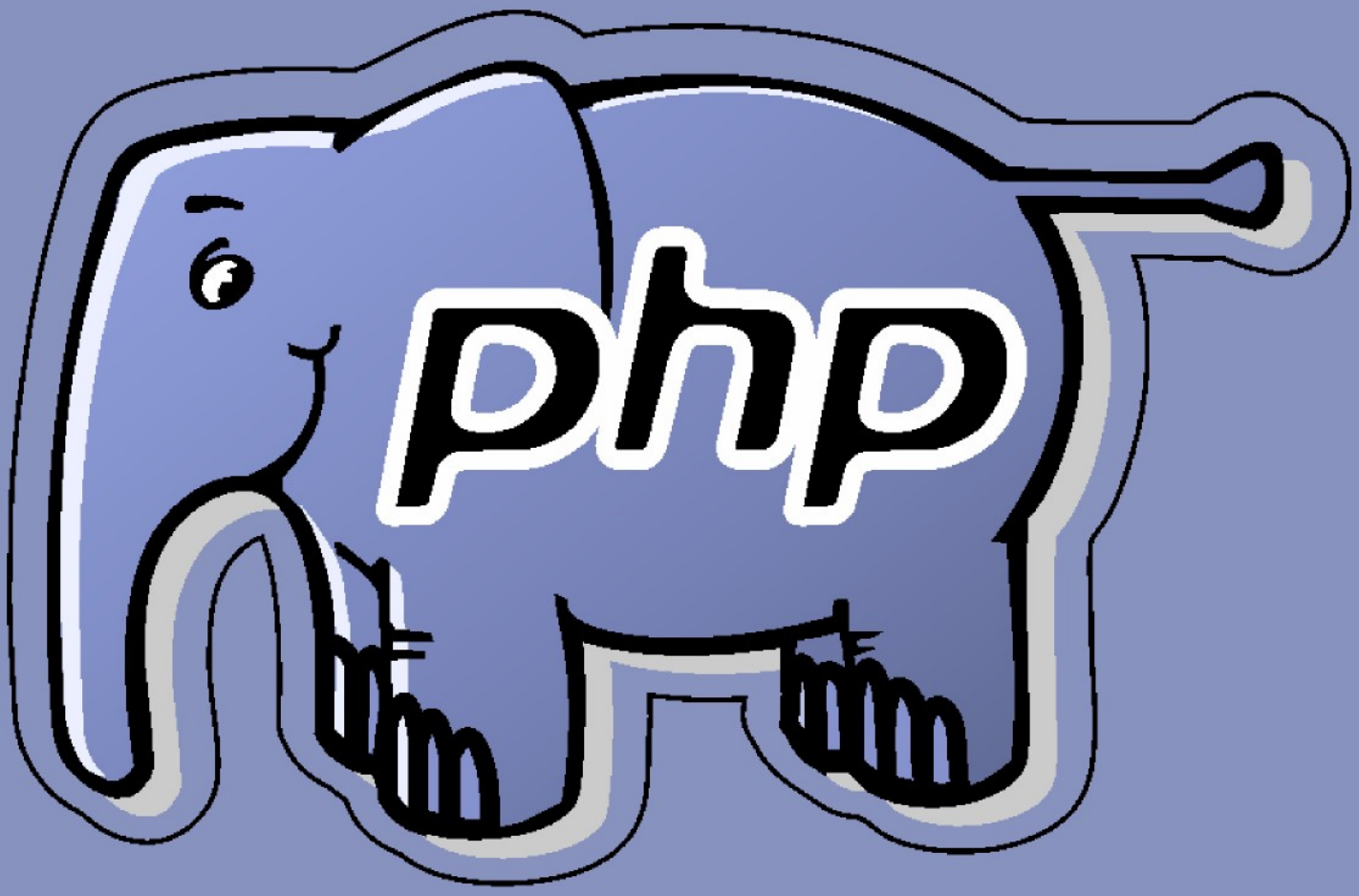
Programación en



Versión 2.0

php

Gugler
Laboratorio de Investigación Gugler

FCyT
Facultad de Ciencia y Tecnología

UADER
Universidad Autónoma de Entre Ríos

CONTENIDO

Capítulo 3

POO en PHP

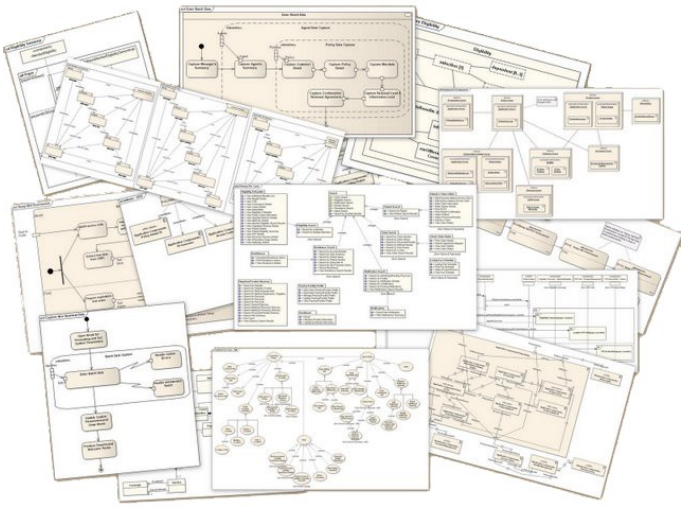
Introducción

Para continuar con nuestra segunda clase de POO en PHP, comenzaremos repasando algunos conceptos de UML lo que nos permitirá abarcar el modelado de desarrollo de nuestro sistema y la posterior representación de los distintos.

UML

¿Qué es?

UML (Lenguaje Unificado de Modelado) es un lenguaje estandarizado el cuál, mediante el uso de recursos gráficos, nos permite especificar, construir y documentar los elementos que forman parte de un sistema software orientado a objetos.



Collage de diagramas UML(Wikipedia)

Impulsado por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh, UML ofrece un estándar para describir un "plano" del sistema, incluyendo aspectos conceptuales tales como procesos de negocio y funciones del sistema, aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes reutilizables.

En UML tenemos distintos tipos de diagramas mediante los cuáles podremos representar todos los aspectos involucrados en el desarrollo. Si agrupamos los diagramas en distintas categorías, tendríamos:

1. **Diagramas de Estructura:** nos permiten analizar y reforzar cada uno de los elementos que deben existir en el sistema modelado. Los diagramas que podemos utilizar:
    - Diagrama de Clases.
    - Diagrama de Objetos.
    - Diagrama de Paquetes.
    - Diagrama de Componentes.
    - Diagrama de Estructura compuesta.
    - Diagrama de Despliegue.

2. **Diagramas de Comportamiento:** permiten analizar y reforzar lo que debe suceder en el sistema. Los diagramas que podemos utilizar:
    - Diagrama de Estados.
    - Diagrama de Casos de Uso.
    - Diagrama de Actividades.
    - Diagrama de secuencia.

3. **Diagramas de Interacción:** permiten analizar y reforzar el flujo de control y de datos entre los elementos del sistema. Los diagramas que podemos utilizar:
    - Diagrama global de interacciones.
    - Diagrama de tiempos.
    - Diagrama de comunicación.

¿Para qué nos sirve?

Visualizar

La mayoría de los programadores piensa que la distancia entre pensar en una implementación y transformarla en código fuente es casi nula. Lo piensas, lo codificas. En este caso el programador está haciendo algo de modelado, aunque sea de forma completamente mental. Esto plantea algunos problemas:

- La comunicación de éstos modelos conceptuales está sujeta a errores, ya que los proyectos y las organizaciones desarrollan su propio "lenguaje".
- Existen cuestiones propias del desarrollo de software que no se pueden entender a menos que se construyan modelos que trasciendan el lenguaje de programación textual. Por ejemplo, el hecho de representar las jerarquías de clases desde el código fuente de la aplicación se hace muy complejo, ya que se debe analizar texto, y no un gráfico.
- Si el desarrollador que escribió el código no documentó los modelos que había en su cabeza, esa información se perderá para siempre.

Los modelos UML permiten una mejor comunicación. Transciende de lo que puede ser representado por un lenguaje de programación. Todos los símbolos de UML poseen una semántica bien definida evitando ambigüedades.

Especificar:

Especificar significa construir modelos precisos, no ambiguos y completos. UML cubre la especificación de todas las decisiones de análisis, diseño e implementación que deben realizarse al desarrollar un sistema con gran cantidad de software.

Construir

UML no es un lenguaje de programación visual, pero sus modelos pueden conectarse de forma directa a muchos lenguajes. Esto nos permite establecer correspondencias desde un modelo UML a un lenguaje como Java, C++, PHP, o incluso a tablas de bases de datos

relacionales.

Las cosas que se expresan mejor gráficamente también se representan gráficamente en UML, mientras que las cosas que se expresan mejor textualmente se plasman con el lenguaje de programación.

Esta correspondencia nos permite ingeniería directa: generación de código a partir de UML y viceversa, obtener a partir de una implementación su representación UML.

Documentar

UML cubre la documentación de la arquitectura de un sistema y todos sus detalles. Posee un lenguaje para expresar los requisitos y pruebas. También permite modelar las actividades de planificación y gestión.

¿Por qué usar UML?

Por que es apropiado para modelar casi cualquier tipo de aplicación (Escritorio, Web, de tiempo real, etc.).

Por que nos permite modelar no sólo sistemas de software sino otro tipo de sistemas reales de la empresa, siempre utilizando los conceptos de la orientación a objetos (OO).

Por que nos permite planificar la estructura y el desarrollo de una plataforma a gran escala, con cientos de módulos que interactúan entre sí, decenas de programadores involucrados, complejos modelos de datos y donde cada dependencia tenga determinado nivel de acceso a este gran mega diagrama, entonces si o si necesitaremos utilizar UML.

Por que es aplicable a:

- Sistemas empresariales.
- Bancos y entidades financieras.
- Telecomunicaciones.
- Electrónica médica.
- Comercio.
- Sistemas basados en la Web.

Modelo conceptual

Para entender UML se necesita adquirir un modelo conceptual del lenguaje, y para ésto necesitaremos aprender tres elementos principales: los *bloques básicos*, las *reglas* que dictan como deben combinarse éstos bloques y algunos *mecanismos comunes* que se aplican a través

de UML

Bloques básicos

El vocabulario de UML incluye tres clases de bloques básicos:

- **Elementos:** son abstracciones que constituyen los bloques básicos de construcción orientados a objetos. Se utilizan para escribir modelos bien formados. Existen cuatro tipos de elementos:
    1. Elementos estructurales.
    2. Elementos de comportamiento.
    3. Elementos de agrupación.
    4. Elementos de anotación.

- **Relaciones:** son las encargadas de ligar a cada uno de los elementos entre sí. Existen cuatro tipos de relaciones diferentes:
    1. Dependencia.
    2. Asociación.
    3. Generalización.
    4. Realización.

- **Mecanismos comunes:** el lenguaje sigue patrones que definen estilos los que se aplican de forma consistente a través del mismo. Esto lo hace más sencillo y simple de leer y comprender. Entre éstos mecanismos encontramos:
    1. Especificaciones.
    2. Adornos.
    3. Divisiones comunes.
    4. Mecanismos de extensibilidad.

Diagramas de clase

Son el tipo de diagrama más utilizado en el modelado de sistemas orientados a objetos. Este tipo de diagramas muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Los diagramas de clase se utilizan para modelar la vista de diseño estática de un sistema. Esto es, modelar el vocabulario del sistema, las colaboraciones o los esquemas del mismo.

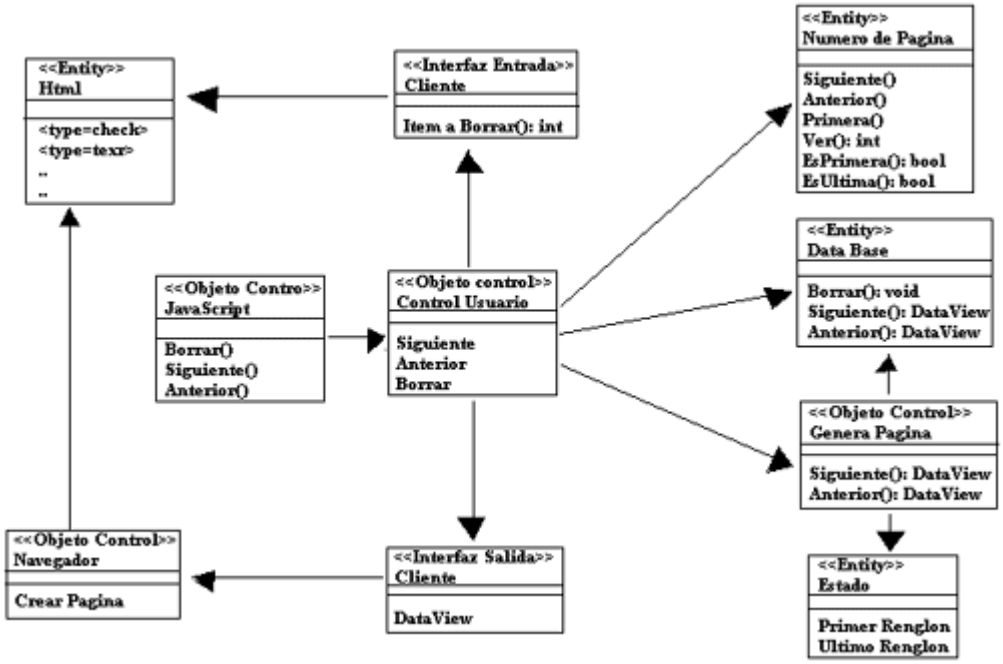


Diagrama de clases

Los diagramas de clase contienen normalmente los siguientes elementos:

- Clases
- Interfaces
- Relaciones de dependencia, generalización y asociación.

Estos diagramas se utilizan para modelar la vista de diseño estático de un sistema. Esta vista soporta principalmente los requisitos funcionales de un sistema, los servicios que el sistema debe proporcionar a sus usuarios finales.

Cuando se modela la vista de diseño estática de un sistema, normalmente se utilizarán los diagramas de clases de alguna de estas tres formas:

1. Para modelar el vocabulario de un sistema: implica decisiones sobre que abstracciones son parte del sistema y cuales quedan afuera de sus límites. Los diagramas de clases se utilizan para especificar estas abstracciones.

2. Para modelar colaboraciones simples: una colaboración es una sociedad de clases, interfaces y otros elementos que colaboran para proporcionar un comportamiento mayor que la suma de todos los elementos.

3. Para modelar un esquema lógico de base de datos: ya sea que necesitemos guardar datos en base de datos relacionales u orientadas a objetos, podremos modelar

esquemas para estas bases de datos mediante diagramas de clases.

Abstracción

Es la capacidad para encapsular y aislar la información del diseño y ejecución. Denota las características esenciales de un objeto, donde se capturan sus comportamientos y atributos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar, cambiar su estado y comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características.

Consiste en la generalización conceptual de un determinado conjunto de objetos y de sus atributos y propiedades, dejando en un segundo término los detalles concretos de cada objeto. ¿Qué se consigue con la abstracción? Bueno, básicamente pasar del plano material (cosas que se tocan) al plano mental (cosas que se piensan).

Relaciones entre clases

Las relaciones que existen entre distintas clases nos indican cómo se comunican los objetos de esas clases entre si. Los mensajes navegan por las relaciones existentes entre las distintas clases.

Existen tres tipos de relaciones:

Asociación (conexión entre clases)
Dependencia (relación de uso)
Generalización / especialización (relaciones de herencia)

Asociación

Es una relación estructural entre entidades (una entidad se construye a partir de otras entidades). La relación de asociación es cuando una clase tiene en su estructura a otra clase, o se puede decir también que se construye una clase a partir de otros elementos u objetos. Pueden ser binarias o n-arias, según se involucren dos o más clases.

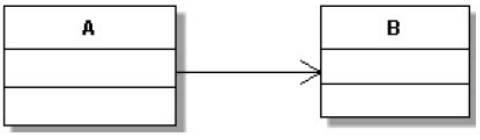


Diagrama genérico de Asociación

Una asociación se dirige desde una clase a otra (o un objeto a otro), el concepto de navegabilidad se refiere al sentido en el que se recorre la asociación. Existe una forma especial de asociación, la *agregación*, que especifica una relación entre las clases donde el llamado *agregado* indica el todo y el *componente* es una parte del mismo.

Algunas deducciones:

- La clase A depende de la clase B.
- La clase A está asociada a la clase B.
- La clase A conoce la existencia de la clase B, pero la clase B no conoce la existencia de la clase A (sentido de la flecha).
- Todo cambio que se haga en la clase B, por la relación que hay con la clase A, podrá afectar a la clase A.

Al codificar, una asociación se representa como un atributo que es una instancia de otra clase. Esta relación se debe entender como la actividad de construir elementos complejos a partir de otros elementos más simples y justamente, la verdadera esencia de la POO.
Codificación:

```
<?php
        require_once 'B.php';
        class A
        {
                private $_b;
                public function __construct()
                {
                        $this->_b = new B();
                }
        }
        $a = new A();
```

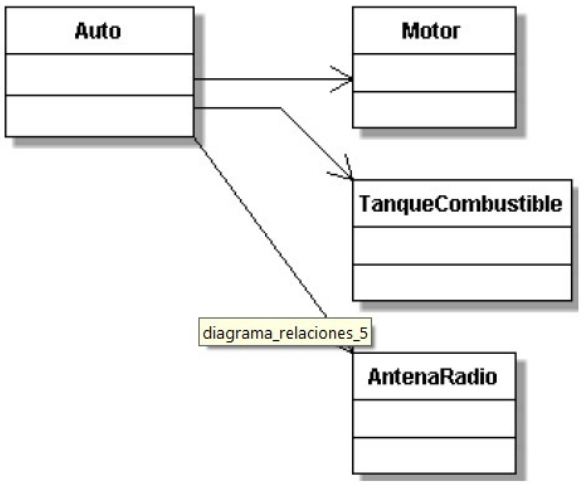
A tener en cuenta:

- En el diagrama, en la especificación de la clase A, no se agrega la definición del atributo *b*, por lo que es decisión de cada uno agregarlo o no en el diagrama.
- Como no se especifica, tampoco parece decir nada sobre su visibilidad, por lo que nos conviene mantener el criterio de que *los atributos por defecto son siempre no públicos.*
- Lo ideal sería que en la misma definición del atributo poder hacer la creación de la instancia del objeto sin depender de un constructor (como sucedería en Java), pero el detalle aquí es que PHP no soporta la creación de instancias en la definición de

atributos, solo podríamos hacer una asignación directa si el atributo es un array, pero no crear una instancia (con “new”). Solo podremos crear las instancias de nuestros atributos en el constructor o dentro de los métodos de la clase.

Ejemplo:

Un auto está compuesto por un motor, un tanque de combustible y una antena de radio, por lo tanto tendríamos: un objeto Auto y como atributos del mismo tendríamos los objetos Motor, Tanque y Antena.



Asociación auto

Lo interesante notar aquí es que el Auto no conoce los detalles internos de cada uno de sus componentes, simplemente se construye a partir de ellos pero cumplen el *Principio de Ocultación*, lo cual fortalece el diseño al desconocer detalles internos que pueden obligarlo a depender fuertemente de cómo están implementados.

Al codificar la clase auto:

```
<?php
	require_once 'Motor.php';
	require_once 'TanqueCombustible.php';
	require_once 'AntenaRadio.php';
	class Auto
	{
		private $_motor;
		private $_tanqueCombustible;
		private $_antenaRadio;
		public function __construct()
		{
			$this->_motor = new Motor();
			$this->_tanqueCombustible = new TanqueCombustible();
			$this->_antenaRadio = new AntenaRadio();
```

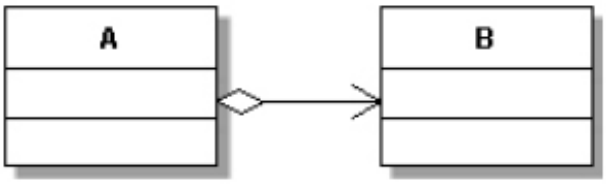
```
        }
}

$auto = new Auto();
```

Los comentarios sobre la lectura de las flechas y los sentidos son los mismos que en los casos anteriores. Según el sentido de las flechas, el Auto conoce a sus componentes y los componentes no conocen al Auto. Si los componentes cambian, afectan al Auto, ya que este está asociado a ellos, no así al revés.

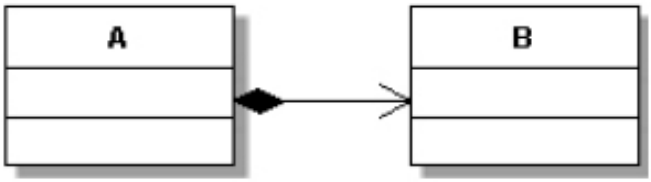
La asociación se puede tipificar de la siguiente forma:

- **Agregación**: es una relación de asociación pero en vez de ser de 1 a 1 es de 1 a muchos, es decir, la clase A agrega muchos muchos elementos de tipo clase B.



Agregación

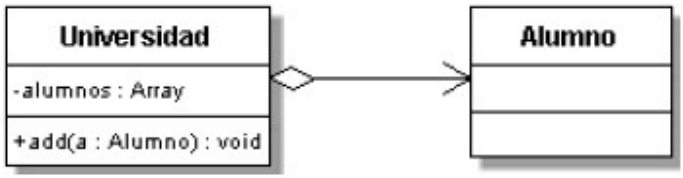
- **Composición**: además de "agregar", existe una relación de vida, donde elementos de B no pueden existir sin la relación con A. Desde el punto de vista de muchos lenguajes (como Java o PHP) esto no necesariamente genera ningún cambio en el código, pero puede ser tomado como parte de la documentación conceptual de cómo debería ser el diseño del sistema.



Composición

Ejemplo:

Tomando un ejemplo de agregación:



Agregación

El elemento del lenguaje que por defecto nos permite contener varios elementos es el array, por lo tanto del diagrama se desprenden que tendremos:

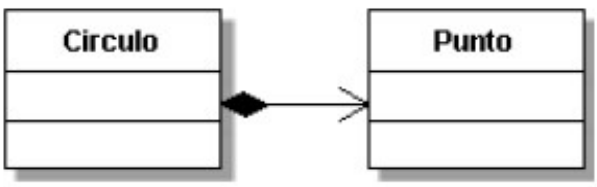
1. Un atributo de tipo array para contener todos los elementos.
2. Por lo menos un método para agregar los elementos al array, que generalmente se representa con un simple "add" (que como siempre queda sujeto a variar según nuestro criterio).

Al codificarlo:

```
<?php
        require_once 'Alumno.php';
        class Universidad
        {
                private $_alumnos = array();
                public function add(Alumno $alumno)
                {
                        $this->_alumnos[] = $alumno;
                }
        }
         $universidad = new Universidad();
        $universidad->add(new Alumno());
        $universidad->add(new Alumno());
        $universidad->add(new Alumno());

/* * Esta universidad contiene 3 alumnos */
```

Como ejemplo de composición, podríamos mencionar el siguiente:



El círculo está compuesto por puntos.

Al existir una relación de vida, no se concibe que existan puntos sueltos sin estar en una figura geométrica

Dependencia

Es una relación de uso entre dos entidades (una usa a la otra). Es cuando una clase depende de la funcionalidad que ofrece otra clase. Esta es la relación más básica entre clases y a su vez comparada con otro tipos de relaciones, la más débil.

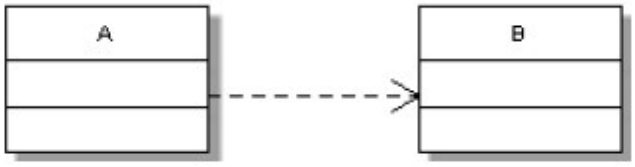


Diagrama genérico de Dependencia

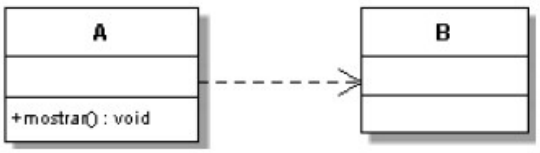
Algunas deducciones:

- La clase A depende de la clase B.
- La clase A usa la clase B.
- La clase A conoce la existencia de la clase B, pero la clase B no conoce la existencia de la clase A (es lo que significa el sentido de la flecha)
- Todo cambio que se haga en la clase B, por la relación que hay con la clase A, podrá afectar a la clase A. Por eso también se le dice *relación de uso*, la clase A "usa" la clase B.

Al codificar la relación de dependencia nos encontramos con que existen dos situaciones posibles:

1. En un método de la clase A instancio un objeto de tipo B y posteriormente lo uso.
2. En un método de la clase A recibo por parámetro un objeto de tipo B y posteriormente lo uso.

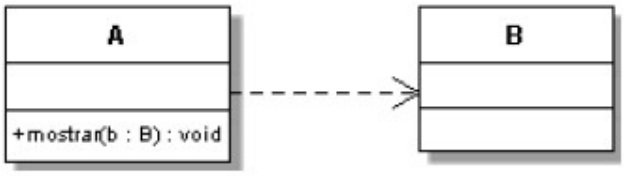
Caso 1 – Instancio un objeto B dentro de un método de A. Código:



```
<?php
        require_once 'B.php';
        class A
        {
                public function mostrar(){
                        $b = new B();
                         /* etc */
        }
}
```

El uso de *require_once* está representando la *dependencia* con otra clase al requerir su fuente. Tomando el código fuente de una clase y siguiendo todos sus require / include podemos determinar la dependencia con otras clases y reconstruir el diagrama (ingeniería inversa).

Caso 2 – Recibo por parámetro de un método de A un objeto B




```
<?php
        require_once 'B.php';
        class A
        {
                public function mostrar(B $b)
                {
                        echo $b; /* etc */
                }
        }
```

Puntos a tener en cuenta:

- En el diagrama UML se representa el parámetro de esta forma b : B, lo cual debe ser leído en el orden *variable:tipo*, pero dependiendo del lenguaje, la traducción generalmente es al revés: Tipo $variable (como se puede observar en el ejemplo con el método "mostrar(B $b)").
- Aparece por primera vez el Type Hinting ("indicando el tipo") que es incorporado en PHP5 (imitando muchos lenguajes de tipado fuerte como Java) y que permite reforzar el diseño al solo permitir que ingresen por parámetro objetos del tipo que

especifiquemos.

Encapsulamiento y principio de ocultación

Encapsulamiento

Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Se refiere a la capacidad de agrupar en un entorno con límites bien definidos distintos elementos.

De manera informal primero generalizamos (la abstracción) y luego decimos: la generalización está bien, pero dentro de un cierto orden: hay que poner límites (la encapsulación), y dentro de esos límites incluimos todo lo relacionado con lo abstraído: no sólo datos, sino también métodos, comportamientos, etc.

Ocultación

Mediante este principio, decimos que cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con él los demás los demás objetos. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a hacerlas

Modificadores de acceso o visibilidad

Como vimos en la clase anterior, los modificadores de acceso son palabras reservadas que especifican el nivel de accesibilidad que tendrán los métodos y propiedades de un objeto.
En POO encontramos tres posibilidades respecto a la visibilidad: público, privado y protegido. PHP5 sin embargo agrega uno más: final.

Visibilidad Pública

Puede considerarse como “sin restricciones”, es decir, lo que es público todos lo pueden acceder o modificar. Por regla antigua de la POO, ningún atributo debe ser público. Para la herencia, todo lo que sea *público* se hereda a sus hijos, de igual forma que cualquiera que acceda a la clase puede ver sus atributos y métodos públicos.

Visibilidad Privada

Aquí me gusta hacer la siguiente metáfora, la cual ayuda a clarificar mucho el funcionamiento de esta “visibilidad” en el contexto de la herencia: un atributo o método privado es como “la ropa interior”, es personal, es privada, y nadie más tiene acceso directo a ella (me imagino que nadie presta ni usa prestada ropa interior, aún siendo familiares. Si es así, mis disculpas por estos ejemplos, no me quise meter en la vida personal de nadie ;-)).

Visibilidad Protegida

Un punto intermedio entre lo privado y lo público, lo “protegido” son los atributos y métodos que pueden ser accedidos de forma directa como “propios” por parte de los “hijos” de la clase “padre”. Para las clases que no tienen una relación de parentesco su significado es igual al de privado.

Setter / Getter

Cuando empezamos a trabajar con objetos, el primer error que cometemos es dejar todos los atributos públicos, lo que permite que cualquier usuario de nuestra clase pueda hacer y deshacer sin control nuestro objeto (modificando y consultando sus valores).

Por un tema de principios de la POO los atributos de los objetos deben ser siempre "privados" (concepto de encapsulación) y se deberán crear métodos públicos que permitan asignar o devolver valores a los atributos cada vez que la situación lo amerite.

set()

Definiremos este método para setear o asignar un valor al atributo en cuestión. Por convención se debe llamar al método como *set<atributo>*, donde <atributo> será el nombre de la variable que recibirá el valor

Veamos un ejemplo:

```
<?php
        class Cliente
        {
                private $_apellido,$_nombre,$_fechaNacimiento;

                public function setFechaNacimiento($fechaNacimiento)
                {
                        $this->_fechaNacimiento=$fechaNacimiento;
                }
```

```
	}

	$cliente= new Cliente();
		$cliente->setFechaNacimiento('18-05-1968');
?>
```

get()

Definiremos este método para obtener el valor asignado a un atributo. Por convención se debe llamar al método como *get<atributo>,* donde <atributo> será el nombre de la variable que nos devolverá el valor.

Veamos un ejemplo:

```
<?php
	class Cliente
	{
		private $_apellido,$_nombre,$_fechaNacimiento,$_edad;

		public function setFechaNacimiento($fechaNacimiento)
		{
			$this->_fechaNacimiento=$fechaNacimiento;
		}


		private  function _calculoEdad()
		{
			$this->_edad= // código que calcula la edad

		}

		public function getEdad()
		{
			$this->_calculoEdad();
			return $this->_edad;
		}

	}

	$cliente= new Cliente();
		$cliente->setFechaNacimiento('18-05-1968');
	echo $cliente->getEdad();

?>
```

Errores comunes:

- Definir todos los "get" y "set" para todos los atributos existentes, es como si fueran todos públicos, careciendo de utilidad.
- Agregar más lógica que asignar un valor o retornar el valor del atributo, con lo cual perderán el concepto de *get / set*.

Puntos a tener en cuenta:

- Por defecto, atributos privados
- Usa métodos comunes y no getter/setter
- Si no queda otra opción, usa solo getter
- Si no queda otra opción, usa setter
- Trata de evitar getter y setter a la vez (efecto "atributo público")

Constantes de una Clase

Definición

Se pueden definir valores constantes en función de cada clase manteniéndola invariable en el transcurso de un script.

Las constantes se diferencian de las variables comunes en que no utilizan el símbolo $ al declararlas o usarlas. Y generalmente, se definen en mayúsculas para diferenciarse del resto de código.

El valor debe ser una expresión constante, no (por ejemplo) una variable, una propiedad, un resultado de una operación matemática, o una llamada a una función.

También es posible tener constantes para interfaces.

Ejemplo de definición y uso

```
<?php
class Constante
{
	// Definición de constantes y atributos
	const CADENA = 'Mi constante de tipo cadena';
	const NUMERICA = 10;
	const NUMERICAREAL = 2.1;
	const BOLEANA = true;

	// Definición de métodos
	public function obtenerCalculoConConstante()
	{
		return 100 * self::NUMERICAREAL;
```

```
        }
}
```

```
echo Constante::CADENA; // Salida: Mi constante de tipo cadena

$oConstante = new Constante();

echo $oConstante->obtenerCalculoConConstante();// Salida: 210

echo $oConstante::NUMERICAREAL;// Salida: 2.1

echo $oPrueba->NUMERICAREAL;// Salida: Undefined property
```

$this vs self y el operador ::

Palabra reservada this

La palabra reserva **this** nos permite acceder a variables y/o métodos de objeto de una instancia particular.

Se puede ver como una variable (**$this**) que apunta a una instancia particular brindando acceso a las propiedades y comportamientos de un objeto dentro de la definición de una clase.

Palabra reservada self

La palabra reserva **self** nos permite acceder a variables estáticas y constantes de clases desde el interior de la definición de la misma.

Se pueden acceder a las constantes y variables sin necesidad de instanciar la clase teniendo en cuenta que son propias de la clase y no de una instancia a la misma.

Self puede ser reemplaza con el nombre de la clase, pero el código queda más prolijo con el uso de **self**.

El operador ::

El Operador de Resolución de Ámbito (también denominado Paamayim Nekudotayim) o en términos simples, el doble dos-puntos, es un token que permite acceder a elementos estáticos, constantes, y sobrescribir propiedades o métodos de una clase.

Cuando se hace referencia a estos items desde el exterior de la definición de la clase, se utiliza el nombre de la clase. Desde la versión 5.3.0 se puede hacer referencia al nombre de la clase utilizando una variable de tipo string que tendrá como valor el nombre de la clase.

Paamayim Nekudotayim podría, en un principio, parecer una extraña elección para bautizar a un doble dos-puntos. Sin embargo, mientras se escribía el Zend Engine 0.5 (que utilizó PHP 3), asi es como el equipo Zend decidió bautizarlo. En realidad, significa doble dos-puntos - en Hebreo!

Ejemplo de uso de constantes de la clase fuera de la definición de la misma.

```
<?php
class Constante
{
	// Definición de constantes y atributos
	const MICONSTANTE = 'Mi constante de tipo cadena';
	// Definición de métodos
} // Prueba.php

$miClase = 'Constante';

echo $miClase::MICONSTANTE; // Salida: Mi constante de tipo cadena

echo Constante::MICONSTANTE; // Salida: Mi constante de tipo cadena
```