**Bases de POO**

* Abstracción: extracción de características fundamentales y esenciales de algo.
* Encapsulación: proceso donde se ocultan detalles del soporte de características esenciales.
* Modulación: descomposición de un sistema en piezas o conjuntos.
* Jerarquización: proceso de estructuración por el que se produce una organización de un conjunto en niveles de responsabilidad, incumbencias o composición de otros. Que programa llama a cuál otro. Como se organizan los módulos. En POO las forman las clases, unas se apoyan en otras ya sea mediante herencia o descomposición.

Acoplamiento --> (encapsular reduce los acoplamientos). Poco acoplado es que las piezas no influyen tanto en otras en la modulación. Ose si cambia algo de una no se rompe todo el programa. (Siempre existe).

Cohesión: en un sistema bien modulado implica que sus módulos deben ser entendibles por si mismos.

EL coche es el elemento principal (OBJETO) que tiene características, como el color o la marca (PROPIEDADES O ATRIBUTOS) y además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar (MÉTODOS).

Clases en POO

Las CLASES son declaraciones, abstracciones de objetos. La definición de un objeto en una clase. Cuando programamos un objeto y definimos sus atributos y métodos en realidad estamos programando una clase.

Las propiedades o atributos son las características de los objetos. Son las variables donde almacenamos los datos relacionados con los objetos.

Los métodos son las funcionalidades asociadas a los objetos. Son como funciones asociadas a un objeto.

Los objetos son ejemplares de las clases. Cuando creamos un objeto 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. Un objeto de la clase fracción es por ejemplo 3/5. El concepto o definición de fracción sería la clase pero cuando ya hablamos de una fracción en concreto como 3/7 la llamamos objeto.

Var miCoche = new Coche()

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

Cuando tenemos un objeto sus propiedades toman valores. Por ejemplo, cuando tenemos un coche el atributo color tomará un valor como rojo o gris. El valor concreto de un atributo de un objeto se llama ESTADO.

miCoche.color = rojo

Un MENSAJE en un objeto es la acción de efectuar una llamada a un método.

miCoche.ponerseEnMarcha()

La HERENCIA sirve para crear objetos que incorporen propiedades y métodos de otros objetos para poder construir objetos a partir de otros sin tener que rescribir todo el código.

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.

**Static**

Los elementos estáticos 'STATIC' también llamados elementos de clase son modificadores 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 al objeto. Cuando defines atributos o métodos solo puedes acceder a ellos a través de un objeto, es la creación de una instancia la que permite el acceso. En cambio, en los elementos static no necesitamos haber creado ningún objeto de esa clase porque en vez de acceder a través de un objeto, accedemos a través del nombre de la clase.

Ejemplo atributo estático: Clase Autobús con el atributo estático precio ya que el precio es igual para todos.

Ejemplo método estático: Clase Fecha con el método estático validación. No hace falta crear una fecha para saber si es válida.

Generalmente se utiliza el modificador "static" para definir los miembros de clase.

Class AutousMetropolitano {

Public static precio = 1.20;

}

Se tiene que asignar un valor a la hora de declarar un atributo estático ya que no se le puede asignar valor a través del constructor.

Para acceder a los métodos y atributos estáticos fuera de la clase se hace de la siguiente forma:

AutobusMetropolitano.precio -->con el nombre de la clase y no del objeto.

En muchos lenguajes podés acceder a esos atributos (o métodos) desde la vista privada de la clase (código de implantación de la clase) a través de la palabra "self" (esto tanto para los atributos como los métodos). Self es para la clase, this es para la instancia.

Class AutobusMetropolitano{

Static precio = 1.20;

Public function aceptarPago(dinero){

If(dinero < self.precio){

Return false

}

//…

}

}

En ocasiones se tiende a usar atributos estáticos para agregar datos a modo de variables globales. Esto nos puede acarrear exactamente los mismos problemas que conocemos por el uso de variables globales en general. También se suelen utilizar métodos estáticos como simples contenedores de unión a diferentes métodos estáticos. Por ejemplo, la clase Math en Javascript 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 POO sin aprovechar los beneficios que nos aporta.

**Herencia**

Puedo tener diversos tipos de jerarquías, como la clasificación o composición.

Composición: es cuando unos elementos están compuestos de otros. Por ejemplo, el sistema respiratorio está compuesto por los pulmones y a su vez los pulmones están compuestos por bronquios y alvéolos.

Clasificación: unos elementos son una especialización de otros. Por ejemplo, los animales, donde tenemos vertebrados, reptiles, mamíferos, dentro de los mamíferos hay vacas, perros, humanos.

Con lo visto en POO ya podemos tener jerarquización de composición, pero necesitamos el concepto de herencia para poder aplicar clasificación.

La HERENCIA es el método fundamental para reutilizar código. La herencia es la transmisión de código de una clase a otra, la clase padre y la clase hija. En muchos lenguajes de programación se declara la herencia con la palabra "extends"(en C# se utiliza dos puntos : ).

Class Hija extends Padre{}

Ejemplo, tenemos alumnos universitarios. Algunos alumnos son normales, otros de intercambio, otros becados. Probablemente tendremos una clase Alumno con una serie de métodos como asistir\_a\_clase(), hacer\_examen() etc, que son comunes en todos los alumnos, pero hay operaciones que son diferentes en cada tipo de alumno como pagar\_mensualidad() o matricularse().

Para saber si está correcto emplear herencia entre unas clases y otras, plantéate la pregunta, ¿Clase hija es una clase padre? (por ejemplo, ¿un perro es un mamífero?

**Polimorfismo**

El POLIMORFISMO es una relajación de los sistemas de tipos, de tal manera que una referencia a una clase acepta direcciones de objetos de dicha clase y de sus clases derivadas. Veremos que polimorfismo y herencia son dos conceptos estrechamente ligados.

¿Pero que es una SISTEMA DE TIPOS?

Existen los lenguajes FUERTEMENTE TIPADOS (TIPADO ESTÁTICO) como Java o C y los DEBILMENTE TIPADOS (TIPADO DINÁMICO) como PHP o Javascript.

En los lenguajes fuertemente tipados es necesario al declarar una variable aclarar el tipo de dato que va a contener esta variable. Esto incluso pasa con los objetos cuando defino un nuevo objeto tengo que definir de que clase va a ser. Ejemplo:

Largometraje miLargo = new Largometraje('Lo que el viento se llevo");

Si tengo una Clase Vehículo con clases hijas moto, auto y autobús y creo un nuevo objeto Coche esta variable nunca podrá ser de clase moto o autobús. Esta rigidez no existe en PHP o Javascript.

Por ende, en los lenguajes fuertemente tipados una clase padre Vehículos no podría contener un array con los objetos de clase auto, moto ó autobús. Tampoco se podría tener un método que acepte todos los hijos si por ejemplo tengo una clase padre Largometraje y un método reproducir() y tengo clases hijas Documental y Película entonces necesitaría tener distintos métodos para cada tipo de clase lo cual crea un problema ya que la idea de la herencia es poder reutilizar código.

Gracias al polimorfismo en nuestro array de Vehículos podrás contener elementos no solo de la clase Vehículo sino también de sus clases derivadas. La variable o método solo tiene que ser nombrada con la Clase padre y listo. Ejemplo;

Vehículo miVehiculo = new Coche('Ford Focus 2.0");

miVehiculo = new Moto('Yamaha YBR");

Y para métodos:

Public reproducir(Largometraje miLargo){

}

**Abstracción**

Una clase ABSTRACTA es aquella sobre la que no podemos crear especímenes concretos es decir es aquella sobre la que no podemos instanciar objetos. ¿Para qué sirven?

Ejemplo, si defines una clase Animal tienes que definir clases como perro, vaca o gallina porque nunca instanciaras un animal de por si, porque no existe un animal como tal. Por lo tanto, animal es un concepto abstracto que implementaremos por medio de una clase abstracta. No instanciaremos como animales como tal en mundo, sino que instanciaremos especímenes de un tipo de animal en concreto.

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. No sé cómo va a nacer un animal, pero sé que todos los animales van a nacer del algún modo, 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, pero que todavía no somos capaces de implementar.

Public abstract function nacer();

Esto quiere decir que todos los animales del mundo heredaran 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

}

"Una clase abstracta es aquella en la que hay definidos métodos abstractos, sobre la que no podremos instanciar objetos", "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".

Gracias a que fueron definidos métodos abstractos en las clases padres, cuando trabaje con elementos de la clase hija, sé 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. Esto me permite que pueda estar seguro de que todos los objetos que reciba puedan responder a acciones determinadas, (si no se definen deberían declararse las clases como abstractas y no podrían ser instanciadas).

**Patrones de Diseño**

Los patrones de diseño resuelven de manera práctica y elegante una infinidad de problemas habituales y concurrentes problemas a los que se enfrentan todo programador.

**MVC**

MVC, Model, View and Controller o Modelo, Vista y Controlador es un patrón de diseño que propone separar el código de los programas en sus diferentes responsabilidades.

MVC es una propuesta de diseño utilizada para implementar sistemas donde se requiere el uso de interfaces de usuario. Ha ganado mucha fuerza gracias a la aparición de frameworks que hacen uso del patrón para la arquitectura de aplicaciones web.

Ejemplo; Antes en HTML se mezclaba tanto el contenido como la presentación y si algún día queríamos cambiar la forma con la que se mostraba una página estábamos obligados a cambiar cada uno de los archivos HTML tocando todas y cada una de las etiquetas que hay en el documento. Con el tiempo se observó que no era práctico y se creó el lenguaje CSS en el que se separó la responsabilidad.

Al escribir un programa cualquiera de nosotros comienza mezclando tanto el código PHP como el código HTML e incluso Javascript. Esto produce lo que se denomina como "CÓDIGO ESPAGUETI" Sería mucho más útil que estuviera separado.

**Modelos**

Es la capa donde se trabaja con los datos. Contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos generalmente los tendremos en una base de datos. En los modelos tendremos las funciones que accederán a las tablas.

Cabe mencionar que cuando se trabaja con MVC lo habitual es también utilizar otras librerías de ORM (

Object-Relational mapping) mapeo objeto relacional, estas librerías nos permiten pasar de los datos que trabajan las bases de datos relacionales a objetos o viceversa, donde cada una de las columnas se convierte en una propiedad o atributo de un objeto. Aparte las librerías de ORM nos permiten cambiar de bases de datos sin tener que cambiar todo el código lo que cambiará será el adaptador o colectores. También ayuda a mejorar el desempeño y velocidad de las aplicaciones por ejemplo con el sistema de cacheo que trata de reducir el número de consultas que se hacen a una base o la carga retrasada que permite recuperar los datos solo cuando son necesarios.

**Vistas**

Las vistas contienen el código que 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 la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las visas requerirán los datos a los modelos y ellas generarán la salida.

**Controladores**

Contiene el código necesario para responder a las acciones que se solicitan en la aplicación. Es una capa que sirve de enlace entre los modelos y las vistas, 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 salida, sino servir de enlace.

Updates 
View 
Sees 
Model 
MVC 
User 
Manipulates 
Controller 
Updates 
danielmiessler.com 

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. 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. Pues en 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.

Lógica de aplicación 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 módulo 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. 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.

**Patrón Adapter**

El PATRÓN ADAPTER permite realizar una envoltura de una API para un desarrollo mantenible cuando usamos APIs de terceros. 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. Es como un "puente de acceso" a funcionalidades residentes en sistemas externos.

El problema cuando trabajas con una API que no controlas es que en cualquier momento te pueden cambiar ese API y con ello te obligan a cambiar tu código fuente. 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. Como dentro de todo proyecto solo hacías llamadas a tu "adapter" y no al API directamente, simplemente tendrás que editar unas pocas líneas de código, reescribiendo las operaciones de tu adaptador, para constatar los nuevos método, parámetros y uso en general de la nueva API de terceros.

**Inyección de dependencias**

La inyección de dependencias es un patrón de diseño de software usada en la POO, que trata de solucionar las necesidades de creación de los objetos de manera práctica, útil y escalable y con una lata versatilidad del código. Este patrón, como muchos otros, nos ayuda a separar nuestro código por responsabilidades.

En el código de una aplicación con POO tenemos posible separación de código en dos partes, una es 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.

class programador{

ordenador

lenguaje

constructor(){

this.ordenador = new Mac()

this.lenguaje = new ObjectiveC()

}

}

miguel = new Programador()

Vs.

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() )

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.