés. En este artículo estudiaremos con detalle estos conceptos, así como las ventajas de ponerlos en marcha cuando desarrollamos.

MVC es un "invento" que ya tiene varias décadas y fue presentado incluso antes de la aparición de la Web. No obstante, en los últimos años **ha ganado mucha fuerza y seguidores gracias a la aparición de numerosos frameworks de desarrollo web que utilizan el patrón MVC como modelo para la arquitectura de las aplicaciones web.**

> **Nota:** Como ya hemos mencionado, MVC es útil para cualquier desarrollo en el que intervengan interfaces de usuario. Sin embargo, a lo largo de este artículo explicaremos el paradigma bajo el prisma del desarrollo web.

Por qué MVC

La rama de la ingeniería del software se preocupa por crear procesos que aseguren calidad en los programas que se realizan y esa calidad atiende a diversos parámetros que son deseables para todo desarrollo, como la estructuración de los programas o reutilización del código, lo que debe influir positivamente en la facilidad de desarrollo y el mantenimiento.

Los ingenieros del software se dedican a estudiar de qué manera se pueden mejorar los procesos de creación de software y una de las soluciones a las que han llegado es la arquitectura basada en capas que separan el código en función de sus responsabilidades o conceptos. Por tanto, cuando estudiamos MVC lo primero que tenemos que saber es que está ahí para **ayudarnos a crear aplicaciones con mayor calidad.**

Quizás, para que a todos nos queden claras las ventajas del MVC podamos echar mano de unos cuantos ejemplos:

1. Aunque no tenga nada que ver, comencemos con algo tan sencillo como son el HTML y las CSS. Al principio, en el HTML se mezclaba tanto el contenido como la presentación. Es decir, en el propio HTML tenemos etiquetas como "font" que sirven para definir las características de una fuente, o atributos como "bgcolor" que definen el color de un fondo. El resultado es que tanto el contenido como la presentación estaban juntos y si algún día pretendíamos cambiar la forma con la que se mostraba una página, estábamos obligados a cambiar cada uno de los archivos HTML que componen una web, tocando todas y cada una de las etiquetas que hay en el documento. Con el tiempo se observó que eso no era práctico y se creó el lenguaje CSS, en el que se **separó la responsabilidad de aplicar el formato** de una web.

Al escribir programas en lenguajes como PHP, cualquiera de nosotros comienza mezclando tanto el código PHP como el código HTML (e incluso el Javascript) en el mismo archivo. Esto produce lo que se denomina el "Código Espagueti". Si algún día pretendemos cambiar el modo en cómo queremos que se muestre el contenido, estamos obligados a repasar todas y cada una de las páginas que tiene nuestro proyecto. Sería mucho más útil que el **HTML estuviera separado del PHP.** Si queremos que en un equipo intervengan perfiles distintos de profesionales y trabajen de manera autónoma, como diseñadores o programadores, ambos tienen que tocar los mismos archivos y el diseñador se tiene necesariamente que relacionar con mucho código en un lenguaje de programación que puede no serle familiar, siendo que a éste quizás solo le interesan los bloques donde hay HTML. De nuevo, sería mucho más fácil la **separación** del código. Durante la manipulación de datos en una aplicación es posible que estemos accediendo a los mismos datos en lugares

distintos. Por ejemplo, podemos acceder a los datos de un artículo desde la página donde se muestra éste, la página donde se listan los artículos de un manual o la página de *backend* donde se administran los artículos de un sitio web. Si un día cambiamos los datos de los artículos (alteramos la tabla para añadir nuevos campos o cambiar los existentes porque las necesidades de nuestros artículos varían), estamos obligados a cambiar, página a página, todos los lugares donde se consumían datos de los artículos. Además, si tenemos el código de acceso a datos disperso por decenas de lugares, es posible que estemos repitiendo las mismas sentencias de acceso a esos datos y por tanto no estamos **reutilizando código.**

Quizás te hayas visto en alguna de esas situaciones en el pasado. Son solo son simples ejemplos, habiendo decenas de casos similares en los que resultaría útil aplicar una arquitectura como el MVC, con la que nos obliguemos a separar nuestro código atendiendo a sus responsabilidades.

Ahora que ya podemos tener una idea de las ventajas que nos puede aportar el MVC, analicemos las diversas partes o conceptos en los que debemos separar el código de nuestras aplicaciones.

Modelos

Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes *selects, updates, inserts*, etc.

No obstante, cabe mencionar que cuando se trabaja con MCV lo habitual también es utilizar otras librerías como PDO o algún ORM como Doctrine, que nos permiten trabajar con abstracción de bases de datos y persistencia en objetos. Por ello, en vez de usar directamente sentencias SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.

Vistas

Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra aplicación en HTML. En las vistas nada más tenemos los códigos HTML y PHP que nos permite **mostrar la salida.**

En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.

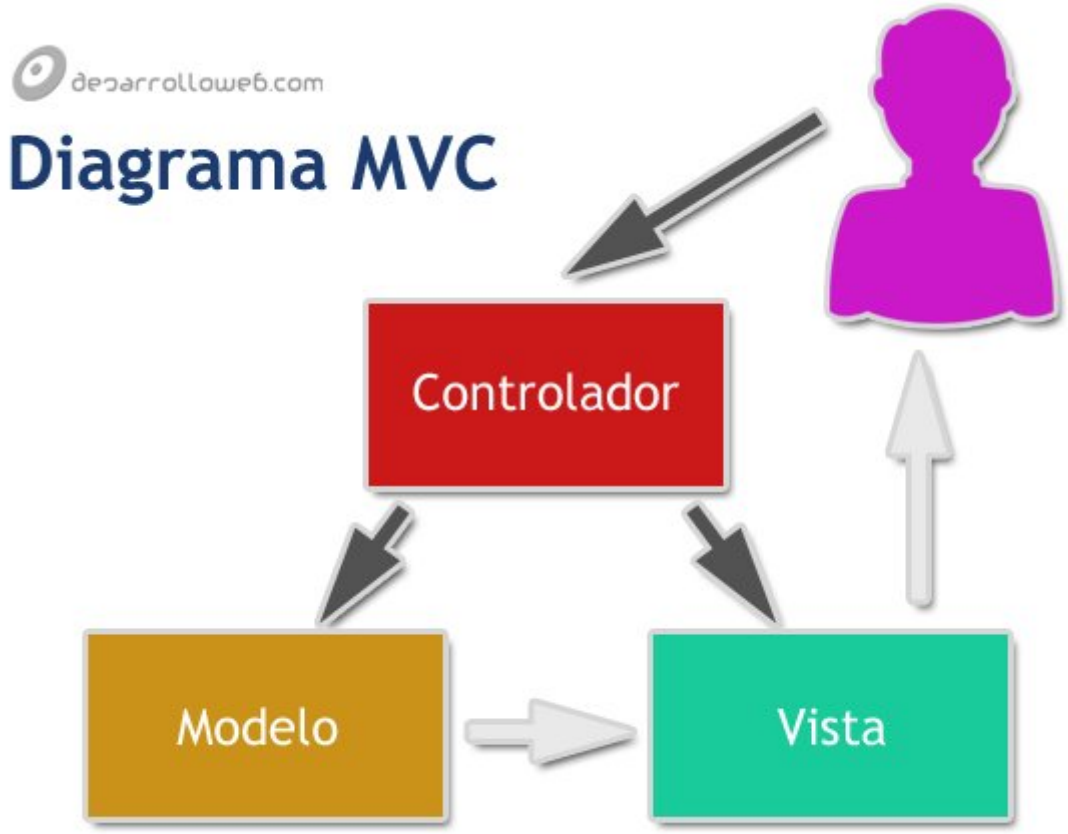
Controladores

Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc.

En realidad es una capa que sirve de **enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación.** Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

Arquitectura de aplicaciones MVC

A continuación encontrarás un diagrama que te servirá para entender un poco mejor cómo colaboran las distintas capas que componen la arquitectura de desarrollo de software en el patrón MVC.



En esta imagen hemos representado con flechas los modos de colaboración entre los distintos elementos que formarían una aplicación MVC, junto con el usuario. Como se puede ver, los controladores, con su lógica de negocio, hacen de puente entre los modelos y las vistas. Pero además en algunos casos los modelos pueden enviar datos a las vistas. Veamos paso a paso cómo sería el flujo de trabajo característico en un esquema MVC.

1. El usuario **realiza una solicitud** a nuestro sitio web. Generalmente estará desencadenada por acceder a una página de nuestro sitio. Esa solicitud le llega al controlador.

El **controlador comunica tanto con modelos como con vistas.** A los modelos les solicita datos o les manda realizar actualizaciones de los datos. A las vistas les solicita la salida correspondiente, una vez se hayan realizado las operaciones pertinentes según la lógica del negocio. Para producir la salida, en ocasiones las **vistas pueden solicitar más información a los modelos.** En ocasiones, el controlador será el responsable de solicitar todos los datos a los modelos y de enviarlos a las vistas, haciendo de puente entre unos y otros. Sería corriente tanto una cosa como la otra, todo depende de nuestra implementación; por eso esa flecha la hemos coloreado de otro color. **Las vistas envían al usuario la salida.** Aunque en ocasiones esa salida puede ir de vuelta al controlador y sería éste el que hace el envío al cliente, por eso he puesto la flecha en otro color.

Lógica de negocio / Lógica de la aplicación

Hay un concepto que se usa mucho cuando se explica el MVC que es la "lógica de negocio". Es un conjunto de reglas que se siguen en el software para reaccionar ante distintas situaciones. En una aplicación el usuario se comunica con el sistema por medio de una interfaz, pero cuando acciona esa interfaz para realizar acciones con el programa, se ejecutan una serie de procesos que se conocen como la lógica del negocio. Este es un concepto de desarrollo de software en general.

La lógica del negocio, aparte de marcar un comportamiento cuando ocurren cosas dentro de un software, también tiene normas sobre lo que se puede hacer y lo que no se puede hacer. Eso también se conoce como reglas del negocio. Bien, pues en el MVC la lógica del negocio queda del lado de los modelos. Ellos son los que deben saber cómo operar en diversas situaciones y las cosas que pueden permitir que ocurran en el proceso de ejecución de una aplicación.

Por ejemplo, pensemos en un sistema que implementa usuarios. Los usuarios pueden realizar comentarios. Pues si en un modelo nos piden eliminar un usuario nosotros debemos borrar todos los comentarios que ha realizado ese usuario también. Eso es una responsabilidad del modelo y forma parte de lo que se llama la lógica del negocio.

> **Nota:** Si no queremos que esos comentarios se pierdan otra posibilidad sería mantener el registro del usuario en la tabla de usuario y únicamente borrar sus datos personales. Cambiaríamos el nombre del usuario por algo como "Jon Nadie" (o cualquier otra cosa), de modo que no perdamos la integridad referencial de la base de datos entre la tabla de comentario y la tabla de usuario (no debe haber comenarios con un id_usuario que luego no existe en la tabla de usuario). Esta otra lógica también forma parte de lo que se denomina lógica del negocio y se tiene que implementar en el modelo.

Incluso, en nuestra aplicación podría haber usuarios especiales, por ejemplo "administradores" y que no está permitido borrar, hasta que no les quitemos el rol de administrador. Eso también lo deberían controlar los modelos, realizando las comprobaciones necesarias antes de borrar efectivamente el usuario.

Sin embargo existe otro concepto que se usa en la terminología del MVC que es la "lógica de aplicación", que es algo que pertenece a los controladores. Por ejemplo, cuando me piden ver el resumen de datos de un usuario. Esa acción le llega al controlador, que tendrá que acceder al modelo del usuario para pedir sus datos. Luego llamará a la vista apropiada para poder mostrar esos datos del usuario. Si en el resumen del usuario queremos mostrar los artículos que ha publicado dentro de la aplicación, quizás el controlador tendrá que llamar al modelo de artículos, pedirle todos los publicados por ese usuario y con ese listado de artículos invocar a la vista correspondiente para mostrarlos. Todo ese conjunto de acciones que se realizan invocando métodos de los modelos y mandando datos a las vistas forman parte de la lógica de la aplicación.

> **Nota:** Este concepto Lógica de aplicación no está tan extendido entre todos los teóricos, pero nos puede ayudar a comprender dónde está cada responsabilidad de nuestro sistema y dónde tenemos que escribir cada parte del código.

Otro ejemplo. Tenemos un sistema para borrar productos. Cuando se hace una solicitud a una página para borrar un producto de la base de datos, se pone en marcha un controlador que recibe el identificador del producto que se tiene que borrar. Entonces le pide al modelo que lo borre y a continuación se comprueba si el modelo nos responde que se ha podido borrar o no. En caso que se haya borrado queremos mostrar una vista y en caso que no se haya borrado queremos mostrar otra. Este proceso también está en los controladores y lo podemos denominar como lógica de la aplicación.

De momento ¡eso es todo! Esperamos que este artículo haya podido aclarar los distintos conceptos relacionados con el MVC y aunque no hayamos visto una implementación en código, te sirva para poder investigar a partir de aquí. En DesarrolloWeb.com hemos tratado con mayor detalle algunos aspectos de MVC y la relación entre vistas y modelos y sus interpretaciones en un artículo que seguro te interesará. Además podrás ver cómo trabajar con MVC en el Manual de Codeigniter

, en el Manual de Laravel y en el Manual del framework ASP.NET MVC.

Paralelamente queremos que conozcas nuestra plataforma para la formación EscuelaIT, donde podrás aprender con nosotros todo sobre este MVC y otros asuntos relacionados con la arquitectura del software en este curso de MVC y otras Técnicas de desarrollo de aplicaciones web en PHP.

Por Miguel Angel Alvarez

2.2.- Patrón Adapter para desarrollo con API

El patrón adapter permite realizar una envoltura de una API para un desarrollo mantenible cuando usamos una API de terceros.

Hoy cada vez es más frecuente que programemos determinados comportamientos en una web usando APIs Rest o servicios web de terceros. Incluso es común encontrar aplicaciones web completas que están basadas en APIs de terceros, como es el caso del API de Twitter o el API de Facebook, por poner dos ejemplos de API utilizadas con especial frecuencia. Para hacernos una idea de este tipo de proyectos, basta nombrar plataformas tan conocidas como Tweetdeck, Hootsuite o Bufferapp, que trabajan con el API de Twitter u otros de redes sociales.

En este artículo vamos a relatar una buena práctica que te facilitará disponer de un desarrollo más mantenible cuando estás trabajando con un API de terceros, sobre el que no tienes control. Se trata de crear una interfaz propia que centralice todo el trabajo, de modo que si cambia el API del tercero, solo tengas que cambiar la programación de esa interfaz propia.

Nota: Por si no queda claro, en este caso se usa la palabra "interfaz" desde el punto de vista de la programación, como una lista de métodos disponibles para el acceso a funcionalidades diversas que están disponibles en un sistema.

Esta buena práctica no es algo que nos estemos inventando ahora. Ya tiene nombre y es un patrón de diseño de software comúnmente aceptado y usado. Se trata del patrón de diseño "Adapter" y en líneas generales se usa para proveer una interfaz de programación de un sistema al que no tienes acceso. Es como un "puente de acceso" a funcionalidades residentes en sistemas externos.

Nota: Este artículo lo introducimos dentro del Manual de los conceptos de Programación Orientada a Objetos, aunque realmente no es un concepto de este paradigma de la programación, sino algo más específico para un desarrollo que permita un mejor mantenimiento cuando nos basamos en un API.

Problemática cuando basamos nuestro desarrollo de negocio en un API

Se dice que si desarrollas un negocio basado en la plataforma de una empresa o red que no depende de ti, estás "vendido". El problema cuando trabajas con API que no controlas es que en cualquier momento te pueden cambiar ese API y con ello te obligan a cambiar tu código fuente.

Imagina tener un proyecto que consume el API de Twitter para implementar una funcionalidad nueva para un negocio *online* que has ideado. Tu aplicación consume y envía tuits, trabaja con listas de usuarios y todo lo que se te pueda ocurrir. Probablemente en tu gran aplicación tendrás miles de líneas de código que trabajan con el API de Twitter.

Ahora imagina qué pasaría si actualizan el API y te cambian radicalmente la manera de comunicar con ella, ¿qué pasaría con tu proyecto? Pues probablemente estarías "vendido" y necesitarías cambiar rápidamente todas las líneas de código que trabajan con el API para adaptarlas a las nuevas maneras de uso. Esto te puede ocasionar cientos de horas de trabajo o una inversión extraordinaria que no habías previsto.

Si no tenemos bien planificado el proyecto a nivel de desarrollo probablemente nos encontremos en una difícil situación que nos obliga a ser muy minuciosos revisando una maraña de clases y cientos o miles de líneas de código dispersas por todo el proyecto ¿Te atrae la idea? ¿Quizás sea mejor abandonar el proyecto para dedicarse a plantar patatas y criar gallinas en en una granja?

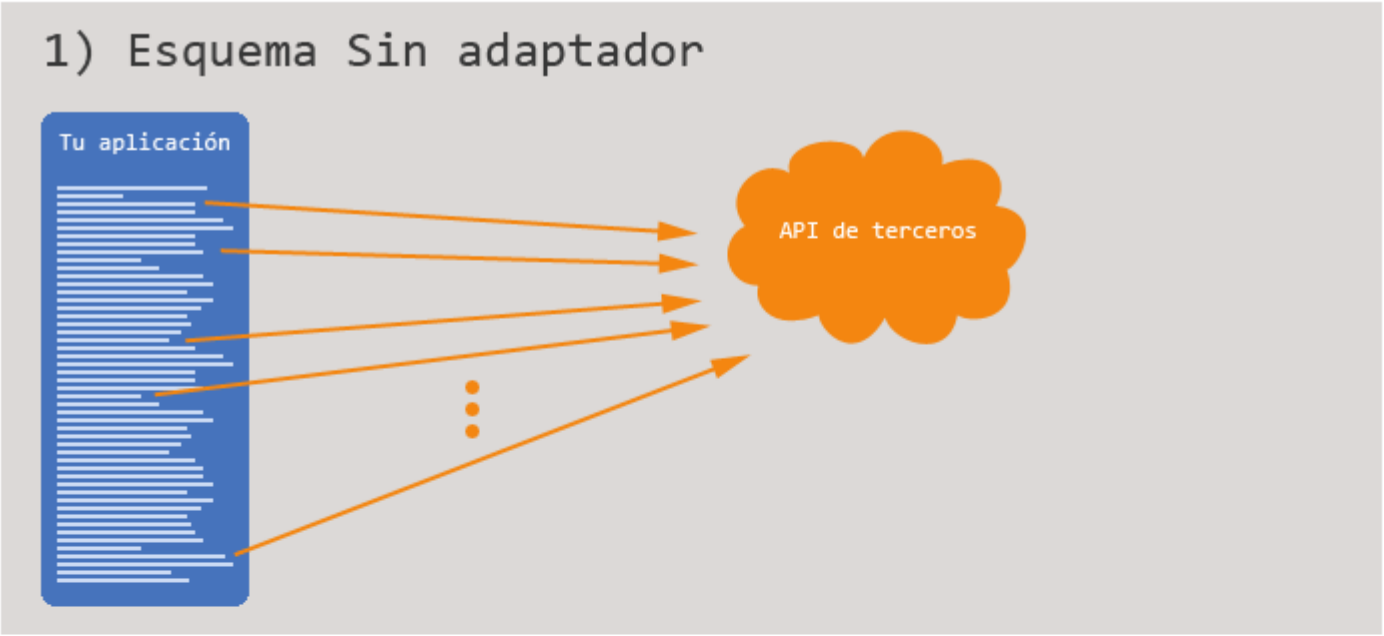
Afortunadamente, hay maneras en las que podrás mitigar esta problemática y reducir de manera drástica la oleada de mantenimiento de tu aplicación provocada por un cambio en la API.

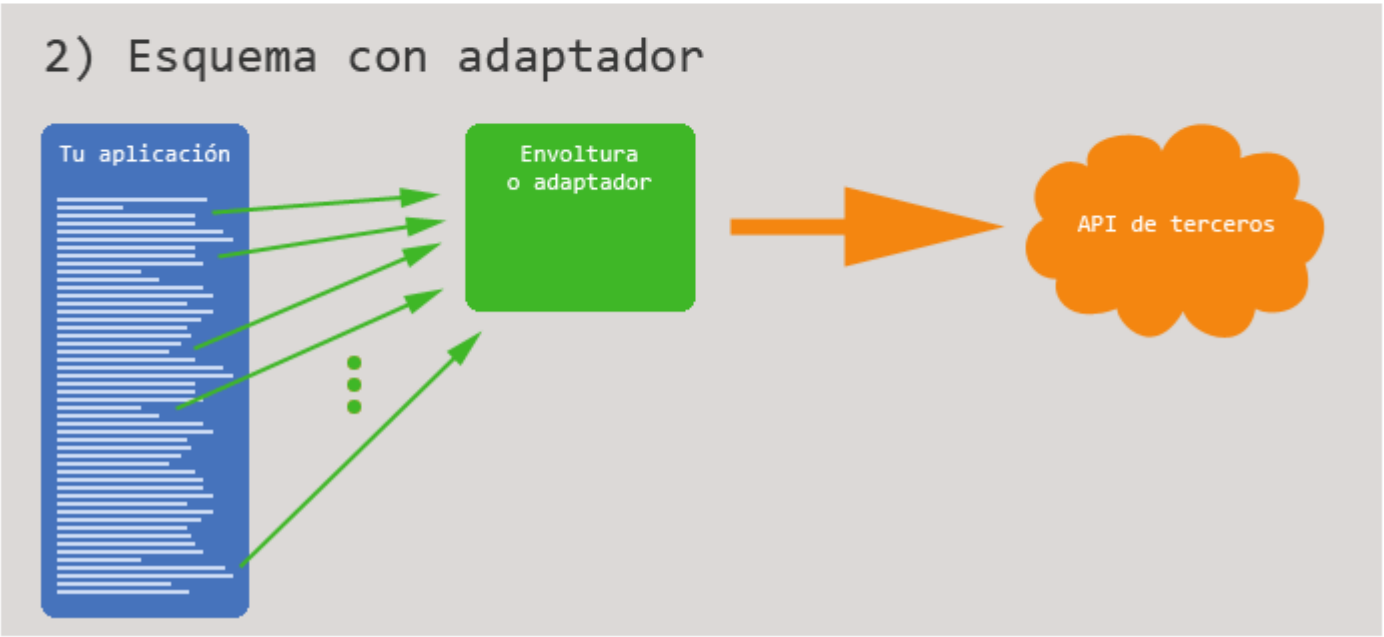
Envoltura de un API

La idea es crear una envoltura o adaptador para la comunicación con el API. Todas las operaciones de tu aplicación cuando trabaja con el API de terceros las debes centralizar en esa clase de POO que realiza la "envoltura".

De este modo, si te cambian el API, el único sitio donde tendrás que cambiar tu código es en aquella clase que realiza el papel de adaptador. La diferencia es sustancial: sin un adaptador probablemente tengas que cambiar el código de tu aplicación en miles de líneas, dispersas en decenas o incluso cientos de clases. Si tienes una envoltura, será solo cambiar la clase que hace uso del API.

Para darnos una idea mejor sobre esta envoltura puedes consultar esta imagen:





Pseudocódigo de ejemplo de envoltura

Este artículo está más enfocado a explicar el concepto, de modo que puedas entender esta buena práctica y ponerla en marcha si alguna vez implementas un API en tus proyectos. No obstante, vamos a colocar aquí un poco de pseudo-código que nos puede dar una idea más concreta.

Insisto, no se trata de ver cómo es un API en concreto y cuáles son sus métodos y esas cosas, sino más bien ver las ventajas de usar una envoltura.

```
//uso una supuesta API de Twitter
//tiene los métodos
// get(usuario, num_tuits);
// put(usuario, texto);
$tuits = get("midesweb", 10);
put("midesweb", "otro tuit");
/// ...
$tiuts = get("micromante", 5);
put("micromante", "otro tuit");
// ...
//Puedes estar usando esos métodos en otros miles de líneas de código
```

Ahora, ¿qué pasa si me cambian los nombres de funciones y le llaman "get_tuits()" y "put_tuits()"?. Realmente, este cambio podría ser tan simple como buscar y reemplazar los nombres de los métodos en todo el código del proyecto, pero cuando te cambian un API no suele ser para cambiarte simplemente los nombres de los métodos, sino para cambiar a veces radicalmente el modo de funcionar, con nuevos métodos, nuevos parámetros, etc.

En este caso tienes que editar todos los sitios donde llamas a las funciones (imagina que tienes miles o cientos de miles de líneas de código).

Si fuiste precavido y te has creado una maravillosa envoltura para trabajar con el API, tendrás algo como esto:

```
// mi envoltura
function dame_tuits(nombre, cantidad){
  get(nombre, cantidad);
}
function manda_tuit(nombre, tuit){
  put(nombre, tuit);
}
/// Las usas así
$tuits = dame_tuits("midesweb", 10);
manda_tuit("midesweb", "otro tuit");
...
$tiuts = dame_tuits("micromante", 5);
manda_tuit("micromante", "otro tuit");
```

Ahora, en caso que te cambiasen el API con la que estás trabajando, tienes simplemente que cambiar el código de tu envoltura.

```
function dame_tuits(nombre, cantidad){
get_tuits(nombre, cantidad);
}
function manda_tuit(nombre, tuit){
put_tuits(nombre, tuit);
}
```

Como dentro de todo el proyecto solo hacías llamadas a tu "adapter" (la envoltura) y no al API directamente, simplemente tendrás que editar unas pocas líneas de código, reescribiendo las operaciones de tu adaptador, para constatar los nuevos métodos, parámetros y uso en general de la nueva API de terceros.

Esto es todo, espero que puedas aplicar esta práctica a cualquier proyecto con el que debas trabajar con un API del que no tienes el control, así como para acceder a cualquier otro tipo de interfaz de programación que no depende de ti directamente.

Por Miguel Angel Alvarez

2.3.- Inyección de dependencias

Qué es la Inyección de dependencias. Que es el contenedor de dependencias, elementos de un patrón de diseño de software usado en la mayoría de los grandes frameworks.

En este artículo vamos a fundir dos textos de DesarrolloWeb.com dedicados a la inyección de dependencias. Ambos te explican el patrón de diseño de software orientada a objetos, sobre dos enfoques diferentes.

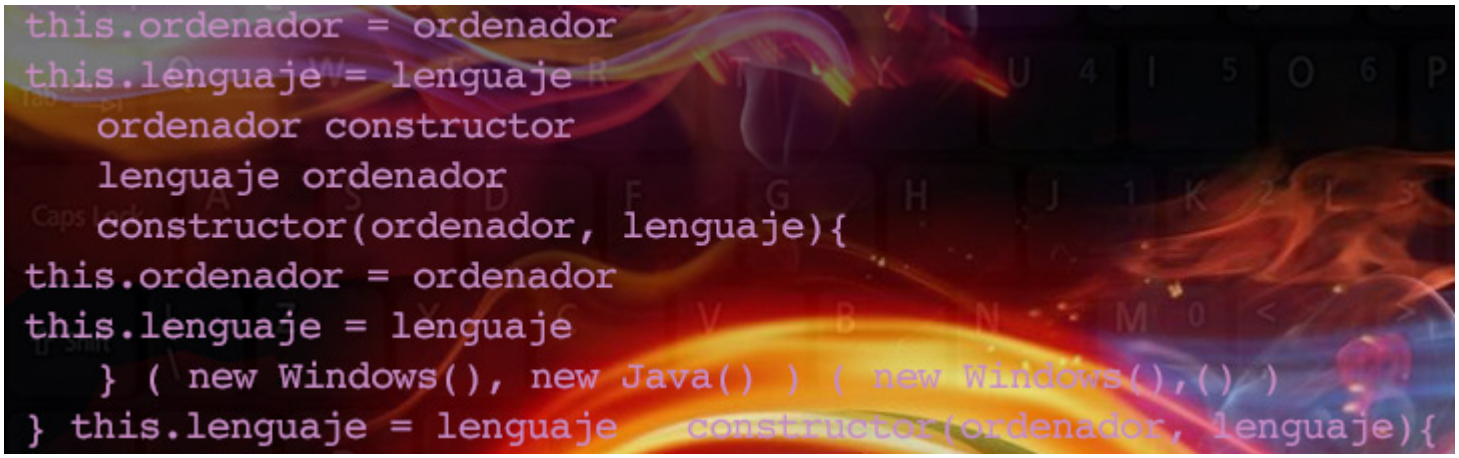
1.- La primera parte del artículo es más nueva, escrita en 2015 por Miguel Angel Alvarez. Relata en rasgos generales el patrón de inyección de dependencias y contenedor de dependencias. Aunque está inspirada en el uso que se le da al patrón en la comunidad de desarrolladores de PHP y en numerosos frameworks de desarrollo, pretende abarcar el conocimiento básico en cualquier lenguaje de programación orientado a objetos.

2.- La segunda parte es un artículo más antiguo, publicado en 2011, pero que sigue de actualidad porque el patrón en sí no ha sufrido cambios conceptuales. La segunda parte la escribe José Miguel Torres y relata la Inyección de dependencias enfocada en la programación en .NET.

Introducción

La inyección de dependencias es un patrón de diseño de software usado en la Programación Orientada a Objetos, que trata de solucionar las necesidades de creación de los objetos de una manera práctica, útil, escalable y con una alta versatilidad del código.

En la mayoría de los frameworks actuales se aplica la Inyección de dependencias como parte de las herramientas y modelos que facilitan al programador. Como cualquier patrón de diseño de software trata de solucionar de una manera elegante un problema habitual en el desarrollo de software, por lo que también es idóneo utilizar este patrón en el desarrollo de proyectos a pequeña escala.



Qué es la inyección de dependencias

Aparte de un patrón de diseño de software, vamos a explicar qué idea hay detrás de ese nombre. Este patrón, como muchos otros, nos ayuda a separar nuestro código por responsabilidades, siendo que en esta ocasión sólo se dedica a organizar el código que tiene que ver con la creación de los objetos.

Como ya sabemos, uno de los principios básicos de la programación, y de las buenas prácticas, es la separación del código por responsabilidades. Pues la inyección de dependencias parte de ahí.

En el código de una aplicación con OOP (Programación Orientada a Objetos) tenemos una posible separación del código

en dos partes, una en la que creamos los objetos y otra en la que los usamos. Existen patrones como las factorías que tratan esa parte, pero la inyección de dependencias va un poco más allá. Lo que dice es que los objetos nunca deben construir aquellos otros objetos que necesitan para funcionar. Esa parte de creación de los objetos se debe hacer en otro lugar diferente a la inicialización de un objeto.

Por ejemplo, este código no sería el mejor:

> **Nota:** Voy a escribir con pseudocódigo porque la inyección de dependencias realmente sirve para cualquier lenguaje y verdaderamente no importa aquí el código sino entender el concepto.

```
class programador{
 ordenador
 lenguaje
 constructor(){
  this.ordenador = new Mac()
  this.lenguaje = new ObjectiveC()
 }
}
miguel = new Programador()
```

El problema que nos encontramos es que la clase Programador está fuertemente acoplada con la el ordenador Mac o el lenguaje ObjectiveC. Si mañana queremos tener programadores de C que usan Windows, tal como está el código, tendríamos que crear una nueva clase Programador, porque esta no nos valdría.

Ahora veamos esta alternativa de código mucho más versátil.

```
class programador{
 ordenador
 lenguaje
 constructor(ordenador, lenguaje){
  this.ordenador = ordenador
  this.lenguaje = lenguaje
 }
}
miguel = new Programador( new Mac(), new ObjectiveC() )carlos = new Programador( new Windows(), new Java() )
```

Ahora nuestro programador es capaz de adaptarse a cualquier tipo de ordenador y cualquier tipo de lenguaje. De hecho observarás que hemos podido crear un segundo programador llamado "carlos" que es capaz de programar en Java bajo Windows.

Claro que esto es solo un ejemplo ridículo pero si has podido apreciar la diferencia, podrás entender el resto que viene detrás del concepto de inyección de dependencias. En realidad es tan sencillo como apreciar que al constructor de los objetos se les están pasando aquellas dependencias que ellos tienen para poder realizar sus tareas.

El hecho en sí, de enviar por parámetros los objetos que son necesarios para que otro objeto funcione, es la inyección de dependencias.

Contenedor de dependencias

El código que has visto anteriormente es muy sencillo, pero en estructuras más complejas observarás que hay muchos objetos que dependen de otros objetos. Esos otros objetos a su vez dependen de otros, que dependen de otros. Como has observado, dado el patrón de inyección de dependencias, necesito tener listos todos los objetos de los que depende el que voy a construir, porque se los tengo que enviar por parámetro al constructor.

Por ejemplo, el programador depende del ordenador y los lenguajes, pero el ordenador depende del sistema operativo y el teclado y ratón. A su vez el teclado depende de una conexión USB y de un conjunto de teclas, la conexión USB depende de las líneas de comunicación de datos y de la electricidad, la electricidad depende de que hayas pagado tu factura el mes anterior y de que así haya tensión en la red. Así podríamos continuar hasta donde nos lleve la imaginación.

Todo eso nos indica que, para conseguir instanciar un programador necesito haber instanciado antes la red eléctrica y las líneas de comunicación del USB, la conexión USB, cada una de las teclas del teclado, el teclado, el ratón, el sistema operativo, el ordenador, los lenguajes... y cuando tengo todo eso, por fin puedo invocar al constructor de la clase Programador para obtener el objeto que quería.

¿Complicado? No. Pero sí es laborioso. Quizás tengas una docena de líneas de código, o más, para poder hacer lo que tú querías, que era instanciar un programador al que necesitas inyectarle todas las dependencias, conforme nos dicta el patrón.

La solución a esta problemática nos la trae el contenedor de dependencias, también llamado inyector de dependencias, contenedor de servicios, etc.

Básicamente es como una caja a la que le pido construir las cosas. Esa caja sabe qué tiene que hacer para construir cada uno de los objetos de la aplicación. Si queremos un programador, simplemente le pedimos al contenedor de dependencias que nos cree un objeto de esa clase y él se dedica a crearlo. Finalmente el "dependency container" lo devuelve listo para ser usado.

```
miguel = contenedorDependencias.crear("Programador");
```

El contenedor de dependencias nos permite que la instanciación de un objeto, por muchas dependencias que tenga, vuelva a ser tan simple como una llamada a un método. ¿Dónde está la magia? el contenedor de dependencias simplemente tiene todos los objetos que puedas necesitar para crear cualquier objeto complejo y si no cuenta en ese instante con las dependencias necesarias, sabe cómo conseguirlas en el acto.

El uso de este contenedor de dependencias ya depende del lenguaje que estés usando y la librería o framework con la que trabajes. Además habitualmente hay que hacer algunas configuraciones básicas para que funcione. Si te fijas, en la línea anterior estoy diciendo que me cree un programador, pero en algún lugar le tendré que decir cómo puedo obtener ese objeto deseado y si tal como está ese programador será experto en Java, ObjectiveC o PHP.

> **Nota:** Si nuestro código está correctamente escrito es muy probable que las configuraciones sean mínimas, porque podrá decidirlo a través del análisis de los tipos de datos indicados en las cabeceras de los métodos constructores, pero generalmente hay que configurar algunas cosas básicas.

En futuros artículos experimentaremos con alguna librería para implementar de manera sencilla un contenedor con el que poner en marcha este patrón de inyección de dependencias.

Inyección de dependencias enfocada en el desarrollo en .NET

A continuación te ofrecemos la segunda parte de este artículo sobre la inyección de dependencias, en la que vamos a conocer más detalles sobre este patrón de diseño de software, particularmente enfocada en el mundo del .NET.

Introducción

Si nos remontamos a los primeros años de la programación, nos encontraremos con programas rígidos repletos de código monolítico y lineal. La propia evolución hizo aparecer conceptos hoy por hoy imprescindibles como la modularidad y la reutilización de componentes, conceptos fundamentales en el paradigma de la Programación Orientada a Objetos.

Figura 1

La modularidad y reutilización de clases conlleva un flujo de comunicación entre instancias cuyo mal uso deriva en un hándicap que limita la flexibilidad, robustez y reusabilidad del código debido a la dependencia o alto acoplamiento entre las clases.

En la figura 1 podemos ver un sencillo diagrama de clases de un sistema de adquisición y control de datos meteorológicos. Existen dos clases participantes: una para la captura de la temperatura, y otra que representa a la estación meteorológica. Ambas tienen una responsabilidad a la hora de mostrar los datos, como puede apreciarse en el listado 1.

```
public class EstacioMeteorologica
{
public void MostrarDatos()
{
Console.WriteLine(
string.Format("Datos a {0} n", DateTime.Now));
Termometro termometro = new Termometro();
termometro.MostrarTemperaturaActual();
}
}
public class Termometro
{
public int Valor { get; set; }
public void MostrarTemperaturaActual ()
{
Console.WriteLine(
string.Format("Temperatura: {0} °", Valor));
}
```

```
}
```

Identificando el problema

Cuando hablamos en términos de calidad, solemos utilizar los adjetivos "bueno" o "malo" para definir la calidad de un diseño. Sin embargo, no siempre utilizamos los argumentos o criterios que sustentan la afirmación "éste es un mal diseño". Existe un conjunto de criterios más allá del siempre subjetivo TNTWIWHDI (Thats Not The Way I Would Have Done It, "Yo no lo habría hecho así") acuñado por Robert C. Martin, y son los que miden el nivel de rigidez, la fragilidad y la inmovilidad del sistema.

En nuestro ejemplo de la estación meteorológica, podemos afirmar que el diseño es rígido, porque cualquier cambio será difícil de llevar a cabo, ya que no conocemos el impacto que la modificación de una clase de bajo nivel (clase Termometro) tendrá sobre la clase de alto nivel (clase EstacioMeteorologica).

Cuando los cambios tienen una repercusión en otras entidades, no necesariamente dependientes, se dice que un sistema o aplicación es frágil. Si nos fijamos en el listado 1, la clase EstacioMeteorologica depende tanto de Termometro como de System.Console. Un cambio del flujo de salida de datos del programa (por ejemplo, a una impresora en lugar de System.Console) repercutiría en las clases de bajo nivel.

El termino inmovil lo utilizamos para medir el nivel de dependencia entre una parte del diseno y otros datos no directos. El ejemplo es inmovil porque la clase Estacio] Meteorologica depende de las clases Termometro y System. Console para mostrar los datos. Dicho en otras palabras, no podriamos extraer la clase de mayor nivel y utilizarla con otras entidades. Lo mismo pasaria con la clase de bajo nivel por su dependencia de System.Console.

> **Nota:** Entre los criterios que permiten determinar si un diseño es bueno o malo están los que miden su nivel de rigidez, fragilidad e inmovilidad.

Planteemos un nuevo diseño a nuestro sistema. En primer lugar, eliminemos la dependencia que la clase Termometro tiene de System.Console, ya le que estamos otorgando la responsabilidad de salida por pantalla cuando realmente no le corresponde. El resultado sería el que se muestra en el listado 2.

```
public class EstacioMeteorologica
{
public void MostrarDatos()
{
Termometro termometro = new Termometro();
string temperatura =
termometro.MostrarTemperaturaActual();
Console.WriteLine(
string.Format("Datos a {0} n{1}",
DateTime.Now, temperatura));
}
}
public class Termometro
{
public int Valor { get; set; }
public string MostrarTemperaturaActual ()
{
return string.Format("Temperatura:{0} º", Valor);
}
}
```

Ahora la clase Termometro ha quedado libre de dependencias, y por tanto es reutilizable. Sin embargo, aún EstacioMeteorologica depende tanto de System.Console como de Termometro. Por otro lado, la clase Termometro no es más que una representación de un valor referencial meteorológico cualquiera; por tanto, podríamos abstraer la interfaz

IMeteoReferencia, tal y como se muestra en el listado 3, y hacer que la clase Termometro la implemente. Esto es un ejemplo de aplicación del patrón Fachada (Façade), mediante el cual simplificamos la firma de varias clases a través de una única interfaz.

```
public interface IMeteoReferencia
{
int Valor { get; set; }
string Mostrar();
}
public class Termometro : IMeteoReferencia
{
public int Valor { get; set; }
public string Mostrar()
{
return string.Format("Temperatura:{0} º", Valor);
}
}
```

Ahora que hemos abstraído la interfaz, ésta nos servirá como contrato para las clases que quieran utilizarla. Esto nos permitirá desacoplar la clase EstacioMeteorologica de Termometro, tal y como muestra el listado 4.

```
public class EstacioMeteorologica
{
private IMeteoReferencia termometro;
public EstacioMeteorologica()
{
termometro = new Termometro();
}
public void MostrarDatos()
{
Console.WriteLine(
string.Format("Datos a {0}", DateTime.Now));
Console.WriteLine(termometro.Mostrar());
}
}
```

Sin embargo, aún no hemos solucionado el problema, pese a que estamos más cerca. Lo que pretendemos es eliminar completamente la instanciación de la clase Termometro, y la solución pasa por inyectar la dependencia directamente a través del constructor, como se muestra en el listado 5.

```
public class EstacioMeteorologica
{
private IMeteoReferencia termometro;
public EstacioMeteorologica(
IMeteoReferencia termometro)
{
this.termometro = termometro;
}
public void MostrarDatos()
{
Console.WriteLine(
string.Format("Datos a {0}", DateTime.Now));
Console.WriteLine(termometro.Mostrar());
}
}
```

El Principio de Inyección de Dependencias

Robert C. Martin afirma en el Principio de Inyección de Dependencias:

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.*

Imaginemos por un momento la solución inicial de la estación meteorológica (listado 1). La clase de alto nivel EstacioMeteorologica depende de la clase de bajo nivel Termometro (o Barometro, Anemometro, etc.). Toda la lógica de la solución se implementaría en la clase de alto nivel, y cualquier modificación en las clases de bajo nivel tendría repercusión no únicamente sobre la definición de la clase de alto nivel, sino sobre la propia lógica de la aplicación, llegando incluso a forzar cambios en la misma, cuando debería ser la clase de alto nivel la que debería forzar el cambio a las clases de bajo nivel sin comprometer la lógica de la aplicación; es decir, justamente lo contrario. Además, la clase de alto nivel sería difícilmente reusable debido a este acoplamiento. Sencillamente, y resumiendo, la clase EstacioMeteorologica no debe depender de la clase Termometro; en todo caso, al contrario.

Existen tres formas de implementación de la Inyección de Dependencias:

- por constructor
- por setter
- por interfaz.

El primer caso lo hemos visto en la sección anterior, donde hemos inyectado la dependencia a través del constructor de la clase; el listado 6 muestra una generalización. La inyección por setter se realiza a través de una propiedad de la clase (listado 7); y por último, la inyección por interfaz se realiza a través de un método, recibiendo como parámetro el objeto a inyectar (listado 8).

```
IMeteoReferencia referencia = ObtenerReferencia();
EstacioMeteorologica estacion =
new EstacioMeteorologica(referencia);

EstacioMeteorologica estacion = new EstacioMeteorologica();
estacioMeteorologica.Referencia = ObtenerReferencia();

EstacioMeteorologica estacion = new EstacioMeteorologica();
estacioMeteorologica.LecturaContador(ObtenerReferencia());
```

Inversión de control y contenedores

No podemos hablar de DI sin dejar de hablar de la Inversión de control (Inversion of Control, IoC). IoC también es conocido como Principio de Hollywood, nombre derivado de las típicas respuestas de los productores de cine a los actores noveles: "no nos llames; nosotros lo haremos".

IoC invierte el flujo de control de un sistema en comparación con la programación estructurada y modular. En el fondo, DI es una implementación de IoC. Aún hoy existe la discusión acerca de si IoC es un principio, un patrón o ambas cosas a la vez. IoC, en definitiva, es una característica fundamental de un framework, y de hecho lo que lo hace realmente diferente a una librería de funciones.

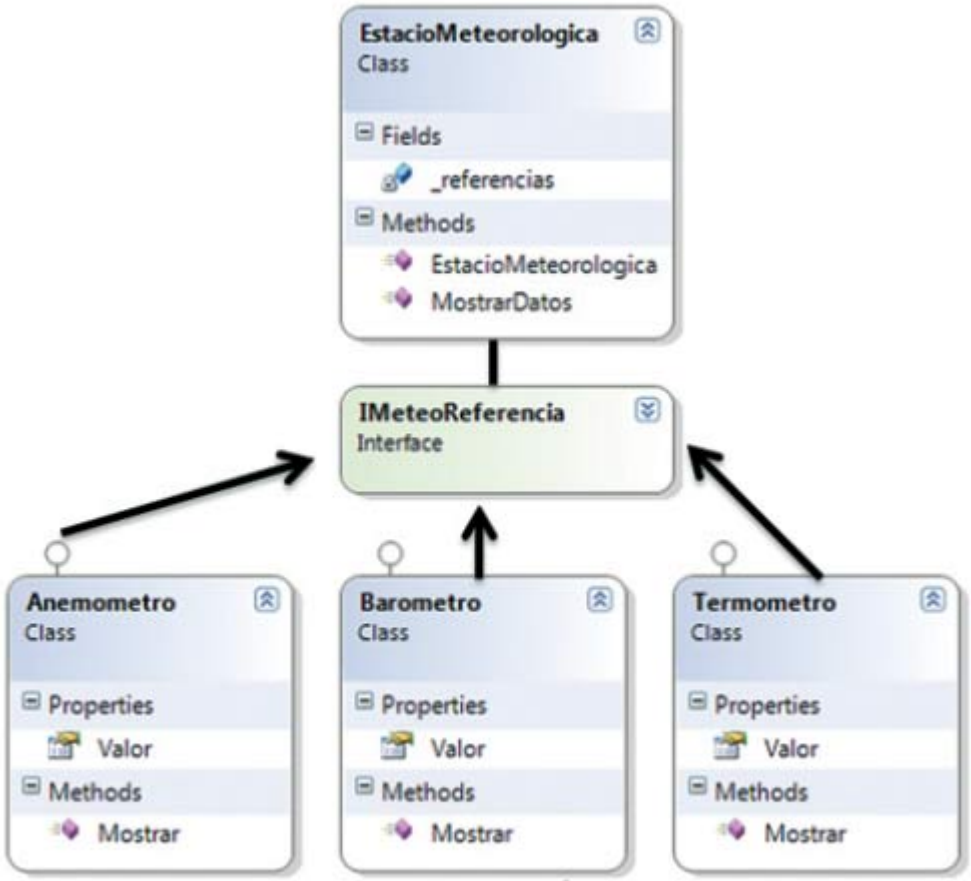


Figura 2

En escenarios de producción, las clases no son tan triviales como la que hemos presentado en este artículo. Imagine por un momento que la interfaz IMeteoReferencia tiene una implementación de IEntradaDatos e IVerificador, y éstas a su vez implementan otras interfaces. En realidad, obtendremos una jerarquía de dependencias (figura 3), cuyo manejo en tiempo de diseño es imposible de gestionar "manualmente"; es aquí donde entra a jugar el término contenedor IoC (IoC Container).

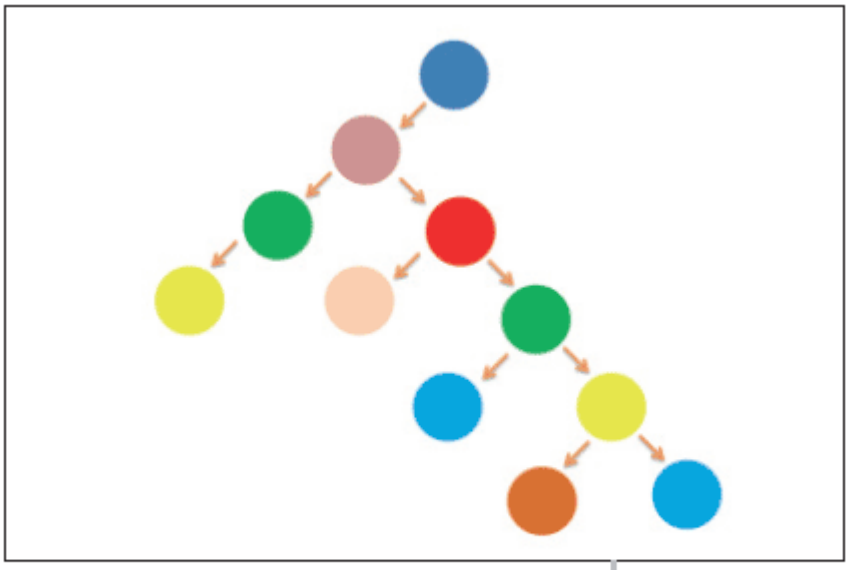


Figura 3.Diagrama de jerarquía de dependencias.

El principal cometido de un contenedor IoC, a diferencia de una factoría, es el de gestionar el ciclo de vida de los objetos. El contenedor IoC registra una implementación específica para cada tipo de interfaz y retorna una instancia de objeto. Esta resolución de objetos tiene lugar en un único punto de las aplicaciones; normalmente, a nivel de infraestructura.

Conclusión

Con este artículo, hemos tratado de mostrar de una forma práctica la relación existente entre dependencias, detalles y abstracciones. Con el Principio de Inyección de Dependencias, ponemos fin a la última de las siglas que componen SOLID. Existen libros íntegros que hablan de este principio, y podrá encontrar en Internet una gran cantidad de recursos relacionados.

A lo largo de esta serie sobre los principios SOLID, hemos presentado aspectos muy importantes que debemos tener en cuenta ante cualquier nuevo desarrollo, y hemos visto cómo muchas de las problemáticas lógicas del diseño pueden ser reducidas mediante la aplicación de estos principios. Trate de entender cada uno de los principios desde un punto de vista práctico. Algunos de ellos (y lo digo por experiencia) son realmente complejos de llevar a la práctica; recuerde además que son principios, no reglas.

Para finalizar, agradecer a Hadi Hariri, quien me ha servido de "enciclopedia de consulta" para esta serie, por su apoyo y ayuda en todo momento.

Por Miguel Angel Alvarez

2.4.- Análisis y Diseño Orientados a Objetos: Por qué, cuándo y cómo

Todo programador debe conocer el Análisis y Diseño Orientados a Objetos. Resumimos el objetivo, la motivación y la necesidad de dominar estas áreas, si te dedicas a la programación profesional.

En este artículo, y en el vídeo que acompaña al final del texto, estamos haciendo un informe sobre los motivos por los que todo programador debería dominar el Análisis y Diseño Orientados a objetos, así como un resumen de los contenidos que deberían conocerse para abarcar estas materias.

El objetivo es, sobre todo, motivar a los programadores a realizar un estudio más profundo acerca de su profesión, lo que debe derivar en un trabajo más correcto y satisfactorio tanto para el profesional como para los proyectos en los que se encuentre involucrado.

En resumen veremos todas las cuestiones relacionadas con estas materias que vamos a abordar: **Análisis y Diseño de Software**, ¿Quién debería conocerlas? ¿Para qué?, ¿Por qué?, ¿Qué incluye?, ¿Cómo se debe abordar el estudio?, ¿Dónde y cuándo?, etc.

Con todo esperamos que los profesionales tengan más claro aquellos puntos y guías para poder avanzar en sus capacidades como programador.

> **Nota:** Este artículo es una transcripción resumida de algunas partes del contenido de la clase impartida por Luis Fernández, profesor de la Escuela Superior de Informática de la Universidad Politécnica de Madrid. La grabación de la clase se puede encontrar al final del artículo. Este texto de transcripción lo ha realizado Miguel Angel Alvarez.

Objetivo de aprender Análisis y Diseño

El software no es como un bebé que entregas en adopción después de dar a luz. Al contrario, es más como tener un hijo en la vida real a lo largo de muchas fases:

1. Durante la gestación la madre se cuida para que el bebé nazca bien, no tenga ninguna deformación o problemas de nacimiento que dificulte una vida normal.

2. Además el software debe acompañarse durante la vida de la persona: durante la vida se debe colaborar para que el niño pueda desarrollarse, crecer, jugar, madurar, trabajar...

Las tasas mayores de éxito del software se tiene justamente cuando se crea una primera versión con muy pocos requisitos y luego se va ampliando su funcionalidad al cabo del tiempo. Eso lo hemos visto en cientos de productos de software, en los que hemos asistido a lo largo de los años a una sofisticación de un programa, según se iban lanzando distintas versiones, muchas veces publicando solo cambios sutiles entre ellas.

Por tanto, para un mejor desarrollo del software es importante esforzarse en alcanzar una elevada calidad. Esto significa crear unas buenas clases, jerarquías, dependencias, etc. y conocer los parámetros que nos indican justificadamente si una clase está bien o mal, si un método es correcto o mejorable, etc.

Al encontrar esa bondad en los componentes del software conseguimos un software legible y no uno viscoso, software flexible en lugar de rígido, robusto en lugar de frágil y reusable en lugar de inmóvil. Buscamos un software cuyas clases,

métodos, jerarquías, herencias nos ofrezcan esas ventajas deseables en todo programa, ya que a lo largo de los años vamos a tener que mantener el software, ya sea para corregirlo, aumentar sus funcionalidades o adaptarlo a nuevas necesidades.

El motivo de buscar ese software correcto es por aumentar la productividad y la eficiencia, no por estética. Pero además y sobre todo es simplemente por ética profesional, lo que se conoce por deontología.

¿Para qué voy a estudiar análisis y diseño de software?

Debemos Realizar un desarrollo con diseño de mayor calidad calidad. Aparte, para permitirme evolucionar correctamente en el campo de la programación.

Esto lo podemos ver por medio de una serie de etapas que todo programador va cubriendo en su desarrollo profesional.

Etapa 1: Aprender programación
Etapa 2: Aprender programación orientada a objetos

Estas 2 primeras etapas, son en sí un primer paso, es algo que todos conocemos o debemos de conocer ya. De hecho, es posible que muchas personas las hayan podido realizar ambas etapas a la vez, puesto que muchas personas pueden haber aprendido a programar directamente con orientación a objetos y no hay problema alguno en ello.

Etapa 3: Análisis y diseño

Esta etapa es fundamental para poder desarrollarse profesionalmente como programador y es a la que nos estamos refiriendo en este artículo. Sin embargo, a pesar de su importancia, muchísima gente se las ha saltado, yendo directamente a las siguientes etapas.

Queremos insistir en que el análisis y diseño es la lanzadera a las siguientes etapas de la programación, porque permite justificar por qué un código está bien y un código está mal. Cuando lo tengas bien claro, podrás pasar a las siguientes etapas con garantías:

Etapas 4 y en adelante: Patrones de diseño, arquitecturas de software, pruebas, TDD, Ecosistema (workflow), metodologías, tecnologías…

El problema es que mucha gente, como decimos, después de aprender programación orientada a objetos, ha pasado directamente a aprender escalones superiores como por ejemplo patrones de diseño, sin pasar por el dominio del análisis y diseño. Aplican un patrón de diseño sin saber realmente cuándo se debe aplicar y si es correcta o necesaria su aplicación. "Saben patrones" (entre comillas) pero no aciertan a aplicarlos correctamente.

Para poder aprender patrones de diseño, o sabes los conceptos generales de análisis y diseño (etapa 3), o lo más seguro es que metas un patrón donde no debías aplicarlo. Lo más seguro es que hagas un software sin arquitectura, o no apliques las arquitecturas correctamente porque no las has podido entender bien. Te incapacites a hacer pruebas automatizadas. Que tengas herramientas habituales para un buen ecosistema (el Workflow de procesos y herramientas, como control de versiones, métricas, check style…) pero no les saques el partido que debes, y de ahí en adelante.

¿Quién debe conocer el análisis y diseño orientados a objetos?

Esta parte la veremos analizando el perfil del estudiante que debería tener claras estas disciplinas (Diseño y Análisis). Es una persona que conoce lenguajes de programación orientados a objetos, como Java, PHP, C#, C++, Ruby o similares. En general cualquier lenguaje que implemente el paradigma de orientación a objetos.

Además debe tener claros los conceptos básicos del mundo de la programación: Tipos primitivos, arrays, punteros y/o referencias, sentencias de control, parámetros y argumentos, acumuladores y flags.

Luego tendrá claros conceptos relacionados con la programación orientada a objetos como clases, atributos, métodos, enumerados. Miembros de instancia Vs miembros de clase (static). Objetos, estado, mensajes, así como herencia, polimorfismo, programación parametrizada, excepciones y aserciones.

Qué contenido debes conocer para dominar análisis y diseño orientados a objetos

Primero tener un contexto teórico, conocer la naturaleza, fundamentos y complejidad del software, economía y crisis del software, lo que son los patrones de diseño y las aquitecturas del software, así como disciplinas y metodologías.

Después de todo ese conocimiento teórico nos vamos al contenido real que debes dominar para el aprendizaje y aquí hay una cantidad enorme de principios, patrones, métricas de calidad, etc. que deberías estudiar y que ahora te enumeramos y resumimos.

Principios DRY, YAGNI, KEEP, Mínima Sorpresa,...
Patrones GRASP (General Responsabilities Assigment Software Patterns): Alta cohesión, bajo acoplamiento, Controlador...
Relaciones entre clases: Composición / agregación, asociación...
Código limpio: Formato, nombrado, comentarios...
Principios SOLID: (Sustitución de Liskov, Abierto / Cerrado de Meyer, Inversión de dependencias...)
Diseño por contrato: precondiciones, postcondiciones...
Métricas de calidad: tamaño de clases, métodos, acoplamiento...
Código sucio: números mágicos, demasiados argumentos...
Anti patrones de desarrollo: BLOB, código muerto...

Todo esto son las materias de estudio de los libros de informática más representativos de los últimos años. Materias que generalmente no se explican en los ciclos universitarios, más concretamente en el "grado universitario" y que solo se abordan para los pocos estudiantes del "post grado". Pero a la vez, son materias esenciales para que las personas que se van a dedicar a la programación puedan asentar el conocimiento posterior, sin dar tumbos.

Sobre todo este contenido de Análisis y Diseño Orientados a Objetos, si te interesa (que debería), vamos a hacer un curso en EscuelaIT a partir de este lunes, con Luis Fernández. Serán 2 semanas de clases diarias, a razón de 2 horas por día. En total están planificadas un mínimo de 20 horas de clase, ya que en la práctica seguro que nos extendemos algo más, en función de la marcha de las sesiones y la cantidad de preguntas que se realicen durante las clases en directo. Puedes encontrar más información en la página del Curso de Análisis y Diseño Orientados a Objetos de EscuelaIT.

Vídeo de la sesión

En este artículo hemos resumido los primeros 25 minutos de la sesión que Luis nos ofreció en abierto antes del curso de EscuelaIT. Sin embargo, como contenido se tocaron muchos otros temas relacionados, como ingeniería del software, una completa bibliografía de los libros más fundamentales que se pueden usar para documentarse en la materia, mecánicas y objetivos del curso de EscuelaIT, etc.

Te recomendamos ver este vídeo porque es tremendamente inspirador. Si te dedicas a la programación deberías invertir un poco de tiempo, pues merece mucho la pena, ya sea para conocer los siguientes objetivos que deberías plantearte en tu carrera profesional o para verificar si te encuentras en el buen camino.

Por Luis Fernández Muñoz