Julio 2009 / Versión 1.8.9

Programación Orientada a Objetos

para

PHP5

*"Aprende de forma simple y definitiva POO para PHP5, deja de ser **Programador de Páginas Dinámicas** y empieza a convertirte en **Desarrollador de Sistemas**"*

por Enrique Place

Usuario: Juan Zapata

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Atribución-No Comercial 3.0 Unported

Usted es libre de:

copiar, distribuir, exhibir, y ejecutar la obra

hacer obras derivadas

Bajo las siguientes condiciones:

Atribución. Usted debe atribuir la obra en la forma especificada por el autor o el licenciante.

No Comercial. Usted no puede usar esta obra con fines comerciales.

- Ante cualquier reutilización o distribución, usted debe dejar claro a los otros los términos de la licencia de esta obra.
- Cualquiera de estas condiciones puede dispensarse si usted obtiene permiso del titular de los derechos de autor.
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"ESTE LIBRO ES UN SERVICIO"

Este Libro está licenciado bajo **CREATIVE COMMONS** y puedes distribuirlo con libertad a quienes consideres que pueda serle útil tenerlo.

Si decides **ADQUIRIR EL SERVICIO COMPLETO** podrás tener acceso a **USUARIOS.SURFORCE.COM** y por el período de tiempo que elijas obtendrás:

1. **Poder hacer CONSULTAS DIRECTAS AL AUTOR**: cualquier parte del libro, tanto dudas sobre ejemplos, capítulos, ejercicios y estas se responderán normalmente durante las próximas 24-48hs (aunque lo más probable que obtengas una respuesta en pocas horas).

2. **Acceso a TODOS LOS FUENTES**: de todos los ejercicios del libro, revisados y comentados por el mismo autor.

3. **ACTUALIZACIONES mensuales:** tanto correcciones como ejemplos o hasta capítulos nuevos, lo que podrá incluir a futuro acceso a material multimedia (screencasts, podcasts, etc).

4. **CAMBIA EL CONTENIDO DEL LBRO**: si consideras que algún capítulo, ejemplo o ejercicio podría mejorarse, o algún tema que ves no se encuentra tratado en el libro, tu sugerencia será recibida y tenida en cuenta para la próxima actualización mensual del libro.

Aprovecha la oportunidad de expandir las posibilidades de un **libro digital** obteniendo todo el soporte que no te podría dar nunca un **libro tradicional** (y de paso salvamos algunos bosques).

ADQUIERE EL LBRO COMPLETO en **SURFORCE** y accede a todos los servicios en

http:/usuarios.surforce.com

[ATENCIÓN: si este material se encuentra impreso, es probable que ya esté desactualizado]

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Versiones del documento

Versión	Fecha	Descripción	Autor
1	1/01/2009	Primera versión	enriqueplace
1.1	15/01/2009	Segunda revisión, recopilación de ejemplos	enriqueplace
1.2	31/01/2009	15 días de revisión de contenidos	enriqueplace
1.3	01/02/2009	Se separa como un capítulo el tema "Paquetes UML" y se agrega un capítulo nuevo sobre "Excepciones"	enriqueplace
1.4	3/02/2009	Agrega capítulo "Debemos Profesionalizarnos" (post del blog)	enriqueplace
1.5	4/02/2009	Error: corrección capítulo 8, diseño 2, cambia **echo por retorno** en clase Persona	andresfguzman (corrector)
1.6	6/2/2009	Agrega nota de autor recalcando el tema de los estándares de codificación definidos por Zend y que todos los ejemplos de este libro lo seguirán	enriqueplace
1.7	6/2/2009	Enumera los Principios que deberíamos seguir los desarrolladores	enriqueplace
1.7.1	10/2/2009	Correcciones en fuentes, espacios, estética	Dennis Tobar (lector)
1.7.2	28/2/2009	Cap.11: Agrega explicación sobre auto-relación con Persona (cuadro de color verde)	Colabora: Antonio L. Gil (lector)
1.7.3	10/3/2009	Cap. 10: Agrega ejemplo y explicación extra en el caso de "qué hacer con las relaciones cíclicas / bidireccionales"	Colabora: Eduardo de la Torre (lector)
1.7.4	22/3/2009	Cap. 14: corrección en la redacción del resumen final	Colabora: Raquel Diaz (lector)
1.7.5	24/3/2009	Cap.11: Agrega explicación de "Confusión común" con respecto a confundir bidireccional con cíclica (cuadro "verde")	enriqueplace

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

1.7.6	26/3/2009	Cap.7: el ejemplo de calcular la edad no está completo, debería retornar un integer y no el valor del atributo "_fechaNacimiento"	Colabora: Carlos Arias (lector)
1.7.7	26/3/2009	Cap.10: amplía la explicación sobre “Multiplicidad”	Colabora: Christian Tipantuña (alumno)
1.7.8	1/4/2009	Cap. 7: ejemplo “decirEdad” tiene un parámetro de más	Colabora: Carlos Arias (alumno/lector)
1.8.0	3/4/2009	Agrega Anexo: "Qué es lo nuevo en PHP5?", basado en el artículo "What's New in PHP5?"	enriqueplace
1.8.1	25/4/2009	Cap.19, parte 2, error, cambia "Copy" por "MaquinaDeEscribir"	Colabora: Karina Diaz (alumna/lector)
1.8.2	25/4/2009	Cap.19, ajusta diagrama UML, cambia parámetro leer:String por texto:String en MaquinaDeEscribir	Colabora: Karina Diaz (alumna/lector)
1.8.3	15/5/2009	Revisión Cap.1	enriqueplace
1.8.4	20/5/2009	Revisión Cap.2	enriqueplace
1.8.5	4/7/2009	Revisión Cap.3, definición de “contexto”	enriqueplace
1.8.6	4/7/2009	Capítulo 3 está repetido, dos capítulos tienen el mismo nombre, se unifican en el capítulo 4, cambiando al nombre de “POO según los Manuales” (incluyendo ahora Wikipedia y el manual Oficial)	enriqueplace
1.8.7	4/7/2009	Cap.4 agrega enlaces a Wikipedia	enriqueplace
1.8.8	5/7/2009	Cap.5 – revisión y ampliación sobre el concepto de “diseño”	enriqueplace

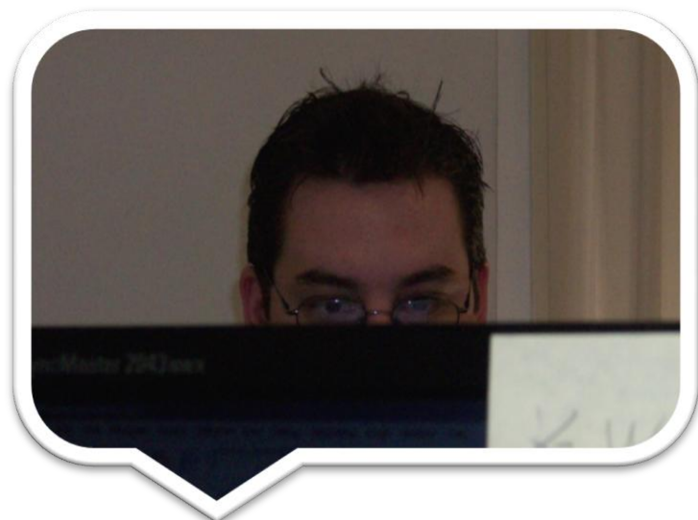
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

1.8.9	5/7/2009	Cap.6 – revisión	enriqueplace

¡Mis más sinceros agradecimientos a lectores y colegas con sus aportes!

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

SOBRE EL AUTOR



Enrique Place (35 años), nacido en Uruguay y actualmente viviendo en Argentina (pero *"ciudadano de Internet"*), es uno de los tantos "emprendedores por naturaleza" que **cambió a los 14 años su fanatismo por las artes marciales** (algunos llegaron a pensar que sería el sucesor sudamericano del *Pequeño Dragón*) **por el fanatismo hacia la informática**.

Por cuestiones que solo el destino sabrá, tuvo la oportunidad de trabajar con los antiguos y míticos dinosaurios de la informática llamados Mainframes y participó en una misión para salvar a la raza humana de su extinción migrando aplicaciones para sobrevivir al colapso del Y2K, convirtiendo a diestra y siniestra código Mantis / Mainframe a Microfocus Cobol y Windows NT / Unix AIX.

Paralelamente, fundó una pequeña empresa llamada **LINUXTECH**, quién fue la primer importadora para Uruguay de SuSE GNU/Linux (Alemania) y que dio los primeros pasos al evangelizar usuarios y empresas brindando servicios profesionales.

De profesión *"Analista Programador",* estudiante y posteriormente **docente** en la Universidad ORT (Uruguay), aprovechó todo lo que pudo aprender de arquitecturas como .Net y Java, conocer de Patrones de Diseño (GOF), como para darse cuenta que **PHP, su verdadero amor informático**, tenía un gran potencial por su **simplicidad y pragmatismo**, y que además su comunidad carecía completamente de una visión amplia como para entender todo lo que aún faltaba recorrer (como lo habían hecho ya otras tecnologías).

Finalmente, el autor **no se considera "gurú"** y simplemente como **"en el país de los ciegos, el tuerto es rey"**, de la mano a su facilidad para enseñar (radicada en que aún es **"alumno de todo"), es que se inicia en el camino de tratar de transmitir nuevos conocimientos a la *Comunidad PHP***.

Este libro se escribe con el objetivo de que los actuales *Programadores PHP* se conviertan en el corto plazo en *Desarrolladores PHP* aprobando la materia que más les cuesta:

"Programación Orientada a Objetos en PHP5"

"Este libro fue escrito para ti, Pequeño Saltamontes"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

AGRADECIMIENTOS

A mi familia: mi amada esposa Laura, mis amadas hijas Micaela y Martina, que tantas veces soportaron que su padre estuviera ausente por tener la cabeza en otro lado. Este fue uno de los tantos proyectos que le robó tiempo a la familia, pero que espero que de alguna forma u otra lo disfruten ellas.

A la Universidad ORT Uruguay, gracias a los buenos docentes (también se aprende de los malos) que me tocaron en mi carrera y que me ayudaron a tener los conocimientos suficientes como para poder trabajar con PHP *"de otra forma"*.

Carlos Cantonnet, docente en *Análisis y Diseño Orientado a Objetos* y cumpliendo el rol de *"Arquitecto de Sistemas"* en multinacionales como TATA TCS, sus excelentes clases lograron abrir mi mente con sus explicaciones simples, directas, y con mucho humor. Ahí empecé a conocer de *Patrones de Diseño* y la magia de los *Principios de Diseño Orientado a Objetos*.

TATA CONSULTANCY SERVICES

Nicolás Fornaro, docente en Programación Java y *Análisis y Diseño*, por sus muy buenas clases que, a diferencia de Cantonnet, me permitían tener una visión menos teórica y mucho más práctica de los *Objetos* y los *Patrones de Diseño*.

A muchos colegas y amigos que de alguna forma u otra siempre apoyaron o inspiraron mis locuras, pero principalmente cuando alguna vez me dijeron frases como *"eso no se puede", "no vale la pena el esfuerzo", "no le veo sentido", "no vas a poder con todo"*.

Me quedo con la frase ***"No sabían que era imposible, por eso lo lograron"***

Sin ustedes no lo hubiera logrado ***¡GRACIAS A TODOS!***

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"REVISORES DE ESTE LIBRO"

Amigos y colegas que ayudarán con la revisión en busca de errores y/o sugerencias. Como es un libro que se irá actualizando mensualmente todas sus mejoras se podrán apreciar a finales de **febrero 2009** (de todas formas creo que mi orgullo me impedirá hacerles caso **;-)**).

Andrés Guzmán, blogger colega de PHP y Zend
Chile
http://bolsadeideas.cl/zsamer/

Christian Serrón, uno de mis mejores alumnos
Uruguay
http://serron.surforce.com

Christopher Valderrama , colega de PHP y Zend, Moderador en Foros del Web / Sección POO – PHP
México
http://web2development.blogspot.com/

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

PRÓLOGO

Estimados Lectores

Tradicionalmente se dice que *el alumno en artes marciales* busca toda su vida el *maestro perfecto* cuando en realidad con los años y habiendo acumulado experiencia se da cuenta que siempre estuvo dentro suyo. **Una de las principales razones de escribir este libro fue no haber encontrado un libro similar para poder leer y consultar**.

A su vez estoy convencido que se pueden escribir libros "pragmáticos", "simples" y "directos", evitando entrar en disertaciones complejas de *"docente que no baja al nivel del alumno que no domina el tema a tratar"*.

El libro es el resultado de apuntes de estudios, investigaciones, experiencia personal, y materiales que fueron desarrollados para **el primer taller de POO para PHP5 (edición 2008), por eso verás ejercicios, soluciones y comentarios de los errores que estadísticamente son muy probables que cometas durante el aprendizaje**. Me tomé el trabajo de revisar todos los capítulos, generar nuevos y a su vez actualizar explicaciones que fueron vertidas en los foros (la discusión con los alumnos enriqueció los ejemplos y debían quedar registrados de forma permanente).

Para no ser menos, acostumbrado a intentar *"un paso más allá",* **este libro no será comercializado solo como un archivo "pdf", la idea es ofrecer un servicio completo** e intentar asegurar al lector el máximo de valor agregado que en este momento puedo concebir.

Traté de hacer el libro que yo compraría, espero satisfacer tus expectativas, y si no lo logro, aún estás a tiempo de que lo solucione, ya que el libro *"está vivo"* y todas las sugerencias generarán nuevas actualizaciones.

Saludos! 😉

Enrique Place

enriqueplace@gmail.com

Blog Personal
http://enriqueplace.blogspot.com

Blog Técnico
http://phpsenior.blogspot.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CONTENIDO

Introducción: 21
”Los Desarrolladores PHP debemos Profesionalizarnos” 21
Esta reflexión se la escribo a todos los "Programadores PHP" 21
Enfrentar la realidad con madurez 21
Mi experiencia personal 22
La culpa es enteramente nuestra 23
Debemos pasar de los dichos a los hechos 24
Capitulo 1 - "Nuestros Principios como Desarrolladores" 25
Principio 1: RTFM - "Lee el Maldito Manual" 27
Principio 2: DRY - "No Te Repitas" 27
Principio 3: KISS - "Mantenlo Simple, Estúpido!" 27
Principio 4: Estándar de Codificación PHP / Zend 27
Capitulo 2 - “Introducción a los Objetos” 29
“Vacuidad: vaciar todos los conocimientos” 31
“La sencillez es la mejor arquitectura “ 31
“Lo más importante es detectar los objetos” 32
En resumen 32
Capítulo 3 - “Cómo Pensar en Objetos” 34
“Lo menos importante es el código” 36
“Un niño pequeño” 36
“El medio de comunicación” 37
Capítulo 4 - “POO según LOS MANUALES” 38
“La POO según Wikipedia” 39
“POO según el manual Oficial de PHP” 42
Capítulo 5 - “Empezar a plasmar los objetos en un diseNo” 44
“Cómo representar la estructura de los objetos” 46
En Resumen 51
Capítulo 6 - “Introducción a UML” 53
“UML, el medio y no el fin en sí mismo” 55

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"UML y el público objetivo" 55
"UML es independiente del lenguaje" 56
En resumen 57
Capítulo 7 - "Cómo representar una clase en UML" 58
"Conceptos Generales" 60
Mi primer diagrama UML 60
Cómo se traduce en código 62
Sección #1 – nombre de la clase 62
Sección #2 – los atributos de la clase 62
Sección #3 – los métodos de la clase 64
"El Constructor" 64
Probando el objeto en un contexto determinado 66
En Resumen 67
Capítulo 8 - Ejercicio "Micaela y el Perro" 68
Requerimientos 70
Solución 71
Aplicación del "Principio KISS" 72
Sugerencias para enfrentar los diseños 72
Propuesta de Diseño 1 73
Diagrama UML 73
Traducción de UML a PHP 74
Propuesta de Diseño 2 76
Cambios 76
Diagrama UML 76
Traducción UML a PHP 77
En Resumen 79
Capítulo 9 - Los métodos "getter / setter" o "accesores / modificadores" 80
Requerimiento 1 84
Requerimiento 4 85
En Resumen 88
Capítulo 10 - "Cómo representar las Relaciones entre clases en UML" 89
La Relación de Dependencia 91

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Representación UML ... 91
Cómo se traduce a código ... 92
Caso 1 – instancio un objeto B dentro de un método de A ... 92
Caso 2 – recibo por parámetro de un método de A un objeto B ... 93
La Relación de Asociación ... 94
Representación UML ... 94
UML del ejemplo del Auto ... 96
Variaciones de Asociación: Relación de Agregación y Relación de Composición ... 97
Relación de Agregación ... 97
Relación de Composición ... 97
Ejemplos de Agregación y de Composición ... 97
¿Qué relaciones se deberían evitar? ... 99
Ejemplo de relación cíclica ... 99
Resumiendo rápidamente algunos conceptos complementarios ... 101
“Navegabilidad” ... 101
“Definición de Roles” ... 101
Resumen ... 102
Capítulo 11 - Ejercicio "Micaela, el Perro y la Escuela” ... 103
Requerimientos ... 105
Solución ... 106
“Errores Comunes” ... 107
Detalle importante a tener en cuenta con las relaciones ... 111
El método público toString() ... 113
Lo que dice el manual de toString() ... 115
Agregando las relaciones que se ven desde Index ... 116
Crear la clase Index a partir de la representación UML ... 117
Segunda propuesta de diseño ... 118
Diagrama UML ... 119
En Resumen ... 120
Capítulo 12 - Ejercicio "La Escuela y los coches escolares" ... 121
Requerimientos ... 122
Guía ... 122

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución ... 123
"Errores Habituales" ... 123
"Comentarios Generales" ... 125
Navegabilidad ... 126
Cómo deberían ser las búsquedas ... 127
Diagrama UML ... 128
Resumen ... 129
Capítulo 13 - Ejercicio "Implementar diagrama de diseNo UML" ... 130
Requerimientos ... 131
Solución ... 132
¿Qué sucedería si se están usando mal las relaciones? ... 133
Comentarios adicionales ... 134
Errores habituales ... 135
Consejos ... 136
Diagrama UML ... 137
Ejemplo codificado ... 138
Resumen ... 139
Capítulo 14 - Ejercicio "Sistema para empresa que realiza encuestas" ... 140
Requerimientos ... 141
Solución ... 142
Posibles dificultades ... 142
Requerimientos presentados ... 142
Requerimientos Erróneos ... 142
Muchos Diseños Posibles ... 143
Paso 1 – "Crear las clases y atributos" ... 144
Paso 2 – "Relacionar las clases" ... 145
"Receta de cocina" ... 149
"El código" ... 150
Index creado a "prueba y error" ... 155
Resumen ... 158
Capítulo 15 - "Herencia, Interfaces y Polimorfismo" ... 159
La Herencia ... 161

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Representación UML 162
Cómo se traduce a código 163
Caso 1 –Usuario hereda de Persona 164
Caso 2 –Persona agrega un constructor 166
Caso 3 –Persona agrega su toString 167
Caso 4 – Los usuarios necesitan un id y una fecha de ingreso 169
Caso 5 – Los usuarios necesitan un id único “autogenerado” 170
Explicación sobre la visibilidad de los atributos y métodos 173
“Visibilidad Pública” 173
“Visibilidad Privada” 173
“Visibilidad Protegida” 173
Caso 6 – Ejemplos varios 174
Clase abstracta 175
Herencia Múltiple 176
“Sobre-escritura” de métodos 177
Evitar la herencia y la “sobre-escritura” de métodos 178
“Generalización” versus “Especialización” 179
Entonces, ¿qué es Polimorfismo? 180
La implementación 181
“Diseño más robusto” 182
¿La herencia está limitada? 182
Las interfaces: “el poder desconocido” 184
Implementación 185
Cómo funciona 186
Detalles importantes 186
Repaso de lo visto hasta el momento 187
Las interfaces son “contratos de implementación” 188
Anexo 188
Resumen 189
Capítulo 16 - Ejercicio “Clase de Persistencia” 190
Requerimientos 191
Solución 193

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Primer escenario: “crear una conexión a la base de datos” 194
Segundo escenario: “crear una clase de persistencia” 194
Tercer escenario: “abstraer el tipo de base de datos” 194
Diseño UML 195
Ejemplo codificado 196
Principio de diseño “Abierto / Cerrado” 200
Resumen 201
Capítulo 17 - Ejercicio “Librería y la búsqueda de Libros” 202
Requerimientos 203
Solución 204
Diseño “Borrador” (con partes incompletas) 205
Diseño “Final” cumpliendo con todos los requerimientos 206
Comentarios sobre el diseño 208
Resumen 214
Capítulo 18 - “Los Paquetes en UML” 215
Cómo se representan 216
¿Qué es una Arquitectura de 3 capas? 217
¿Que son entonces los Namespaces? 218
- PHP5: Diseño en 3 capas y problemas con subdirectorios 218
- "Petición para soporte de Name Spaces en PHP5" 218
En Resumen 219
Capítulo 19 - Ejercicio “Programación ‘Orientada a la Implementación’ vs ‘Orientada a la Interface’" 220
Requerimientos 222
Solución 224
Parte 1: hacer una “máquina de escribir” siguiendo el ejemplo del artículo 225
Parte 2: Agregar interfaces al diseño propuesto en la parte 1 229
Implementación 231
¿Y el Diagrama de Paquetes? 234
El *“Principio de Inversión de dependencias (DIP)”* 235
Resumen 237
Comentarios adicionales 238

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 20 - Ejercicio “Desarrollar un sistema de ABM de usuarios” 239
Requerimientos 240
Solución 243
Cambios que se aplicaron en la resolución 244
Diagrama tradicional de paquetes 245
Diagramas de clases y sus paquetes de origen 246
Diagramas de Secuencia 247
Partes esenciales del código de la solución 249
Resumen 257
Capítulo 21 - Anexo: “Manejo de excepciones” 258
Introducción 259
Básicamente cómo funcionan las excepciones 261
Estructura interna de una clase Exception 262
Importante: PHP no tiene excepciones por defecto 264
¿Cuan grave es no tener Excepciones predefinidas y por defecto? 264
Ejemplo funcional de excepciones en PHP5 265
Importante: el orden de las excepciones 266
Beneficios de las Excepciones 267
En Resumen 268
Capítulo 22 - Cierre del libro y reflexiones finales 269
Anexo I: "Qué es lo nuevo en PHP5?" 273
Características del Lenguaje 274
1.Modificadores de acceso "public/private/protected" 274
2. El método reservado __construct() 275
3. El método reservado __destructor() 275
4. Interfaces 276
5. El operador "instance of" 277
6. Operador "final" para los métodos 277
7. Operador "final" para la clase 278
8. Método reservado __clone para clonado de objetos 279
9. Atributos Constantes para las clases 281
10. Miembros estáticos o de clase (static) 282

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

11. Métodos estáticos o de clase (static) 284
12. Clases Abstractas 286
13. Métodos abstractos 286
14. Validación de Tipo a través de Clases (type hints) 287
15.Soporte a invocaciones anidadas de objetos retornados 288
16. Iteradores 290
17. __autoload() 291
Anexo II: Recopilación de FRASES 292
- 101 citas célebres del mundo de la informática 294
- Otras 101 citas célebres del mundo de la informática 294

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Si adquieres el **[libro + servicios]** puedes hacer todas las consultas que necesites de forma directa al autor.

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

INTRODUCCIÓN:

"LOS DESARROLLADORES PHP DEBEMOS PROFESIONALIZARNOS"

Basado en el post:

Los Desarrolladores PHP debemos profesionalizarnos o quedaremos descartados por obsoletos

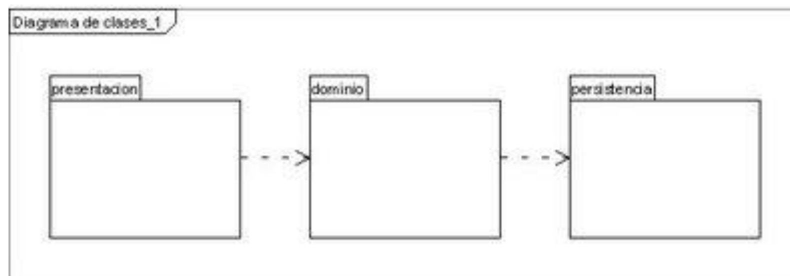
Esta reflexión se la escribo a todos los "Programadores PHP"

Al día de hoy la mayoría de los institutos o universidades de muchos países **siguen enseñando PHP4, o mejor dicho, programación "scripting" básica**. Se mueven en el viejo concepto de la "programación estructurada", trabajando constantemente sobre código que mezcla html y sintaxis PHP, todo como si de una ensalada estuviéramos hablando.

Casi paralelamente, los jóvenes autodidactas siguen por el mismo camino, tal vez ayudados por la **gran cantidad de material repetido y obsoleto que se encuentra tanto en la web como en las editoriales de turno**, donde a pesar que un libro haya sido impreso recientemente, los autores siguen siendo los mismos y escribiendo -una y otra vez- sobre los mismos temas elementales.

Enfrentar la realidad con madurez

Solo nos damos cuenta que estamos en un grave problema cuando nos enfrentamos a la realidad: **salimos al mercado laboral y con inocente sorpresa vemos que se habla mayoritariamente de Java o .Net, de UML, desarrollos en 3 capas, lógica de negocios, persistencia, polimorfismo, frameworks, patrones de diseño, refactoring...** y tú solo tienes una vaga idea de algunos conceptos, pero nulo conocimiento de si es realmente posible hacerlo con PHP...



Diseño UML: Representación de Paquetes y sus relaciones

¿No crees que algo está pasando y que tú estás quedando fuera de la "conversación"?

Este es el gran problema de la mayoría de los "Programadores PHP": se quedan en el "lenguaje", en la programación lisa y llana, rechazando todo lo que sea objetos hasta que no les queda otra salida que aprender a usarlos mínimamente... pues todas las nuevas herramientas solo hablan "ese" idioma.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¿Hasta donde piensas que podemos llegar con tan poca preparación?

Mi experiencia personal

De lo que trato de hablar en este blog es de "profesionalizarnos", de copiar y mejorar, de aprender y evolucionar. **La mayoría de los temas que expongo no son nuevos, trato de basarme en autores reconocidos y darle más prioridad a los conceptos que al lenguaje, y por sobre todas las cosas: ser simple y directo (pragmático antes que dogmático, pero sin olvidarme de lo último).**

Hay muchas cosas que desconozco de PHP y otras que directamente no uso, y nunca me baso en la memoria, siempre voy a buscar hasta lo más elemental al manual (doy prioridad al razonamiento por sobre la retención mecánica de conocimientos). Siguiendo esta metodología, mañana deberías poder cambiar de lenguaje y seguir trabajando sin problemas, pues los conceptos base los tendrías claros y estos se aplican sin importar la plataforma que estés usando.

Muchas veces comento que los temas sobre los que escribo son elementales para muchos desarrolladores Java de nivel medio y alto, pero en el ambiente PHP esto cambia (todavía no hemos madurado hacia el concepto de "arquitectura") donde *"en el mundo de los ciegos puedo ser rey".*

Debemos cambiar la mentalidad ahora que existe PHP5 y que su nueva sintaxis nos permite hacer muchas cosas que son habituales en el mundo Java.

"la vida es demasiado corta para Java", detrás de la remera dice "Usa PHP" (verídico)

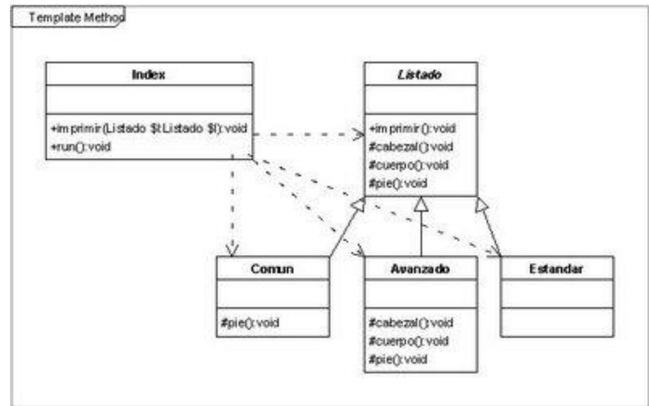
Por lo tanto, tenemos todas las herramientas para "evolucionar" y no quedarnos en las excusas.

Programador versus Desarrollador

Desarrollar Orientado a Objetos va más allá que crear objetos aislados que solo contienen datos, programar usando algunos objetos es distinto a desarrollar 100% Orientado a Objetos, **ser programador es distinto a ser un desarrollador, un sitio web no es lo mismo que un sistema web. Existen**, además de los objetos, "Principios de Diseño (OO)", "Patrones de Diseño (OO)", el lenguaje de diseño UML, frameworks, etc, y todo es perfectamente aplicable usando PHP.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Es más, muchos de estos conceptos e ideas son independientes al lenguaje si este cumple mínimamente con las características de la OO, cosa que **sucede a partir de PHP5 en adelante y que PHP4 casi carece por completo**.



Ejemplo de Patrón de Diseño: Template Method

Finalmente, es mi visión que un programador resuelve problemas aislados usando un lenguaje, pero **un desarrollador diseña e implementa una solución global**, une los componentes en un único sistema integrado y es lo suficientemente inteligente y estratega para poder reutilizar la experiencia y conocimientos adquiridos en favor de los próximos desarrollos.

Los sistemas que van quedando atrás nunca serán un lastre porque podrán ser mantenidos con el mínimo costo posible, permitiendo que el desarrollador pueda afrontar nuevos y enriquecedores desafíos.

Todos estos detalles los percibimos claramente cuando nuestros desarrollos dejan de ser un "programa menor" y necesitamos crecer, pero vemos que con los conocimientos que contamos hasta el momento todo se nos hace cuesta arriba.

La culpa es enteramente nuestra

No podemos quejarnos que a los programadores Java se les paga el doble que a nosotros y que a la mayoría de los proyectos PHP se los desvalorice, se los trate como "algo menor", "poco serio", todo porque es un "simple lenguaje web" limitado en sus posibilidades.

El "Simple Lenguaje" lo hacemos todos, al ser "Simples Programadores PHP" y nuestras son las limitaciones fundamentales. Perfectamente podemos tratar de trabajar "más seriamente" como lo hacen los "desarrolladores Java", y tratando con creatividad de suplir las carencias momentáneas (como el muy sonado caso de la falta de "namespaces").

PHP se está orientando a convertir en una "arquitectura", a parecerse a un J2EE pero mucho más simple y directo.

El proceso hace tiempo que inició.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Debemos pasar de los dichos a los hechos

De la misma forma, creo que **nos hace falta tener más "sentimiento de comunidad",** como sucede habitualmente -hasta de forma exagerada- en el mundo GNU/Linux. No es posible que nos sigamos quejando que los proveedores de hosting siguen usando PHP4.

Deberíamos hacer campañas para promover la migración a las nuevas versiones de PHP, pero fundamentalmente, incorporar en nuestros desarrollos las características avanzadas del lenguaje, e invitar a usarlo como si fuera una arquitectura, actualizar a nuestro equipo de desarrolladores, visitar empresas, universidades, etc.

¿Tú, qué vas a hacer? ¿Te vas a quedar donde estás o te vas a subir al tren?

¿Eres parte del problema o parte de la solución?

Tenemos que especializarnos y profesionalizarnos, el mundo pide POO, arquitecturas, capas, etc, y habla en "UML"... tú, ¿en qué idioma hablas? 🙂

Post Publicado el 30 de Abril 2006

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPITULO 1 - "NUESTROS PRINCIPIOS COMO DESARROLLADORES"

Peor que usar un mal estándar o un estándar incorrecto es no seguir ninguno*, de la misma forma, lo peor que podemos hacer es no tener ningún criterio para enfrentar los desarrollos. Para aumentar nuestra productividad, tanto individualmente como en equipo, debemos siempre seguir estándares y fijar criterios de desarrollo. Nuestro objetivo debería ser contar con una “plataforma de desarrollo” que nos evite tener que repensar problemas típicos y cotidianos, y concentrarnos solo en los problemas nuevos.*

Empecemos por el principio, por lo más básico y elemental… nuestros principios base.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Nadie debe empezar un proyecto grande. Empiezas con uno pequeño y trivial y nunca debes esperar que crezca; si lo haces solamente sobre-diseñarás y generalmente pensarás que es más importante de lo que lo es en esta etapa. O peor, puedes asustarte por el tamaño de lo que tu esperas que crezca. Así que empieza pequeño y piensa en los detalles. No pienses acerca de la foto grande y el diseño elegante. Si no resuelve una necesidad inmediata, seguramente está sobre-diseñado. Y no esperes que la gente salte a ayudarte, no es así como estas cosas funcionan. Primero debes tener algo medianamente usable y otros dirán "hey, esto casi funciona para mí" y se involucrarán en el proyecto."

- ***Linus Torvalds***

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Principio 1: RTFM - "Lee el Maldito Manual"

RTFM es una sigla que significa "lee el maldito manual", algo muy usado en los foros como respuesta hacia los novatos que lo último que hacen es leerlos (lamentablemente *todo* se encuentra ahí)

http://es.wikipedia.org/wiki/RTFM

Principio 2: DRY - "No Te Repitas"

"No te repitas" significa algo muy simple: si cuando desarrollas ves que al programar "copias" un código para "pegarlo en otro lado", es muy probable que estés haciendo algo mal, ya que ese código debería estar aislado y ser usado a través de parámetros.

Generalmente no existe razón para tener que duplicar el código, si estamos muy apurados, seguro, lo pagaremos muy caro dentro de poco.

http://es.wikipedia.org/wiki/DRY

Principio 3: KISS - "Mantenlo Simple, Estúpido!"

Hay una frase que dice "la mejor arquitectura es la sencillez". La sencillez es escalable, si resolvemos pequeños problemas y luego los unimos, será más fácil que hacer un sistema complejo de entrada (así funciona Unix / Linux).

http://es.wikipedia.org/wiki/Principio_KISS

Principio 4: Estándar de Codificación PHP / Zend

El lenguaje PHP y su comunidad, por años, no ha tenido ningún referente único para seguir un estándar, por consiguiente los desarrolladores usan o inventan el que más le quede cómodo. La empresa **Zend Technologies**, **principal desarrolladora de PHP (junto al autor original) y creadora de Zend Framework**, ha tomado cada vez más protagonismo (principalmente por su framework), por lo que debería ser de ahora en más nuestra referencia a seguir.

Evita crear un estándar propio, usa el definido por Zend.

HTTP://FRAMEWORK.ZEND.COM/WIKI/DISPLAY/ZFDEV/PHP+CODING+STANDARD+(DRAFT)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Nota del Autor: *si prestas atención verás que en todos los ejemplos las llaves {} inician a la izquierda cuando es una clase o un método, no se usa el tag de cierre de php* **?>**, *atributos y métodos privados iniciando con "_", etcétera.*

Bien, todo eso lo define el estándar de Zend, así que te recomiendo que lo leas y lo sigas al pié de la letra, el estándar nos hará más fuertes ;-)

"Evita crear un estándar propio, usa el definido por Zend Technologies"

HTTP://FRAMEWORK.ZEND.COM/WIKI/DISPLAY/ZFDEV/PHP+CODING+STANDARD+(DRAFT)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPITULO 2 - "INTRODUCCIÓN A LOS OBJETOS"

Aquí es donde inicia todo, el primer viaje con destino aprender las profundidades de la POO ***;-)***

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

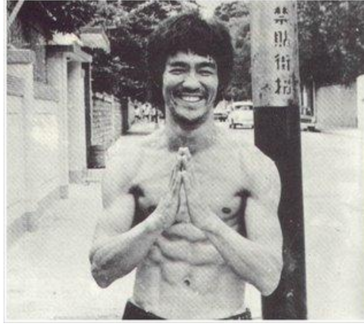
"Un sistema complejo que funciona resulta invariablemente de la evolución de un sistema simple que funcionaba. Un sistema complejo diseñado desde cero nunca funciona y no puede ser arreglado para que funcione. Tienes que comenzar de nuevo con un sistema simple que funcione."

– John Gall

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Vacuidad: vaciar todos los conocimientos"

Como diría el legendario Bruce Lee, si tienes ***un vaso medio lleno*** no vas a poder recibir por completo toda la enseñanza que te quiero transmitir, por consiguiente, **"vaciarse" de todo lo que crees que sabes es lo más importante**, más que acumular conocimientos y luego simplemente rechazar lo nuevo creyendo que eso *" ya lo sabes"*.



Así que de aquí en adelante, olvida todo lo que crees que sabes sobre objetos y empecemos de cero.

"La sencillez es la mejor arquitectura "

Los "objetos" como concepto -fuera de la informática- existen desde antes de la programación (obviamente).

¿Qué es lo que intenta hacer entonces la *Programación Orientada a los Objetos (POO)*?

Lisa y llanamente intentar simplificar la complejidad (*"simplificar las abstracciones"*) y tratar de representar de forma simple lo que vemos y manejamos todos los días… los objetos que nos rodean.

> "Controlar la complejidad es la esencia de la programación"
> -- *Brian Kernigan*

Un niño, cuando empieza a hablar, nos demuestra que ya entiende el concepto de "objetos", empieza a nombrar esas "cosas":

- "vaso",
    - "agua",
    - "papá",
    - "mamá",
    - "casa",
    - "guau guau" ("perro"),
    - etc.

Todos estamos sumidos en un mundo que tiene reglas (sepamos o no que existen, nos afectan), como cuando tiramos un objeto y este cae (gravedad), cuando pisamos la cola de un gato y este llora (y probablemente nos arañe).

Nos vamos dando cuenta que generalmente no manipulamos los objetos directamente ni tenemos un completo control sobre ellos, muchas veces solo interactuamos con ellos, por más que no queramos que el gato se defienda, este lo hará. **Nos damos cuenta que cada uno tiene dentro una *"programación"* que le dice cómo reaccionar ante determinados estímulos o situaciones**, y descubrimos luego que un gato reacciona distinto de otro gato (a pesar que ambos son gatos) y que no es lo mismo la reacción de un perro con respecto a la de un gato (a pesar que entre ellos existe una relación que los une como mamíferos).

Por lo tanto, así son los objetos: pueden ser de un tipo determinado (perro, gato), también pertenecer a una misma familia (mamíferos) y a su vez ser únicos ("el gato llamado Risón")

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Lo más importante es detectar los objetos"

Entonces, la *Programación Orientada a Objetos* no es más que eso, detectar los objetos existentes en nuestro contexto real y construirlos como si fuéramos *"Dios",* dándoles un comportamiento para que estos sepan solos cómo reaccionar ante la interacción con otros objetos.

A esta actividad la llamaremos *"diseño"* *y* será cuando debamos decidir cómo serán y cómo se comportarán nuestros objetos ante la interacción con otros objetos.

Nada más y nada menos... y hemos logrado empezar a hablar del tema sin mostrar -hasta el momento - una sola línea de código **;-)**

En resumen

Abre tu cabeza, borra todo lo que sabes hasta el momento de objetos y de código, y empecemos de cero, es más simple de lo que crees.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"La complejidad es destructiva. Chupa la sangre de los desarrolladores, hace que los productos sean difíciles de planificar, construir y probar, introduce problemas de seguridad y provoca la frustración de usuarios finales y administradores"

-- Ray Ozzie

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 3 - "Cómo pensar en Objetos"

Veremos en este capítulo cómo detectar los objetos sin preocuparnos aún del código.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Mucho software hoy día es en cierta medida como una pirámide egipcia con millones de piedras apiladas una encima de otra, sin integridad estructural, construido por fuerza bruta y miles de esclavos."

Por Alan Kay (científico de la computación)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Lo menos importante es el código"

Lo más importante –por lo menos al principio- no es jugar con el código (ya que este podrá funcionar, no dará error, pero conceptualmente nuestro sistema no funcionará como un sistema *"Orientado a Objetos"* ni aprovecharemos todas sus virtudes), **es detectar los objetos dentro de un contexto determinado.**

> *"Todo problema está sujeto a un determinado contexto, no existe un diseño que se adapte a todos los posibles contextos"*

El código con el que se construyen los objetos es meramente circunstancial, una vez que tengamos claro el diseño conceptual, luego será seguir la receta a través del manual del lenguaje de turno.

"Un niño pequeño"

Empecemos por el diálogo de un niño que recién empieza a hablar (en este caso mi hija):

> *Micaela, de 5 años, dice: "mira el perro negro y blanco, se llama Tito, le toco la cabeza y mueve la cola, y si le doy de comer, al rato, hace caca".*

Claramente tenemos **un objeto de tipo *"Perro"*** con características bastante definidas (y probablemente con algún problema en sus esfínteres).

> *"Más importante que codificar es detectar los objetos sin preocuparnos -aún- del código que los representará"* – EP

Tenemos entonces:

- ***"Un perro"***, el "objeto" propiamente dicho.
- ***"Es de color negro y blanco"***, el color es un atributo del objeto "perro"
- ***"reacciona si le tocan la cabeza"***, el comportamiento ante un estímulo externo.
- ***"mueve la cola"***, tiene acciones.
- ***"come"***, otras acciones relacionadas con su exterior/interior
- ***"hace caca"***, tiene otras acciones que están relacionadas con su interior, y que posteriormente se exteriorizan de alguna forma.

También es importante destacar, un poco más sutil, que existe **otro objeto** en este escenario y se llama **"Micaela",** y además existe (aunque no lo veamos) **un contexto** (*"todo sistema tiene un contexto, un sistema no puede aplicarse en absolutamente todos los contextos"*) donde

"viven" los objetos y que permite que se genere una interacción entre ambos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"El medio de comunicación"

De alguna forma u otra, ambos objetos tienen cosas en común: existe un medio que les permite comunicarse, pero a su vez ellos tienen los elementos para generar ese "diálogo", como así también existen "acciones" que son "internas" e "implícitas" de cada uno:

- **Aunque Micaela –y aún el perro- no lo entienda**, ambos tienen internamente distintos mecanismos de digestión y ninguno controla el mecanismo del otro.
- **El perro, que sabe que cuando está nervioso mueve la cola**, no logra entender del todo por qué si Micaela lo acaricia, esta también se mueve. Micaela sabe que si lo acaricia su cola se moverá.
- **Micaela tiene una mano y el perro una cabeza**, Micaela tiene acceso a su cabeza, y la cabeza es accesible para que la mano pueda acariciarla.

Parece tonto y simple, pero así son los objetos, y en esos temas tenemos que pensar cuando diseñamos:

El contexto, los objetos, sus atributos, sus acciones, cuáles pueden ser conocidos por otros objetos y cuales son (o deben ser) naturalmente internos del propio objeto, para finalmente hacerlos interactuar como en una obra de teatro o en un cuento, con varios posibles principios y finales según la historia que necesitemos contar.

Cuando veamos los primeros ejemplos codificados entenderán un poco más de lo que hablo **;-)**

Nota del Autor: *no, no estoy bajo los efectos de alucinógenos, la POO tiene más de observación de la realidad de lo que normalmente se cree, nunca deben diseñar un sistema* que ***"no sea coherente con la realidad".***

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPÍTULO 4 - "POO SEGÚN LOS MANUALES"

En la era "web" es muy probable que antes de consultar un libro impreso usemos nuestro navegador y busquemos material de lectura. Esto no está mal, el problema es que **existe mucha información antigua o incorrecta**, también existen demasiados "charlatanes", por lo que deberemos ser desconfiados y selectivos con lo que leemos.

Dos fuentes que son confiables y que tenemos más a mano son **Wikipedia**, principalmente para consultar los conceptos fundamentales que se aplican a cualquier lenguaje Orientado a Objetos, y **el manual oficial de PHP**, que aunque es un manual de programación y no de desarrollo (se concentra más en la sintaxis que en los conceptos), tiene mucho para enseñarnos (el resto lo haremos a través de este libro **;-)**).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"La POO según Wikipedia"

Amén que la introducción conceptual de **Wikipedia** está bien, en algunas partes puede generar confusión o no estar suficientemente explicadas. Intentaremos resaltar algunos puntos fundamentales sin reescribir el capítulo original.

De todas formas, se sugiere primero la lectura del capítulo en Wikipedia

http://es.wikipedia.org/wiki/POO

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Depurar código es dos veces más difícil que escribir código, por lo tanto si escribes el código tan inteligentemente como te sea posible, por definición no serás lo suficientemente inteligente como para depurarlo"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Luego de haber leído el capítulo original en Wikipedia, quiero resaltar los siguientes puntos fundamentales:

1. **La POO es un paradigma que tiene sus orígenes desde antes de 1990** (a partir de este año se empieza a popularizar). Por lo tanto no es ninguna excusa (menos como Desarrollador PHP) seguir a la fecha desconociendo cómo trabajar con POO o discutiendo si realmente es útil su adopción.

2. *"Los objetos son entidades que combinan **estado, comportamiento e identidad**"*

3. Fundamental, los beneficios que obtenemos usando este paradigma:

   - *"La programación orientada a objetos expresa un programa como **un conjunto de estos objetos, que colaboran entre ellos para realizar tareas**. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar."*

4. La razón de por qué no es necesario que todos los objetos que creemos tengan un id como si fuera una clave primaria de una tabla (con el fin de ubicar un objeto en particular):

   - *"De esta forma, **un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase**, al poder tener valores bien diferenciados en sus atributos."*

5. **Diferencias con respecto a la Programación Estructurada versus Programación Orientada a Objetos:** la primera se pensó como **funcionalidad por un lado y datos por otro**, es decir, llamar a una función y pasarle constantemente datos para que los procese, mientras que **la POO está pensada para tener todo integrado en el mismo objeto**.

   - *"En la programación estructurada sólo se escriben funciones que procesan datos. Los programadores que emplean éste nuevo paradigma, en cambio, **primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos**."*

6. **Muy importante es tener SIEMPRE en claro los conceptos FUNDAMENTALES**, si no los tienes claros cuando programas OO, algo está mal, seguro errarás el camino que define el paradigma: *Clase, Herencia, Objeto, Método, Evento, Mensaje, Atributo, Estado Interno, Componentes de un objeto y Representación de un objeto*. No dudes en volver a repasarlos todas las veces que lo necesites, por más experto que te consideres, siempre viene bien una relectura de nuestras bases.

7. **Características de la POO:** igual que el punto anterior, es fundamental tener claros estos conceptos cada vez que desarrollamos, con principal énfasis en el ***Principio de Ocultación*** (que es muy común confundir con Encapsulamiento), lo que explica **por qué no deberían existir los atributos públicos ni abusar de los setter/getter** (tema que veremos más adelante).

Si alguno de estos puntos no quedaron claros, sugiero su relectura en la Wikipedia.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"POO según el manual Oficial de PHP"

De la misma forma que en el punto anterior, es muy importante hacer una lectura de Referencias del Lenguaje (la base para empezar comprender PHP) y posteriormente del capítulo sobre POO en el manual oficial, **aunque algunos capítulos no aborden en profundidad cada tema** (lo cual es entendible si comprendemos que hablamos de **un manual de sintaxis** y no un tutorial para aprender a programar).

Todos estos temas los veremos más adelante y haré todas las referencias oportunas al manual oficial, pero aquí la lista de temas básicos que trata.

Clases y Objetos (PHP 5)

Table of Contents

1. Introducción
2. Las Bases
3. Auto carga de Objetos
4. Constructores y Destructores
5. Visibilidad
6. Alcance del operador de resolución (::)
7. La palabra reservada 'Static'
8. Constantes De la Clase
9. Abstracción de Clases
10. Interfaces de Objetos
11. Sobrecarga
12. Interacción de Objetos
13. Patrones
14. Métodos mágicos
15. La palabra reservada 'Final'
16. Clonado de Objetos
17. Comparación de Objetos
18. Reflección
19. Type Hinting

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Medir el progreso de programación en líneas de código es como medir el progreso de construcción de un avión en peso"

Bill Gates

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 5 - "Empezar a plasmar los objetos en un diseño"

En este capítulo empezaremos a ver cómo transformar los objetos que detectamos en nuestra observación de la "realidad" en algo "informáticamente" palpable.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Hay dos maneras de diseñar software: una es hacerlo tan simple que sea obvia su falta de deficiencias, y la otra es hacerlo tan complejo que no haya deficiencias obvias"
-- C.A.R. Hoare

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Cómo representar la estructura de los objetos"

En un capítulo anterior habíamos hablado de una situación de la realidad donde una niña (mi hija) interactuaba con un perro, **así que vamos a ir por partes y tratar de representar lo que podemos interpretar del texto** que está en un lenguaje natural y nos describe claramente los objetos y sus relaciones:

> *Micaela, de 5 años, dice: "mira el perro negro y blanco, se llama Tito, le toco la cabeza y mueve la cola, y si le doy de comer, al rato, hace caca".*

Si nos concentramos en el perro, tenemos que:

Atributos

- Color
- Nombre

Comportamiento

- Se le puede tocar la cabeza
- Mueve la cola
- Puede comer
- Sabe hacer sus necesidades

Y se desprende prestando atención a la lectura del texto que define nuestro contexto.

1) ***"Los Objetos tienen Atributos, Comportamientos y Estados"***

Todos los objetos tienen "atributos","comportamientos" ("métodos") y un "estado". Este último no es más que la información que tienen los atributos en un momento dado.

2) **Un perro se llamará "Tito" y otro "Ruffo"**, pero el atributo es el mismo ("nombre"). Si mañana el perro cambia de nombre, lo que cambia es "su estado" y el mecanismo para cambiar el estado serán sus métodos ("cambiar el nombre").

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

3) "La clase, un molde para construir objetos"

Este ejemplo es muy usado para tratar de transmitir el concepto que hay detrás. **Podríamos decir que si *"jugáramos a ser Dios"*, primero definiríamos un diseño de cómo va a ser la criatura que queremos crear**, haríamos un *"molde"* a partir de ese diseño, y posteriormente podríamos crear *"vida"* con similares características, pero que siempre serán "objetos únicos".

"Crear Personas"

Por ejemplo, si quiero crear *"Personas"* diría que todas tendrán un sexo (masculino / femenino), dos piernas, dos brazos, una cabeza y pelo sobre ella (es un diseño simple para un contexto simple, si fuera otro el contexto, muy probablemente debería cambiar mi diseño).

"Crear Perros"

Por ejemplo, podría decir que los *"Perros"* también tendrían un sexo, pero ahora tendrían cuatro patas, una cabeza y pelo sobre todo su cuerpo.

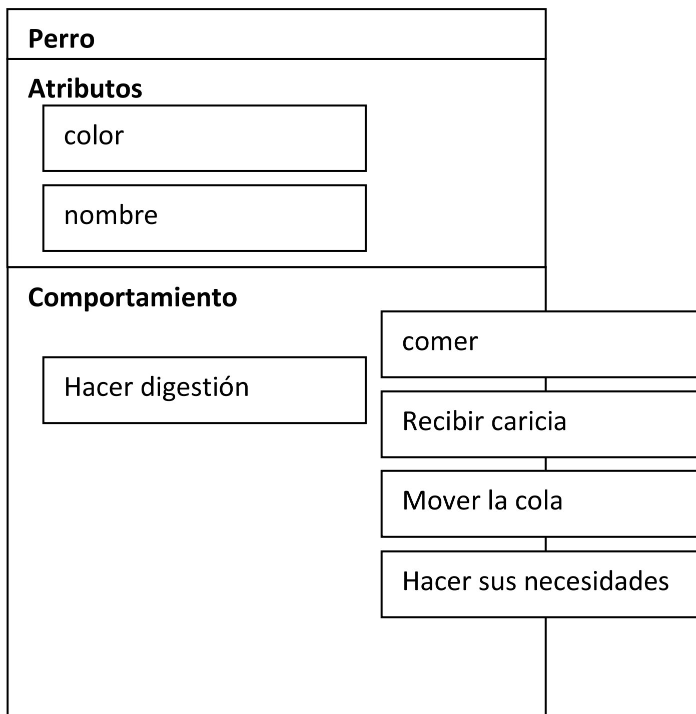
Para ambos ejemplos **ya cuento con dos moldes, el de Personas y el de Perros**, por lo tanto ahora puedo crear a *Micaela* y a *Tito*, pero también podría crear a *Martina* y a *Ruffo*, por lo que tendríamos dos personas y dos perros, con características similares, pero que serían a su vez criaturas únicas, identificables y distinguibles entre las demás criaturas, aún entre las criaturas del mismo tipo (aunque se llamaran igual y tuvieran los mismos rasgos, serían solo parecidos).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

4) ***"Los atributos y comportamientos pueden ser públicos o privados"***

Existirá información y comportamientos que serán conocidos por otros objetos (**"acceso público"**) y esto permitirá que se pueda generar una interacción entre ellos. También existirá información y comportamientos que serán internos de cada objeto (**"acceso privado"**) y no serán conocidos por los demás objetos.

Un diseño posible para nuestro Perro podría ser:



Cómo se debería leer este diagrama:

- **Todo lo que se encuentre dentro de la representación "Perro" es *"interno y privado"*** (invisible para el exterior) y lo que se encuentre **"una parte dentro y otra afuera" será nuestra *"interfaz"* con el exterior, los *"comportamientos públicos"* del objeto** que serán invocados por los "mensajes" enviados por otro objeto que quiere interactuar con nosotros.

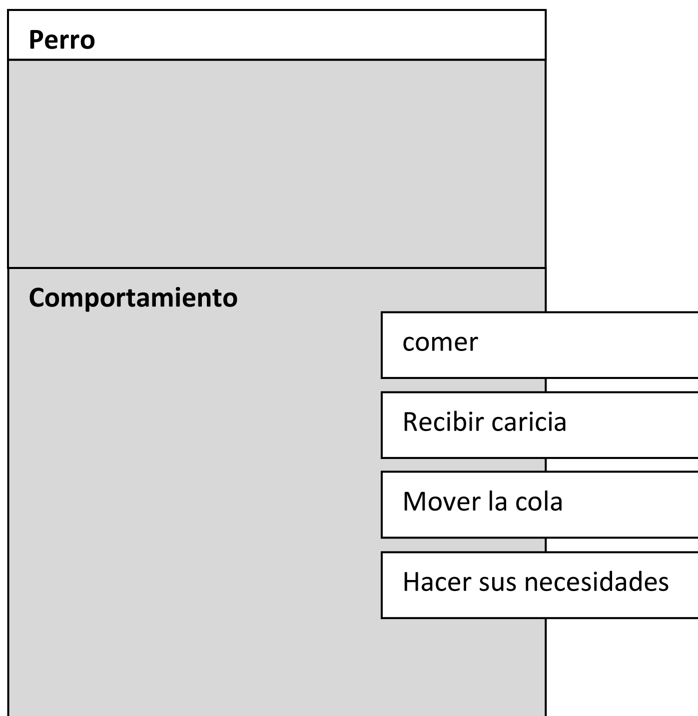
- Un objeto de tipo "Perro" tiene como atributos su "color" y su "nombre"

- Dentro de los posibles comportamientos públicos de nuestro perro podrían ser "comer", "recibir una caricia", etc.

- Dentro de los posibles comportamientos privados de nuestro perro podría ser "hacer la digestión", que muy probablemente será activada a través de otros métodos (como "comer").

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Qué es lo que vería Micaela desde el exterior:



Solo podríamos interactuar con lo que es "público" del diseño, como así lo decidimos. Ahora la pregunta sería **"¿Qué debería ser público y qué debería ser privado?"**, bueno, intentemos usar el sentido común:

- el **perro necesita poder interactuar con su exterior de alguna forma**, pero existirán "detalles" que no conciernen al exterior ni nos interesa que otro objeto pueda modificar a su antojo.

Por ejemplo, **Micaela no tiene por qué saber cómo es el mecanismo de digestión de un perro**, qué órganos entran en juego durante todo el proceso, es más, por la vida del perro creo que **tampoco Micaela debería poder tener la potestad para cambiarlo a su antojo** (ya que seguro estaría en peligro la vida del perro).

Lo que sí sucederá es que **Micaela generará indirectamente que se active el mecanismo interno de digestión al darle de comer al Perro** ("Micaela le dice al perro que coma", lo que se debe traducir como "Micaela le envió un mensaje al perro"), pero esto ocurrirá sin que Micaela sepa que sucede, solo podrá apreciar sus resultados cuando el perro haga sus necesidades (ya que las necesidades no salen solas).

Ahora bien, **¿de quién es la responsabilidad de definir todas estas reglas?**

Si, obviamente, la responsabilidad es nuestra, y un diseño será más débil o más robusto de acuerdo a cómo nosotros pensemos que deben reaccionar nuestros objetos a los mensajes que recibirán del exterior y cuanto oculten de sus propios detalles de implementación.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Nota del Autor: esto es un ejemplo teórico para tratar de transmitir varios conceptos, como todo diseño, tendrá debilidades y fortalezas, y si cambiamos de contexto, muy probablemente este diseño no servirá (no dudo que en este momento estarán pensando ***"que pasaría si…"*** y muy probablemente se encuentren con que algo no se puede hacer... bueno, vayamos paso a paso y veremos de a poco el horizonte, pero ya les voy adelantando: **no existen diseños que pueda funcionar en absolutamente todos los contextos posibles**, por esta razón es importante definir "el contexto" de nuestro sistema.

> **Nota:** a menos que estemos hablando concretamente de patrones de diseño, pero aún así se define en qué contextos se podrían aplicar y en cuales no.

Por ejemplo: el diseño de una clase *Persona* no será exactamente el mismo si estamos hablando de un sistema de reserva de películas, una universidad con alumnos o un sistema de salud. Tendrán cosas comunes, pero su diseño no será necesariamente el mismo.

En mi opinión no existe el **"100% de reuso puro"**, existirán componentes que dada su naturaleza sí podrán usarse en distintos contextos y otros directamente no, a menos tal vez que hayamos pensado de entrada que así debíamos diseñar nuestros componetes: *"reusables en varios contextos"*, aunque esto podría aumentar exponencialmente la complejidad del componente o de los sistemas.

Para más información sobre el apasionante tema sobre "diseño" se recomienda leer el siguiente artículo de Martín Fowler:

"¿Ha muerto el diseño?"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

Las clases se construyen en la etapa de diseño donde definimos qué es lo que queremos crear. Lo que creamos a partir de ellas es un objeto que "tendrá vida" (será lo que verdaderamente se ejecutará en nuestro sistema) y a la vez "único" (podrán existir muchos objetos del mismo tipo, pero podremos interactuar con ellos e identificarlos de forma única).

Dentro de las definiciones de la clase **tenemos los atributos y los comportamientos** que tendrá nuestra creación, algunos de ellos **serán públicos y otros serán privados**. Todo lo que definamos como público será nuestra "conexión" con el exterior y permitirá la interacción entre los objetos. Esta interacción se dará a través de **envíos de mensajes entre objetos**, como por ejemplo *"Micaela le da de comer al perro"*, por lo que existirán dos objetos, uno que puede *"comer"* y otro que puede decirle al otro *"que coma"* (si no existen estos métodos, el perro directamente no hará nada y estará desconectado con el exterior).

Esto se clarificará cuando empecemos a usar los diagramas UML (similares al diagrama presentado anteriormente) y a traducirlos en código concreto y ejecutable.

Tienes dudas? Quieres hacer una consulta?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Codifica siempre como si la persona que finalmente mantendrá tu código fuera un psicópata violento que sabe dónde vives"

-- Martin Golding

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 6 - "Introducción a UML"

*De la misma forma que a veces nos apoyamos en un dibujo para tratar de comprender y razonar un problema, muchas veces complejo, considero que **es fundamental contar con una "herramienta gráfica" (para armar "diagramas") que nos permita discutir y elaborar diseños sin tener que distraernos en los superfluos y cirscunstanciales detalles de la codificación según el lenguaje que necesitemos usar**.*

Diseñar Orientado a Objetos es independiente del lenguaje de programación, por lo tanto usaremos UML, un lenguaje "gráfico" independiente de la implementación.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"En las batallas te das cuenta que los planes son inservibles, pero hacer planes es indispensable"

- Dwight E. Eisenhower

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Aquí es donde entra el *Lenguaje Unificado de Modelado (UML)*, y a pesar que existe mucho software para documentar usando diagramas, pero bien podemos decantarnos por una opción "libre" como *ArgoUML (multiplataforma), StarUML (win) o Umbrello (lin)*

"UML, el medio y no el fin en sí mismo"

Como bien dice la frase, los diagramas UML son el medio y no el fin, sirven para simplificar notablemente las discusiones sobre "abstracciones" y **mejoran la comunicación entre personas**, ya sean desarrolladores como otros roles dentro de un mismo proyecto. Otra de las ventajas es que atrasa la *"codificación temprana"* y facilita estar más tiempo en la etapa de diseño.

Existen distintos tipos de diagramas, cada uno más adecuado que el otro según la situación (si tratáramos con el cliente directamente los **diagramas de Casos de Uso** serían lo más indicado).

Para lo que a nosotros nos concierne, vamos a apoyarnos en ellos para simplificar nuestro razonamiento y poder detectar más fácilmente cuando un diseño no es correcto y discutir cómo mejorarlo y para ellos nos concentraremos en los **diagramas de clases y diagramas de paquetes**.

"UML y el público objetivo"

A pesar que existen reglas para hacer cualquier diagrama UML, **los diseños siempre deben estar sujetos a interpretación y su nivel de detalle dependerá de nuestro "público objetivo"**.

Si vamos a usar un diagrama para presentarles a **programadores inexperientes** cómo deberían desarrollar un sistema, este diagrama debería disponer de la mayor cantidad de información y no dar por sobreentendido nada.

Por el contrario, si nuestro público son **desarrolladores "seniors"** con experiencia en UML, bastará con detallar solo la información pertinente al problema que se quiere resolver y el resto de información se obviará para evitar "ruido extra".

Un equipo con experiencia en UML debería poder recibir cualquier diagrama e implementarlo de forma mecánica sin tener siquiera que razonar el diseño, lo cual significa que el diagrama siempre se traduce de la misma forma, un elemento del diagrama siempre tendrá la misma traducción en código, no importa el lenguaje que usemos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"UML es independiente del lenguaje"

Como bien lo dice la frase, UML es independiente del lenguaje. Sirve para representar muchas cosas, pero en lo que respecta a la POO, cada dato que se impregne en el diagrama podrá ser traducido a cualquier lenguaje que permita implementar el paradigma de los Objetos. Si el lenguaje de turno no lo soportara en un 100%, habrá que interpretar el diseño UML y hacer los ajustes pertinentes en el código.

Algunas veces el diagrama no puede representarse exactamente en un lenguaje, por lo que deberemos ser creativos y quedarnos con el concepto que nos busca transmitir el diseño (este caso lo veremos concretamente con PHP y su problema histórico de representar explicitamente los "paquetes").

Consejo #1

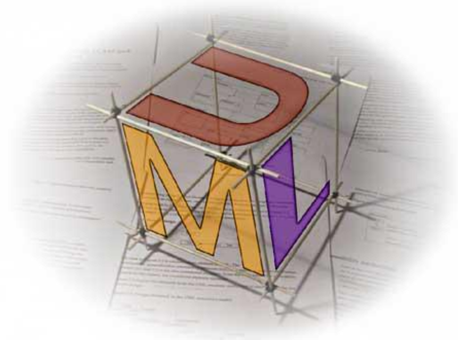
"No aprendas POO ni te apoyes en libros sobre POO que no usen UML"

Consejo #2

"No empieces a desarrollar con POO si no tienes antes –como mínimo- un pequeño diagrama de clases explicando la relación entre ellas"

Consejo #3

"Lávate los dientes antes de dormir, cuando te levantes, y luego de cada comida"



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En resumen

Los diagramas UML permiten unificar y simplificar la comunicación en un proyecto, como así también apoyar el razonamiento en la etapa de diseño de una solución.

Existen gran variedad de diagramas y son tan importantes como los diagramas MER/DER que se usan para diseñar una base de datos. **Nadie que desarrolle un sistema se le ocurriría crear una base de datos directamente en el servidor sin definir anticipadamente una estrategia en un documento**.

De la misma forma deberíamos pensar a la hora de hacer sistemas Orientado a Objetos.

Quieres solicitar más información sobre algún punto?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 7 - "Cómo representar una Clase en UML"

Empezamos con los primeros pasos en el Lenguaje Unificado de Modelado (UML) y usa las herramientas que te sean más cómodas, puede ser ArgoUML , Gliffy, Dia, etc (existen muchos programas más), o directamente dibujando en un papel o una pizarra (lo que puedes luego fotografiar y subir a una wiki).

Lo que importa son los conceptos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Primero resuelve el problema. Entonces, escribe el código"

-- John Johnson

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

PHP tiene Estándares: aunque no lo crean, Zend, la empresa que desarrolla el lenguaje y el framework, tiene definido un documento que especifica los estándares de codificación –que para no ser menos- seguiremos al pié de la letra.

"Conceptos Generales"

Primer Regla: de nomenclatura**, los nombres de las clases** son siempre en singular y la primer letra de cada palabra en mayúsculas (CamelCase), al revés de los nombres de las tablas de una base de datos, generalmente en plural y en minúsculas:

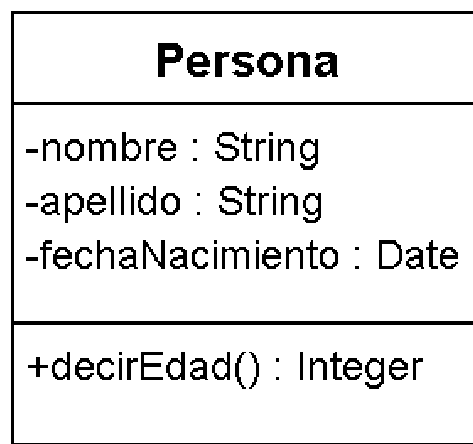
- **Tablas en base de datos:** personas, animales,usuarios, usuarios_administradores
- **Clases en POO:** Persona, Animal, Usuario, UsuarioAdministrador

Mi primer diagrama UML

Definamos un contexto para luego diseñarlo con UML:

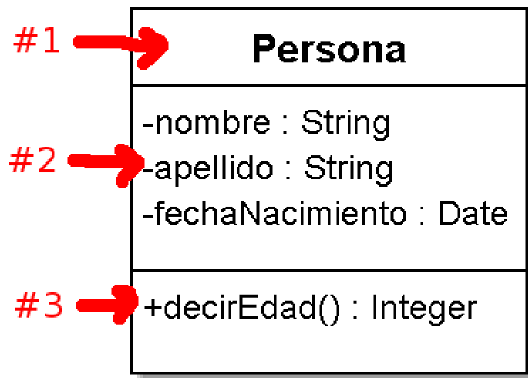
"Una persona tiene nombre, apellido y fecha de nacimiento, cuando se le pregunta qué edad tiene, responde con su edad que calcula en base a la fecha de nacimiento"

Y el diagrama UML debería verse de la siguiente manera:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Este diagrama se debe leer de la siguiente forma:



- **La Sección #1** es para definir el **nombre de la clase** (como explicamos al principio, usando CamelCase).
- **La Sección #2** es para definir **los atributos de nuestra clase (CamelCase, pero a diferencia de las clases, inician con minúscula)**, la visibilidad de cada uno (el signo "-" para privado, el signo "+" para público) y qué tipo de dato debería ser (no importa si el lenguaje lo soporta exactamente, recuerda que UML es independiente del lenguaje)
- **La Sección #3** es para definir **los métodos de nuestra clase (CamelCase, igual que con los atributos)**, la visibilidad de cada uno, los parámetros que pueden recibir y si retornan o no alguna información (para ambos casos, especificando el tipo de dato).

Regla: "todos los atributos de una clase son por defecto no-públicos".

Existe una convención (que casi no se discute) desde los principios de la POO que si cualquiera desde el exterior puede conocer y modificar los atributos de un objeto, y por ende, su "estado", todo diseño será débil (este tema lo veremos más en profundidad cuando tratemos el tema *"métodos accesores / modificadores"*).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo se traduce en código

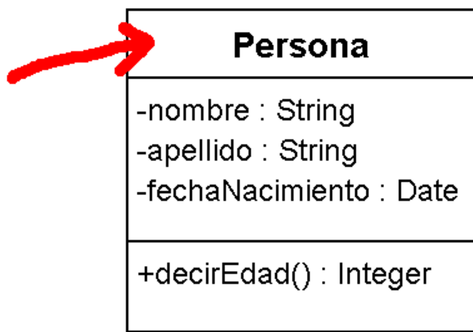
Vayamos por partes (dijo Jack, "El Destripador"), empieza por arriba y sigue secuencialmente hacia abajo, y la traducción se hará muy simple y natural.

Sección #1 – nombre de la clase

Siguiendo la nomenclatura del estándar, deberíamos primero crear un archivo / fichero con el mismo nombre que la clase: **Persona.php**

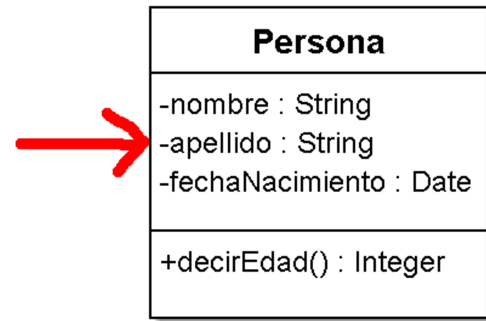
Posteriormente, todo se traduce en:

```
<?php
class Persona
{
}
```



Sección #2 – los atributos de la clase

Según el diagrama, tenemos **3 atributos**, todos "privados" (signo "-") y dos serán de tipo "String" (cadena de caracteres) y uno de tipo Date (fecha).



Aquí es donde tenemos que interpretar el diagrama y ajustarlo a nuestro lenguaje. Si quisiéramos traducir esta clase a Java no tendríamos problema porque existen "clases base" (ya que en Java *"todo es un objeto"*) como String, Integer, Date, etc. Pero, como no tenemos esta posibilidad en PHP (aún, espero que en un futuro cercano sí), podemos enfrentarlo de dos maneras, la manera **simple y pragmática**, o la **estricta y dogmática**.

Estándar de Codificación

Como podrán apreciar en el primer código de ejemplo sucede algo atípico: nuestra clase tiene las llaves a la izquierda y no existe el tag de cierre **?>**

Todo corresponde al **estándar de codificación de Zend**, particularmente para el último caso se omite cuando el archivo usará solo código PHP y no con HTML embebido.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Versión pragmática

```
<?php
class Persona
{
   private $_nombre;
   private $_apellido;
   private $_fechaNacimiento;
}
```

Comentario aparte: perfectamente podríamos haber usado una traducción directa de los atributos y decir que son $nombre, $apellido y $fechaNacimiento, pero como estamos siguiendo el estándar definido por Zend, usaremos el agregado **$_** para todos los atributos que son "privados". No olvidar, esto no es un problema que debe contemplar UML, ya es un problema del lenguaje de turno, en nuestro caso, PHP.

Versión dogmática

Por ejemplo, como Java es un lenguaje de "fuertemente tipado", cada uno de los atributos se traducirían de la siguiente forma:

Código JAVA:

```
public class Persona{
   private String nombre;
   private String apellido;
   private Date fechaNacimiento;
}
```

Pero como PHP es un lenguaje de "tipado dinámico" no requiere definir inicialmente un tipo para sus variables, el tipo es definido dinámicamente cuando se hace una asignación de valor (y su uso también dependerá del contexto en donde se encuentre)

Lo máximo que podríamos hacer es:

```
<?php
class Persona
{
   private $_nombre = '';
   private $_apellido = '';
   private $_fechaNacimiento = '';
}
```

Al asignarle un valor por defecto de tipo String, su atributo sería de tipo String y cumpliríamos exactamente con el diagrama, a no ser por la fecha, porque no tenemos nada parecido a Date.

Si tuviéramos la necesidad de definir un Integer, simplemente asignaríamos un 0 a nuestro atributo, por ejemplo:

private $_edad = 0;

SUGERENCIA, sean PRAGMÁTICOS: no conviene perder mucho más tiempo con esto ya que como PHP es de asignación dinámica de tipos, el hecho que definamos en el comienzo un valor por defecto no asegura que se vaya a mantener, y **UML es un medio y no un fin en sí mismo**. El mensaje que nos debe quedar es que ese atributo debe manejar este tipo de dato, y nosotros como desarrolladores estamos siendo notificados de ello y somos responsables de ver luego qué controles deberemos agregar para que se cumpla en el resto de la clase.

Por ejemplo, imaginemos lo siguiente: ***"la documentación que nos entregó nuestro director de sistemas dice que manejarán valores de tipo Date en el atributo fechaNacimiento"*** por consiguiente no deberíamos permitir el uso de algo que no sea de tipo Date, así tendrá coherencia el diseño con la implementación. Si el código no respeta el diseño, el código está mal y hay que corregirlo.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

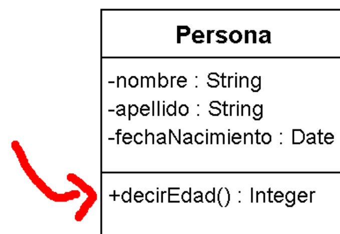
Sección #3 – los métodos de la clase

Finalmente, la traducción de los métodos. Aquí tenemos **un método público (signo "+")**, que no recibe parámetros (los paréntesis están vacíos) y luego de los ":" representan qué tipo de valor retornará (en este caso será un número entero que representará la edad).

El resultado en código sería:

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

   public function decirEdad()
   {
      /* calculo la fecha a partir de
$_fechaNacimiento */

      return $edadCalculada;

   }
}
```

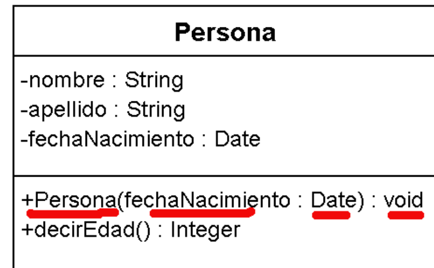
En este ejemplo se puede ver un detalle importante: **¿cómo y cuando se define el valor de la fecha de nacimiento de la persona?**



Respuesta: en ningún momento, el diagrama UML no dice nada, aquí está supeditado al público objetivo de la documentación y a su interpretación. Cabe aclarar que cuando se entrega un diagrama UML no solo van los dibujos, también se acompañan con más información escrita explicando los posibles detalles o problemas de la implementación, o simplemente se hará una reunión para tratar el tema y sacar dudas.

"El Constructor"

Si tuviéramos la necesidad de documentar alguna particularidad del **constructor**, lo que generalmente se hace **es agregar un método con el mismo nombre de la clase**, lo que representaría "el constructor" de la clase (sintácticamente sería similar a como lo hace Java y cómo lo hacía PHP4, ya que cambia en PHP5 por la palabra reservada __construct()).



Nota: al no existir el tipo Date lo que se hace es recibir un parámetro y pasarle el valor al atributo que se encuentra protegido internamente en la clase ("atributo privado").

El elemento "void": significa "vacío", "nada", es decir, debemos leerlo como **"este método no retorna nada"**. En los diagramas UML podemos encontrar que todo lo que no devuelva nada o no tenga tipo, es de tipo "void". También podemos encontrar que directamente se omita y no se muestre nada al final del método (eliminado ":void" del final).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Persona.php

```
<?php
class Persona
{
    private $_fechaNacimiento;

    /**
     *
     * @param string $fechaNacimiento 5/8/1973
     */
    public function __construct($fechaNacimiento)
    {
        $this->_fechaNacimiento = $fechaNacimiento;
    }
    public function decirEdad()
    {
        return $this->_calcularEdad();
    }
    private function _calcularEdad()
    {
        $diaActual  = date(j);
        $mesActual= date(n);
        $añoActual = date(Y);

        list($dia, $mes, $año) = explode("/", $this->_fechaNacimiento);

        // si el mes es el mismo pero el dia inferior aun
        // no ha cumplido años, le quitaremos un año al actual


        if (($mes == $mesActual) && ($dia > $diaActual)) {
            $añoActual = $añoActual - 1;
        }
        // si el mes es superior al actual tampoco habra
        // cumplido años, por eso le quitamos un año al actual

        if ($mes > $mesActual) {
            $añoActual = $añoActual - 1;
        }
        // ya no habria mas condiciones, ahora simplemente
        // restamos los años y mostramos el resultado como su edad
        $edad = $añoActual - $año;

        return $edad;
    }
}
```

¿Cómo se prueba?

```
$persona = new Persona('5/8/1973');

echo $persona->decirEdad();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Probando el objeto en un contexto determinado

Nuestra clase ya está lista y definida, ahora habría que probar de crear a partir del "molde" un "objeto" y probarlo.

> **CRITERIO:** como todo debería ser un objeto, todos los archivos deberían tener la misma nomenclatura. Pero en nuestro caso particular de la web, siempre tenemos una página inicial "index", por lo que fijaremos de ahora en más el siguiente criterio: **en los diagramas representaremos la clase "Index" pero a la hora de implementarla crearemos el archivo en minúsculas "index.php"** (o de lo contrario nuestro servidor web no lo encontrará) y en su interior podemos crear o no una clase de Index (quedará librado para discutir más adelante).

Para simplificar y no perder el foco en lo importante, probar la clase Persona, haremos el mínimo código necesario para index.php.

Listo, creamos nuestro contexto en index.php, donde creamos el objeto *unaPersona* a partir de la clase *Persona*, definimos su fecha de nacimiento, y posteriormente le pedimos que nos diga su edad.

Probando que los objetos son únicos

Creamos dos instancias a partir de la clase Persona y luego imprimimos en pantalla la estructura interna de cada uno de los objetos creados con un var_dump.

Como salida obtendremos la radiografía interna de cada objeto:

```
object(Persona)#1 (3) {
  ["_nombre:private"]=> NULL
  ["_apellido:private"]=> NULL
  ["_fechaNacimiento:private"]=> string(8) "5/8/1973"
}
object(Persona)#2 (3) {
  ["_nombre:private"]=> NULL
  ["_apellido:private"]=> NULL
  ["_fechaNacimiento:private"]=> string(8) "5/8/1973"
}
```

Si prestamos atención, el primero dice:

object(Persona) #1 y ***object(Persona) #2***

Ambos números (#1 y #2) son los identificadores internos del sistema. Por cada objeto que creamos en el contexto de index.php (mientras esté ejecutando) se irá incrementando secuencialmente y por más que los datos de los atributos sean idénticos ("estado") estamos hablando que existen en realidad "objetos distintos y únicos".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

Vimos cómo se representa una clase en UML, que no necesariamente la traducción es literal ya que dependemos del lenguaje, pero que UML siempre es independiente, no habla de sintaxis particular y siempre está sujeto a una interpretación.

¿Algo no quedó claro? No te quedes con dudas

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPÍTULO 8 - EJERCICIO "MICAELA Y EL PERRO"

Lo más importante es tener bien claros los conceptos, pero fundamental es saber aplicarlos. Iremos planteando varios ejercicios que tú mismo puedes hacer y revisar su solución más adelante.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Programar sin una arquitectura o diseño en mente es como explorar una gruta sólo con una linterna: no sabes dónde estás, dónde has estado ni hacia dónde vas"

-- Danny Thorpe

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

A partir de todos lo expuesto anteriormente se solicita:

1. **Hacer los diagramas UML de las clases** (y codificar cada una de ellas) del ejemplo "Micaela y el Perro" (se sugiere revisar todo el material, ya que hay referencias en varios capítulos).

2. **Crear todas las clases en archivos independientes**: Persona.php y Perro.php

3. **Crear un index.php** que use las clases que se crearon.

Atención:

1. **Solo con lo visto hasta el momento**, nada de diseños complejos, solo clases involucradas, atributos y métodos.

2. Evitar toda codificación extra e innecesaria (KISS).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

La idea de cada tarea es aplicar y extender los conocimientos que se debieron haber adquirido con la lectura de los capítulos anteriores.

Estos mismos ejercicios fueron realizados por alumnos de los talleres a distancia y estos son los errores más habituales que se pueden cometer:

- A partir de las clases UML se deben generar siempre archivos aparte, por clase, con la nomenclatura **Persona.php** y no **persona.class.php** o **persona.php** (debemos usar el Estándar de Codificación de Zend)

- **Las llaves "{}" van abajo solo en las clases y métodos**, para ningún caso más, ni if, ni while, etc (estándar Zend)

- **Los atributos y métodos privados** deben llevar delante "_" (estándar Zend), por ejemplo: $_nombre, _hacerDigestion()

- **Siempre realizar "exactamente" lo que se solicita** (a menos que solicite explícitamente lo contrario), no más, debemos ajustarnos siempre a los requerimientos que nos solicitan (**Principio KISS). Debemos evitar la "sobre-ingeniería"** y extender la funcionalidad de una clase, aumentando innecesariamente su complejidad. En un proyecto, donde hay más de un desarrollador es importante que la arquitectura se mantenga "simple", de lo contrario, cuando crezca, será más costoso mantener el sistema.

- **Mi sugerencia es no hacer uso de los __set y __get** que incorpora PHP5, ya que en la mayoría de los casos –según mi experiencia- debilitan los diseños al generar efectos de *"atributos públicos"*. No siempre lo genérico es bueno, como en este caso.

- **Si el método retorna algo de tipo "Comer", por ejemplo darComida():Comer**, **eso significa que el retorno es una clase Comer**, por lo cual no es correcto si el retorno en sí es un String, por lo tanto lo correcto es: darComida():String.

- **Los atributos deben iniciar siempre en minúsculas**, igual que los métodos

- **No es necesario especificar el constructor del diagrama** si este no aporta nada relevante.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Aplicación del "Principio KISS"

Si en una empresa/proyecto para hacer un sistema cada desarrollador (con toda la mejor voluntad del mundo) agrega un "extra" y trata de hacer la "super clase" (agregar funcionalidad no solicitada), la complejidad aumentará exponencialmente.

Es recomendable ajustarse a los requerimientos, a menos que estos digan que deben cubrir todos los posibles errores, situaciones, etc, pero de todas formas habrá que definir un criterio o un límite.

Recuerda: un diseño está atado siempre a un contexto, si cambia el contexto, muy probablemente deberá cambiar el diseño. No existen diseños que sirvan para absolutamente todos los contextos posibles, por lo tanto, es recomendable hacer un diseño concreto para un contexto concreto.

Sugerencias para enfrentar los diseños

En general, tratar de:

- **Empezar a pensar el diseño a partir de los objetos de más bajo nivel** (los más simples) y en ese momento "olvidarse del bosque" (así no se pierde el foco de lo que se está tratando de abstraer, el objeto concreto como entidad independiente).
- **Siempre cuando diseñen una clase, "pararse literalmente sobre ella"** razonando qué es lo que debería ver o conocer esta, sin entrar a pensar qué es lo que deberían hacer las demás clases. Tan importante como diseñar qué hacen es entender qué no deberían hacer.
- **Seguir el principio de diseño OO: *"Una clase, una única responsabilidad",*** si tiene más de una responsabilidad, debe estar haciendo más de lo que le corresponde, y deberá descomponerse en otra clase.
- **Finalmente, con él o los objetos de más alto nivel, empezar a interactuar con los objetos de más bajo nivel**, sin interesar cómo están construidos por dentro y qué hace su código interno; solo hay que tomar los objetos y pedirles la información o el comportamiento que necesitamos de ellos ("interfaz pública").
- Y el más importante de todos, Mantener un "Diseño Simple" en todos los sentidos, evitar complejidad innecesaria. Por ejemplo, puedes tomar la siguiente métrica: si un método no se puede visualizar en una sola pantalla, hay algo que evidentemente está mal y hay que descomponerlo en varios métodos (refactoring), etc.

Sugerencia:

Prefiere "Simple" sobre "Complejo"

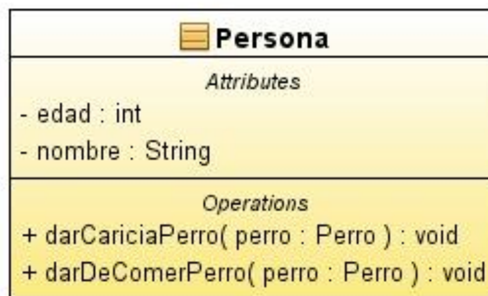
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

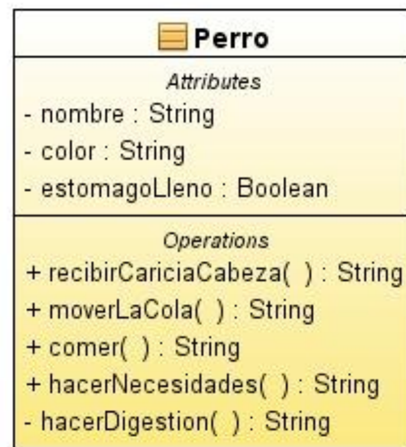
Propuesta de Diseño 1

Para este caso planteo otro posible diseño siguiendo al pié de la letra lo comentado en los capítulos anteriores.

Diagrama UML

Diagramas usando Netbeans 6.5 + plugin UML





Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Traducción de UML a PHP

Perro.php

```
<?php

class Perro
{
  private $_color;
  private $_nombre;
  private $_estomagoLleno = false;

  public function __construct($nombre, $color)
  {
      $this->_nombre = $nombre;
      $this->_color = $color;
  }
  public function recibirCariciaCabeza()
  {
    return $this->moverLaCola();
  }
  public function moverLaCola()
  {
      return 'estoy moviendo la cola!';
  }
  public function comer()
  {
      $this->_estomagoLleno = true;
      sleep(5);
      return $this->_hacerDigestion();
  }
  public function hacerNecesidades()
  {
    return 'hago caca!';
  }
  private function _hacerDigestion()
  {
      $retorno = null;

      if($this->_estomagoLleno){
          $this->_estomagoLleno = false;
          $retorno = $this->hacerNecesidades();
      }
      return $retorno;
  }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Persona.php

```
<?php
class Persona
{
    private $_edad;
    private $_nombre;

    public function __construct($nombre, $edad)
    {
        $this->_nombre = $nombre;
        $this->_edad = $edad;
    }
    public function darCariciaPerro($perro)
    {
      echo $perro->recibirCariciaCabeza() . '<br>';
    }
    public function darDeComerPerro($perro)
    {
      echo $perro->comer() . '<br>';
    }
}
```

index.php

```
<?php
require_once 'Persona.php';
require_once 'Perro.php';

$persona = new Persona('Micaela', 5);
$perro = new Perro('Tito', 'blanco y negro');

$persona->darCariciaPerro($perro);
$persona->darDeComerPerro($perro);
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Propuesta de Diseño 2

Atención con la "sobre-ingeniería", hay que evaluar si estas mejoras no son demasiadas porque el contexto no lo requiere.

Cambios

- **Se hace más genérico**, no es dependiente de un perro, puede acariciar otras cosas que sean "acariciables" (otros animales).
- **Se crea una clase Index para mostrar cómo sería hacerlo 100% POO,** usando la invocación sin instancia, directamente ejecutando la clase (más adelante profundizaremos).
- **Las clases nunca imprimen información, siempre retornan**. Solo la primer clase que invoca toda la acción del sistema ("Clase Controladora") hace la impresión que genera la página html (ubicada en el archivo index.php)

Diagrama UML

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Traducción UML a PHP

Perro.php

```php
<?php
class Perro
{
  private $_color;
  private $_nombre;
  private $_estomago;

  public function __construct($nombre, $color)
  {
      $this->_nombre = $nombre;
      $this->_color = $color;
  }
  public function tocan($lugar)
  {
      $retorno = null;

      if($lugar == 'cabeza'){
        $retorno = $this->_moverCola();
      }
      return $retorno;
  }
  private function _moverCola()
  {
      return 'estoy moviendo la cola!';
  }
  public function comer($comida)
  {
      $this->_estomago = $comida;
      sleep(5);
      return $this->_hacerDigestion();
  }
  private function _hacerDigestion()
  {
      $retorno = null;

      if(isset($this->_estomago)){
          $this->_estomago = null;
          $retorno = $this->_hacerNecesidades();
      }
      return $retorno;
  }
  private function _hacerNecesidades()
  {
    return 'hago caca!';
  }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Persona.php

```php
<?php
class Persona
{
    private $_edad;
    private $_nombre;

    public function __construct($nombre, $edad)
    {
        $this->_nombre = $nombre;
        $this->_edad = $edad;
    }
    public function tocar($objeto, $lugar)
    {
        return $objeto->tocan($lugar);
    }
    public function darComer($objeto, $comida)
    {
        return $objeto->comer($comida);
    }
}
```

index.php

```php
<?php
require_once 'Persona.php';
require_once 'Perro.php';

class Index
{
    public function ejecutar()
    {
        $persona = new Persona('Micaela', 5);
        $perro = new Perro('Tito', 'blanco y negro');

        echo $persona->tocar($perro, 'cabeza') . '<br>';
        echo $persona->darComer($perro, 'carne') . '<br>';
    }
}

$index = new Index();
$index->ejecutar();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

Siempre se pueden plantear más de un diseño y esto quedará a discusión del equipo de trabajo y ajustándose a un contexto determinado. Se plantearon un par de diseños que no necesariamente son *"la única solución posible"* y se busca extender la comprensión de los conceptos tratados.

Encontraste algún error? Quieres enviar una sugerencia?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 9 - Los métodos "getter / setter" o "accesores / modificadores"

He visto mucha documentación que habla sobre el tema de los métodos "getter / setter", o traducido al castellano los métodos "accesores / modificadores", y la mayoría se va a los extremos, perdiendo de explicar de forma simple la razón de su existencia, y algo más importante, por qué no deberíamos abusar de esta práctica.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Si la depuración es el proceso de eliminar errores, entonces la programación debe ser el proceso de introducirlos"

-- Edsger W. Dijkstra

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

El origen de la necesidad de los "set y get" para los atributos

En **PHP4 todos los atributos de un objeto son siempre atributos públicos**, es decir, cuando creamos una instancia del objeto a partir de la definición de la clase, tanto para leerlos como para modificarlos.

Definición de la clase "Usuario" en PHP4

```
class Usuario{
    var $nombre;
    var $nombreReal;
    var $edad;
    var $clave;
}
```

Creo el objeto "unUsuario":

```
$unUsuario = new Usuario();
```

En este momento el usuario está vacío, pues sus atributos no tienen ningún valor asignado. Por lo tanto, le daremos sentido a este objeto:

```
$unUsuario->nombre = "eplace";
$unUsuario->nombreReal = "Enrique Place";
$unUsuario->edad = "33";
$unUsuario->clave = "pepe";
```

Definamos un contexto de ejemplo

Supongamos ahora que nuestro sistema es mantenido por varios desarrolladores y que una parte del sistema es mantenida por un "estudiante de programación" que decidió unilateralmente que cuando le llegue el objeto "unUsuario" le pondrá siempre el nombre en mayúsculas y le colocará una clave por defecto si esta está vacía.

Nosotros, como *"desarrolladores experimentados"*, nos sentimos molestos por la tontería que acaba de hacer el "estudiante"... ahora la pregunta es, como evitamos que esto suceda?

```
$unUsuario->nombre = strtoupper($unUsuario->nombre);

if (is_null($unUsuario->clave)){
    $unUsuario->clave="clavePorDefecto";
}
```

Uno de los problemas aquí es que PHP4 no soporta la definición del "alcance" (o también llamada "visibilidad") de los atributos y métodos, algo habitual en cualquier lenguaje serio de tipo *"Orientado a Objetos"*. Esto hace que -aunque no nos guste- el desarrollador de turno pueda hacer lo que quiera con los atributos de cualquier objeto a su alcance.

Migremos la sintaxis a PHP5

PHP5, consciente de este problema, implementa la definición de "visibilidad" de los atributos y métodos, como lo haría Java o .Net. Si hiciéramos una migración "mecánica" de nuestro código, este cambiaría la sintaxis a esta forma (ya que la sintaxis "var" pasa a desuso y esta definía a los atributos siempre "públicos"):

```
class Usuario
{
    public $nombre;
    public $nombreReal;
    public $edad;
    public $clave;
}
```

Ahora disponemos de las siguientes opciones gracias a la nueva sintaxis: public, private y protected.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Ahora bien, si queremos que el *"estudiante"* no pueda modificar nuestros datos, podemos pasar a colocar todos los atributos como "private":

```
class Usuario
{
    private $_nombre;
    private $_nombreReal;
    private $_edad;
    private $_clave;
}
```

Listo, ahora cuando el *"estudiante"* quiera ver o modificar un atributo, el sistema le enviará un error. **El problema ahora es que nosotros queremos que:**

1. La edad se pueda saber y cambiar en todo momento.
2. Se pueda saber el nombre del usuario, pero no modificarlo
3. No nos interesa que se sepa el nombre real del mismo
4. Pero queremos que pueda colocar una nueva clave si el usuario se la olvida, pero no saber la que existe actualmente

Esto no lo podemos hacer ni teniendo todos los atributos públicos como sucedía con PHP4 ni restringiendo toda la visibilidad como nos permite ahora PHP5.

¿Cómo se hace entonces?

Por un tema de principios de la POO los atributos de los objetos deben ser siempre "privados" (concepto "encapsulación": no son accesibles desde fuera del objeto, solo el objeto tiene la potestad de usarlos directamente) y se deberán crear métodos públicos que sustituya una u otra operación, o ambas, cada vez que la situación lo amerite:

- un **método "setter" para "cargar un valor"** (asignar un valor a una variable)
- un **método "getter" para "retornar el valor"** (solo devolver la información del atributo para quién la solicite).

Veamos cómo se resuelve, paso a paso.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimiento 1

La edad se puede acceder y modificar en todo momento, por consiguiente se deben agregar los dos métodos, un "get" y un "set" para ese atributo:

```
class Usuario
{
    private $_nombre;
    private $_nombreReal;
    private $_edad;
    private $_clave;

    public function getEdad()
    {
       return $this->_edad;
    }
    public function setEdad($edad)
    {
       $this->_edad = $edad;
    }
}
```

Pronto, ahora el atributo se puede consultar o modificar no directamente, solo a través de los métodos "get / set". En este caso no se nota la utilidad, pero pasemos al siguiente requerimiento.

Recordatorio

Estamos usando el estándar de codificación definido por la empresa Zend. Por ejemplo, los métodos y atributos privados deben iniciar con "_" y todas las llaves de métodos y clases inician a la izquierda (ninguna otra más, como un if, for, etc).

Requerimiento 2

Poder saber el nombre del usuario pero no modificarlo, para eso hay que agregar solo un método get para ese atributo:

```
class Usuario
{
    private $_nombre;
    private $_nombreReal;
    private $_edad;
    private $_clave;

    public function getEdad()
    {
     return $this->_edad;
    }
    public function setEdad($edad)
    {
     $this->_edad = $edad;
    }
    public function getNombre()
    {
     return $this->_nombre;
    }
}
```

Ahora se puede consultar el nombre pero no modificar, pues el atributo no es visible desde fuera del objeto, solo a través de los métodos públicos que vamos definiendo.

Requerimiento 3

No nos interesa que se sepa el nombre real del usuario

Lo dejamos como está y queda inaccesible desde fuera del objeto.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimiento 4

Queremos que pueda colocar una nueva clave si el usuario se la olvida, pero no saber la que existe actualmente

Para eso, hacemos lo contrario que con el atributo nombre, agregamos un método "set" pero no el "get":

```
class Usuario
{
   private $_nombre;
   private $_nombreReal;
   private $_edad;
   private $_clave;

   public function getEdad()
   {
       return $this->_edad;
   }
   public function setEdad($edad)
   {
      $this->_edad = $edad;
   }
   public function getNombre()
   {
      return $this->_nombre;
   }
   public function setClave($clave)
   {
      $this->_clave = $clave;
   }
}
```

Pronto, usando simples métodos podemos reforzar el diseño de nuestro objeto, restringiendo según nuestra necesidad el acceso a sus atributos.

Formalicemos: "Getter/Setter es solo un tema de conceptos"

Cuando empezamos a aprender a usar 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).

Generalmente nuestros objetos deberían contener determinada información que no necesariamente el usuario de nuestro objeto debería saber, porque no conviene, o porque directamente no corresponde y permitirle acceder a la misma es aumentar innecesariamente la complejidad del uso del objeto.

Regla: "Evitar que el objeto muestre detalles de su implementación"

Por ejemplo: si tu sabes que tu objeto "Persona" tiene una fecha de nacimiento y calculas su edad, **no deberías poder permitir que alguien "de afuera del objeto" cambie la edad**, pues está relacionada con otra información (fecha de nacimiento) y a partir de ella es que se genera (calcularEdad()).

Lo correcto sería modificar la fecha de nacimiento para que el propio objeto la vuelva a calcular.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Los detalles del funcionamiento del objeto son internos, y el usuario del objeto (otro desarrollador u otro sistema) no debe ni necesita conocerlos. Por consiguiente ya tenemos otro caso claro, el atributo "edad" no debería poder ser modificado externamente, pero sí consultado cada vez que se necesite.

Si este fuera un atributo público sería imposible restringir una operación y habilitar la otra. De ahí que nace el concepto de "getter / setter", o de "métodos accesores / modificadores", o como muchos autores se refieren a estos métodos especiales como "propiedades" (pero creo que puede generar confusiones con el concepto "atributo", pues muchos otros autores usan las mismas palabras indistintamente).

Si tú colocas atributos privados, estos serán solo *"vistos / accedidos / usados / modificados"* dentro de la propia clase. **Si tu quieres que puedan ser accedidos desde fuera de la clase, debes crear métodos públicos que internamente "accedan" a los atributos, pero que los dejarán "resguardados" dentro del objeto** (no hay que olvidar que en un caso pueden hacer una operación u otra, no necesariamente ambas).

Recalco, digo "nomenclatura", puesto que los nombres de los métodos pueden llamarse como quieras, pero si cumplen con ambas definiciones (get/set), cumplirá con la esencia de la funcionalidad. El tema es que, por convención, se tiende a reconocer así a los métodos que solo sirven para *"hacer operaciones básicas como si trabajáramos con atributos públicos"*, y que además, no deben tener más complejidad que la que describo.

De lo contrario, ya sería mejor que crearas un método común y corriente, asignándole un nombre claro a su acción.

Errores más comunes

Definir todos los "get" y "set" para todos los atributos existentes. Es casi lo mismo que si fueran todos públicos, careciendo de utilidad.

Lo habitual es que esto no debería suceder, cuanto más escondamos de nuestro objeto mejor será, pues disminuimos la complejidad de su uso y evitamos que cambie a un estado que no queremos, y **cuando menos se sepa de cómo trabaja internamente, más fácil será poder reutilizarlo en contextos distintos** (deberá ser raro que debas dejar disponible todo y no existan algunos que sean solo de uso interno).

Otro error común es agregarle más lógica que asignar un valor o retornar el valor del atributo. Si necesitas agregarle lógica, ya dejan de ser "get / set", lo cual es mejor que cambiemos de nombre y lo dejemos con los demás métodos comunes de nuestra clase.

Por ejemplo: si para cambiar el nombre del usuario, antes de asignar el valor voy a la base de datos y verifico que no exista otro igual, y luego en caso de nombre duplicado genero otro alternativo, etc, **preferiría o desglosar el método setNombre en varias llamadas a métodos privados**, o directamente crear un método nuevo que se llame **cambiarNombre**, reflejando un proceso fuera del simple get/set.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Nota: esto es muy a nivel personal, hay opiniones encontradas sobre tener una lógica elemental dentro de un set/get o hacerlo tan extenso como necesitemos. Claramente estoy en el primer grupo.

Como detalle, para que quede más evidente y visualmente claro

Como todo es convención a la hora de definir la forma de trabajo en un entorno de desarrollo, **es habitual que los atributos siempre vayan juntos al principio de la clase e inmediatamente -antes de empezar a definir los métodos- deberían estar los métodos "accesores / modificadores"**. Lo que se suele hacer, pues son métodos muy simples y generalmente carentes de mucha lógica (como se explica en el punto anterior), tal vez sería mejor hacerlos en una sola línea, sin indentación:

```
class Usuario
{
  private $_nombre;
  private $_nombreReal;
  private $_edad;
  private $_clave;

  /** getter / setter */
  public function getEdad(){return  $this->_edad;}
  public function setEdad($edad){$this->_edad = $edad;}
  public function getNombre(){return $this->_nombre;}
  public function setClave($clave){$this->_clave = $clave;}
}
```

Nota

De todas formas no tomar esto más de un ejemplo, ya que el estándar oficial de codificación para PHP (el que define la empresa Zend) no sugiere en ningún momento esta práctica.

Trata de seguir las siguientes reglas

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

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

El tema no es tan complejo, de forma genérica esta es la explicación de qué es "getter/setter" y para qué sirve y cómo se usan.

También quiero destacar que a pesar de existir esta técnica, no quiere decir que deba ser usada o que su uso esté completamente permitido. **Hay que entender que la POO no debería hacer uso de de los getter y setters** ya que tendríamos acceso de forma directa al "estado del objeto" (la información que tienen los atributos de un objeto) y permitir el acceso o modificación genera el mismo efecto de manipulación que si fuera una operación de "corazón abierto".

Nuestro objeto debería estar protegido del exterior y no permitir tener acceso directo a sus atributos, y trabajar siempre sobre sus métodos ("los métodos definen el comportamiento del objeto").

En caso de no poder hacerlo, se podría priorizar disponer de "get" (obtener valor de un atributo de forma directa) y no "set" (modificar un valor directo de un atributo), y **en el peor de los casos, tener un uso muy controlado de los "set".**

Para más información sobre diseño OO, discusiones teóricas, buenas prácticas, etc, consultar escritos de gurúes como Martín Fowler.

Quieres solicitar un ejemplo adicional?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 10 - "Cómo representar las relaciones entre clases en UML"

Definir las relaciones entre las clases es la parte medular de la POO y donde verdaderamente construimos el Diseño OO. Es importante hacer un buen diseño individual de clases, pero más aún saber qué significa cada relación, cuando se deben usar y que impacto tiene sobre todo el sistema.

IMPORTANTE

Aquí entenderemos por qué **considero** **FUNDAMENTAL** estar apoyado por diagramas para poder visualizar, razonar y discutir un diseño OO. Las relaciones, el significado y sus consecuencias son el corazón de la disciplina. Por ejemplo, nadie concibe enseñar Base de Datos sin empezar primero por los diagramas de relaciones DER/MER.

Empecemos con las relaciones más básicas y fundamentales: ***"la dependencia y la asociación"***

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Unas buenas especificaciones incrementará la productividad del programador mucho más de lo que puede hacerlo cualquier herramienta o técnica"

Milt Bryce

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

La Relación de Dependencia

Definición: *"es una relación de uso entre dos entidades" (una usa a la otra)*

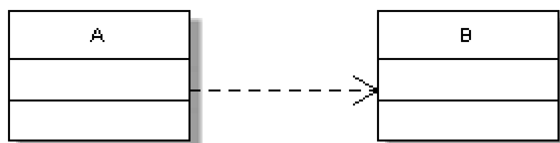
La relación de dependencia es cuando una clase depende de la funcionalidad que ofrece otra clase. Si lo vemos del punto de vista "Cliente / Servidor" (existe una entidad que necesita de un "servicio" y otra entidad que lo provee), se puede decir que una clase es "cliente" del servicio de otra clase.

Esta es la relación más básica entre clases y a su vez comparada con otro tipos de relaciones, la más débil.

Representación UML

La representación en UML es una "flecha punteada" o "discontinua" que va desde la clase "cliente" del servicio/funcionalidad hasta la clase que ofrece el "servicio/funcionalidad".

Un diagrama genérico sería:



Todos los diagramas tienen varias interpretaciones, este en concreto nos dice que:

- 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.

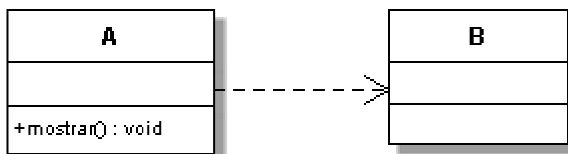
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo se traduce a código

Solo 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.

Por consiguiente, completemos los diagramas de ejemplo:



Caso 1 – instancio un objeto B dentro de un método de A

Y traducido a código será:

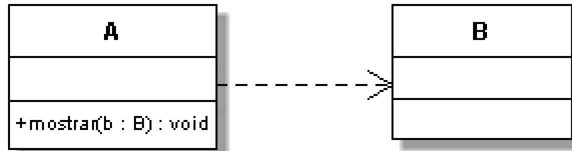
A.php:

```
<?php

require_once 'B.php';

class A
{
   public function mostrar()
   {
       $b = new B();

       /* etc */
   }
}
```

Presta especial atención en el "require_once", ya que esta sentencia 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).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

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



A.php
```php
<?php

require_once 'B.php';

class A
{
   public function mostrar(B $b)
   {
       echo $b;

       /* etc */
   }
}
```

Varios detalles 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 –sin llegar a perder el tipado dinámico- reforzar el diseño al solo permitir que ingresen por parámetro objetos del tipo que especifiquemos (ampliaremos más adelante).

Finalmente, **la "flecha punteada" representa que es una "relación débil"** en contraposición a la siguiente relación, la "asociación", que se representa con una "flecha continua" y representa una relación "más fuerte".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

La Relación de Asociación

Definición: *"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. Si lo codificáramos esto 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.**

Por ejemplo

Un auto está compuesto por un motor, un tanque de combustible y una antena de radio, por lo tanto tendríamos:

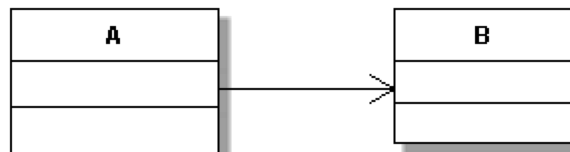
- un objeto Auto y como atributos del mismo tendríamos los objetos Motor, Tanque y Antena.

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.

Representación UML

La representación en UML es una "flecha continua" que va apuntando desde la clase "compuesto" hasta la o las clases "componentes".

Un diagrama genérico será:



Nuevamente, los diagramas tienen varios mensajes que nos transmiten:

- 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.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo se traduce a código

Creando una clase A que tiene como atributo un objeto de tipo clase B.

A.php

```php
<?php

require_once 'B.php';

class A
{
   private $_b;

   public function __construct()
   {
       $this->_b = new B();
   }
}

$a = new A();
```

Varios detalles a tener en cuenta:

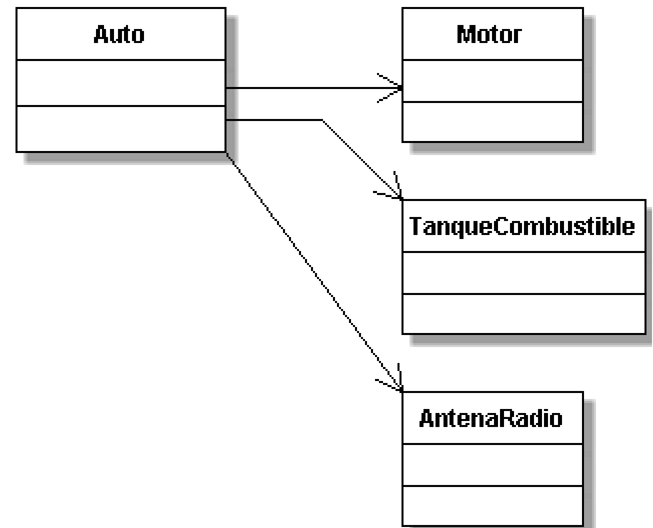
- **En el diagrama no se especifica en la clase A, en la zona de atributos, el atributo "b",** pero se puede inferir de la relación representada en el diagrama. Por lo tanto está sujeto a decisión de cada uno agregar o no más detalle en el diagrama (lo cual dependerá siempre del público objetivo del mismo).

- **Como no se especifica, tampoco parece decir nada sobre su visibilidad:** mantendremos siempre el criterio que los atributos por defecto son siempre "no públicos", por lo tanto por defecto son "privados" (más adelante veremos otras posibilidades).

- **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.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

UML del ejemplo del Auto



Auto.php

```php
<?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.

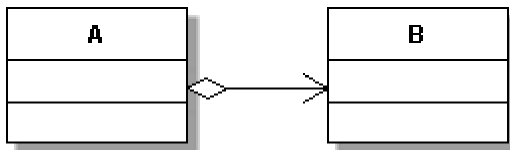
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Variaciones de Asociación: Relación de Agregación y Relación de Composición

Se puede especializar la relación "asociación" en dos tipos más (note que las flechas siguen siendo continuas pero agrega un rombo en su inicio).

Relación de Agregación

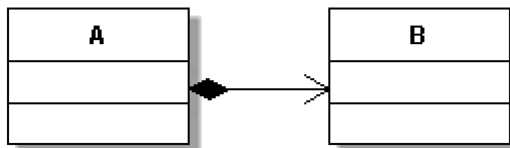
Es una relación de asociación pero en vez de ser "1 a 1" es de "1 a muchos", la clase A agrega muchos elementos de tipo clase B



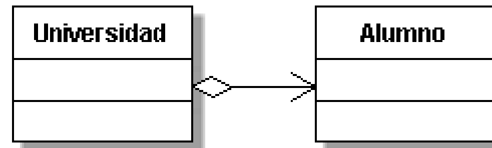
Relación de Composición

Similar a la agregación, solo aporta semántica a la relación al decir que además de "agregar", existe una relación de vida, donde elementos de B no pueden existir sin la relación con A.

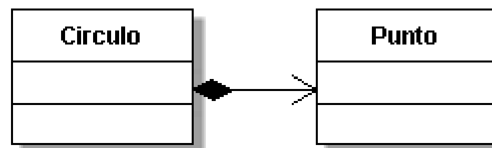
Desde el punto de vista de muchos lenguajes (como Java o PHP) esto no necesariamente genera ningún cambio en el código, pero puede ser tomado como parte de la documentación conceptual de cómo debería ser el diseño del sistema.



Ejemplos de Agregación y de Composición



"La Universidad agrupa muchos Alumnos"



"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

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Ejemplo de traducción a código

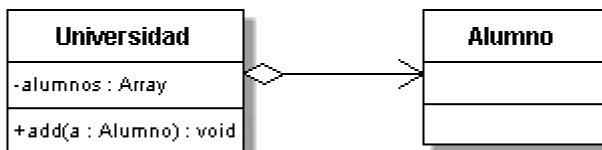
La forma de representar ambas variantes es simple y como en todos los casos, mecánica.

El elemento del lenguaje que por defecto nos permite contener varios elementos es el array, por lo tanto del diagrama se desprenden (aunque no esté explícitamente) 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).

Teniendo en cuenta lo anterior, podría apuntar mi diagrama a un público que no le fuera evidente su implementación, por lo tanto podría ser tan específico cómo:



Aunque podría pecar de redundante, ya que el diagrama original ya representa toda esta implementación.

Ejemplo de traducción a código

Universidad.php

```php
<?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
 */
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¿Qué relaciones se deberían evitar?

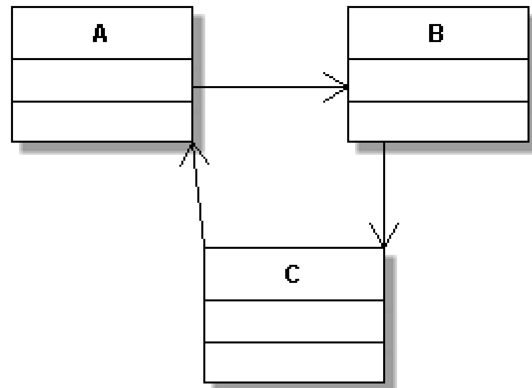
Las **relaciones bidireccionales** y las **relaciones cíclicas**, ya que la visibilidad es en ambas partes (A y B) y un cambio en una clase genera un impacto en la otra y viceversa, logrando una inestabilidad muy peligrosa.

Ejemplo de cómo se representa una relación bidireccional

ElHuevoOLaGallina.php

```php
<?php

class A
{
   private $_b;

   public function __construct()
   {
      $this->_b = new B();
   }
}

class B
{
   private $_a;

   public function __construct()
   {
      $this->_a = new A();
   }
}
```

Ejemplo de relación cíclica



Creo que es evidente el problema sin necesidad de codificar el ejemplo, un cambio en A afecta a C, como afecta a C, también afecta a B, que a su vez afecta a A… y sigue.

Caso de Ejemplo

"¿Qué sucedería si tenemos una clase Curso y otra Profesor? El profesor puede impartir varios cursos y un curso puede tener varios profesores. ¿Cómo haríamos las relaciones? ¿Dos flechas de agregación, una de Curso a Profesor y otra de Profesor a Curso?"

Aquí es cuando entra la "decisión de diseño" en base al análisis de lo más conveniente según el problema que queremos resolver y de evitar "todo lo posible" hacer relaciones cíclicas / bidireccionales.

La sugerencia es **"deberíamos optar por uno de los dos diseños" (no ambos)** de acuerdo a la navegación que sea más conveniente de acuerdo a las necesidades del sistema, qué información requerirá con más frecuencia.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Si va a ser más frecuente buscar los cursos de un docente o si a partir de un docente vamos a necesitar saber sus cursos.

Aún eligiendo uno u otro diseño, siempre podremos "navegar" por ambas formas, solo que una va a costarnos más que la otra.

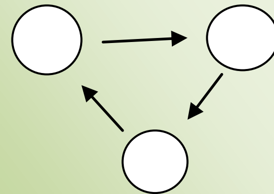
Si tenemos Curso -> Profesor

- Se resuelve directamente si necesitamos los profesores de un curso.
- Pero, para obtener de un docente cuales son sus cursos, deberíamos recorrer todos los cursos para recién saber qué cursos tiene asignado.

Siempre, La última alternativa debe ser optar por una relación bidireccional ó cíclica, porque aumentas problemas y costos de desarrollo.

Confusión común

A veces podemos cometer el error de decir "cíclicas" cuando deberíamos decir "bidireccionales", en sí el efecto es el mismo, solo que en bidireccional no hay un ciclo porque es una relación de 2 clases ("en dos direcciones") y para hacer una relación "cíclicla" se necesita por lo menos 3 clases para poder hacer un ciclo.



Comentario final : Imaginar cómo sería el funcionamiento de un sistema con relaciones bidireccionales y a su vez cíclicas entre varias clases… lo más parecido a un *"Castillo de Naipes"* **;-)**

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumiendo rápidamente algunos conceptos complementarios

"Navegabilidad"

La lectura que debe hacerse al interpretar los diagramas es:

A -> B

- "B no tiene idea de A"
- "A puede invocar métodos de B, pero no al revés"

A – B

- Bidireccionalidad: "A ve a B, pero en algún momento B ve a A"

"Definición de Roles"

Las relaciones tienen nombre: "tiene", "contiene", "fabrica", etc.

A -tiene-> B

Y es parte de la documentación especificar los roles de cada relación para ayudar a su interpretación y codificación posterior.

"Multiplicidad"

Parte de la documentación de UML, en una relación A->B, poder especificar la cantidad de objetos de B que se pueden ver en A

Por ejemplo:

- 1
- 1..*
- 0..*
- 3..14

Donde en el primer caso decimos que en A veremos siempre un objeto de B, en el segundo caso "uno o muchos", en el tercer caso "ninguno o muchos" y el en último caso "de 3 a 14".

Por ejemplo, podrías decir que tienes una Escuela -> Alumnos, entonces podrías agregar a la flecha:

- 1..N
- 0..N

Primer caso, significa que **en algún momento la escuela ve 1 o más alumnos (N)**, pero si haces el segundo caso, estás diciendo que en algún momento la escuela puede que no tenga alumnos.

Imagina una relación Esposo -> Esposa, podría ser 1..1, ya que si haces 1..2 ya serías bígamo

¿Cómo nos afecta cuándo debemos traducilo a código? Que si tienes 1..1 debes controlar que no pueda suceder (o que de error) si alguna vez no se cumple esta regla.

Es un elemento más que se puede agregar en la documentación y que aporta información a quién tiene que interpretar el diagrama y generar el código a partir de él.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Vimos cómo se representan **las relaciones básicas entre las clases**, cómo la dependencia es una relación **más débil** que la asociación, y cómo la agregación **es una variante** de asociación, y que la composición se diferencia de la agregación solo por **"semántica".**

Fundamental es entender la visibilidad entre las clases, el sentido de las relaciones, **y por qué es importante que estas se den de forma acotada y medida**, ya que existir flechas que salgan hacia todos lados, de forma bidireccional o cíclica, generan que nuestro sistema sea completamente inestable y con un **alto costo de mantenimiento**, ya que cualquier cambio en una clase generará un impacto contra las demás clases del sistema.

De aquí en más se empezará a entender que aprender POO o diseñar un sistema OO **no puede hacerse sin plantear un diagrama UML** analizando meditadamente las relaciones que deben existir entre las clases, ya que es la única forma de graficarlas y visualizarlas con facilidad.

Tranquilos, sobre estos temas vamos a ir presentado ejercicios de ejemplo con sus soluciones a partir de varios capítulos a continuación, ya que este es un tema medular ;-)

Quieres bajar el código fuente de los ejemplos?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPÍTULO 11 - EJERCICIO "MICAELA, EL PERRO Y LA ESCUELA"

Ampliemos progresivamente los ejercicios prácticos aumentado gradualmente la complejidad.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Cuando se está depurando, el programador novato introduce código correctivo; el experto elimina el código defectuoso"

-- Richard Pattis

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

Partiendo como base el problema presentado anteriormente, ahora se agregan los siguientes requisitos:

"Micaela, además de tener un perro e interactuar como vimos, **ella va a la escuela 'Dos Corazones' junto a 5 niños más**. Uno de esos niños es hermana de Micaela y se llama Martina (3 años), ambas son dueñas del mismo perro. Tienen un amigo que se llama Marcos (6 años) y él es dueño de un gato de muy mal carácter, que si le tiran de la cola, es seguro que araña!"

Guía

Primero

- Representar las relaciones entre las clases Index, Persona y Perro.

Segundo

- Representar la Escuela, que debe poder contener a todos los niños de la letra.

- Representar la relación de "hermano" entre Micaela – Martina.

- Representar el Gato

- Usar toString() (ver manual) para que cada objeto sepa cómo convertirse a "cadena de texto".

- Fundamental, se hará más hincapié que el diseño y código sea KISS, cuanto más código no solicitado se encuentre, más puntos se perderán.

Se espera

Que el Index esté representado en los diagramas y ser la *"Clase Controladora"* que inicia el sistema), y **ninguna clase puede imprimir por su cuenta (hacer un "echo"), solo Index (todos los métodos deben retornar datos hacer ningún tipo de impresión).**

Explicación: *"Controladora"*

Se dice ***"clase controladora"*** a una clase que toma el control de la solución de un problema o situación concreta desde una visión de alto nivel. Generalmente a partir de ella se crean y usan la mayor parte de los objetos involucrados en la solución.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

La idea de cada tarea es aplicar y extender los conocimientos que se debieron haber adquirido con la lectura de los capítulos anteriores. Se define un nuevo contexto para evaluar la evolución del lector.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Errores Comunes"

Estos fueron los errores más comunes que han cometido alumnos en cursos a distancia:

- **El constructor demasiado atado al contexto del problema:** he visto casos donde en el constructor de la Persona se agrega la posibilidad de recibir un "hermano". **La sugerencia es que el constructor sea siempre simple y que contenga la mínima información que represente a la clase para crearla**, como podría ser el nombre y el apellido, tal vez la fecha de nacimiento, no mucha información más. Nuestra decisión luego es, si la persona al crearse no cuenta con su fecha de nacimiento, ¿permito que se pueda crear? ¿O requiero esa información siempre? De la misma forma **considero que definir un hermano estaría separado de la creación de la persona**, por lo tanto sería más conveniente que si tiene un hermano, poder hacer esa relación con un método posterior (agregarHermano() o setHermano(), según lo que necesitemos hacer).

- **Comprender la utilidad de Index, más allá de lo evidente:** index.php podría ser mañana admin.php, no importa su nombre, lo que importa es que **ese es el punto de partida donde construimos la solución a nuestro problema**, implementamos nuestros requerimientos, juntando todos los objetos que implementamos y empezamos a hacer que interactúen para que trabajen para nosotros. Esto nos da una idea de que cuando trabajamos diseñando una clase nos tenemos que **olvidar del resto del mundo**, y recién **"la visión de bosque"** la tenemos cuando nos paramos sobre un "index.php" (o el nombre de turno).

- **Olvidar representar en código las flechas del diagrama UML:** hay que entender que –por ejemplo- si desde index hacemos "require_once" a 4 clases, si o si, significa que hay 4 flechas hacia esas clases, y viceversa, si hay 4 flechas desde Index, debemos crear 4 "require_once" en Index.

- **Confundir desde donde hacer los "require_once":** debemos entender que PHP "suma" todos los fuentes como uno solo y luego lo ejecuta, por lo tanto, si colocamos todos los "require_once" solo en el index, el sistema funcionará, pero no es lo correcto. **Cada clase debe mantener sus relaciones de acuerdo al diagrama**, ya que mañana funcionará en otro contexto y ella debe saber qué clases necesita para funcionar**. Si todo está en Index, esto no representa las relaciones desde la clase correcta**. Tampoco es problema que las flechas o "require_once" se repitan (el mismo "_once" hace que solo tome la primera invocación y el resto se ignore). **Por lo tanto deben ir desde Escuela hacia Persona y desde Persona hacia sus mascotas.**

- **No respetar el estándar de codificación de Zend**, en particular el uso de las llaves {} para los métodos y las sentencias internas. Las llaves inician a la izquierda solo cuando es una clase o un método, en ningún otro lado más.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

- **No hacer "validación de tipo" ("Type hinting"):** en el caso de la Escuela, al agregar Personas, validar que solo se puedan agregar del tipo la clase "Persona" y no otro, lo mismo para el caso en que el diseño de Persona fuera "agregar" Mascotas (refuerza el diseño, es más robusto ante situaciones no esperadas).

- **Los atributos van siempre privados (99,99% de los casos), tanto por UML como por código.**

- **Auto relación Persona -> Persona:** a pesar que una persona tenga elementos de tipo Persona, no hace falta en la clase Persona agregar un require_once hacia Persona.

- **Evitar relaciones cíclicas:** se vieron diseños que planteaban que la Mascota tuviera una colección de Dueños. En sí el diseño no está mal a menos que genere una relación cíclica entre las clases, es decir, Persona ve a la Mascota y la Mascota a la Persona (A<->B), por lo tanto cualquier cambio en una clase afecta a la otra y así sigue en ciclos. Por lo tanto, este diseño debería evitarse para anular la relación cíclica (o romperla de alguna forma).

- **El toString es un método reservado __toString:** PHP5 fija para varios métodos especiales (constructor, destructor, toString, etc) una sintaxis distinta, le agrega "__" (dos guiones inferiores) al principio del método. Por lo tanto, para que toString funcione hay que implementarlo así, de lo contrario no funcionará al hacer **echo $persona;**

- **Se presentan algunos casos de "desborde de ingeniería"**. Hay que recordar el **Principio KISS**, por lo tanto debemos realizar las siguientes preguntas:

  - ¿Si para implementar un ejercicio tan acotado necesito tanto código, para un proyecto real cuando más necesitaré?

  - ¿Podré manejar toda esa complejidad?

  - ¿Cumpliré los tiempos de entrega?

  - **¿Y si trabajo en equipo?** ¿seré lo suficiente claro como para que otro integrante de mi equipo pueda seguirme y repartirnos todo el trabajo? ¿o debo hacerlo todo yo porque no entienden lo que hago?

  - **Hacer las cosas complicadas no es una virtud**, hacer simples las cosas complicadas debería ser el verdadero arte y objetivo en nuestra actividad como desarrolladores profesionales.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

1) ***El ejercicio busca llevar al extremo las herramientas que disponemos***

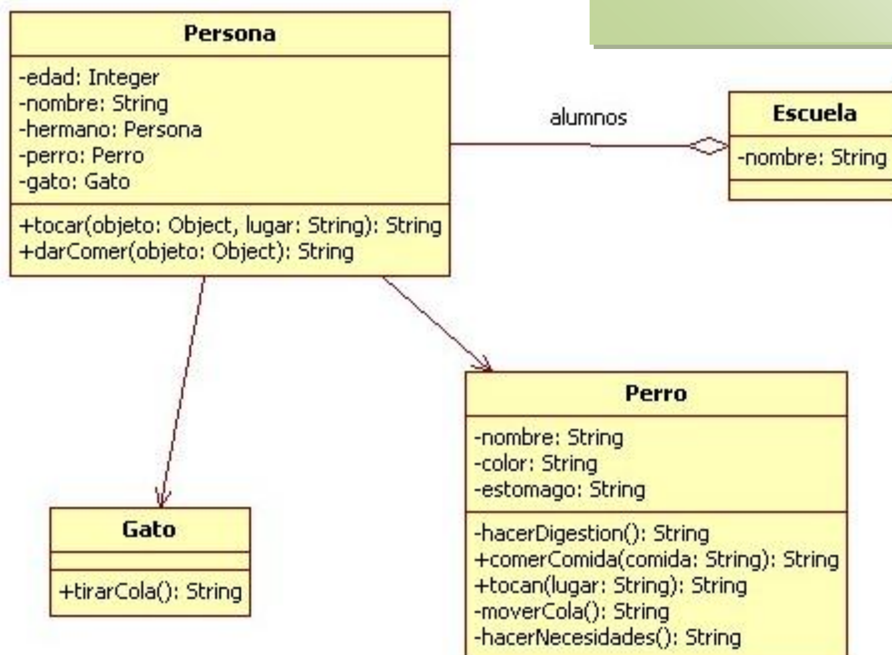
El ejercicio busca doblar de alguna forma las herramientas que disponemos en la actualidad para con la práctica darnos cuenta que nos faltan elementos para representar toda la realidad que tenemos delante. Por consiguiente, cuando veamos las restantes relaciones entenderemos cómo terminar de armar un diagrama más completo.

2) ***Tarea 2 – propuesta de diseño 1***

En este caso se usa la herramienta StarUML y vamos a representar la relación "hermano" con un atributo simple de tipo "Persona".

Tener en cuenta en muchas veces se ven "flecha de agregación" (con el rombo en su inicio) sin la punta -> pero el significado es el mismo.

Diagrama UML (usando StarUML)



Para no perder foco en las relaciones entre los elementos y confundirse con todas las flechas, **se omite intencionalmente en este ejemplo la clase Index (ver ejemplo completo más adelante)**.

¿Cómo se probaría este diseño y verificar que cumpla con la letra?

Auto-relación "Persona"

Si tú eres *"Persona"* y tienes una relación con (otra) Persona, se dice que tienes una **auto-relación**, por lo tanto se podría representar con una flecha que sale de *Persona* y vuelve otra vez a la clase *Persona*. Como poder, se puede hacer y está permitido, pero a veces por claridad y evitar tantas flechas, se deja solo lo necesario de lo que quieres comentar / documentar, por lo tanto **si tienes un atributo de tipo Persona se sobre-entiende que es una auto-relación**, lo mismo, tampoco es necesario en Persona hacer un require_once a Persona, ya estás en la misma clase.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Veamos primero cómo sería el código de index.php sin crear la clase para probar paso a paso toda la letra:

index.php:

```php
<?php
require_once 'Persona.php';
require_once 'Perro.php';
require_once 'Gato.php';
require_once 'Escuela.php';

/* Micaela tiene un perro */
$persona  = new Persona('Micaela', 5);
$perro = new Perro('Tito', 'blanco y negro');

$persona->setPerro($perro);

/* Martina, dueña del mismo perro... */
$persona1 = new Persona('Martina', 3);
$persona1->setPerro($perro);

/* ... y hermana de Micaela */
$persona->setHermano($persona1);

/* Marcos es dueño de un gato */
$persona2 = new Persona('Marcos', 6);
$persona2->setGato(new Gato());

/* Escuela Dos Corazones */

$escuela = new Escuela('Dos Corazones');

/* ... Micaela va junto con 5 niños más ... */

$escuela->addAlumno($persona);
$escuela->addAlumno($persona1);
$escuela->addAlumno($persona2);
$escuela->addAlumno(new Persona('Julio',5));
$escuela->addAlumno(new Persona('Martín',4));
$escuela->addAlumno(new Persona('Carla',4));
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Detalle importante a tener en cuenta con las relaciones

PHP4 era un lenguaje donde todo era "por valor" y había que especificar explícitamente cuando era "por referencia". PHP5 se pone a tono con los lenguajes POO y de ahora en más **todos los objetos son siempre "por referencia"**, por lo que no hay que especificar nada (y en ningún otro lenguaje POO).

¿Qué significa esto?

Si fuera por valor, cuando hacemos:

```
$persona = new Persona();

agregar_hermano($persona);

var_dump($persona);

/* Funciones */

function agregar_hermano($persona){
      $hermano = new Persona();
      $persona->setHermano($hermano);
}
```

El var_dump final no mostraría al hermano, ya que cuando se envió por parámetro, **pasó por valor** y lo que cambió fue solo eso, no el objeto original, y fuera de la función, no ocurrió nada.

Para poder modificar este comportamiento en el contexto de PHP4, habría que hacer cosas como cambiar la función para que retorne el objeto modificado y sustituirlo por el que quedó fuera (así forzamos a que cambie):

```
$persona = agregar_hermano($persona);
```

Todo esto cambia cuando usamos por defecto "pasaje por referencia" (lo que ocurre en PHP5), ya que si pasamos un objeto es la referencia a este y no importa donde estemos, si cambia, estamos cambiando el objeto original (y único).

Se deberá entender entonces que **no es necesario retornar nada cuando modificamos un objeto dentro de una función**.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Por lo tanto, **el orden de la asignaciones entre los objetos no son realmente necesarias** si manejamos correctamente el concepto de las referencias.

Por ejemplo:

```
$persona = new Persona();

$escuela->addAlumno($persona);

$persona->addPerro($perro);
```

¿El perro quedó asociado a la persona que entró a la escuela? ¿Si? ¿no?

¿Si agregamos a la persona antes de asociarla con el perro, no debería primero haber agregado al perro y luego mandar a la persona a la escuela?

Repuesta: no es necesario, la persona que mandamos a la escuela es la misma que tenemos luego, ya que es una referencia. Cualquier referencia que modifiquemos, afecta al mismo objeto del sistema.

¿Y ahora, cómo puedo probar el contenido de cada uno implementando correctamente los toString() de los objetos?

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

El método público toString()

Todas las clases que tengan nombre (Persona, Perro) podrían tener un toString de este tipo:

Perro.php:
```
<?php

class Perro
{
   /* … */

  public function __toString()
  {
      return $this->_nombre;
  }
}
```

Así cuando necesitemos "convertir un Objeto a String" ("imprimirlo"), simplemente haremos:

```
echo $perro;
```

Que es lo mismo decir (y funciona):

```
echo $perro->__toString();
```

toString es un método público, pero que no hace falta ejecutar literalmente, ya que lo hace de forma automáticamente con cualquier operación que obligue/necesite hacer que un objeto deba comportarse como un String ("cadena de texto").

El único caso que podría ser más complejo sería el toString de Escuela, que podría ser solo la impresión del nombre, pero si queremos que muestre todo su contenido (que no es necesario hacerlo en el toString, ya que lo anterior es lo justo).

Ejemplos:

- **¿Si tenemos un objeto Persona, cual sería la mínima información en texto que lo describiría?** Generalmente el nombre y apellido.

- **¿Si tenemos un objeto usuario, cual sería la mínima información en texto que lo describiría?** Generalmente el nombre de usuario.

- **¿Si estamos hablando de una mascota?** Simplemente el nombre.

- **¿Si estamos hablando de una pieza de repuesto de un auto?** La descripción de la misma.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Por ejemplo, para la Escuela podría ser:

```
<?php
class Escuela
{
  private $_nombre;
  private $_alumnos = array();

  public function __construct($nombre)
  {
      $this->_nombre = $nombre;
  }
  public function addAlumno(Persona $persona)
  {
      $this->_alumnos[] = $persona;
  }
  public function __toString()
  {
      $retorno = '';

      foreach ($this->_alumnos as $alumno) {
          $retorno .= $alumno .' ';
          /*
           * Es lo mismo que decir
           *
           * $retorno .= $alumno->__toString() .' ';
           *
           * solo que el objeto sabe cómo convertirse en
            * String, tema que detecta cuando se hace
            * una operación  de suma de cadenas
            * con el punto ".".
           */
           }
           return $retorno;
        }

      }
}
```

Aunque de todas formas se sugiere la simplicidad, **lo correcto es que la Escuela solo retorne en el toString() el nombre de la misma**, y que la información sobre los alumnos sea dada por un método específico para eso.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Lo que dice el manual de toString()

En resumen, toString() (a pesar de lo sintético del manual oficial) es:

"El método reservado "toString()" es la forma que todo lenguaje POO tiene para que nosotros los desarrolladores podamos definir cómo convertir un objeto en "texto plano"."

En Java es muy común solicitar una colección de objetos de tipo Persona y luego tirarle directamente esta información a un "combo" (otro objeto de la interfaz), y este sabrá qué información mostrar a partir que cada objeto tiene claramente definido su toString().

Si tomamos nuestra clase Persona y hago:

```
echo $micaela;
```

¿Qué hace? ¿Cómo puedo imprimir esto? ¿Cómo lo traduzco?

El método toString sirve para eso, **la regla sintáctica** es que ese método solo puede retornar un String, **la regla conceptual** dice que ese String debe representar con muy poca información la esencia del objeto que estoy tratando de imprimir.

Para el caso de Micaela, como es una persona, diría que lo más adecuado sería (solo agrego el método nuevo):

```php
<?php
class Persona
{
   public function __toString()
   {
      return $this->_nombre . '' .$this->_apellido;
   }
}
```

Si fuera un Usuario y no solo una Persona, tal vez agregaría en el toString el id + nombre + apellido.

En resumen: acorta el camino, se estandariza una acción muy recurrente, y evita que tengamos un getNombre() ó getDefinición() o similar que dependa de nosotros agregarlo.

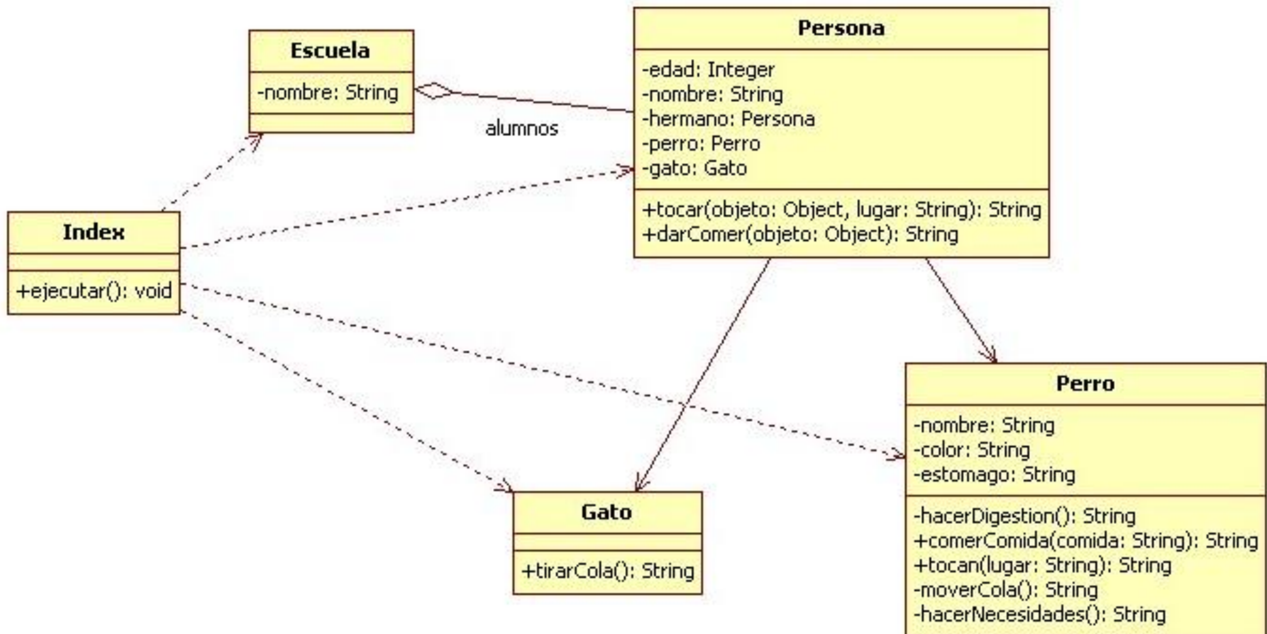
Por todas estas razones todo objeto debería tener definido siempre su toString().

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Agregando las relaciones que se ven desde Index

Ahora bien, una vez que tenemos resuelto nuestro sistema, ¿cómo sería el diagrama UML completo con las relaciones a todas las clases que está usando?

Si hiciéramos "ingeniería inversa", deberíamos traducir en flechas por cada requiere / include de nuestro index.php y el diagrama nos quedaría así:



Index está necesitando conocer a todas las clases para luego relacionarlas entre ellas (hace "new" de las clases y luego las relaciona).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Crear la clase Index a partir de la representación UML

Lo más importante ver aquí, además que todo el código presentado anteriormente en el index iría en un método ejecutar de la clase Index, son cómo todas las flechas se representan de acuerdo al diagrama, y así hay que hacerlo para cada una de las clases del diseño (revisar las flechas y representarlas exactamente con un require_once).

```
<?php
require_once 'Persona.php';
require_once 'Perro.php';
require_once 'Gato.php';
require_once 'Escuela.php';

class Index
{
   public function ejecutar()
   {
      /* Micaela tiene un perro */
      $persona  = new Persona('Micaela', 5);
      $perro = new Perro('Tito', 'blanco y negro');
      $persona->setPerro($perro);

      /* Martina, dueña del mismo perro... */
      $persona1 = new Persona('Martina', 3);
      $persona1->setPerro($perro);

      /* ... y hermana de Micaela */
      $persona->setHermano($persona1);

      /* Marcos es dueño de un gato */
      $persona2 = new Persona('Marcos', 6);
      $persona2->setGato(new Gato());

      /* Escuela Dos Corazones */
      $escuela = new Escuela('Dos Corazones');

      /* ... Micaela va junto con 5 niños más ... */
      $escuela->addAlumno($persona);
      $escuela->addAlumno($persona1);
      $escuela->addAlumno($persona2);
      $escuela->addAlumno(new Persona('Julio',5));
      $escuela->addAlumno(new Persona('Martín',4));
      $escuela->addAlumno(new Persona('Carla',4));

      echo $escuela;
   }
}

Index::ejecutar();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Segunda propuesta de diseño

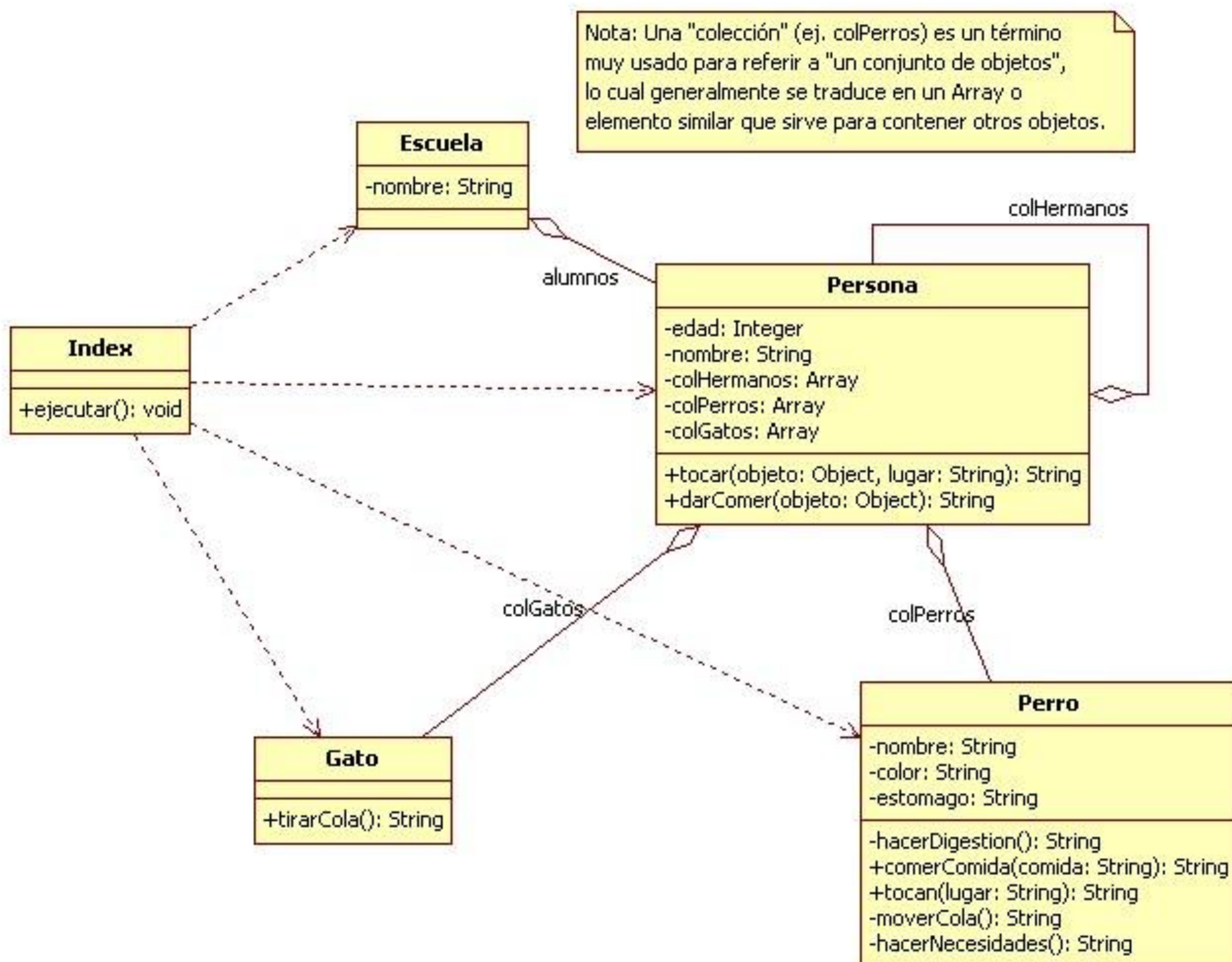
Atención con la "sobre-ingeniería", hay que evaluar si estas mejoras no son demasiadas porque el contexto podría no requerirlo.

Cambios

- **Se cambian las relaciones de asociación por agregación** en el caso que decidamos que pueden existir más de un elemento de Perros, Gatos y Hermanos (varios hermanos, varias mascotas, etc).
- **Se agrega una Nota UML,** que sirve para agregar explicaciones (puede ser hasta pequeñas porciones de código) para entender cómo traducir a código. Generalmente lo que se tiende a explicar es el "index", el punto de arranca para ver cómo hacer funcionar todo el sistema (ya que a veces interpretar el diagrama de clases no es tan obvio).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diagrama UML



Nota

Se omite la traducción de UML a PHP por considerar que su implementación a esta altura es simple y haría extremadamente extensas las explicaciones con código que ocupa decenas de páginas, cuando mostrando las principales clases nos enfocamos en lo importante de la solución.

De todas formas puedes entrar a http://usuarios.surforce.com y obtener los fuentes completos y explicados de todo el libro.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

Siempre se pueden plantear más de un diseño y esto quedará a discusión del equipo de trabajo y ajustándose a un contexto determinado. Aquí se plantean un par de diseños que no necesariamente son "la única solución posible" y se busca extender la comprensión de los temas tratados hasta el momento.

Lo fundamental de este ejercicio es comprender:

- Las distintas **relaciones entre clases y su representación en código**
- **Reafirmar que los objetos pasan siempre por referencia y no se necesita ser retornados** o en determinados casos, ni siquiera un orden específico en sus asignaciones, ya que siempre estamos tratando con una referencia al mismo objeto (no copias).
- **La importancia del "index" como concepto** de punto de partida de resolución del problema general.
- **El uso correcto del toString(), más allá de lo sintáctico**, cual es el concepto que debemos tener en cuenta a la hora de definirlo.

¡Cualquier duda, envía una consulta! **;-)**

Crees que al capítulo le faltó algo?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 12 - Ejercicio "La Escuela y los coches escolares"

En este ejercicio observaremos cómo se van incrementando las relaciones entre las clases y cómo una clase se compone de varias otras clases y que las clases externas no necesariamente tienen que conocerlas.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

*"Tenemos **una escuela** que está ubicada en Ignacio Media 1212 y cuenta con **3 coches escolares** para transportar a **los niños** de su **casa** a la **escuela**. Cada coche puede llevar como **máximo a 5 niños** y tienen un **barrio/zona asignada**.*

*Existe un **conductor** por coche y un **ayudante**, y cada vez que un niño sube a la camioneta se le pregunta su nombre como forma de control. Para seguridad de la escuela y de los padres, se desea **poder saber en todo momento el contenido de niños de todas las camionetas**, el **contenido de niños de una camioneta en particular** y poder **consultar por un niño en particular** y saber **en qué camioneta está**"*

Guía

Empezar a hacer un diagrama UML que represente esta realidad y posteriormente realizar en index.php lo que se solicita en la letra que debe hacer el sistema:

- Saber el contenido de todas las camionetas
- Saber el contenido de una camioneta
- Poder buscar a un niño y saber en qué camioneta está
- Representar como Personas al conductor y al ayudante
- El diagrama UML no debe tener información de tareas anteriores (no va ni perro, gato, etc)

Entender que desde index.php es el lugar *"donde los padres consultarían esta información"* (no representar con clases a los padres), por lo tanto desde ahí se debe preguntar a la escuela y esta devolver la información, nunca a un coche de forma directa.

Nota: se debe usar solo lo visto hasta el momento en los capítulos anteriores.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

Este ejercicio vuelve a poner foco en el tema (fundamental) de las relaciones entre clases y la importancia de entender al "objeto" como "unidad de trabajo" sin desarmarlo de forma innecesaria.

Excepciones: cómo aún no disponemos de la herencia para estos ejercicios se tomarán como válidos diseños con clases específicas más allá de Persona (Niño, Ayudante, Conductor, etc).

"Errores Habituales"

La mayoría de los alumnos al llegar a este nivel los errores fueron pocos, principalmente por descuidos, otras por subestimar los requerimientos (no leerlos con atención) y pensar que los requerimientos solicitados no podían ser implementados (como buscar el niño y saber su coche):

- **Existieron diseños que implementaban una clase Camioneta, pero tenía como atributos camioneta1, camioneta2 y camioneta3**, lo cual no es correcto. Se define una clase Camioneta y luego una instancia para cada una de ellas, a lo sumo, contendrá un atributo "alumnos" para almacenar sus pasajeros.

- **Confusión en el UML al definir el tipo cuando debíamos armar las colecciones de objetos:** en algunos casos usaban *colCoches:Coche*, cuando en realidad el tipo es *Array* (que sí contendrá dentro coches).

- **Dificultad para determinar cuota de responsabilidad de cada uno de los objetos relacionados:** cuando diseñamos una clase, siempre deberíamos pararnos en ella y preguntar *"¿Cuál debería ser su responsabilidad? ¿qué no debería hacer y sí pedirle a otra clase que haga (porque es responsabilidad de otra clase)?"*. Si hacemos un diseño de Escuela con Camionetas que tienen Pasajeros que a su vez cada uno tiene un nombre, la solución directa debería ser:

    - **El "Padre" le pregunta a la Escuela "¿Dónde está mi niño?"** (pregunta por su nombre, ya que quiere saber donde está "la instancia").

    - **La Escuela no lo sabe directamente, tiene que preguntarle a todas las camionetas** y hacer la misma pregunta que le hicieron los padres: "¿Dónde está el niño X?"

    - **La camioneta está en similar situación que la Escuela, solo puede recorrer sus pasajeros** y preguntar uno por uno cómo se llama hasta que alguno coincida. Evidentemente el Pasajero debe poder decir cómo se llama.

    - Luego, **se retorna una referencia al objeto Niño o al objeto Camioneta** (según sea la necesidad y nuestro diseño).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

- **No diferenciar cuando usar setAlgo(objeto) y addAlgo(objeto):** pensar que el primero caso es cuando asocio un objeto con otro y en el segundo caso cuando puedo "agregar" varias asociaciones (de una a muchas asociaciones).

- **Uso de tipo Object y no uno concreto (como ser Persona, Coche, etc):** a pesar que en todo lenguaje 100% Orientado a Objetos "todo objeto hereda de un objeto llamado Object", no se aplica en nuestros problemas no definir el tipo concreto y dejar uno genérico de tipo "Object"

- **Desarmar los objetos, retornar valores y no objetos:** en algunos casos he visto que si se buscaba una persona se retornaba un String que podría ser el nombre del objeto.

  Sugerencia, acostúmbrense a mantener el criterio que **nuestra "unidad" debería ser "un objeto"**, por lo tanto retornemos objetos y no los desarmemos. Si buscamos una persona a partir de un nombre que nos pasaron (String), en vez de retornar un Boolean (true = encontrado), retornar el objeto encontrado o de lo contrario **null**.

- **Parámetros/métodos/atributos no descriptivos o abreviados (UML):** he visto casos en donde se documentaba ***buscarPersona(n:String):Persona***, donde el "n" no se entiende con claridad a qué se refiere. No ahorren caracteres, es mejor dejar claro con el nombre del parámetro a qué nos referimos. No siempre es mejor documentar comentando "de más" un código o un diagrama; si hacemos las cosas detalladas se entenderán directamente sin "anexos". **Una variable, atributo, parámetro, método bien especificado se auto-documenta, no necesita un comentario.**

- **Los fuentes PHP deben evitar el tag de cierre de PHP:** el estándar solicita esto, ya que es casi histórico que PHP requiera apertura y cierre de tag. Se argumenta que también evita muchos errores de caracteres ocultos en la última línea. A menos que lo necesites para embeber PHP/HTML, no cierres con el tag ?>

- **Las variables del lenguaje null, false, true van siempre en minúsculas:** otro error que cometo, ya que se pueden definir tanto en mayúsculas como en minúsculas, y siempre las entendí como "constantes" y estas deben estar en mayúsculas. Esto último no es correcto, deben ir siempre en minúsculas (estándar Zend).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Comentarios Generales"

- **La búsqueda se hacía directamente con el objeto a encontrar:** no necesariamente quiere decir que esté mal, pero quiero recalcar todas las alternativas posibles.

    a. **En la mayoría de los casos se hace desde la escuela un *buscarAlumno($alumno):Coche*,** y justamente el alumno es el que se pasa por parámetro. Se podría hacer una variante más genérica como preguntar por el nombre y retornar la referencia del coche con el niño dentro (una forma de saber en qué coche se encuentra), o en otra variante si simplificamos, una búsqueda por solo el niño (retorna Niño) con solo ingresar su nombre.

    b. **Otra posibilidad crear un "objeto parámetro"**, una instancia vacía de Niño y donde los parámetros que se carguen sirvan para hacer la búsqueda del niño. Por ejemplo, crear un niño con solo el nombre asignado, o si quisiéramos, asignaríamos la edad para que la búsqueda la haga por nombre y edad a la vez.

- **Para el Coche Escolar definir una constante en la clase LIMITE_CAPACIDAD = 5:** esto es una técnica de refactoring que se llama *"eliminar los números mágicos"* y ayudar a documentar sin necesidad de agregar comentarios de más (un buen código es el que no necesita comentarios para explicarse).

```
class CocheEscolar
{
   private $_colAlumnos = array();

   const LIMITE_CAPACIDAD = 5;

   public function estaLleno()
   {
      return count($this->_colAlumnos) >= self::LIMITE_CAPACIDAD;
   }

}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Navegabilidad

Navegabilidad se le dice **al camino que debemos hacer para obtener la información necesaria a partir del diseño de los objetos del problema y sus interacciones**. Ya vimos que lo más natural es pensar ***Escuela -> Coche -> Niño***, pero existen alternativas y ninguna es menos válida que la otra (más si el recorrido entre clases ayuda y no dificulta extraer la información que necesitamos).

Una alternativa: cambiar la *"navegabilidad"*

- La escuela "conoce" a los niños
- Los niños "conocen" a sus camionetas

Por lo pronto este diseño permite fácilmente obtener un niño y preguntarle ¿cual es su camioneta?. Si no hubiera ninguna flecha que se comunique con las camionetas deberíamos pasar por los niños para hacer cualquier cosa que requiera de las camionetas.

¿Cual es mejor?

Depende de la información que estamos buscando que el sistema deba procesar, ya que según el caso una ruta nos será más conveniente que la otra y la razón la debemos decidir nosotros.

Consejo

No debemos tener miedo de optar por una u otra navegabilidad, es completamente normal tener que hacerlo, **no existe una única navegación posible y es fundamental ajustarla para obtener el acceso óptimo a las clases**. No es sano quedarnos con una navegación que nos parece más "natural" pero que nos complique a la hora de extraer o encontrar los objetos que necesitamos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo deberían ser las búsquedas

Debemos mantener el foco en que para un sistema "Orientado a Objetos" la unidad de trabajo son siempre "objetos", por lo tanto para la búsqueda de un Alumno, si lo encontramos, debemos retornar la instancia del objeto, de lo contrario retornamos "null" (como estándar de *"no encontrado"*).

Repasando cómo funciona

Si se agrega desde A la clase B, simplemente es:

$a->agrego(b);

Y si se necesita retornar la clase B desde A, es simplemente

$claseB = $a->retornoB();

Ya que internamente una vez que se encuentra la clase B del array de la clase A, simplemente se devuelve una referencia con un "return" (el objeto sigue siendo único, pero ahora hay dos referencias al mismo objeto).

Si es un coche, deberían preguntar en el método de búsqueda un dato que permita buscar dentro del array, y lo que debemos hacer es preguntar a cada instancia por esa información.

Si estamos hablando de Escuela y Coche, y suponemos que el dato es la matrícula, deberíamos hacer:

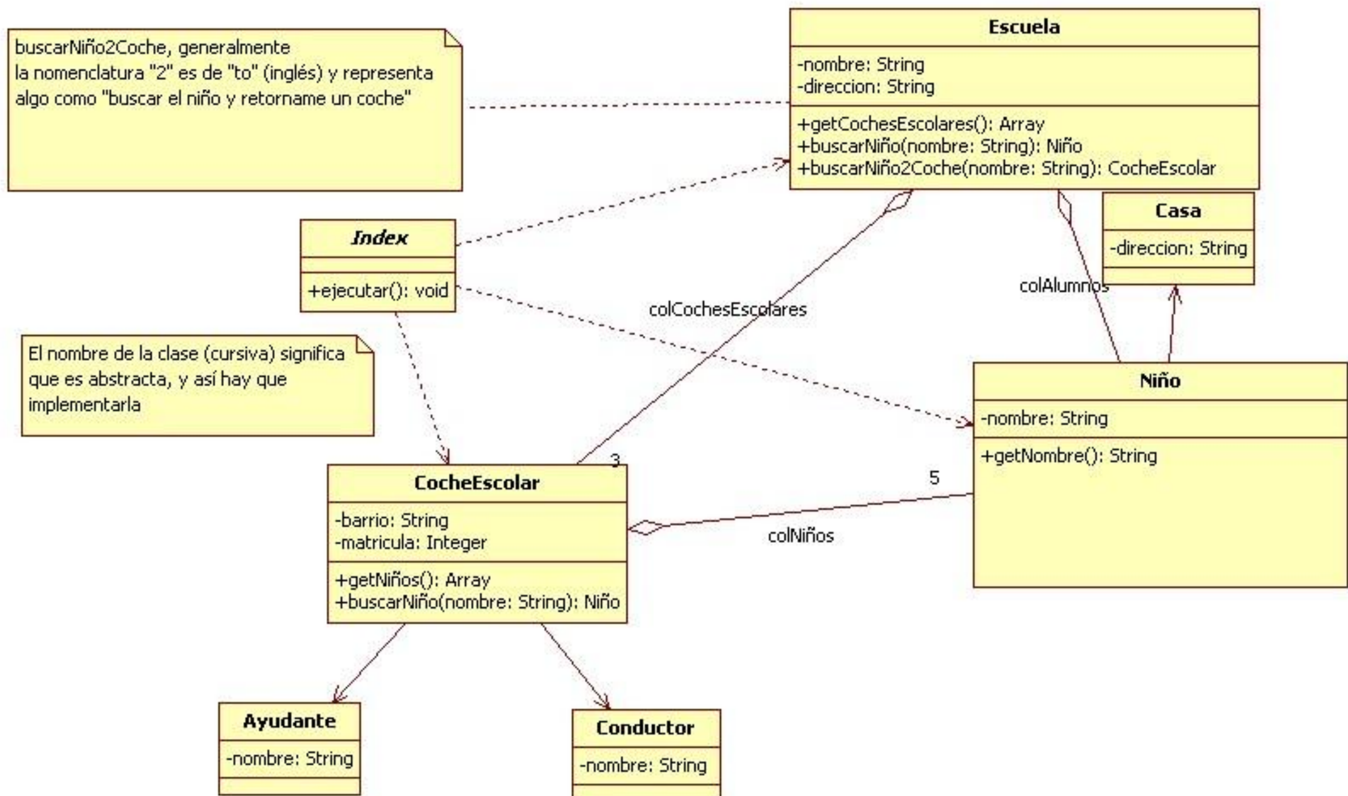
$cocheEncontrado = $escuela->buscarCoche('123456');

y dentro del método hacer un foreach que pregunte por ese dato, si lo encuentra, lo retorna.

```
01. class Escuela
02. {
03.   public function buscarCoche($matricula)
04.   {
05.     foreach ($this->_colCoches() as $coche){
06.       if ( $coche->getMatricula() == $matricula){
07.         return $coche;
08.       }
09.     }
10.   }
11. }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diagrama UML



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

En este ejercicio podemos observar las relaciones entre las clases y cómo se van componiendo unas clases a partir de otras, como también que no es necesario saber o preguntar por un "id" como si fuera una base de datos para obtener un objeto de terminado, no existiendo necesidad ya que todos los objetos de por sí son únicos, aún si son de la misma clase.

Desearías que se ampliara este capítulo?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 13 - Ejercicio "Implementar diagrama de diseño UML"

Daremos vuelta la dinámica de los ejercicios y ahora recibimos un diagrama UML y debemos interpretarlo y codificarlo.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

A partir del diseño UML presentado en la solución del ejercicio anterior, traducir "exactamente" a código PHP.

"No subestimar el ejercicio"

Debemos cuidar al detalle cada traducción de UML a PHP, a esta altura no pueden haber errores.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

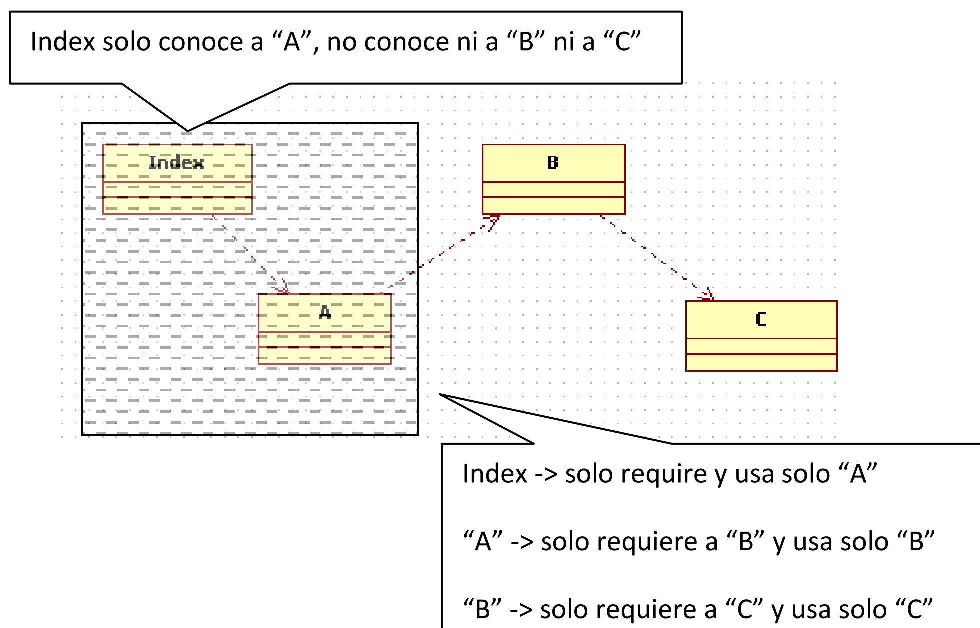
Solución

Aquí vemos que lo que podría ser un ejercicio repetitivo y simple para muchos, gran cantidad de alumnos en los cursos a distancia **falló en representar correctamente las relaciones** (lo más importante) y posteriormente traducir sin errores toda la información vertida en el UML.

Se esperaba

Que no se creara desde index dependencias si no estaban representadas en el diagrama: como estaba planteado el UML, no había relaciones con Ayudante, Conductor y Casa. La idea era tentarlos, pero que finalmente resolvieran que crear **Ayudante y Conductor solo era un problema de CocheEscolar** y que **Casa era un problema de Niño**.

Aunque incluyendo todas las clases desde Index se tenga "técnicamente" acceso a las mismas (porque PHP "suma" todas las clases y las ejecuta como un único fuente) no se ajusta al diseño UML hacer uso de esas clases. Por lo tanto, y siguiendo el diagrama de relaciones**, Index solo debe tener relación directa con las clases que están definidas en las flechas** y así el resto de las relaciones (más adelante veremos en los casos muy específicos que esta situación podría cambiar).



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¿Qué sucedería si se están usando mal las relaciones?

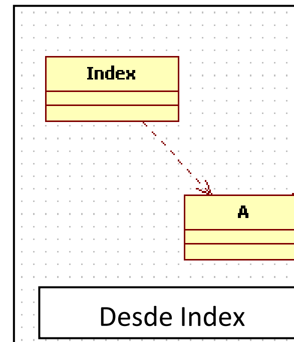
El diseño es frágil y al reusar las clases en otros contextos darán error al no tener el require_once correspondiente para poder encontrar los fuentes necesarios.

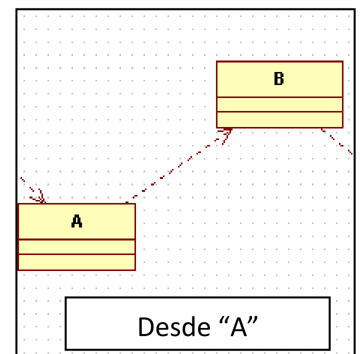
Por ejemplo, si desde Index usamos la clase "B" (que técnicamente su fuente está disponible porque lo incluye "A") y mañana "A" cambia su diseño interno y decide no usar más "B" y usar de ahora en más "C", ¿cómo nos afectaría?

Si las relaciones están mal implementadas, en este caso los cambios internos de "A" afectarán a Index porque conoce directamente a "B" (y lo usa), mientras que si se hubiera respetado el diseño, Index solo conocería "A" y baja enormemente las probabilidades que este cambio le afecte directamente.

Aunque la probabilidad existe, esta es menor al no conocer sus componentes internos ni depender de ellos de forma directa (de todas formas, volvemos al punto inicial, quien armó el diseño UML especificó las relaciones de esta forma con un objetivo específico).

Por lo tanto, para verlo de forma gráfica, tenemos la siguiente visibilidad





Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Comentarios adicionales

- **Se podían hacer variantes para estas clases que no se ven desde Index**, crear todo a partir de información que llega a los constructores o agregar métodos set / get para cargarlos (pasando solo datos, no instancias).

- Cabe acotar que de todas formas, aunque pasemos solo los datos, **teóricamente estamos pasando un "objeto desarmado"** (por decirlo de alguna forma).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Errores habituales

- **Lo más importante y lo medular del ejercicio:** a pesar que hagamos un "require_once" de todas las clases que necesitemos desde el index y todo nos funcione, **no quiere decir que esté bien**. Cada require_once debe respetar las flechas, de lo contrario las clases cuando sean reusadas y no estén ejecutando desde el index, no funcionarán por la falta de sus dependencias/asociaciones. **Eso son las flechas, aunque las inclusiones se repitan (como Escuela -> Niño, Coche ->Niño) cada clase debe saber sí o sí de quién depende y eso se representa con un require_once por clase que deba representarse.**

- Una convención que definimos al principio es que por nuestro estándar web **es requerido contar con un index.php**, y que nosotros lo íbamos a presentar en el UML como una clase, pero que físicamente era el único archivo en minúsculas (para que nuestro navegador lo encuentre).

- **Debemos hacer "require_once" y no "include" o "require" solos:** si se requieren dos veces una clase, con "_once" solo trae en el primer intento, en los otros casos da error por repetir la definición de una clase.

- **"include" no es lo mismo que "require":** el primero da un warning y sigue, el segundo falla y no continúa ("requerir" es más fuerte, nuestra clase no puede continuar si le faltan sus partes).

- **Omitir el Index por index, o usar dos, index.php e Index.php:** tengan en cuenta para el último caso que en Linux/Unix funciones, pero en Win son archivos duplicados y se sustituyen entre sí.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Consejos

- **Para simplificar la creación de múltiples instancias**, muchas veces se omite la variable temporal que luego es asignada a otro objeto (esto se ve mucho en Java, ya que ayuda a disminuir un poco el exceso de verborragia y kilómetros de código):

Ejemplo

```
$niño15 = new Niño('alfredo beltran','Girasoles 7453');

$escuela->addAlumnos($niño15);
```

Simplificación

```
$escuela->addAlumnos( new Niño('alfredo beltran','Girasoles 7453') );
```

- **Aprendan a usar la conversión de tipos:** en vez de hacer esto para convertir a texto:

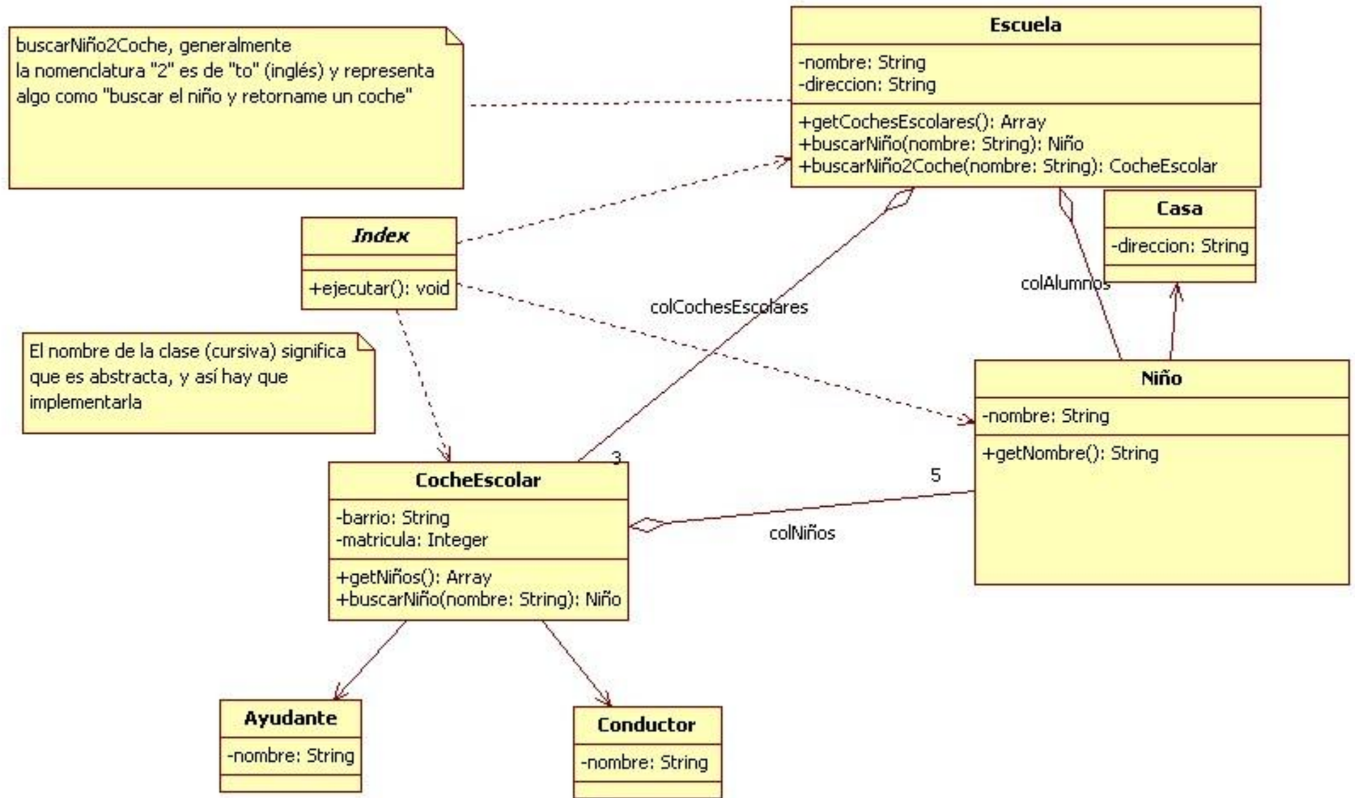
```
return $this->getMatricula()."";
```

cambiar por

```
return (string)$this->getMatricula();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diagrama UML



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Ejemplo codificado

index.php

```
<?php
require_once 'CocheEscolar.php';
require_once 'Nino.php';
require_once 'Escuela.php';

abstract class Index
{
  public function ejecutar()
  {
    $escuela = new Escuela('Dos Corazones','Ignacio Media 1212');

    $coche1 = new CocheEscolar('Barrio1','Matricula1','Conductor1','Ayudante1
');

    $coche1->addNiño(new Niño('Micaela', 'Direccion1'));
    $coche1->addNiño(new Niño('Martina', 'Direccion2'));

    $niñoEncontrado = $coche1->buscarNiño('Micaela');
  }
}

Index::ejecutar();
```

Ejemplo de cómo sería la creación interna de un objeto, sin necesidad que esté creado en el exterior para luego recibirlo por parámetros:

Nino.php:

```
<?php
require_once 'Casa.php';

class Niño
{
  private $_nombre;
  private $_casa;

  public function __construct($nombre, $direccion)
  {
      $this->_nombre = $nombre;
      $this->_casa = new Casa($direccion);
  }
  public function getNombre()
  {
      return $this->_nombre;
  }
}
```

> **Importante**
>
> Prestar especial atención cómo se corresponde la flecha del diagrama UML con el **require_once** de la clase.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Queda en evidencia que no se pueden subestimar los conocimientos ni la letra de ningún requerimiento. A pesar que podría parecer un ejercicio rutinario en los cursos existieron dudas en la mayoría de los trabajos.

Tan importante como hacer los diagramas es aprender a recibirlos e implementarlos. Es muy habitual que si trabajamos en una empresa de gran tamaño sea de todos los días trabajar con diagramas diseñados por un *"Arquitecto de Sistemas"*.

Algún diagrama no se entendió?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 14 - Ejercicio "Sistema para empresa que realiza encuestas"

Un ejercicio más complicado en las relaciones y en la interpretación de los requerimientos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

Una empresa que realiza encuestas necesita implementar un sistema que cumpla con los siguientes requerimientos:

1. La empresa registra primero a **las personas** que van a contestar la encuesta
2. Posteriormente se le asigna **una encuesta** a la persona; existe una encuesta para cada sexo.
3. La encuesta está compuesta por **varias preguntas**
4. Se requiere saber en todo momento **cuales preguntas fueron respondidas**
5. Y si estas fueron **correctas o no**
6. Finalmente, tener un **resumen del resultado** de cada encuesta.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

El foco de este ejercicio sigue siendo la resolución de las relaciones, pero ahora agregando nuevas dificultades (tratando de acercarnos a la realidad): un contexto más confuso (no es resoluble fácilmente y de forma tan directa), muchas posibles soluciones y con errores en los requerimientos.

La pregunta es:

¿Cómo lo resolverán? ¿Mantendrán el foco en la simplicidad? ¿En *"dividir y conquistarás"*? ¿discutirán los requerimientos?

Posibles dificultades

- Cómo armar las relaciones entre la Empresa y sus componentes
- Qué debe verse desde el exterior (que lo representa nuestro Index)
- Cómo registrar una encuesta y las respuestas a las preguntas
- Cómo manejar requerimientos ambiguos y/o erróneos

Requerimientos presentados

"Una empresa que realiza encuestas necesita implementar un sistema que cumpla con los siguientes requerimientos:

1. *La empresa registra primero a las personas que van a contestar la encuesta*
2. *Posteriormente se le asigna una encuesta a la persona; existe una encuesta para cada sexo.*
3. *La encuesta está compuesta por varias preguntas*
4. *Se requiere saber en todo momento cuales preguntas fueron respondidas*
5. ***Y si estas fueron correctas o no***

Finalmente, tener un resumen del resultado de cada encuesta"

Requerimientos Erróneos

Primer dificultad, **el punto 5** no corresponde con el contexto, una encuesta no es un examen, no tiene preguntas correctas o incorrectas, el sistema solo debe registrar las respuestas para poder analizarlas. También se podría pensar que fuera una encuesta "cerrada" donde hay múltiples opciones ya definidas (1-5), pero nuevamente, no tenemos una "correcta".

Simplemente un poco de "ruido" ;-)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Muchos Diseños Posibles

Un posible problema es tratar de hacer un sistema complejo de primera:

> *"Un sistema complejo que funciona resulta invariablemente de la evolución de un sistema simple que funcionaba. Un sistema complejo diseñado desde cero nunca funciona y no puede ser arreglado para que funcione. Tienes que comenzar de nuevo con un sistema simple que funcione."*
>
> *– John Gall*

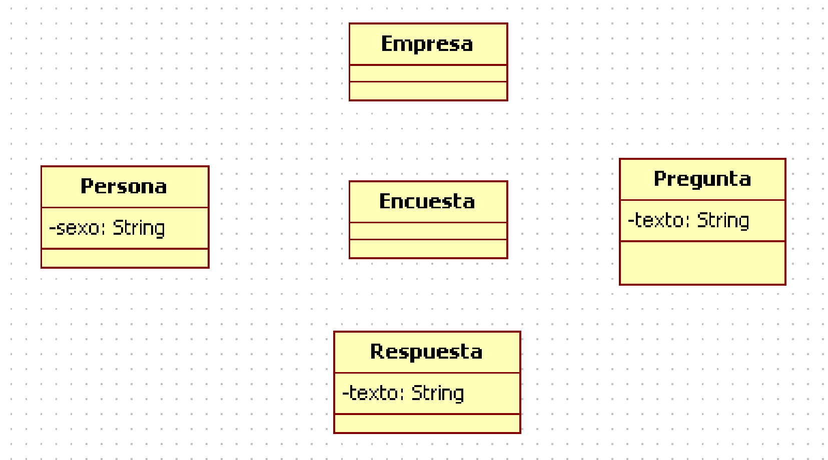
Un enfoque es armar los objetos como elementos independientes (casi olvidándose de su entorno) y luego ver de "jugar" con los objetos haciendo que se comuniquen entre ellos ("el objeto A le pide información al objeto B, etc.").

Por lo tanto, sin importarnos ahora si todo se ve desde Index, el primer planteo podría ser el siguiente:

1. Crear las clases con sus atributos a partir de los requerimientos
2. Relacionar las clases según lo que necesitamos cumplir de acuerdo con los requerimientos.
3. Agregarle los métodos a cada Clase según la responsabilidad que detectamos debería tener cada una y sus relaciones con las demás clases de la solución.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Paso 1 – "Crear las clases y atributos"

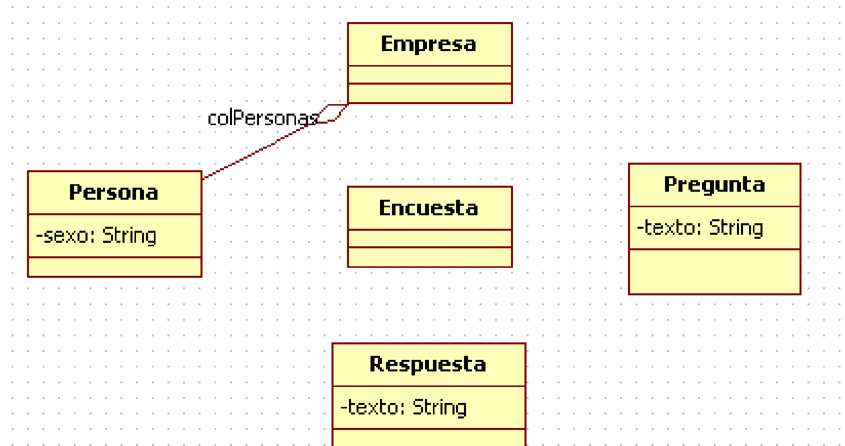


A partir de los requerimientos podemos distinguir todas estas entidades y por lo menos los siguientes atributos:

- *"Persona tiene Sexo"*
- La pregunta tiene –por lo menos- que tener un texto.
- La respuesta tiene –por lo menos- que registrar el texto que responde la Persona.

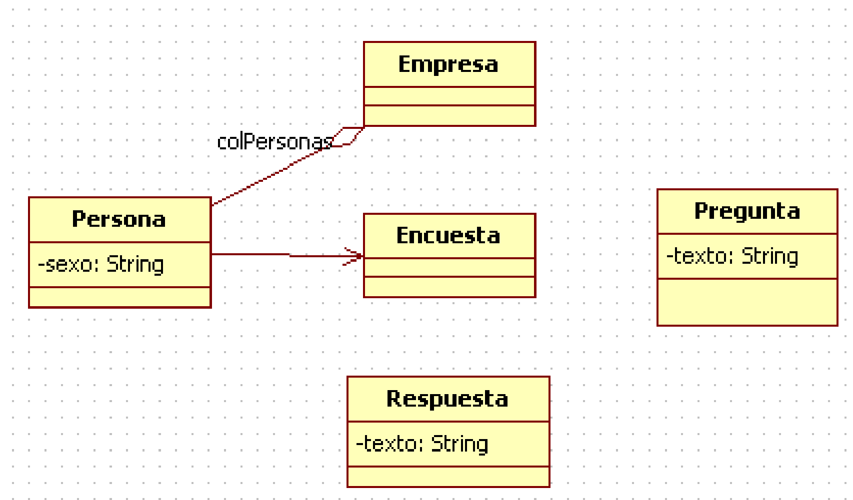
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Paso 2 – "Relacionar las clases"



"La empresa registra primero a las personas que van a contestar la encuesta"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

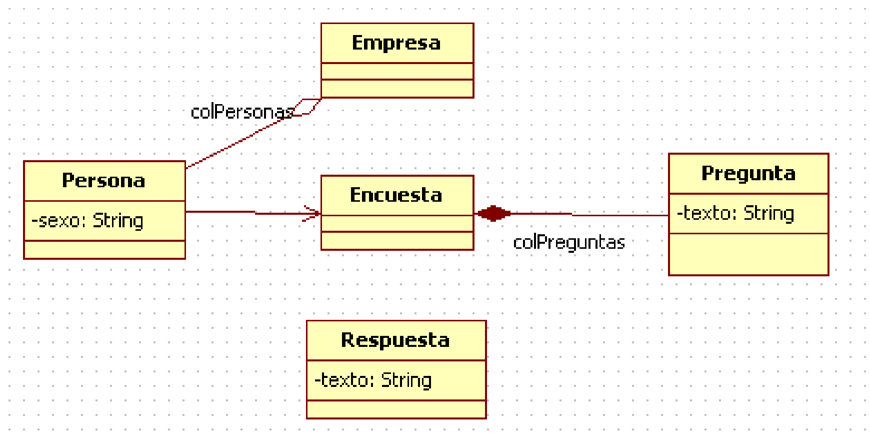


"Posteriormente se le asigna una encuesta a la persona; existe una encuesta para cada sexo"

La asignación es simplemente una relación entre una persona y una encuesta, que exista una para cada sexo en sí no se representaría a través de las relaciones en el diagrama, solo es una regla a tener en cuenta (que hay que documentar) a la hora de crear Encuestas. Si tengo 2 encuestas, preguntaré cual es el sexo de la persona y le asignaré una u otra. **Para este ejercicio no aporta nada implementar esta situación.**

También se asume que según los requerimientos, solo hay una encuesta posible por persona, por lo tanto no manejamos la posibilidad de tener varias encuestas y crear una *"agregación de encuestas"* desde Persona.

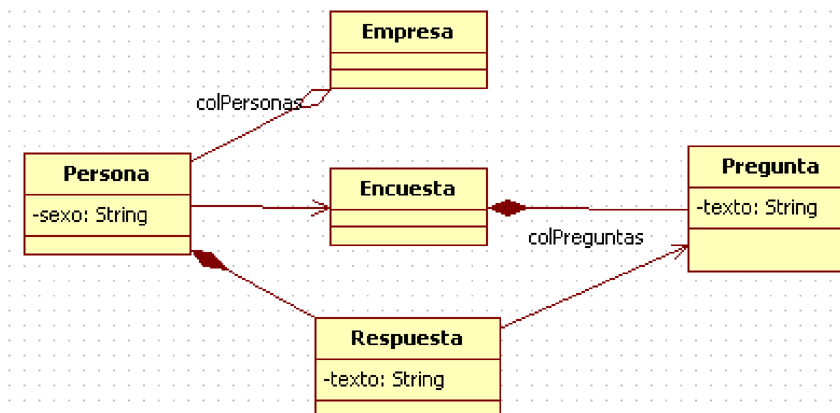
Licencia: http://creativecommons.org/licenses/by-nc/3.0/



"La encuesta está compuesta por varias preguntas"

En esta situación podríamos llegar a documentar que las preguntas no tienen sentido de existir si no están relacionadas con la encuesta. De todas formas, se pueden dar varios contextos, como por ejemplo que la Empresa sea la encargada de crear preguntas y tener un conjunto disponible (*"pool de preguntas"*) para luego seleccionar algunas para una encuesta (con la posibilidad de reusar entre encuestas).

Por ahora no compliquemos el diseño, optemos por una solución más simple (siempre hay tiempo para complicarlo).

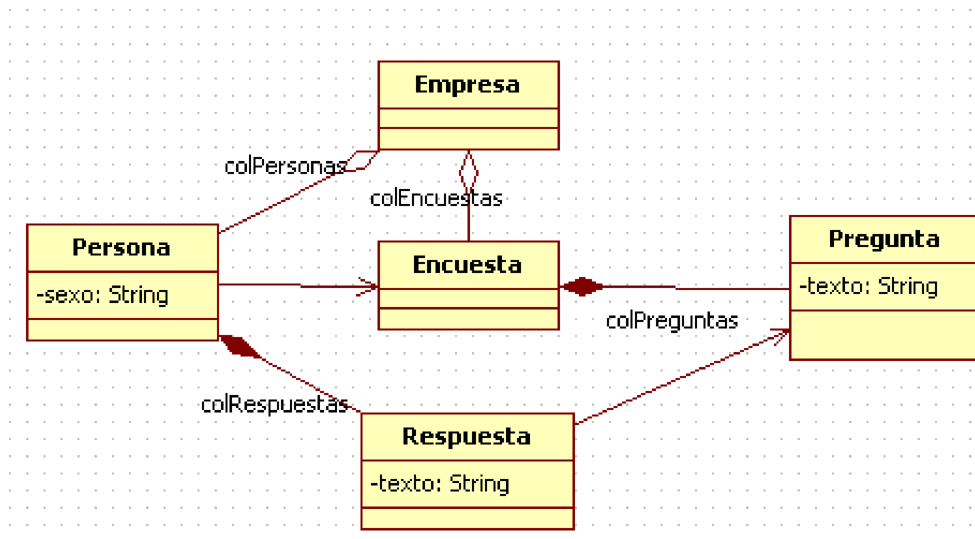


"Se requiere saber en todo momento cuales preguntas fueron respondidas"

Aquí tal vez es lo más complicado de ver y pueden salir varios diseños posibles para registrar las respuestas. Pensemos simple, tenemos una persona, una encuesta y por cada persona tenemos que registrar sus respuestas… ¿y si asociamos por cada respuesta a qué pregunta corresponde? (no tenemos que preocuparnos a qué encuesta corresponden ya que la Persona ya lo sabe de antemano).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

También podemos decir que las respuestas no deberían existir si no existe una asociación con la Persona (composición).



¿Quién debería crear las Encuestas? ¿La Persona o la Empresa? Tiene más sentido que sea responsabilidad de la Empresa conocer a sus encuestas y crearlas.

¿Quién debería crear las Preguntas? ¿La Empresa o la propia Encuesta, serán las preguntas un problema interno y la Empresa solo pasarle la información para crearlas?

¿Index, qué debería conocer?

En primera instancia haremos que conozca a casi todas las clases sin ningún pudor (dije “casi”, no “todas”).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Receta de cocina"

Definir Index: ¿qué necesitamos para armar el sistema?

1. Creo la Empresa
2. Podemos decidir que crear una encuesta y sus preguntas es completamente problema de la Empresa, por lo tanto cuando se cree la empresa ya existirá una encuesta con sus preguntas.
3. Creo a la Persona y le pido una Encuesta a la Empresa y se la asigno a la Persona (para poder ubicar a una encuesta determinada se le agrega un atributo "nombre").
4. Posteriormente solicito las preguntas de la encuesta y las recorro
5. Por cada Pregunta simulo una respuesta hipotética de una persona, por lo tanto registro en la Persona qué pregunta estoy respondiendo y mi respuesta (recordar, la persona ya tiene asociada a qué encuesta corresponde las respuestas y las preguntas, solo tendremos una encuesta por persona).
6. Hacer un resumen de las preguntas que fueron respondidas, para ello hay que preguntarle a la persona qué preguntas respondió. Si fueran muchas personas se le puede pedir a la Empresa que recorra a las personas y le pregunte a cada una lo mismo.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"El código"

index.php

```
<?php
require_once 'Empresa.php';
require_once 'Persona.php';
require_once 'Respuesta.php';

abstract class Index
{
    static function run()
    {
        $empresa = new Empresa();
        $persona = new Persona('masculino');

        /* Defino la Encuesta en base al sexo */
        $sexo       = $persona->getSexo();
        $encuesta = $empresa->getEncuesta($sexo);

        $persona->setEncuesta($encuesta);
        $preguntas = $encuesta->getPreguntas();

        /* Respondo las preguntas de la encuesta */
        foreach($preguntas as $pregunta){
            echo $pregunta . "<br>";
            $persona->addRespuesta(new Respuesta($pregunta, "si"));
        }
        /* Puedo Preguntarle a la Persona sus respuestas */
        echo $persona->getResumenPreguntasRespondidas();

        /* Si la empresa tuviera muchas personas, sería el proceso similar al caso anterior, solo que
         * la empresa recorre todas las personas encuestadas de una encuesta determinada */
    }
}
Index::run();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Empresa.php

```php
<?php
require_once 'Persona.php';
require_once 'Encuesta.php';

class Empresa
{
    private $_colPersonas = array();
    private $_colEncuestas = array();

    public function __construct()
    {
        $encuesta = new Encuesta('masculino');

        $encuesta->addPregunta("Tienes conocimientos de POO?");
        $encuesta->addPregunta("Tienes conocimientos de Base de Datos?");

        $this->_colEncuestas[] = $encuesta;

    }
    public function addPersona(Persona $persona)
    {
        $this->_colPersonas[] = $persona;
    }
    public function addEncuesta(Encuesta $encuesta)
    {
        $this->_colEncuestas[] = $encuesta;
    }
    public function getEncuesta($nombre)
    {
        foreach ($this->_colEncuestas as $encuesta) {
            if($encuesta->getNombre() == $nombre){
                return $encuesta;
            }
        }
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Encuesta.php

```php
<?php
require_once 'Pregunta.php';

class Encuesta
{
    private $_nombre;
    private $_colPreguntas = array();

    public function __construct($nombre)
    {
        $this->_nombre = $nombre;
    }
    public function addPregunta($texto)
    {
        $this->_colPreguntas[] = new Pregunta($texto);
    }
    public function __toString()
    {
        return $this->_nombre;
    }
    public function getNombre()
    {
        return $this->_nombre;
    }
    public function getPreguntas()
    {
        return $this->_colPreguntas;
    }
}
```

Pregunta.php

```php
<?php
class Pregunta
{
    private $_texto;

    public function  __construct($texto)
    {
        $this->_texto = $texto;
    }
    public function __toString()
    {
        return $this->_texto;
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Persona.php

```php
<?php
require_once 'Encuesta.php';
require_once 'Respuesta.php';

class Persona
{
    private $_sexo;
    private $_encuesta;
    private $_colRespuestas = array();

    public function __construct($sexo)
    {
        $this->_sexo = $sexo;
    }
    public function setEncuesta(Encuesta $encuesta)
    {
        $this->_encuesta = $encuesta;
    }
    public function addRespuesta(Respuesta $respuesta)
    {
        $this->_colRespuestas[] = $respuesta;
    }
    public function getSexo()
    {
        return $this->_sexo;
    }
    public function getResumenPreguntasRespondidas()
    {
        $retorno = '';
        foreach($this->_colRespuestas as $respuesta){
            $retorno .= $respuesta->getPregunta() . ': '.$respuesta .'<br>';
        }
        return $retorno;
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Respuesta.php

```php
<?php
require_once 'Pregunta.php';

class Respuesta
{
    private $_texto;
    private $_pregunta;

    public function __construct(Pregunta $pregunta, $texto)
    {
        $this->_pregunta = $pregunta;
        $this->_texto = $texto;
    }
    public function getPregunta()
    {
        return $this->_pregunta;
    }
    public function __toString()
    {
        return $this->_texto;
    }
}

```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Index creado a “prueba y error”

Si se hicieron todas las relaciones y a *“prueba y error”* se creó la clase Index y no se sabes cómo definir correctamente las relaciones, se puede simplemente buscar todos los “new” y estos representarán todas las dependencias de este contexto.

¿Qué sucede con las otras clases que uso pero que no creo de forma directa?

Sí, de alguna forma dependes de ellas, pero de forma “indirecta”, fue la clase responsable de esos componentes que nos entregó sus elementos internos.

Por ejemplo:

1. **$encuesta = $empresa->getEncuesta($sexo);** - La empresa nos entregó un objeto Encuesta, pero en realidad nosotros desde index dependemos de la Empresa. Perfectamente podríamos desarmar el objeto entregado y retornar solo datos, sin conocer los objetos internos (tema sujeto a criterios y discusión).

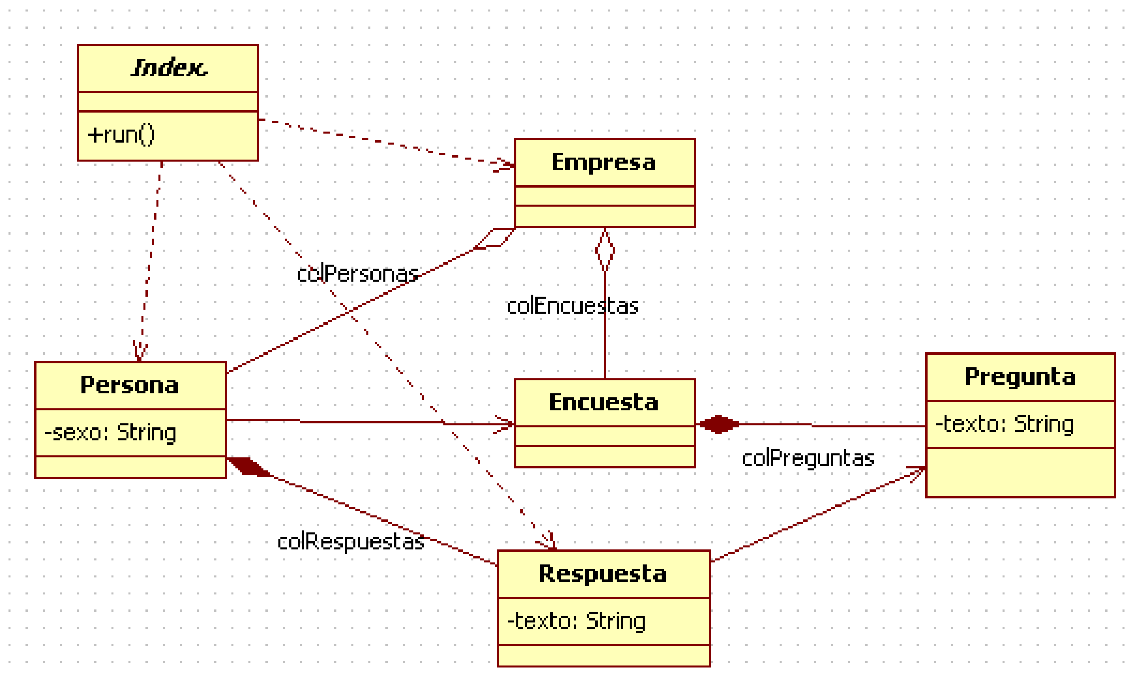
2. **$preguntas = $encuesta->getPreguntas();** - Para empeorarla aún más, la encuesta nos entrega un array con objetos de tipo Pregunta. Nuevamente, la relación es indirecta, la encuesta, para facilitar la operación de procesar preguntas nos entregó varios objetos Pregunta.

Resultado final, Index en este diseño dependería explícitamente de:

require_once 'Empresa.php';
require_once 'Persona.php';
require_once 'Respuesta.php';

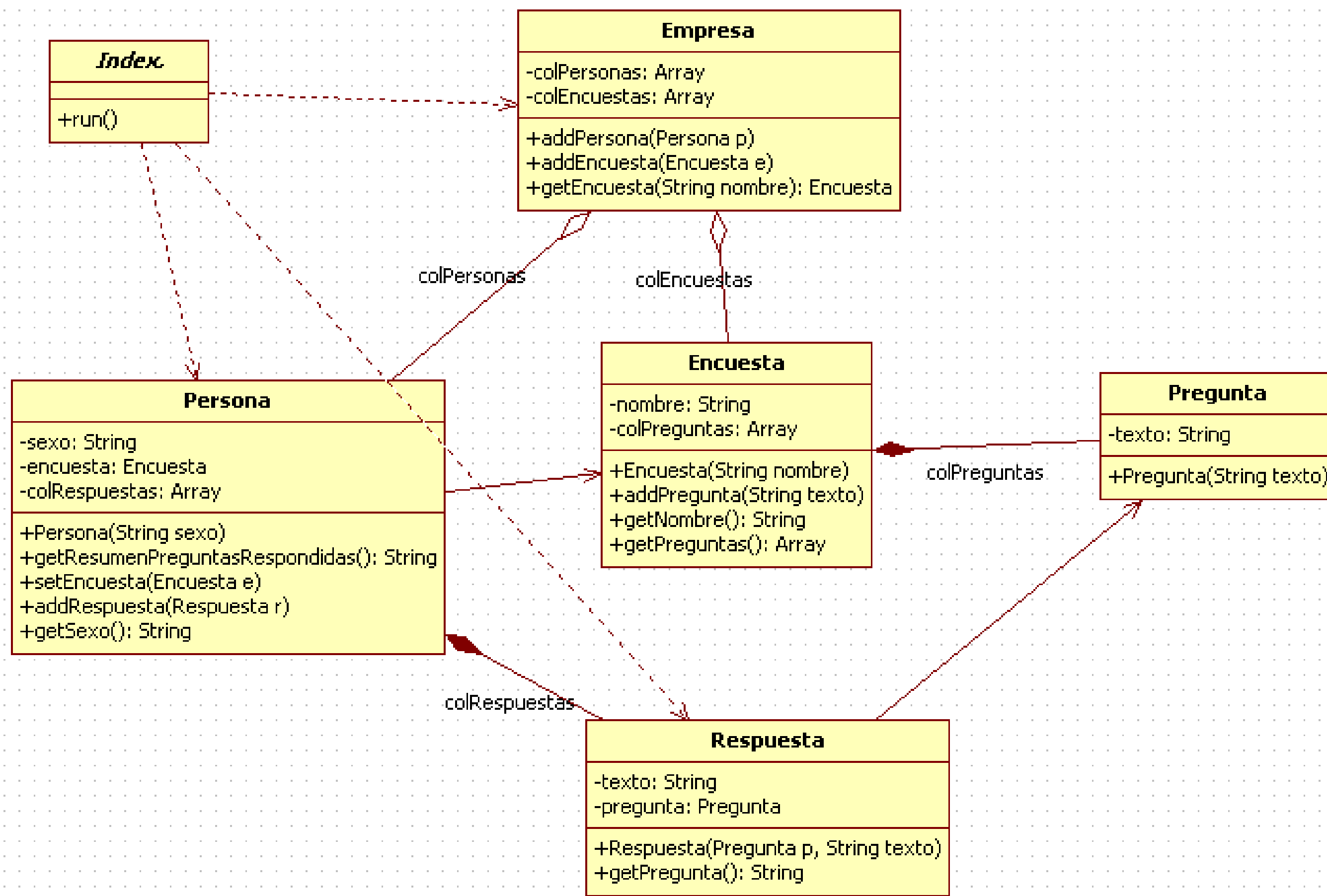
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Finalmente, el diagrama UML de diseño posible para nuestro problema podría verse así (enfocado a las relaciones generales de las clases):



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Como otra opción, un diagrama con absolutamente todos los atributos y métodos, aún si estos se interpretan de las relaciones (de todas formas se omiten los toString por considerarlos triviales en este caso).



Perfectamente se podría armar un diseño que desde Index tenga una flecha de dependencia con todas las clases del diagrama y crear libremente y de forma directa todo y luego relacionar objetos entre ellos.

Para este diseño se trató de evitar (dentro de lo posible) que para usar esta solución las "clases clientes" deban conocer todos los componentes involucrados (*Index* en este caso).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Se deja abierto el tema a criterio y discusión del alumno para determinar en cada caso cual sería la mejor solución partiendo de las siguientes reflexiones:

- **Es mejor que las clases conozcan lo menos posible de los componentes de otras clases a las que usa**.
- **Tampoco es bueno complicar un diseño por intentar esconder absolutamente todas las clases e intentar que exista una única relación contra una sola clase** (por poner un ejemplo), ya que esto dificultaría el trabajo de relacionarlas (es mucho más simple relacionar objetos que tratar datos directamente, de la misma forma, intentar evitar retornar un objeto interno cuando en realidad deberíamos usar siempre "objetos de alto nivel" como unidad).
- A pesar que enviemos los datos por parámetros y no un objeto ya creado, "conceptualmente" esos datos son "el objeto".
- **Cada vez que una clase tiene un "new" de otra clase, indudablemente hay una flecha que sale hacia esa clase** y tiene que haber un correspondiente "require_once" que represente en el código esta relación. "New" y "require_once" van juntos, siempre.

"La perfección es enemigo de lo bueno", no existe un único diseño y no hay solo *"bien" o "mal"*, todas las soluciones dependen del contexto concreto y qué alcance se le busca dar.

Dudas? Espero tu consulta! ;-)

Necesitas el fuente completo de los diagramas UML?

Ingresa a http://usuarios.surforce.com

Capítulo 15 - "Herencia, Interfaces y Polimorfismo"

Luego de haber conocido las relaciones más elementales de la POO (dependencia y asociación) y sus variantes (agregación y composición), empezaremos a conocer dos nuevas relaciones: la herencia y las interfaces.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

La herencia es una de las relaciones donde más errores conceptuales se comenten, principalmente porque es fácil visualizar el *"reuso mecánico de código"*, lo que lleva a ***"heredar por el simple hecho de disponer de código"***.

Las "interfaces" son las "grandes desconocidas" en el mundo PHP lo que hace que generalmente los programadores directamente no las usen, pero lamentablemente se pierden de una gran herramienta que es fundamental para poder implementar buenas soluciones de diseño.

Finalmente un tema que va anidado a la herencia y las interfaces es el **"polimorfismo",** considerado *"el patrón estratégico más importante de la POO".*

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

La Herencia

Definición: *"es una relación de **parentesco** entre dos entidades" (una es padre de la otra, una es hija de la otra)*

Antes de abordar lo "mecánico de la herencia" quiero destacar lo resaltado :

> **Parentesco**. No puede existir herencia si no existe alguna relación "familiar" entre ambas entidades. Uno de los peores errores que se puede cometer -y que demuestra la falta de conocimientos en Objetos- es usar el mecanismo de la herencia con el único objetivo de "reusar código" de otra clase.

A partir de estos errores no hay sistema que pueda ser bien implementado si todos heredan de clases por el simple hecho de querer usar sus funcionalidades.

"La herencia no se debe aplicar de forma mecánica, si no hay una relación literal y coherente de padre-hijo, la herencia no es aplicable"

Ya hemos visto en el documento anterior que **a través de una relación de dependencia** se puede hacer uso de funcionalidades de otras clases y **a través de la asociación** construir objetos más grandes y complejos, sin necesitar siquiera de conocer sus mecanismos internos de funcionamiento (en otras palabras, "sin conocer su código").

Metáfora: como en la genética, **un padre hereda a su hijo determinadas características que podrá usar a su favor o en su contra**. Todo lo que esté en su contra será culpa de nuestro diseño, de cómo habremos hecho al padre y de cuanta información en exceso dejamos que llegue a sus hijos.

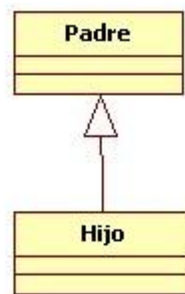
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Representación UML

La representación en UML es **una "flecha continua"** que va desde la clase "hija" hasta la clase "padre", similar a la relación de "asociación", pero con la diferencia que **termina con una punta en forma de triángulo**.

> **Nota:** entender la diferencia de las flechas: si es punteada ("no continua") la relación es más débil que una "flecha continua". Por lo tanto la Herencia es una relación igual de fuerte que la asociación, pero más fuerte que una relación como la "dependencia" ("línea punteada").

Un diagrama genérico sería:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo se traduce a código

Se pueden hacer varias lecturas:

1. Todos los atributos y métodos se heredan del Padre al Hijo, pero…

2. **Los atributos y métodos que son de tipo "público" (public) o "protegido" (protected)** son visibles directamente desde el Hijo (en el caso particular de "público" son visibles de todos lados, incluyendo desde el exterior del objeto).

3. **Los atributos y métodos que son de tipo "privado" (private)** se heredan del padre a su hijo pero no son "visibles" de forma directa, solo se podrá hacer a través de métodos del padre que sean públicos o protegidos.

Funcionalmente hablando podríamos decir que un método público "saludar" del padre que su hijo hereda se ejecuta de la misma forma que si el hijo lo hubiera implementado él mismo, solo que el código se encuentra "guardado/almacenado" en el padre.

Si hacemos un var_dump de un objeto hijo veremos la estructura interna del objeto y todos sus atributos y métodos privados, pero si intentamos usarlos normalmente no podremos visualizarlos, pero siguen estando ahí.

Un ejemplo podría ser que la clase Persona tiene atributos privados nombre y apellido, el constructor del Padre donde carga estos datos es público, de la misma forma que el toString, por consiguiente la clase Hijo podrá construir el objeto, cargarlo con su nombre y apellido y desplegar sus datos cuando haga un echo de Hijo, todo, sin acceder directamente a los atributos de Persona (porque son privados) pero sí pudo hacer uso de ellos a través de métodos públicos o protegidos desde la clase Hijo y que fueron heredados de su padre.

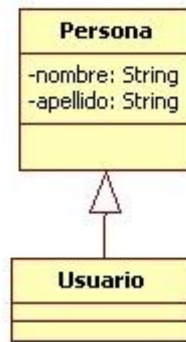
Por consiguiente, veamos algunos ejemplos reales: definiremos una clase muy genérica llamada Persona que usaremos luego para crear distintas clases un poco más concretas y particulares, que se basarán siempre en esa clase. Aquí hay un claro reuso de código, ya que todas aprovecharán lo que ya tiene definido *Persona*, pero también es claro que existe una relación de parentesco con sus hijos (*"no se hereda por el simple hecho de necesitar acceder al código"*).

Más información:

Manual de PHP, sección Visibilidad

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Caso 1 –Usuario hereda de Persona



Y traducido a código será:

```
class Persona
{
    private $_nombre;
    private $_apellido;
}

class Usuario extends Persona
{
}
```

Presta atención, esta herencia no hace nada más que definir que un usuario es, además de tipo "Usuario", de tipo "Persona", pero como no hay atributos o métodos "no privados" (public, protected), **no tiene acceso directo ("no es visible") a lo que es "privado" en su Padre**.

A pesar que es un "mal padre" (chiste), el usuario es ahora de dos tipos, por consiguiente un *"Type Hinting"* (validación de tipos) que filtre personas aceptará al usuario *porque "el Usuario es una Persona"*.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"El Usuario es una Persona"

Si la Persona estuviera completa (con constructor y toString), este código funcionaría sin problemas en la clase Usuario, ya que heredaría de forma automática el constructor y el toString de su Padre, y usaría los atributos privados del Padre que el hijo tiene acceso a través de los métodos públicos o protegidos (según el caso).

```
01. class Impresora
02. {
03.     function imprime(Persona $persona)
04.     {
05.         echo $persona;
06.     }
07. }
08.
09. $impresora = new Impresora();
10.
11. $impresora->imprime(new Persona('Enrique'));
12. $impresora->imprime(new Usuario('Enrique'));
```

Usuario será aceptado sin ningún problema por el *"Type Hinting / Validación de Tipo"* reconoce al usuario como un tipo de Persona, lo cual es válido y permite su ingreso al método "imprime".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Caso 2 –Persona agrega un constructor

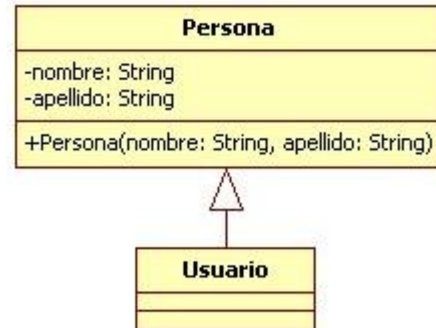
En este caso tendremos un método construir que es público, por lo tanto podemos usarlo desde la clase "hija" de forma automática.

Hay que prestar especial atención que **en PHP el constructor del padre no se ejecuta de forma automática cuando el hijo define su propio constructor**, por lo tanto hay que invocar de forma explícita el constructor del padre.

Repasando: si el hijo **no define su constructor**, igual que sucede con el toString, el destructor, etc, estos serán heredados de su Padre y **sí son ejecutados de forma automática.**

Si el usuario define su propio constructor:

```
class Persona
{
      private $_nombre;
      private $_apellido;

      public function __construct($nombre, $apellido)
      {
         $this->_nombre = $nombre;
         $this->_apellido = $apellido;
      }
}

class Usuario extends Persona
{
      public function __construct($nombre, $apellido)
      {
         parent::__construct($nombre, $apellido);
      }
}

/* Forma de uso */
$usuario = new Usuario('Enrique','Place');
```



Aquí, la clase usuario visualiza el método público de su padre, pero no ve directamente los atributos privados, pero, como ahora "Usuario es una Persona", aunque no podamos "manipular directamente" los atributos, estos están ahí.

> **Nota**
>
> **Para el ejemplo el constructor de Usuario no aporta nada nuevo**, la idea aquí es que agregue posteriormente la inicialización de sus atributos particulares y luego use el constructor del Padre para inicializar los atributos heredados.

Bueno, hagamos algo útil, usemos esos atributos que parecen "invisibles" ;-)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Caso 3 –Persona agrega su toString

Definamos que Usuario es una Persona y que además de tener todos los datos de ella, se define ahora en la clase Padre el método toString

Persona.php:

```
<?php

class Persona
{
   private $_nombre;
   private $_apellido;

   public function __construct($nombre, $apellido)
   {
      $this->_nombre = $nombre;
      $this->_apellido = $apellido;
   }
   public function __toString()
   {
      return $this->_nombre." ".$this->_apellido;
   }
}
```

Usuario.php:

```
<?php

require_once 'Persona.php';

class Usuario extends Persona
{
   public function __construct($nombre, $apellido)
   {
      parent::__construct($nombre, $apellido);
   }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

index.php:

```
<?php
/* Forma de uso en index.php */

/* Solo vamos a requerir la clase que
vamos a usar, no la clase padre, ese es
problema de la clase Usuario (recuerden
las flechas de relaciones, cada clase debe
saber a quién necesita para funcionar) */

require_once 'Usuario.php';

$usuario = new Usuario('Enrique','Place');

echo $usuario;
```

¡Y funciona! ;-)

La explicación es que, a diferencia que el constructor, **el toString se hereda porque es público pero este se ejecuta de forma automática**, sin necesidad de explicitar como en el anterior caso (constructor).

Aquí es donde deberíamos entender que el **principio de ocultación** refuerza los diseños al ocultar y cerrar el acceso a detalles internos, pero no por eso nos impide poder aprovechar las implementaciones realizadas. Como es este caso, no tenemos acceso a los atributos de forma directa y no podemos modificarlos, pero perfectamente podemos asignarles información y usarlos.

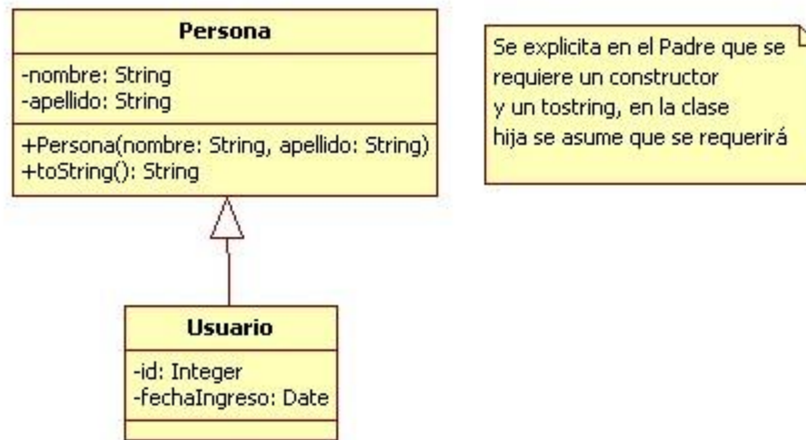
Sigamos modificando este ejemplo.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Caso 4 – Los usuarios necesitan un id y una fecha de ingreso

Imaginen lo siguiente, nuestro sistema requiere que los usuarios tengan un id y además una fecha de ingreso que será definida de forma automática desde el constructor, por lo tanto también ajustaremos la información del toString para que muestre el id de usuario.

El diseño sería el siguiente:



La clase Persona sigue igual, lo que cambia es la clase Usuario y se ajusta la invocación desde Index:

Usuario.php:

```
<?php
require_once 'Persona.php';

class Usuario extends Persona
{
    private $_id;
    private $_fechaIngreso;

    public function __construct($id, $nombre, $apellido)
    {
      parent::__construct($nombre, $apellido);

      $this->_id = $id;
      $this->_fechaIngreso = date('w');
    }
    public function __toString()
    {
        return $this->_id.' '.parent::__toString();
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

index.php:

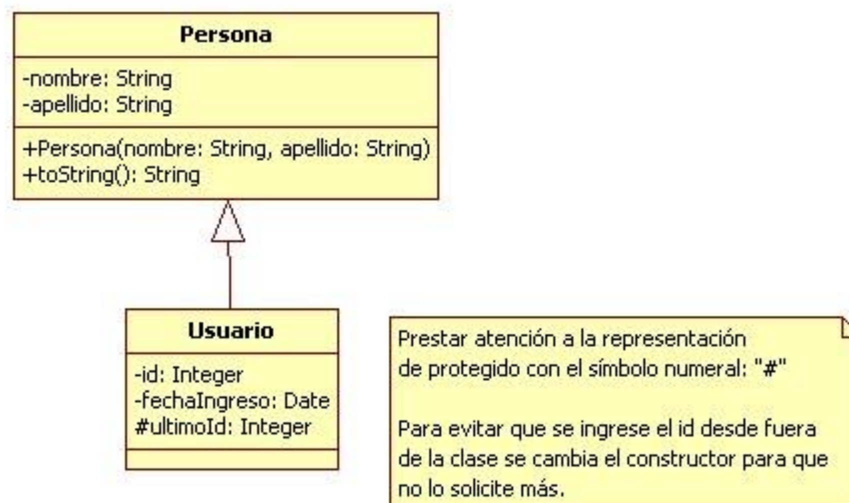
```
<?php
require_once 'Usuario.php';

$usuario = new Usuario(1, 'Enrique','Place');

echo $usuario; // salida: "1 Enrique Place
```

En la clase Usuario armamos el toString juntando la información de una Persona más lo que agrega de específico el Usuario.

Caso 5 – Los usuarios necesitan un id único "autogenerado"

Aprovecharemos la situación para explicar los "atributos estáticos" (static), o lo que podríamos traducir como ***"atributos / métodos de Clase".*** Estos elementos son de acceso común para todas las instancias de la misma clase. Un ejemplo práctico puede ser que la clase sepa autogenerar un id nuevo (que no se repita cada vez que se construye un objeto), para ello, la clase tienen que contar el id actual para asignarle **+1** al próximo usuario.

El diseño sería el siguiente:



La clase Persona sigue igual, lo que cambia es la clase Usuario y se agrega un atributo protegido (lo cual permitirá que una clase herede de Usuario y acceder directamente al atributo como propio) y se ajusta el constructor de la clase para no solicitar más el id.

Atención: para que funcione el autogenerado de id el atributo **ultimoId** debe ser "estático"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Usuario.php:

```
<?php
require_once 'Persona.php';

class Usuario extends Persona
{
    private $_id;
    private $_fechaIngreso;
    static protected $_ultimoId = 0;

    public function __construct($nombre, $apellido)
    {
      parent::__construct($nombre, $apellido);

      self::$_ultimoId += 1;

      $this->_id = self::$_ultimoId;
      $this->_fechaIngreso = date('w');
    }
    public function __toString()
    {
        return $this->_id.'-'.parent::__toString();
    }
}
```

La forma de acceder al atributo estático es usando **self::$atributo**

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

index.php:

```
<?php
require_once 'Usuario.php';

$usuario1 = new Usuario('Enrique','Place');
$usuario2 = new Usuario('Bruce','Lee');
$usuario3 = new Usuario('Linus','Torvalds');

echo $usuario1.'<br>'.$usuario2.'<br>'.$usuario3;
```

Y la salida será

1-Enrique Place

2-Bruce Lee

3-Linus Torvalds

IMPORTANTE

Cabe destacar que esto ocurre dentro del mundo "stateless" *("estado desconectado"),* por consiguiente él o los objetos se pierden luego de terminada la ejecución de la página, iniciando todo nuevamente (si restaura la pantalla volverá a contar desde "1").

Explicación sobre la visibilidad de los atributos y métodos

Existen tres posibilidades: público, privado y protegido.

"Visibilidad Pública"

Se puede considerar como "sin restricciones", lo que es público todos lo pueden acceder o modificar.

Por regla antigua de la POO, en el 99.99% de los casos, 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"

Siguiendo este paralelismo, un padre puede prestar el auto a su hijo de la misma forma que podría prestarlo a otro familiar, pero no a "todo el mundo" sin restricciones.

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.

Recomendación

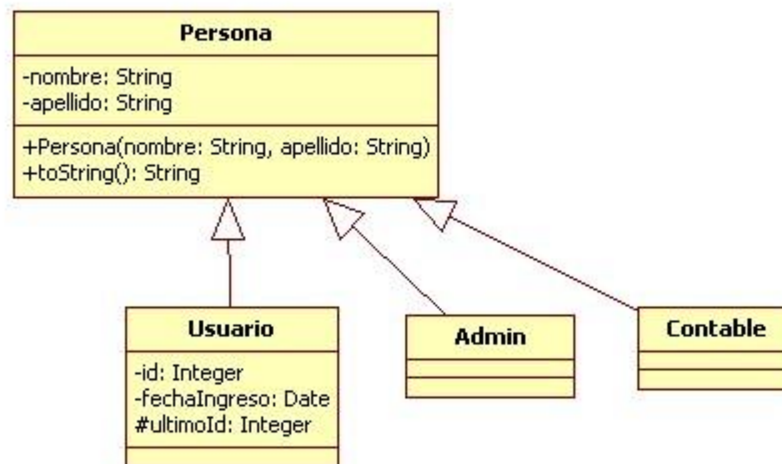
No hacer abuso de la visibilidad protegida, solo en casos excepcionales donde se justifique. **En los casos habituales todos los atributos deben ser privados** y en casos excepcionales a través de algunos métodos getter/setter permitir de forma restringida cargar un valor o consultarlo, de la misma forma crear atributos y métodos protegidos, nunca aplicar estas posibilidades de forma general, mecánica y sin meditar.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Caso 6 – Ejemplos varios

Supongamos que nuestro Usuario genérico siguió creciendo en atributos y métodos. Ahora necesitamos definir un administrador, pero decidimos por diseño que no es *"un tipo de usuario"* ya que las operaciones que tiene implementadas no se ajustan a lo que estamos buscando, es un administrador que tiene la misma información que cualquier otra persona, pero se le agrega la lista de los sistemas que es responsable.

El diseño sería el siguiente:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Clase abstracta

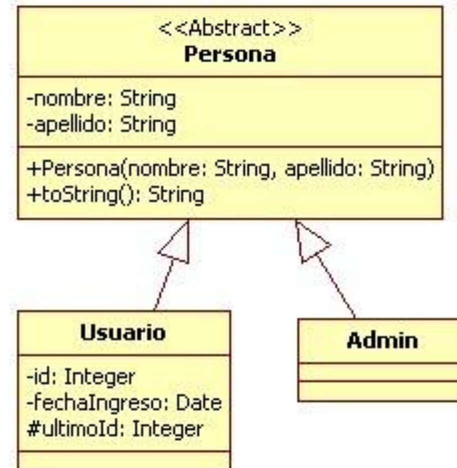
Hay que tener cuidado con la traducción al castellano del manual de PHP donde dice *"abstracción de clases",* cuando en realidad es *"clases abstractas".*

Técnicamente lo que hacemos definir que la clase no puede ser instanciada, por lo que se le antepone a "class" la palabra "abstract" y en UML se representa con el nombre de la clase en cursiva ó agregando un "estereotipo" (texto que va arriba del nombre de la clase entre <<>>).

Conceptualmente lo que estamos diciendo es que la clase no puede ser usada directamente y que nos servirá de molde para crear otras clases que sí serán concretas y candidatas a instanciar.

Por ejemplo, imaginen que tenemos en nuestro equipo un desarrollador novato que se le encomienda hacer unas modificaciones en el sistema y en vez de usar la clase Usuario usa directamente la clase Persona y el sistema empieza a fallar ya que requiere que los objetos sean de tipo Persona, pero las operaciones que requiere solo están implementadas en la clase Usuario y la clase admin.

En UML el diseño más robusto sería así:



En código se traduce en:

```
<?php
abstract class Persona
{
   /*sigue el código de la clase*/
}
```

Lo cual impide hacer

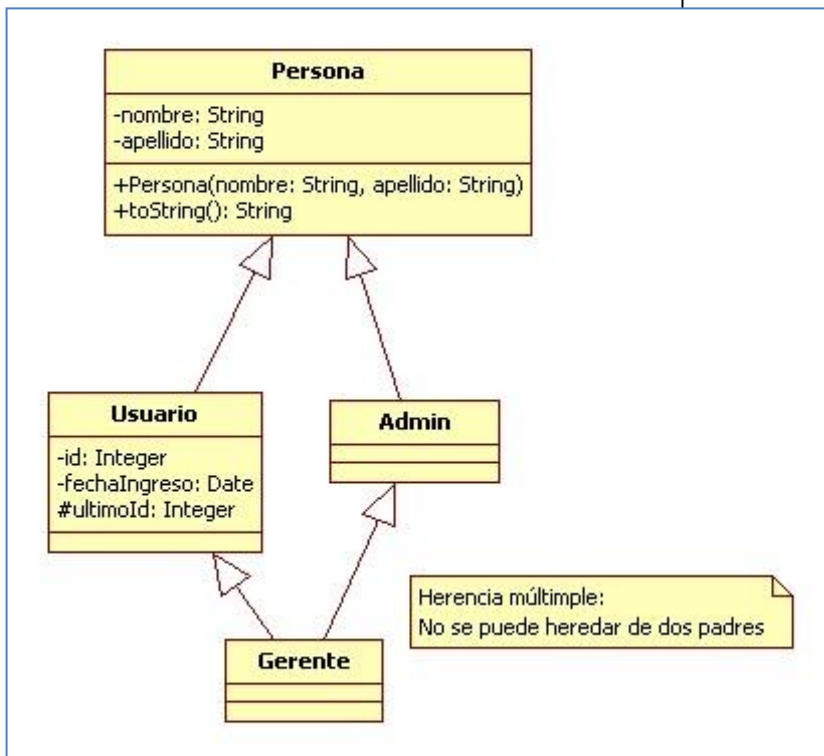
$persona = new Persona();

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Herencia Múltiple

La mayoría de los lenguajes POO **no implementan la herencia múltiple**, no porque les falte, se considera que lo único que aporta es más posibilidades de errores en los diseños.

Lo que no se podría hacer es heredar de dos o más padres (siempre uno solo):



Aclaración

PHP no implementa Herencia Múltiple como la gran mayoría de los lenguajes POO por considerarse inconveniente. Tampoco es un sustituto hacer uso de las interfaces (que veremos en este mismo documento) y argumentar que *"se puede hacer herencia múltiple"*, ya que no es lo mismo y aunque se pudiera, seguiría siendo **no recomendado su uso**.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

“Sobre-escritura” de métodos

Otro elemento del lenguaje es poder sobre escribir un método que heredamos de nuestro padre. Esto se puede hacer simplemente volviendo a definir en la case “hija” un método con el mismo nombre.

```
class Padre
{
   public function metodo()
   {
      /* código */
   }
}
class Hijo extends Padre
{
   public function metodo()
   {
      /* nuevo código */
   }
}
```

Si quisiéramos usar el comportamiento original y solo modificar parte en la clase hija, deberíamos hacer lo mismo que vimos anteriormente en el constructor, invocando desde “parent”.

```
class Hijo extends Padre
{
   public function metodo()
   {
      parent::metodo();
      /* nuevo código */
   }
}
```

Consejo

Nuevamente, si heredamos de una clase para luego sobre escribir el método que recibimos **significará que muy probablemente el diseño está mal (o que necesita ser rediseñado)**, ya que carece de sentido heredar algo para luego modificarlo.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Evitar la herencia y la "sobre-escritura" de métodos

Qué pasaría si necesitáramos endurecer el diseño y evitar que puedan indiscriminadamente modificar nuestros métodos o hasta clases a diestra y siniestra? Deberíamos usar la palabra "final", la que impide la herencia de la clase o la sobre-escritura del método, de acuerdo en donde lo apliquemos.

Impedir heredar la clase Admin

```
final class Admin
{
}
```

Impedir alterar un método que pueda ser heredado

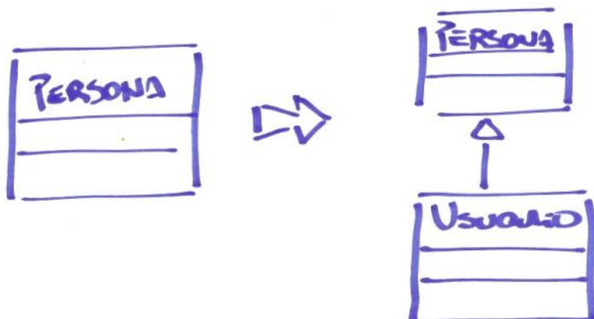
```
class Admin
{
    final function metodo()
    {
        /* código */
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

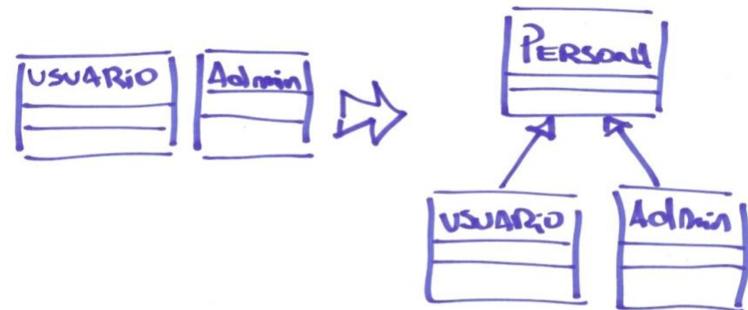
"Generalización" versus "Especialización"

El término es muy usado para definir la "herencia". La explicación más simple es que cada término es usado para referirse a la misma herencia, pero de acuerdo a la lectura que le queramos dar a la relación o al análisis de diseño que estemos haciendo.

Por ejemplo, imaginemos que **solo contamos con la clase Persona y descubrimos que necesitamos una clase Usuario**, podemos decir entonces que a partir de Persona **hicimos una "especialización"** al crear la clase Usuario, ya que tomamos un comportamiento más genérico y lo adecuamos a una clase mucho más concreta y aplicable a entornos más particulares.



Imaginemos lo contrario, contamos **con dos clases que están al mismo nivel, sin relaciones entre ellas, que son Usuario y Admin**, ambas clases repiten muchos atributos y métodos (como nombre y apellido), por lo que decidimos hacer **una "generalización"** y creamos una clase Persona haciendo un factoreo de todo lo común entre ambas clases.



En resumen: podemos decir que son dos formas distintas de referirnos a la relación de "herencia".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Entonces, ¿qué es Polimorfismo?

En sí la palabra solo refiere a *"muchas formas"* y aplicado a la POO es la capacidad de poder tomar varias clases que tiene operaciones iguales y poder invocar a cualquier instancia el mismo método común.

Ventajas: Poder hacer un diseño genérico que se apliquen a varios objetos que tienen operaciones comunes.

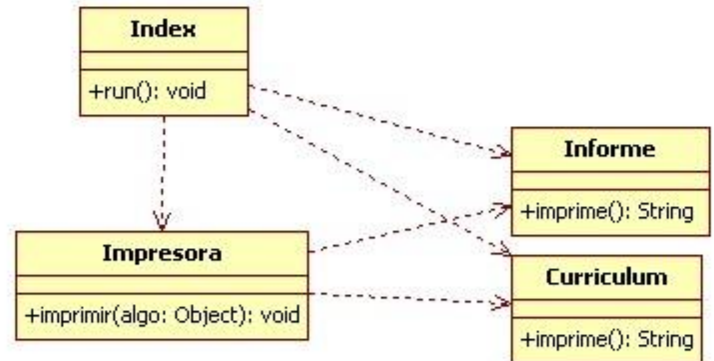
Definamos el siguiente contexto: tenemos una clase Impresora que se encarga de recibir cualquier cosa y la imprime con un "echo"(a futuro podrá ser una impresora).

```php
<?php
class Impresora
{
  public function imprimir($algo)
  {
      echo $algo;
  }
}

$impresora = new Impresora();
$impresora->imprimir('Hola mundo!);
```

Listo, tenemos nuestra clase Impresora. Ahora bien, luego de analizar este diseño **decidimos que deberíamos poder recibir objetos que ellos mismos sepan cómo imprimirse**, y que la impresora se encargue solo de ejecutar la acción y ella provee su "echo" (que mañana podrá ser una operación a bajo nivel que la conecte con una impresora real).

Entonces, luego de discutir con nuestro equipo de desarrollo **convenimos que todos los objetos que quieran imprimirse deberán tener un método "imprime"** y que cada uno deberá saber qué información debe entregar a la impresora.

Este es el diseño



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

La implementación

index.php

```
<?php
require_once 'Impresora.php';
require_once 'Informe.php';
require_once 'Curriculum.php';

Impresora::imprimir(new Informe());
Impresora::imprimir(new Curriculum()
);
```

Impresora.php

```
<?php
abstract class Impresora
{
  public function imprimir($algo)
  {
      echo $algo->imprime();
  }
}
```

Informe.php

```
<?php
class Informe
{
  public function imprime()
  {
      return 'imprimo informe';
  }
}
```

Bien, sin darnos cuenta y casi sin necesidad de entenderlo, **estamos haciendo uso del polimorfismo**. Nuestra clase de impresión recibe distintos tipos de objetos que comparten el mismo nombre común de un método que se requiere para que la impresora pueda funcionar. En distintas etapas de nuestro sistema veremos distintas instancias haciendo uso del mismo código de impresión: “polimorfismo”, “muchas formas”.

Comentario

Este caso en particular lo llamo ***“polimorfismo de palabra”*** *o* ***“polimorfismo débil”*** ya que nada me asegura que se respeten las reglas del diseño, es más, las reglas no están explícitas por código.

Veamos a continuación cómo reforzar el diseño.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Diseño más robusto"

Bien, el ejemplo anterior funciona, pero **el diseño es débil**, ya que se puede decir que entre los desarrolladores hay un "compromiso de palabra" (muy probablemente no documentado) que toda clase que de ahora en más quiera imprimirse debe implementar este método, o de lo contrario fallará.

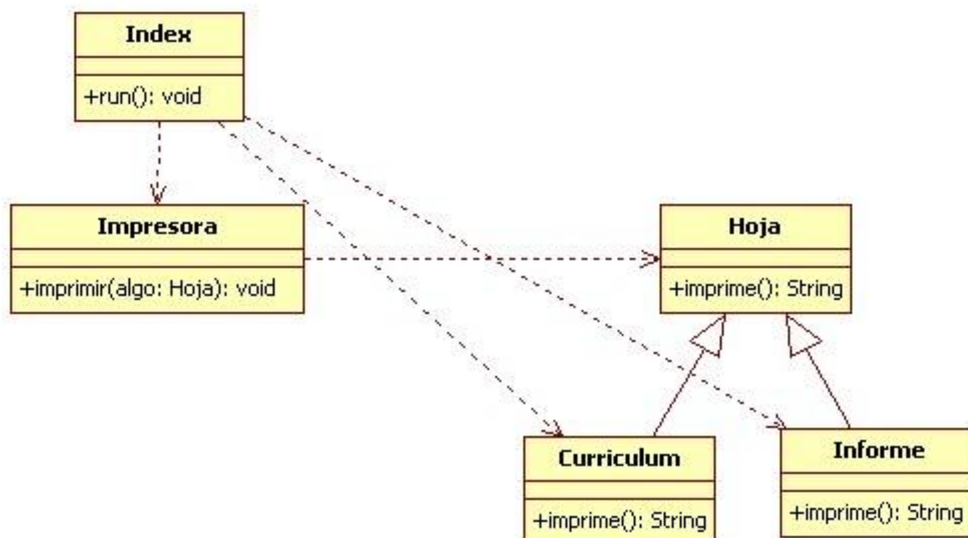
Esto es "débil" ya que **no está garantizando nada y dejando supeditado al desarrollador cumplir con lo que se requiere** (que muy probablemente en caso de olvidar van a tener que recurrir al fuente para entender cómo funciona la clase).

Entonces, mejoremos el diseño.

En este caso hicimos las siguientes modificaciones:

- **Aplicamos una "generalización"** buscando lo común de ambas clases y creando una clase "Hoja" que tendría toda la información base para cualquier documento que quiera imprimirse. Por defecto la hoja sabe imprimirse, y en caso de necesitar modificar el comportamiento, sobre-escribiremos el método imprime en la clase concreta (Currículum o Informe).
- **Se agrega el tipo "Hoja"** en el método imprimir de la clase Impresora, por consiguiente no permitirá a ningún objeto que no sea de tipo "Hoja" pasar por allí.

Bien, esto es el tradicional ***"polimorfismo por herencia".***



Ahora bien, el diseño se nos complicaría si queremos poder imprimir cosas que no necesariamente son Hojas, como un Libro, una revista, una etiqueta, etc.

¿Qué hacemos? ¿Creamos tantas impresoras como temas distintos enfrentemos?

¿La herencia está limitada?

Lamentablemente aquí es donde más se queda corto el manual oficial de PHP al explicar tan brevemente qué es una interfaz (pero bueno, es un manual técnico y no una guía de programación).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Siguiendo el problema anterior, ahora tenemos varios tipos de elementos que no necesariamente son simples "hojas".

La pregunta es:

¿A nuestra impresora le debe interesar cómo son los elementos y cómo tienen que imprimirse?

¿o es un problema de cada objeto saber cómo hacerlo?

Hasta ahora veníamos bien, pero si entendimos todo lo expuesto anteriormente, no podemos modificar la herencia para que un ***"Libro sea hijo de Hoja"*** **(incorrecto)**, o que la ***"Impresora ahora controle libros y que todo herede de Libro"*** **(incorrecto)**, o todas las variantes que se nos ocurran… el problema es que la herencia no cubre estos casos, la herencia solo se aplica cuando hay parentesco, si no lo hay, no sirve…

… a menos que usemos las interfaces!

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Las interfaces: "el poder desconocido"

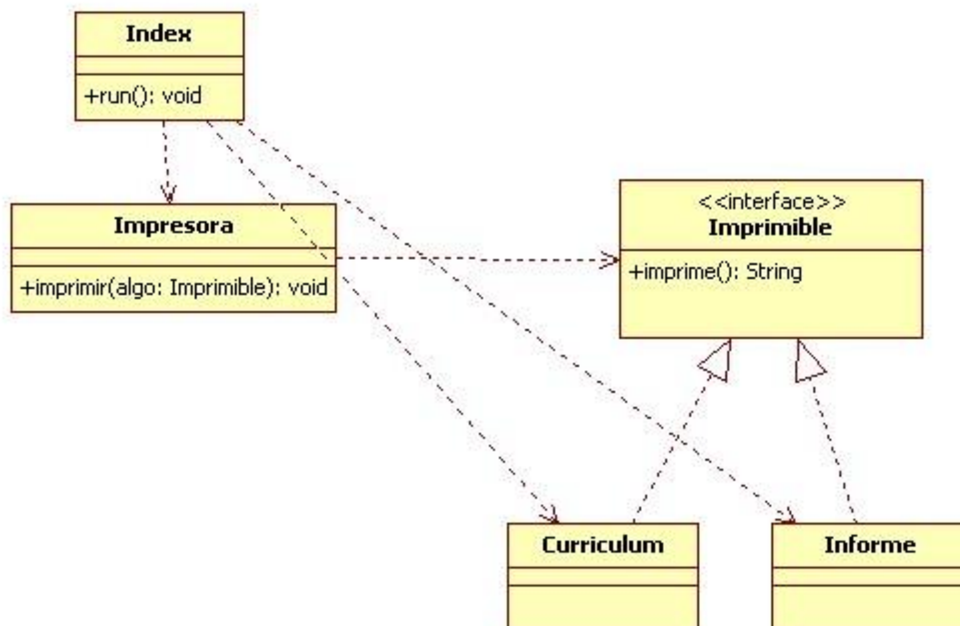
(bueno, por los programadores PHP ;-))

Las interfaces son similares a la herencia: la herencia "agrupa" clases que "son lo mismo" (relación de parentesco) y la interface "agrupa" clases que "que realizan las mismas operaciones" (pero no necesariamente son iguales).

¿Como solucionaríamos nuestro problema?

Lo que une a las clases es que todas quieren poder ser aceptadas por la Impresora y por lo tanto todas deben tener su método "imprime"

El diseño de la solución sería así:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Implementación

ImprimibleInterface.php

```
01. interface Imprimible
02. {
03.     public function imprime();
04. }
```

Curriculum.php

```
01. require_once 'ImprimibleInterface.php';
02.
03. class Curriculum implements Imprimible
04. {
05.     /* Inmediatamente de hacer el implements,
06.     si intentamos ejecutar el sistema, nos dirá
07.     que nos falta el método imprime, por lo tanto
08.     lo agregamos a continuación */
09.
10.     public function imprime()
11.     {
12.         return 'El currículum se imprime distinto que una hora común';
13.     }
14. }
```

Impresora.php

```
01. require_once 'ImprimibleInterface.php';
02.
03. class Impresora
04. {
05.     public function imprimir(Imprimible $algo)
06.     {
07.         echo $algo->imprime();
08.     }
09. }
```

index.php

```
01. require_once 'Impresora.php';
02. require_once 'Curriculum.php';
03. require_once 'Informe.php';
04.
05. abstract class Index
06. {
07.     public function run()
08.     {
09.         $impresora = new Impresora();
10.
11.         $impresora->imprimir(new Curriculum());
12.         $impresora->imprimir(new Informe());
13.     }
14. }
15. Index::run();
```

Explicación

Si en index.php intentamos imprimir una clase que no cumpla con la interfaz, dará error la **ejecución del imprimir de Impresora**, ya que solo puede aceptar objetos que implementen la interfaz Imprimible.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo funciona

1. Inmediatamente que agregamos el tipo *Imprimible* en el método de la Impresora, cualquier objeto que quiera pasar por ahí deberá implementar la interface *Imprimible*.
2. Una vez que lo implemente, el compilador de PHP le dirá que su clase no tienen el método "imprime" (esto lo obliga la interface, usted no la está cumpliendo), por lo tanto para que pueda ser aceptada tiene que contar con ese método.

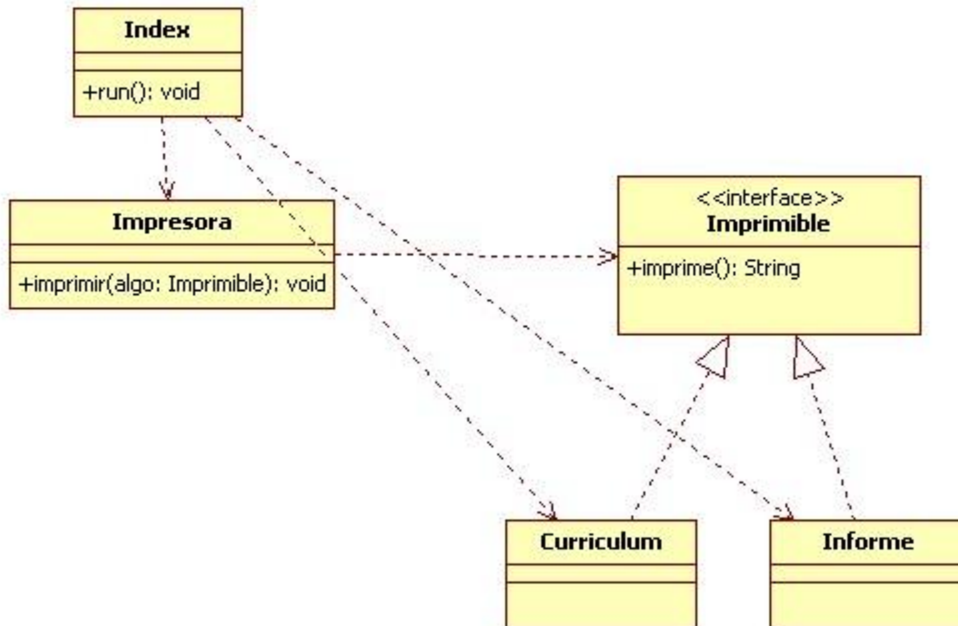
Detalles importantes

- **La flecha es similar a herencia**, solo que es "punteada" ("discontinua"), en vez de decir "hereda" debemos decir "realiza/implementa/cumple" con la interfaz.
- **Todos los métodos de la interfaz son "firmas",** es decir, solo va el nombre y la lista de parámetros, no existe implementación, ni siquiera van las llaves {}.

Sobre los diagramas: en el anterior podemos ver que la representación es similar a una clase, solo que no tiene la división donde deberían ir los atributos, solo el nombre y los métodos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Repaso de lo visto hasta el momento



Detalles a tener en cuenta: en este diagrama se ve la interface como si fuera una clase sin la zona de atributos (por que no tiene) y para reforzar la documentación (no haría falta igual, se sobre entiende) le agrego un comentario como estereotipo “interface”.

Dependiendo de la herramienta, el libro, el autor, el docente, **la interfaz se puede representar también como un círculo**, lo cual en lo personal prefiero (si la herramienta me deja o encuentro como) ya que a simple vista se detectan fácilmente todas las interfaces.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Las interfaces son "contratos de implementación"

Una interface es conceptualmente lo que se dice un *"contrato de implementación"*, ya que para usar un servicio (Impresora) tiene que cumplir un contrato (interfaz) que lo obliga a cumplir una serie de requisitos (las firmas de métodos que aparecen en la interfaz).

De esta forma obtenemos un diseño robusto, ya que nada queda supeditado a la palabra y no depende del conocimiento o desconocimiento del código, ya que al solo hacer un "implements" el mismo compilador nos va guiando con los métodos que nos faltan cumplir

Anexo

Se recomienda leer:

- Herencia de clases y el "Principio de Liskov" (actualizado 15/10/2007)
- "Herencia múltiple en PHP5"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Hicimos la primer introducción a la Herencia y las Interfaces, las diferencias entre ambas, y cómo se aplica el polimorfismo y los distintos tipos que hay. También destacamos que conceptualmente el Polimorfismo es un "patrón estratégico" y que las interfaces son "contratos de implementación".

¿Dudas? ¡envíame tu consulta! ;-)

Necesitas más ejemplos para entender este capítulo?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 16 - Ejercicio "Clase de Persistencia"

Ya que estamos en un capítulo bastante avanzado en los conceptos, en esta oportunidad te voy a soltar un poco más de la mano para que puedas probarte hasta donde puedes llegar (trata de hacer el ejercicio sin mirar la solución) 😉

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

Se necesita diseñar e implementar una **clase de persistencia genérica llamada *BaseDeDatos*** que pueda recibir distintos tipos de **"Manejadores de Base de Datos"** como ser MySQL, Postgres, etc, desde su constructor. Para ello deberán hacer uso de las interfaces, creando una que se llame *ManejadorBaseDeDatosInterface* y que tenga en su "contrato de implementación" las operaciones más básicas y elementales: conectar, desconectar, traer datos ejecutando una consulta SQL, etc.

Posteriormente y para dividir responsabilidades, necesito que creen una **clase SQL** que se encargue de procesar todos los pedidos SQL en partes, así poder a futuro hacer mayores controles.

Por ejemplo, debería tener métodos como:

- addCampo
- addTabla
- addWhere
- addOrder
- etc

Esta primera versión debe contemplar las operaciones de un SELECT y las condiciones serán por defecto "AND".

Con esta clase lo que haremos es evitar que la clase de persistencia tenga el trabajo de hacer estos controles y se lo derivaremos a una clase especializada para esa tarea.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo deberá ser el contenido de la clase Index en index.php:

```
require_once 'BaseDeDatos.php';
require_once 'MySQL.php';
require_once 'Sql.php';

$bd = new BaseDeDatos(new MySQL());

$sql = new Sql();

$sql->addTable("usuarios");
$sql->addWhere("id = 1");
$sql->addWhere("id = 1");
$sql->addWhere("nombre = 'enrique' ");

$usuario = $bd->ejecutar($sql); // esto genera SELECT * FROM usuarios WHERE id = 1 AND
nombre = 'enrique';
```

El resultado es un array asociativo que se obtiene de la consulta a la base de datos.

Deberán realizar el UML y el código de toda la implementación de Index funcionando (no más código que este) y convertido en una clase (el código anterior en un método "run").

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

El objetivo de este ejercicio fue obligar a enfrentar una situación muy cotidiana, como puede ser las clases de *"persistencia de datos"* (estas preguntas se repiten una y otra vez en los foros).

Definición

Con ***"Persistencia"*** nos referimos tanto a guardar como recuperar datos de un medio de almacenamiento permanente, por defecto hablamos de una base de datos, pero perfectamente podría ser un archivo de texto plano u otro medio.

Hay un frase que dice:

"Cualquier problema en computación puede resolverse añadiendo otra capa de abstracción"

La forma de separar el problema de tener distintos motores de base de datos es, justamente, agregando una nueva capa de abstracción y evitar conocer directamente los detalles concretos de una base de datos en particular.

Veamos la evolución, paso a paso…

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Primer escenario: "crear una conexión a la base de datos"

Lo primero que implementamos si necesitamos persistir nuestros datos en una base de datos son las funciones de conexión de un motor de base de datos como MySQL. Entonces, buscamos cada una de las sentencias en el manual y en el peor de los casos nuestro código se plaga de sentencias tan específicas como: mysql_connect, mysql_query, etc.

Lo que podría suceder luego es que **si quisiéramos cambiar de motor de base de datos, tendríamos que modificar todas las sentencias de nuestro sistema**, con el costo que esto tiene.

Segundo escenario: "crear una clase de persistencia"

Tratando de prever esta situación podríamos crear una **"nueva capa de abstracción"** y generar un clase genéricas (conectar, traer datos, desconectar) que nos permita "esconder" el código específico de MySQL. En caso de necesitar alguna funcionalidad extra o cambiar de motor, simplemente el impacto en nuestro sistema estaría reducido a modificar nuestras funcionalidades de una clase que oculta los detalles de implementación.

De todas formas, aún seguiríamos dependiendo de tener una implementación para una u otra base de forma exclusiva.

Tercer escenario: "abstraer el tipo de base de datos"

Imaginen que nuestro producto crece y lo empezamos a comercializar para muchas plataformas, y que uno de nuestros nuevos requerimientos es poder contemplar varios motores de bases de datos. Para ello tenemos que **"contener el foco de cambio"(*)** y adaptar a nuestro sistema para que solo agregando *"código nuevo sin modificar lo existente"* pueda así soportar más motores de bases como Oracle, MSSQL Server, Informix, Sybase, Firebird, etc, etc.

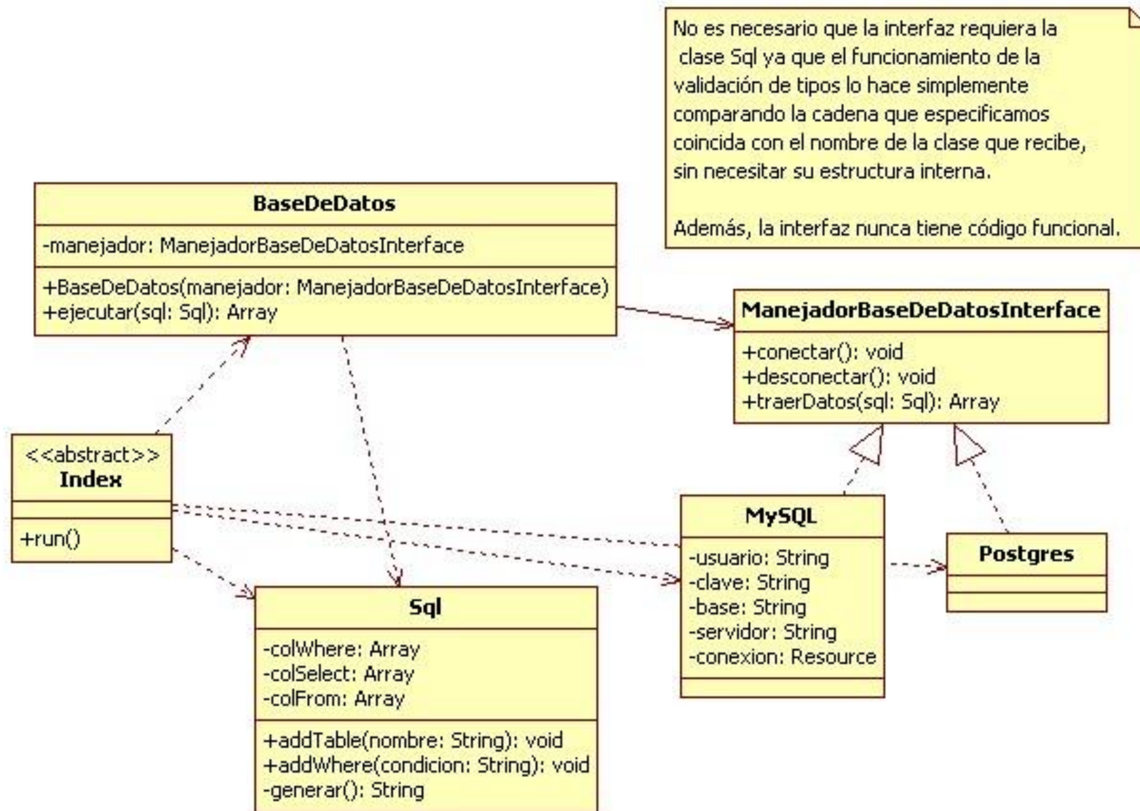
Por lo tanto, lo que aplicaremos a continuación es **un principio de diseño que se llama "Abierto / Cerrado"** y utiliza el polimorfismo a través de herencia o de interfaces.

Definición

(*)Foco de cambio: se le dice al lugar o lugares de nuestro sistema que por los requerimientos que estamos desarrollando, es seguro que ahí habrán cambios (agregar, modificar, etc) por consiguiente debemos preverlo de alguna forma para bajar su costo de modificación / mantenimiento.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diseño UML



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Ejemplo codificado

index.php

```php
<?php
require_once 'BaseDeDatos.php';
require_once 'MySQL.php';
require_once 'Sql.php';

abstract class Index
{
    public function run()
    {
        $bd = new BaseDeDatos(new MySQL());

        $sql = new Sql();

        $sql->addTable('usuarios');
        $sql->addWhere('id = 1');
        $sql->addWhere('id = 1');
        $sql->addWhere("nombre = 'enrique' ");

        $usuario = $bd->ejecutar($sql);

        echo highlight_string(var_export($usuario, true));

    }
}

Index::run();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

ManejadorBaseDeDatosInterface.php

```
<?php
interface ManejadorBaseDeDatosInterface
{
  public function conectar();
  public function desconectar();
  public function traerDatos(Sql $sql);
}
```

BaseDeDatos.php

```
<?php
require_once 'ManejadorBaseDeDatosInterface.php';
require_once 'Sql.php';

class BaseDeDatos
{
 private $_manejador;

 public function __construct(ManejadorBaseDeDatosInterface $manejador)
 {
     $this->_manejador = $manejador;
 }
 public function ejecutar(Sql $sql)
 {
     $this->_manejador->conectar();

     $datos = $this->_manejador->traerDatos($sql);

     $this->_manejador->desconectar();

     return $datos;
 }

}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

MySQL.php

```php
<?php
require_once 'ManejadorBaseDeDatosInterface.php';

class MySQL implements ManejadorBaseDeDatosInterface
{
    const USUARIO = 'root';
    const CLAVE = '';
    const BASE = 'tarea5';
    const SERVIDOR = 'localhost';

    private $_conexion;

    public function conectar()
    {
        $this->_conexion = mysql_connect(
            self::SERVIDOR,
            self::USUARIO,
            self::CLAVE
        );

        mysql_select_db(
            self::BASE,
            $this->_conexion
        );
    }
    public function desconectar()
    {
        mysql_close($this->_conexion);
    }
    public function traerDatos(Sql $sql)
    {
        $resultado = mysql_query($sql, $this->_conexion);

        while ($fila = mysql_fetch_array($resultado, MYSQL_ASSOC)){
            $todo[] = $fila;
        }
        return $todo;
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Sql.php

```
<?php
class Sql
{
  private $_colWhere = array();
  private $_colSelect = array('*');
  private $_colFrom = array();

  public function addTable($table)
  {
      $this->_colFrom[] = $table;
  }
  public function addWhere($where)
  {
      $this->_colWhere[] = $where;
  }
  private function _generar()
  {
      $select = implode(',',array_unique($this->_colSelect));
      $from   = implode(',',array_unique($this->_colFrom));
      $where  = implode(' AND ',array_unique($this->_colWhere));

      return 'SELECT '.$select.' FROM '.$from.' WHERE '.$where;
  }
  public function __toString()
  {
      return $this->_generar();
  }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Principio de diseño "Abierto / Cerrado"

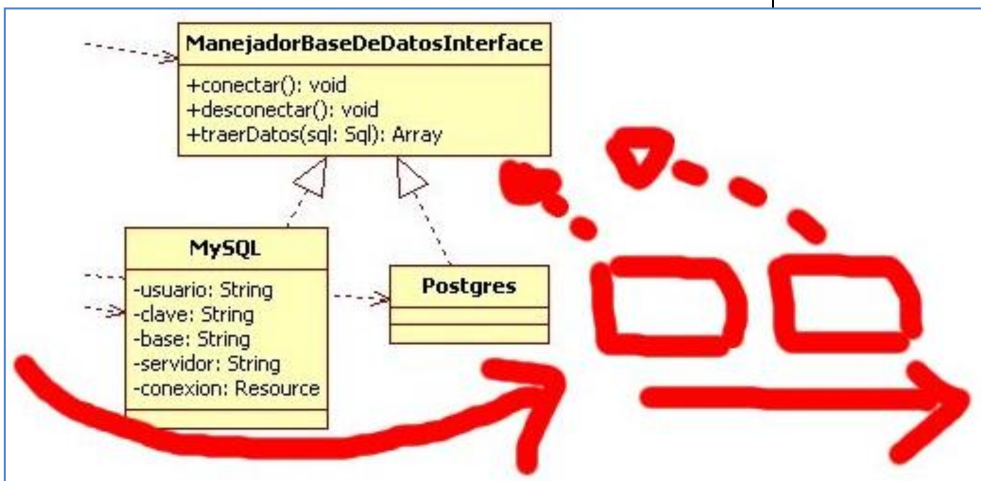
Enunciado formal:

"Entidades de Software (clases, módulos, funciones, etc) deberían ser abiertas para la extensión y cerradas para la modificación"

Lo cual significa que nuestro costo en el desarrollo se da cuando diseñamos un sistema que cuando hay cambios de requerimientos debemos modificar constantemente, por lo tanto un buen diseño deberá ser "Abierto / Cerrado", se diseñará una vez y cuando se necesite agregar nueva funcionalidad se hará sin modificar el código existente, solo agregando código nuevo.

¿Donde está el principio aplicado?

En la siguientes "flechas":



Imaginar que las clases MySQL y Postgres están "colgadas" desde sus flechas en un punto de apoyo como es la Interface, para agregar una nueva funcionalidad simplemente debemos agregar nuevas clases "colgadas de la interfaz" –por ejemplo- creando una nueva clase para el nuevo motor de base de datos, implementamos la interfaz y esta nos obligará a crear todos los métodos necesarios para que la clase genérica *BaseDeDatos* nos acepte y ejecute.

Comentario

Se dice que **todo buen diseño** (como las implementaciones de los *Patrones de Diseño*) de alguna forma u otra termina cumpliendo con este principio**.**

Pasando en limpio: cada vez que necesitemos soportar otro motor de base de datos crearemos una nueva clase, requerimos la interfaz (*"contrato de implementación"*) y esta nos obligará a implementar los métodos requeridos para que todo funcione de acuerdo al diseño.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

En mundos más "arquitectónicos" como es Java, donde no es nada raro aplicar patrones de diseño, el uso de las interfaces es algo habitual y necesario. Cuando diseñan una solución, generalmente crean una clase que ofrece un "servicio" y a su vez una "interfaz" que deberán cumplir todas las clases que quieran utilizar ese servicio, aplicando de esta forma el principio de diseño *"Abierto/Cerrado"*.

Recuerda: trata de mecanizar esta buena práctica, una clase que ofrece un servicio debe implementar en conjunto una interfaz que servirá de guía para todas las clases que quieran usar ese servicio.

Repítelo como un monje budista.

Tienes una pequeña duda? No hay dudas pequeñas…

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 17 - Ejercicio "Librería y la búsqueda de libros"

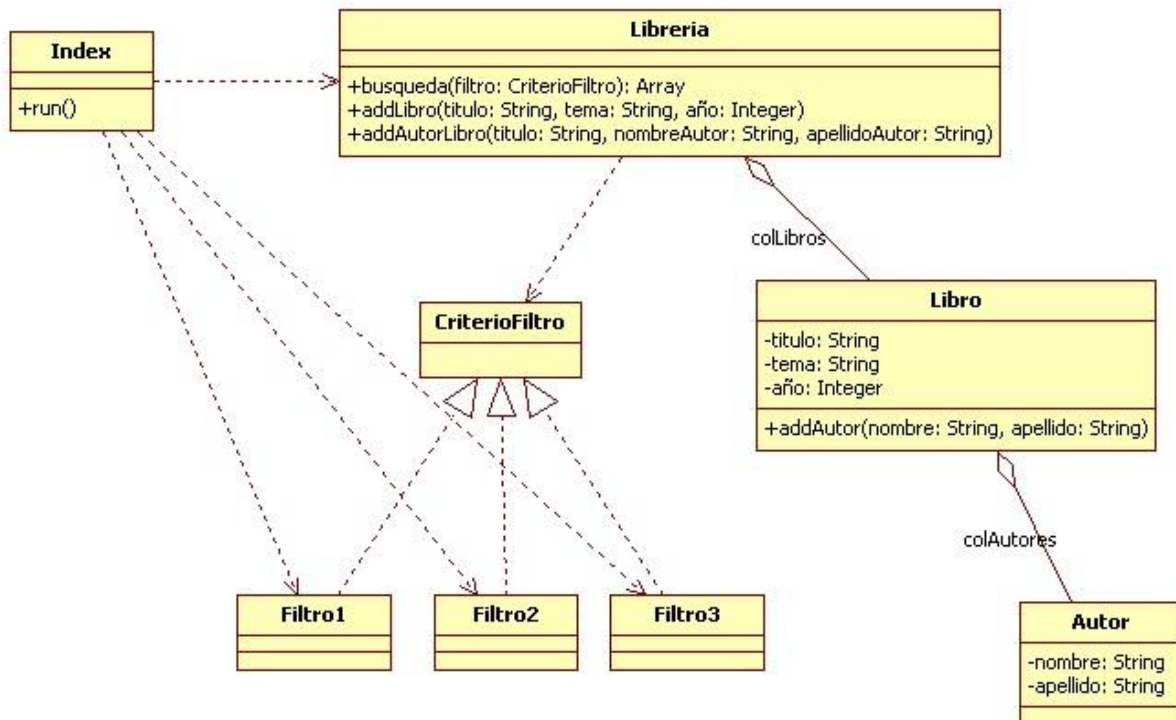
Con el objetivo de reafirmar un tema tan importante como las interfaces, se solicita implementar y resolver un nuevo problema: la búsqueda de libros por distintos filtros que no conocemos en su totalidad y que irán aumentando con el tiempo.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

“Tengo una librería y estoy implementando un sistema para registrar libros. Actualmente mi principal problema es la búsqueda de los libros, ya que sé que luego de implementado mi sistema de búsquedas me pedirán agregar nuevos filtros de información (he detectado el "foco de cambio"), por lo tanto deberé contemplar en el diseño este problema.”

El diseño borrador es el siguiente:



El diseño debe poder soportar las siguientes búsquedas:

- Todos los libros de un año: 2008
- Todos los libros del autor llamado "Enrique Place"
- Todos los libros del tema "PHP"
- Todos los libros que tengan en su título la palabra "Java" (deberán buscar la coincidencia parcial).

Tener muy en cuenta las relaciones presentadas en los diagramas:

- Index no ve directamente a los libros
- Index no ve directamente a los Autores

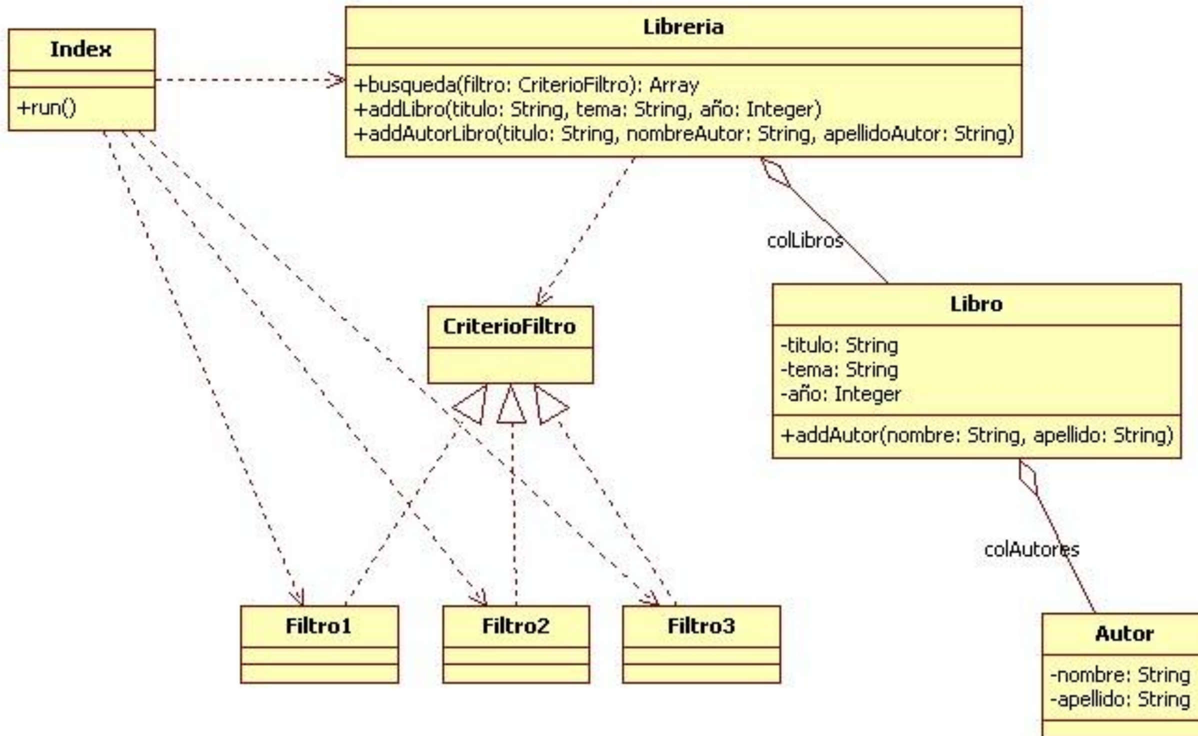
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

El objetivo de este ejercicio es completar el entendimiento de cómo funcionan las interfaces, cómo se aplica el principio de diseño **"Abierto / Cerrado"** y cómo se pueden implementar las relaciones en situaciones donde el acoplamiento es alto entre las clases de la solución.

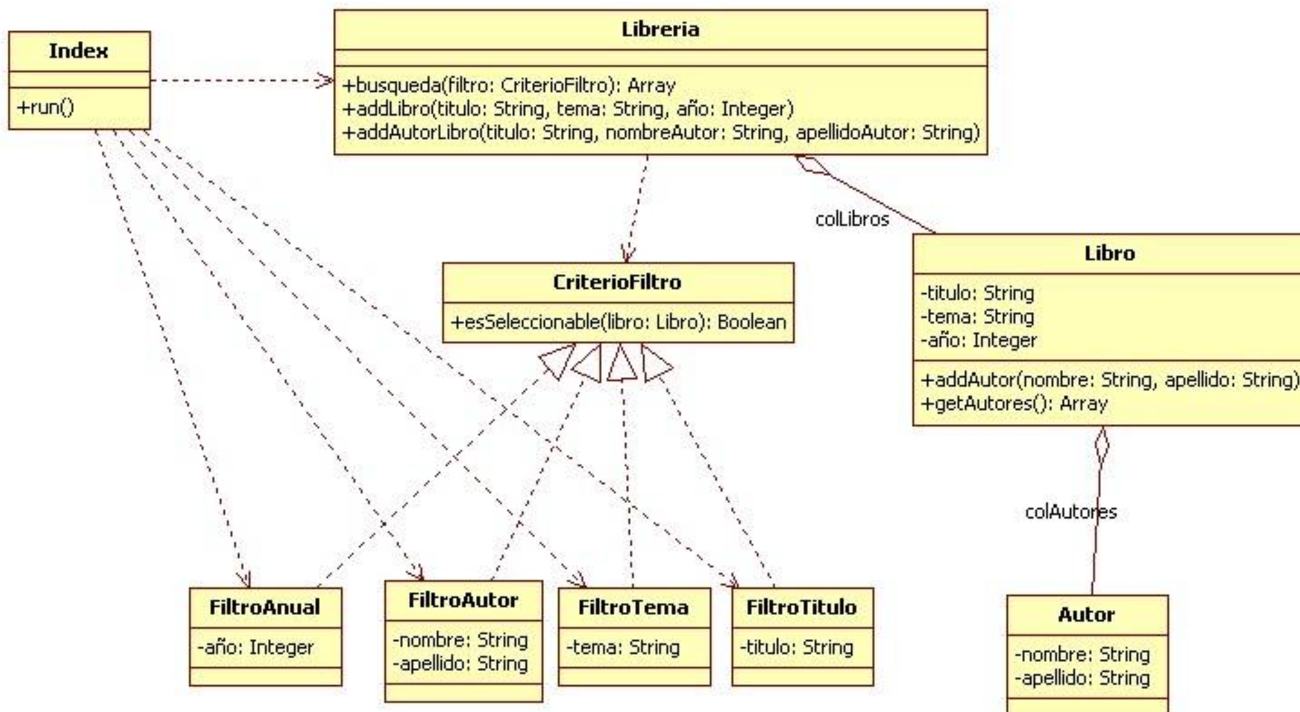
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diseño "Borrador" (con partes incompletas)



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diseño "Final" cumpliendo con todos los requerimientos



Agregados

- Se agrega la firma del método en la interfaz CriterioFiltro
- Los nombres de las clases Filtro y sus atributos
- Se obvian todos los get, set, constructores y toString del diagrama ya que se considera que el lector objetivo es un desarrollador que sabe determinar cuándo necesitarlos implementar (para este diagrama no aportan detalles relevantes).
- Se agrega en Libro un método **getAutores** para poder obtener luego su lista de autores.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

index.php:
```php
<?php
require_once 'Libreria.php';

require_once 'FiltroAnual.php';
require_once 'FiltroAutor.php';
require_once 'FiltroTema.php';
require_once 'FiltroTitulo.php';

abstract class Index
{
    public function run()
    {
        $libreria = new Libreria();

        /*
         * Carga de libros
         */
        $libreria->addLibro('Introduccion a Java','Java',2007);
        $libreria->addLibro('Introduccion a PHP','PHP',2008);
        $libreria->addLibro('Introduccion a los Patrones de Diseño','Patrones',2007);
        $libreria->addLibro('Introduccion a Zend Framework','Zend',2008);

        /*
         * Carga de autores
         */
        $libreria->addAutorLibro('Introduccion a PHP', 'Enrique', 'Place');

        /*
         * Búsqueda de libros
         */
        $libros2008   = $libreria->busqueda(new FiltroAnual(2008));
        $librosAutor  = $libreria->busqueda(new FiltroAutor('Enrique','Place'));
        $librosTema   = $libreria->busqueda(new FiltroTema('PHP'));
        $librosTitulo = $libreria->busqueda(new FiltroTitulo('Java'));

        echo self::_librosEncontrados2Html("Libros del 2008: ",$libros2008);
        echo self::_librosEncontrados2Html("Libros del Autor: ",$librosAutor);
        echo self::_librosEncontrados2Html("Libros del Tema: ",$librosTema);
        echo self::_librosEncontrados2Html("Libros del Titulo: ",$librosTitulo);
    }
    private function _librosEncontrados2Html($titulo, $array)
    {
        return $titulo.': ' . implode(', ', $array).'<br><br>';
    }
}

Index::run();
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Comentarios sobre el diseño

- Se requieren exactamente las clases que dice el diseño UML, ni más ni menos.
- Se cargan libros y autores sin acceder desde Index a las clases Libro y Autor, todo a través de la clase Librería.
- Este diseño permite tener un solo diseño de búsqueda y poder intercambiar fácilmente los algoritmos de los criterios de búsqueda.
- **El principio Abierto / Cerrado nos dice que un buen diseño es *"cerrado al cambio y abierto a la extensión"*,** en este caso el "foco de cambio" será agregar nuevos criterios de búsqueda, tema que está resuelto "*agregando solo código nuevo"* (sin modificar código existente) permite incorporar nueva funcionalidad (esto no es menor si entendemos que tocar algo que funciona puede generar a su vez nuevos bugs o nuevos cambios en cadena). Aquí logramos bajar el costo de mantenimiento de esta parte del sistema.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Libreria.php:

```php
<?php
require_once 'Libro.php';
require_once 'CriterioFiltro.php';

class Libreria
{
    private $_colLibros = array();

    public function busqueda(CriterioFiltro $filtro)
    {
        $librosRetorno = array();

        foreach( $this->_colLibros as $libro){
            if( $filtro->esSeleccionable($libro)){
                $librosRetorno[]= $libro;
            }
        }
        return $librosRetorno;
    }
    public function addLibro($titulo, $tema, $año)
    {
      $this->_colLibros[] = new Libro($titulo,$tema,$año);
    }
    public function addAutorLibro($titulo, $nombreAutor, $apellidoAutor)
    {
        foreach($this->_colLibros as $libro){
            if($libro->getTitulo() == $titulo){
              $libro->addAutor($nombreAutor,$apellidoAutor);
            }
        }
    }
}
```

Comentarios

- **La clase Librería sólo requiere Libro**, no requiere directamente la clase Autor
- **La búsqueda se realiza recibiendo un criterio y este criterio va recibiendo cada uno de los ítems de la colección de libros**, cada vez que el libro coincida con el criterio de filtro este método retornará "true", por lo tanto lo guardaré en un array para luego devolver a todos los objetos Libros encontrados. El filtro sólo tiene una responsabilidad, confirmar si se cumple el criterio o no.
- Para agregar un autor de un libro se hace una búsqueda y posteriormente se le da todos los datos al libro para que guarde su autor, todo sin tener que crear una instancia a este nivel.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Libro.php:

```
<?php
require 'Autor.php';

class Libro
{
    private $_titulo;
    private $_tema;
    private $_año;
    private $_colAutores = array();

    public function __construct($titulo,$tema,$año)
    {
        $this->_titulo = $titulo;
        $this->_tema = $tema;
        $this->_año = $año;
    }
    public function addAutor($nombre,$apellido)
    {
        $this->_colAutores[] = new Autor($nombre,$apellido);
    }
    public function getTitulo()
    {
        return $this->_titulo;
    }
    public function getTema()
    {
        return $this->_tema;
    }
    public function getAño()
    {
        return $this->_año;
    }
    public function getAutores()
    {
        return $this->_colAutores;
    }
    public function __toString()
    {
        return $this->_titulo;
    }
}
```

Comentarios

- El Libro sólo conoce al Autor
- En el UML se obvian todos los get y toString por considerarlos triviales a la hora de abordar el problema real.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Autor.php:

```
<?php
class Autor
{
    private $_nombre;
    private $_apellido;

    public function __construct($nombre, $apellido)
    {
        $this->_nombre = $nombre;
        $this->_apellido = $apellido;
    }
    public function getNombre()
    {
        return $this->_nombre;
    }
    public function getApellido()
    {
        return $this->_apellido;
    }
    public function __toString()
    {
        return $this->_nombre . ' ' . $this->_apellido;
    }
}
```

Comentarios

- Esta clase no aporta nada nuevo

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CriterioFiltro.php:

```
<?php
interface CriterioFiltro
{
    public function esSeleccionable(Libro $libro);
}
```

Comentarios

- No aporta nada nuevo de lo visto a la fecha.

FiltroAutor.php:

```
<?php
require_once 'CriterioFiltro.php';

class FiltroAutor implements CriterioFiltro
{
    private $_nombre;
    private $_apellido;

    public function __construct($nombre,$apellido)
    {
        $this->_nombre = $nombre;
        $this->_apellido = $apellido;
    }
    public function esSeleccionable(Libro $libro)
    {
        $encontrado = false;

        foreach($libro->getAutores() as $autor){
            if($autor->getNombre() == $this->_nombre &&
                $autor->getApellido() == $this->_apellido){

                $encontrado = true;
            }
        }
        return $encontrado;
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Comentarios

- A este nivel sólo conozco a la interfaz
- ***esSeleccionable*** recorre una lista de los autores de un libro para ver si al final coincide con lo solicitado.
- ***Cambia la forma de requerir las clases (sujeto a discusión y definición de criterios):*** podríamos decir que el filtro depende del libro para poder trabajar y no estaría mal implementarlo, **pero para este diseño se toma como criterio que las clases de filtro están *"altamente acopladas"* a los componentes de la clase Librería** y por lo tanto no van a trabajar sin ella: sin depender de nada más, sólo de los componentes asociados a la Librería, sin relación directa con Libro (usará el require_once de Libro existente desde Librería).
- **Se crea una "dependencia indirecta" entre el filtro de autor y el autor propiamente dicho:** se vuelve a repetir el caso donde el objeto (libro) expone sus componentes internos. El filtro sólo depende del Libro pero este es requerido en la Librería, y a través del require_once de Autor en Libro se accede al mismo. **La pregunta que siempre debemos hacer es: ¿qué forma es más conveniente?** ¿permitir exponer componentes internos de una clase a favor de simplificar su uso? ó ¿esconder todos los detalles internos para no violar el *"Principio de Ocultación" y* que los elementos externos no dependan de los elementos internos de las clases con las cuales se relacionan? (efectos en cadena).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Cuando vi por primera vez en la universidad materias como ***Diseño Orientado a Objetos*** y ***Patrones de Diseño*** (todo bajo Java), este fue particularmente uno de los ejercicios que más *"abrió mi mente"* y me hizo terminar de descubrir lo importante que son los diseños, las relaciones, los diagramas y fundamentalmente, que gran herramienta son las interfaces.

No es raro ver en Java que para hacer o usar tal o cual funcionalidad de una clase hay que implementar primero una interfaz para llegar hasta la clase que ofrece la funcionalidad que necesitamos.

Al principio no será natural esta forma de trabajo ya que poco estamos acostumbrados en el mundo PHP a hacerlo de esta forma, pero con el tiempo y la práctica tenemos que mecanizarnos que **cada vez que necesitemos implementar un servicio deberemos hacer en conjunto una interfaz que guíe cómo se debe usar y qué requisitos se deben cumplir.**

Repite como un monje budista:

"Un servicio (clase) debe disponer de una interfaz que obligue a cumplir con un contrato de implementación".

Alguna parte quedó confusa?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 18 - "Los Paquetes en UML"

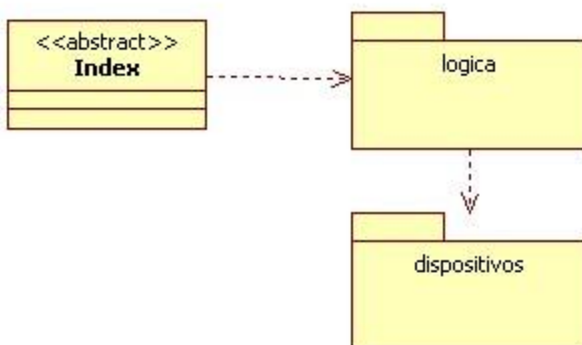
Los paquetes son la representación de un agrupador de clases (o de otros paquetes) en un sistema Orientado a Objetos, lo que generalmente se traduce físicamente en un directorio (aunque dependiendo de la tecnología esto puede ser más "lógico" que "físico"). Esto nos permitirá organizar nuestro sistema y tener un nivel mayor de abstracción

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cómo se representan

En UML se hace gráficamente con el dibujo de una "carpeta" como representación del concepto de "agrupador de clases". Este gráfico **se traduce simplemente en un subdirectorio físico en nuestra aplicación**, y en el caso particular de PHP, cuando una clase de un paquete deba apuntar a una clase en otro paquete, tendremos una referencia con un require_once con la ruta directorio/Clase.php.

Según este diagrama:



La clase Index se encuentra en la raíz de nuestro proyecto, y la relación siempre con y entre los paquetes es de "dependencia". Aquí tenemos un nuevo nivel de abstracción donde "conocemos" que nuestra clase Index depende de un paquete "Lógica" (que a dentro contendrá clases) y que a su vez el paquete de Lógica depende del paquete de "Dispositivos".

Depender de un paquete no significa otra cosa que ***"depender concretamente de una clase del interior del paquete"*** pero para la representación, nosotros dependemos del *"paquete que la contiene"*.

Por ejemplo, es como decir que "***el mecánico trabaja sobre el motor (paquete)"*** y no tener que dar la lista de todos los componentes del motor, o tener que decir que está trabajando concretamente sobre alguna parte en particular del motor (en este contexto no nos interesa tener tanto detalle, estamos abstrayendo complejidad para que sea más fácil de manejar nuestro sistema).

Podríamos diseñar un sistema que tenga los paquetes de *Ventas, Stock y Contabilidad* donde cada uno tendrá una lista extensa de clases pero que en algún momento nos interesará manejarlo a un nivel mucho más abstracto como el "paquete".

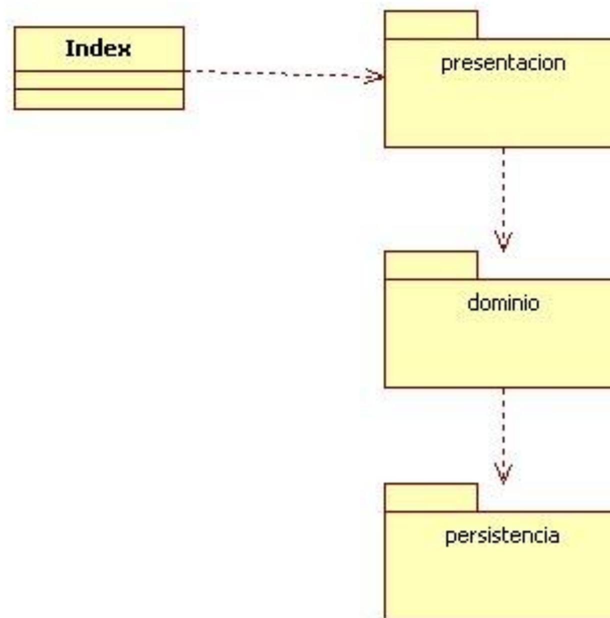
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¿Qué es una Arquitectura de 3 capas?

No es más que crear 3 paquetes con nombres como: presentación, dominio y persistencia, y darle a cada uno una responsabilidad concreta. La regla será que siempre deberán entrar las peticiones desde la interfaz (lo que usa el usuario), pasar por el dominio (donde realmente se resuelve el sistema) y finalmente la capa de persistencia (donde se guardan o recuperan los datos).

Cada capa recibe una petición de la otra, si alguna no tiene nada que hacer con ella, simplemente juega de "pasamanos" para la capa correspondiente según su responsabilidad.

El diagrama tradicional para una arquitectura de 3 capas:



Nota: veremos luego cómo se aplica en el último ejerció práctico de este libro.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¿Que son entonces los Namespaces?

Por muchos años los desarrolladores que conocemos de otros lenguajes notamos en PHP la ausencia del soporte por parte del lenguaje al manejo de "paquetes" a nivel conceptual (inicialmente lo diagramamos en UML pero en algún momento debemos codificarlo).

Tanto Java como .Net permiten codificar claramente un "Paquete UML":

- **Java tiene la sentencia "import"** que permite definir el paquete en el cual pertenece la clase en cuestión y a su vez incluir él o los paquetes que necesitamos acceder desde una clase particular.
- **En .Net, por ejemplo con Visual Basic, el nombre cambia a "namespace"** y a pesar que sirve para implementar los "paquetes", es un concepto un poco más amplio : permite definir que una clase es parte de un paquete, independientemente de donde se encuentre físicamente el archivo de la clase, por lo que podríamos tener varios archivos distribuidos en un sistema pero ser parte del mismo "paquete".

PHP5.3 incorporará, a pesar de haber seguido siempre a Java, por primera vez el concepto de "namespaces". Será la primera versión y aún no está claro la forma de usarlo (a la fecha siguen habiendo discusiones sobre "lo mal qu ese está implementando" esta funcionalidad). Por lo pronto es como si no contáramos con ella y debiéramos esperar hasta PHP6.

Para más información sobre este interesante tema, se recomienda leer

- PHP5: Diseño en 3 capas y problemas con subdirectorios
- "Petición para soporte de Name Spaces en PHP5"
- http://ar2.php.net/manual/es/language.namespaces.php

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

Los paquetes permiten un nuevo nivel de abstracción y que podemos usar para definir una arquitectura dividida en 3 capas, que no es más complejo que separar físicamente los archivos de las clases bajo el criterio único de la responsabildad de la capa. También vimos que esto puede cambiar con un concepto un poco más amplio como son los Namespaces, pero que a los efectos ambos sirven para representar los "paquetes".

Al final de cuentas es un mecanismo más que sirve para estructurar nuestro sistema en elementos de más alto nivel y luego relacionarlos entre ellos.

Este capítulo te pareció muy breve? Solicita ampliarlo

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPÍTULO 19 - EJERCICIO "PROGRAMACIÓN 'ORIENTADA A LA IMPLEMENTACIÓN' VS 'ORIENTADA A LA INTERFACE'"

Penúltimo ejercicio y terminamos cerrando con un concepto que considero fundamental, además del tema de las interfaces, entender la diferencia entre la programación *"orientada a la implementación"* versus programación *"orientada a la interface"*.

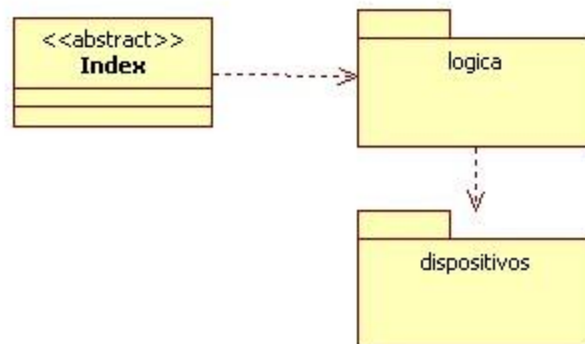
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Abril de 2006 escribí un artículo que cubre el 50 % del tema donde explico lo que es la programación orientada a la interfaz y dejo la promesa de terminarlo algún día. Ese día llegó y lo haremos juntos cómo la penúltimo ejercicio de este libro.

Tomando el artículo de referencia (actualizado al día de hoy):

Programación: "Orientada a la Implementación" versus "Orientada a la Interface" - Parte 1

Index usa la clase MaquinaDeEscribir.php que se encuentra dentro del paquete"logica", por consiguiente, desde la vista de paquetes, Index depende del paquete "logica", y como dentro de lógica la clase MaquinaDeEscribir depende de dos clases que están dentro del paquete "dispositivos", el paquete "logica" (donde está MaquinaDeEscribir), depende del paquete "dispositivos".



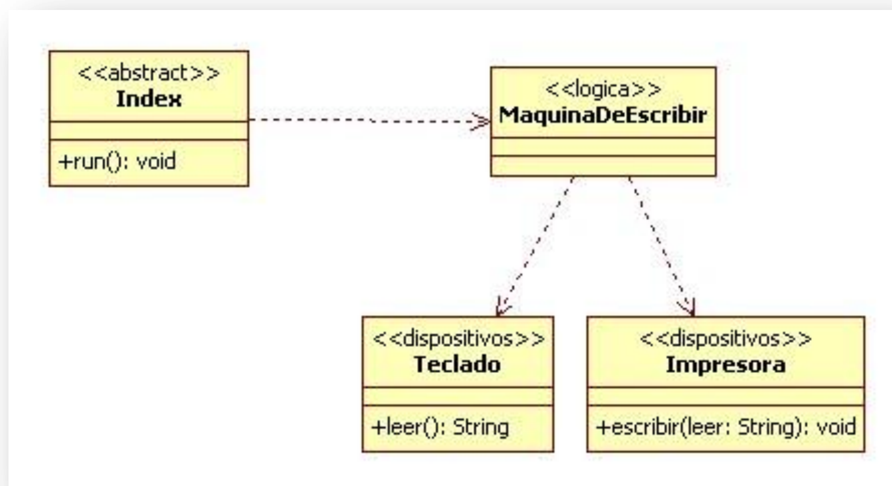
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

Enunciado:"La máquina de escribir tiene un dispositivo de entrada de datos (teclado) y un dispositivo de salida de datos (impresora)"

Como convención los nombres de paquetes son siempre en minúsculas y se traducen como un subdirectorio en nuestro sistema como forma de organizar nuestras clases.

Veamos entonces qué hay dentro de cada paquete. Para este caso se agrega como "estereotipo" de cada clase el nombre del paquete al cual pertenecen:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Lo que solicita el ejercicio es:

1. **Dos implementaciones** que deberán llamarse version1 y version2
2. **La primera tendrá la programación "orientada a la implementación",** deberán completar el diagrama presentado y hacer su implementación.
3. **La segunda deberá ser programación "orientada a la interfaz"** (como dice su nombre, usando "interfaces"), también deberán hacer el diagrama e implementación de la solución (todo de acuerdo a los conceptos vertidos en el artículo presentado oportunamente en el blog):
4. **Deberá depender de las interfaces y no de las clases concretas**.
5. Finalmente, cuando el diseño esté completamente terminado, **ver la forma de solucionar un problema en el diseño**: **cambiar la dependencia entre los paquetes**, ya que la lógica no debería depender de los detalles de implementación de los dispositivos, sino, los dispositivos cambiar en caso de que la lógica cambie ("lo menos importante depende de lo más importante").

Dudas, como siempre, envíala! 😉

Sugerencia

Se recomienda la lectura profunda y meditada del artículo de referencia.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

El objetivo de este ejercicio estaba dividido en dos partes:

> **La primera parte**, repasar el concepto y la forma de requerir las clases y cómo hacerlo en caso de tener subdirectorios (representados en UML como "paquetes" para agrupar clases).
>
> **La segunda parte**, buscaba seguir aplicando con ejemplos por qué cada vez que implementamos un "servicio" debemos de forma casi mecánica incluir una interfaz para dejar el camino definido a cualquier clase que necesite usar ese servicio.

Una vez concluidos estos dos ejercicios, en esta misma corrección, explicaré algunos conceptos sobre ***"Análisis & Diseño Orientado a Objetos"*** que creo (aunque escapa el alcance del temario de este libro) es el momento ideal para abordar.

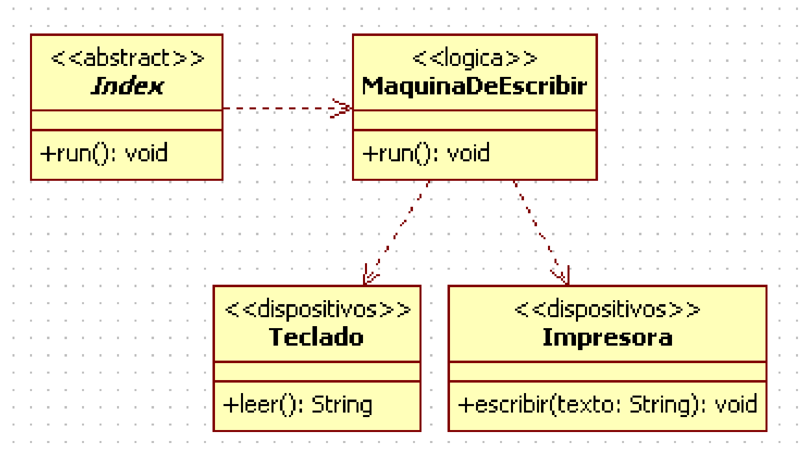
No desaprovechemos la oportunidad ;-)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

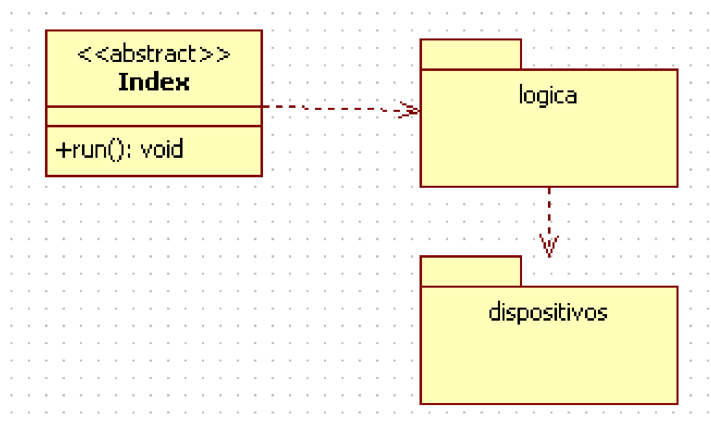
Parte 1: hacer una "máquina de escribir" siguiendo el ejemplo del artículo

Se creará una clase "MaquinaDeEscribir" que estará en el paquete "lógica" (donde se define la forma en que trabajará el sistema). Posteriormente esta "lógica" hará uso de los dispositivos de más bajo nivel para poder acceder a la entrada y salida de datos.

El diseño UML era el siguiente:



Y la vista de paquetes era la siguiente



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Implementación

index.php

```
01. require_once "logica/MaquinaDeEscribir.php";
02.
03. abstract class Index
04. {
05.     static public function run()
06.     {
07.         echo 'Ejecuto máquina de escribir->';
08.         MaquinaDeEscribir::run();
09.     }
10. }
11.
12. Index::run();
```

MaquinaDeEscribir.php (dentro del paquete "lógica")

```
01. require_once 'dispositivos/Teclado.php';
02. require_once 'dispositivos/Impresora.php';
03.
04. abstract class MaquinaDeEscribir
05. {
06.     static function run()
07.     {
08.         $miTeclado = new Teclado();
09.         $miImpresora = new Impresora();
10.
11.         $miImpresora->escribir($miTeclado->leer());
12.     }
13. }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Teclado.php (dentro del paquete "dispositivos")

```
01. class Teclado
02. {
03.     public function leer()
04.     {
05.         // a efectos del ejemplo, solo retornará un
06.         // un texto como representación de una
07.         // entrada de datos
08.         return "texto ingresado";
09.     }
10. }
```

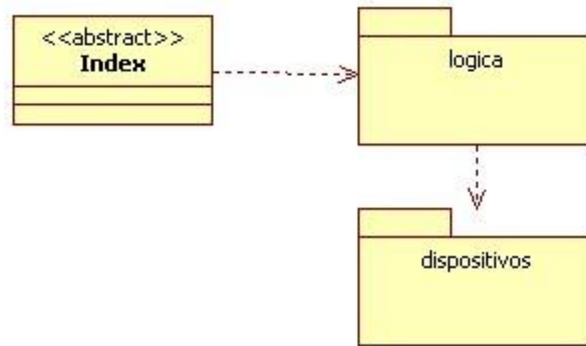
Impresora.php (dentro del paquete "dispositivos")

```
01. class Impresora
02. {
03.     function escribir($texto)
04.     {
05.         echo $texto;
06.     }
07. }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Conclusión

Según se explica en el artículo de referencia (PHPSenior), la lectura de los diagramas y sus relaciones nos hacen llegar a la siguiente conclusión:



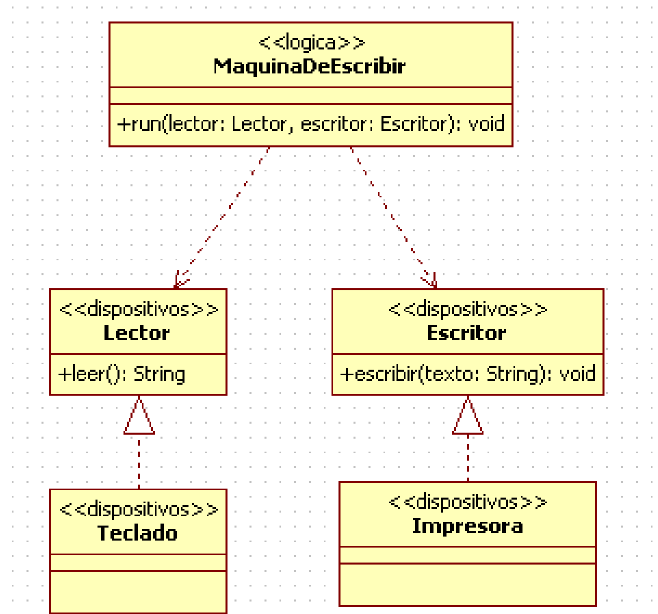
"La lógica de nuestro sistema depende de los dispositivos, por lo pronto esto significa ***un problema en nuestro diseño*** *, ya que según el sentido de las flechas, cualquier cambio en nuestros dispositivos de bajo nivel afectarán el corazón de nuestro sistema"*

Esto es tan peligroso como decir que habría que cambiar la implementación del sistema de gestión de nuestra empresa cada vez que cambie la forma de entrada o salida de datos, algo que es ridículo. Sería correcto si cambia la lógica central de nuestro sistema se vean afectados los dispositivos de bajo nivel y estos deban ser adaptados, **pero no al revés!**

Esto lo podremos revertir usando interfaces, por lo tanto veremos todo el proceso de mejorar nuestro diseño a continuación…

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Parte 2: Agregar interfaces al diseño propuesto en la parte 1



Aquí separamos la clase *MaquinaDeEscribir* de depender directamente de las clases *Teclado* e *Impresora*.

Hay varios detalles importantes:

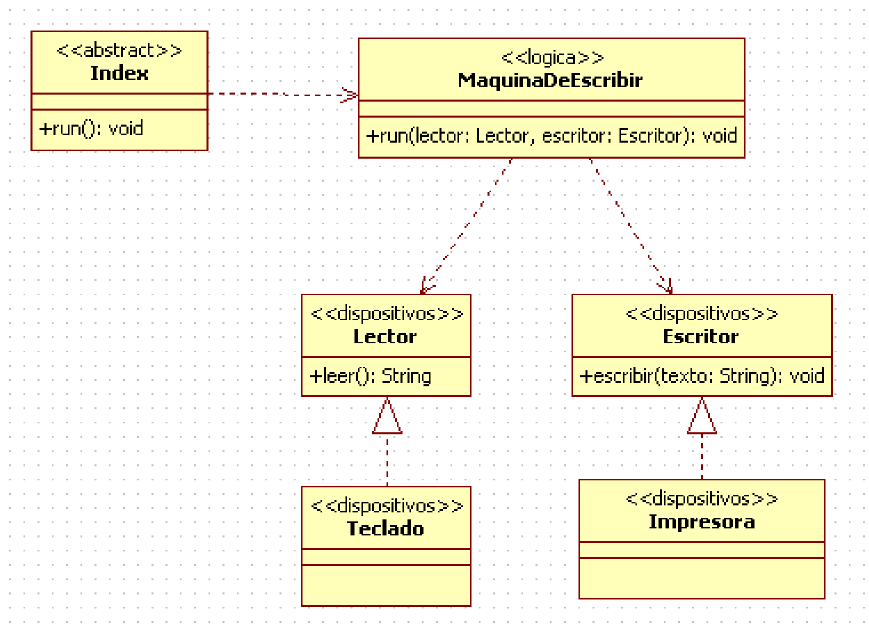
- **Pasamos de depender de "implementaciones concretas" a depender de "implementaciones abstractas"**, ya que para *MaquinaDeEscribir* lo único con que se relaciona es con elementos de tipo las *interfaces* y no se preocupa cómo se implementen las clases que cumplan con el *"contrato de implementación"* (es problema de la interfaz definir lo que se necesita y de la clase que quiere el servicio en cumplir los requisitos necesarios para usarlo).

- Casi sin darnos cuenta **estamos cumpliendo con el principio *"Abierto/Cerrado"*,** ya que vamos a estar *"cerrados al cambio pero abiertos a la extensión"*, el *"foco de cambio"* será agregar nuevos dispositivos pero no necesitaremos modificar el funcionamiento de *MaquinaDeEscribir*, ya que su algoritmo será siempre el mismo.

Aún hay otro detalle más, **no está representada la clase *Index***, la que nos da el contexto de cómo se ejecuta todo este diseño…

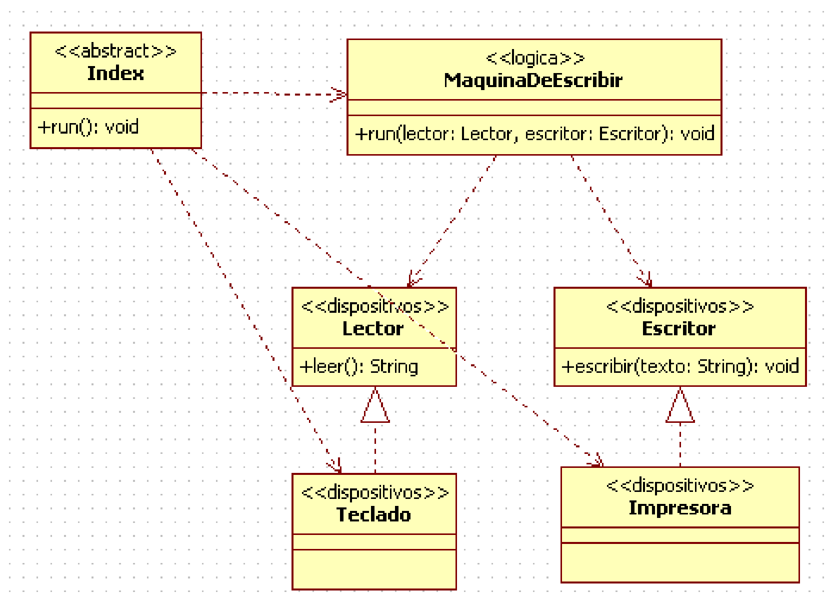
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Se podría decir que solo falta la flecha de Index a la clase *MaquinaDeEscribir*:



Pero no, no es suficiente, ya que para que *MaquinaDeEscribir* pueda funcionar **depende de que le ingresen dos elementos de tipo *Lector* y *Escritor*.** Prestar atención, **la clase *MaquinaDeEscribir* no depende directamente de las clases concretas**, sí las terminará usando a través de **polimorfismo** (a esto se le llama *"indirección"* o *"relación indirecta"*: usa elementos concretos pero depende en realidad de elementos abstractos).

El diagrama correcto debería ser el siguiente:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Implementación

Lo medular de la implementación es el nuevo método “run” de la clase *MaquinaDeEscribir*, que recibe ahora dos tipos de objetos, “lectores” y “escritores”:

index.php

```
1  <?php
2  require_once 'logica/MaquinaDeEscribir.php';
3  require_once 'dispositivos/Teclado.php';
4  require_once 'dispositivos/Impresora.php';
5
6  abstract class Index
7  {
8    static public function run()
9    {
10     MaquinaDeEscribir::run(new Teclado(), new Impresora());
11   }
12 }
13
14 Index::run();
```

MaquinaDeEscribir.php

```
1  <?php
2  require_once 'dispositivos/LectorInterface.php';
3  require_once 'dispositivos/EscritorInterface.php';
4
5  abstract class MaquinaDeEscribir
6  {
7      static public function run(Lector $lector, Escritor $escritor)
8      {
9        $escritor->escribir($lector->leer());
10     }
11 }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

LectorInterface.php

```php
<?php
interface Lector
{
  public function leer();
}
```

EscritorInterface.php

```php
<?php
interface Escritor
{
  public function escribir($texto);
}
```

Impresora.php

```php
<?php
require_once 'EscritorInterface.php';

class Impresora implements Escritor
{
    function escribir($texto)
    {
        echo $texto;
    }
}
```

Nota: como *Impresora* y la interfaz *Escritor* son del mismo paquete, no llevan rutas en el *require_once* que hagan referencia desde donde se ejecuta *Index*, ya que esta ubicación puede cambiar y la relación entre los paquetes debe estar fija por diseño (revisar las flechas del diagrama anterior).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Teclado.php

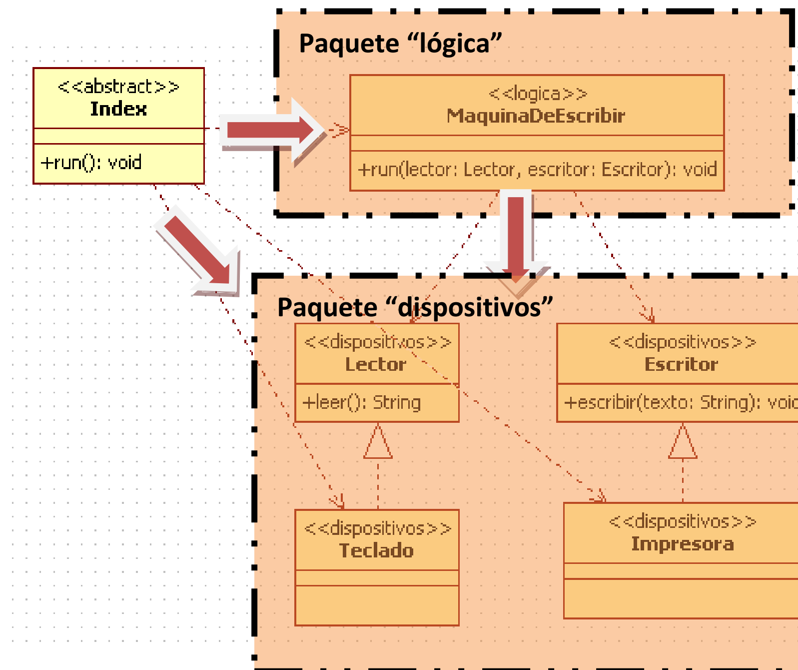
```
1   <?php
2   require_once 'LectorInterface.php';
3
4   class Teclado implements Lector
5   {
6       public function leer()
7       {
8        /* A efectos del ejemplo, solo retornará un
9         un texto como representación de una
10        entrada de datos */
11        return "texto ingresado";
12      }
13  }
```

Nota: el *require_once* es igual que en el caso anterior, se asume la invocación desde el paquete.

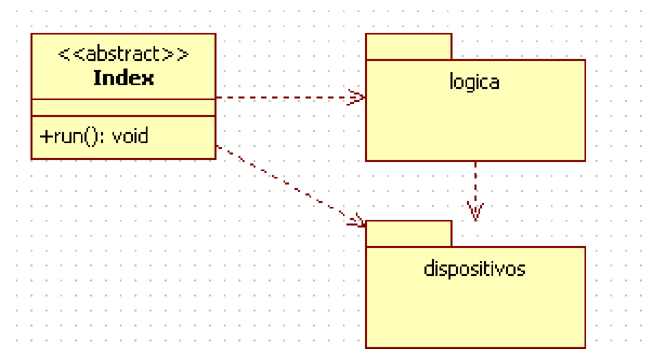
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¿Y el Diagrama de Paquetes?

… ¿cómo quedará ahora la relación vista a nivel de paquetes? Viendo el sentido de las flechas y solo representando una flecha de dependencia (así son siempre entre paquetes) resumiendo todas las existentes (aumentamos el nivel de abstracción) …



Entonces, la vista de paquetes se ve así:



Lo que no debe asustar es que existan dos flechas de dependencia contra el paquete desde *Index*, ya que esto es normal (para usar un servicio o conjunto de clases podremos depender de uno o más paquetes), lo que sí debería preocuparnos (además de evitar las relaciones cíclicas) es que –a pesar de haber mejorado hacia un diseño *"que depende de interfaces y no de implementaciones concretas"*- **nuestro paquete de "lógica" sigue dependiendo del paquete de "dispositivos"**…

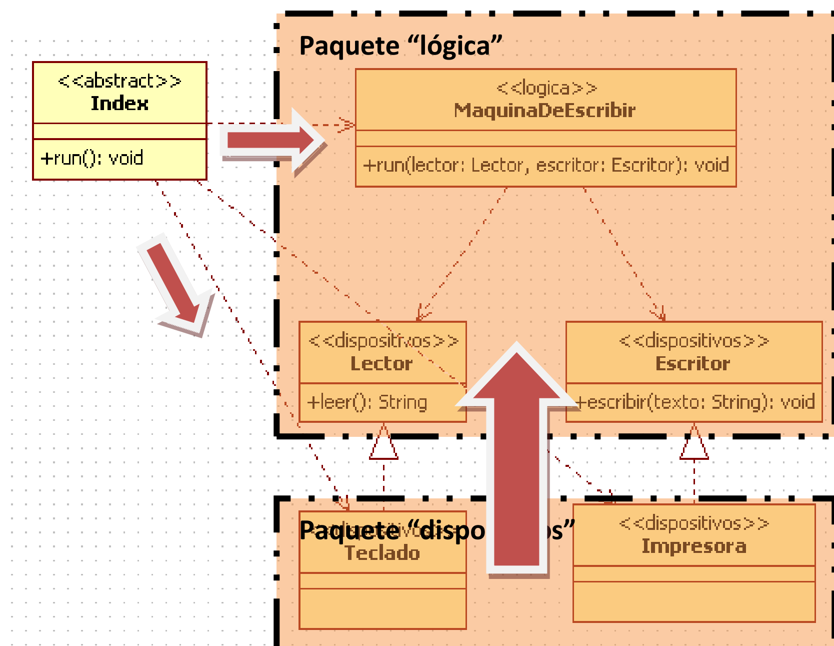
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

El *"Principio de Inversión de dependencias (DIP)"*

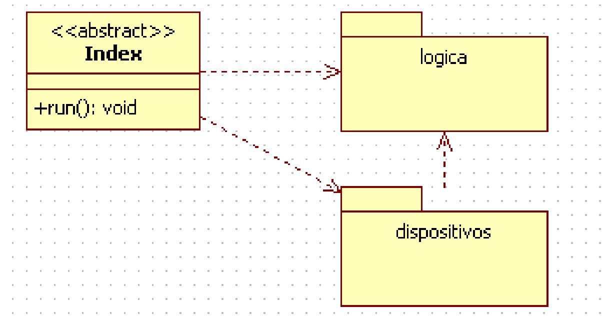
¿Cómo solucionamos esto? Cambio de sentido de flechas entre paquetes

Hay una técnica para cambiar el sentido de las dependencias entre paquetes que es **mover las clases de un paquete hacia el otro paquete**, o en su defecto, **creando un tercer paquete y mover las clases para cambiar el sentido de las direcciones**. Para este caso concreto,

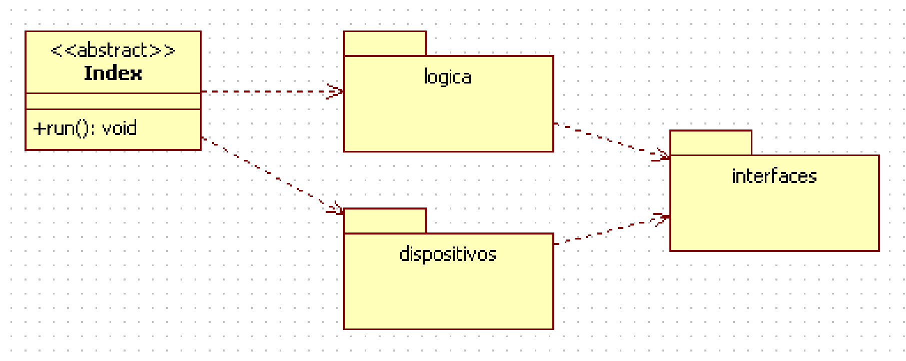
moveremos las interfaces a la clase que provee el servicio, por lo tanto cambiaremos el sentido de las dependencias entre paquetes:



Nuevamente, **prestar atención el sentido de las flechas de implementación**, ahora apunta a las clases que están en el paquete de arriba ("lógica"), así que pasando en limpio el diagrama de paquetes ahora quedaría:



Licencia: http://creativecommons.org/licenses/by-nc/3.0/



Alternativa: como solución alternativa podríamos crear un tercer paquete para las Interfaces, donde la lógica y los dispositivos dependerían, rompiendo la relación directa que antes existía.

¡Ey! ¡Finalmente logramos nuestro objetivo!, ahora el sistema está mejor diseñado, implementado *"Orientado a la Interfaz",* cumple con el principio de diseño *"Abierto/Cerrado"* y el paquete de más *"alto nivel"* no depende de los detalles o cambios del paquete de más *"bajo nivel"!!*

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Premisa: *"En general, las interfaces pertenecen a las entidades que las emplean"*

El paquete de "lógica" conoce y tiene una interfaz para conectarse con el paquete "dispositivos", pero no sabe qué hay dentro del paquete "dispositivos".

Para más información:

DIP - Dependency Inversion Principle (Principio de Inversión de Dependencias)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Comentarios adicionales

Pienso que era difícil intuir el desenlace de esta nueva historia de suspenso **;-)**

A esta altura del libro espero haber logrado transmitir un poco de experiencia y dejarles una "semilla" para seguir investigando y profundizando el tema de la *"Programación Orientada a Objetos"* y que no es solo *"crear clases y hacer objetos"*.

Existe otra disciplina que estudia cómo poder aplicar los conceptos más allá de la *"visión del árbol"*, estudiar el impacto de los paquetes, los patrones y los principios de diseño, y esa *"magia de druidas"* se llama "***Análisis y Diseño Orientado a Objetos".***

Nos estamos encontrando en el próximo desafío, el último ejercicio del libro, donde aplicaremos el último de los conceptos que quiero dejarles: ***"la arquitectura de tres capas"***

Necesitas el código completo de la solución del ejercicio?

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPÍTULO 20 - EJERCICIO "DESARROLLAR UN SISTEMA DE ABM DE USUARIOS"

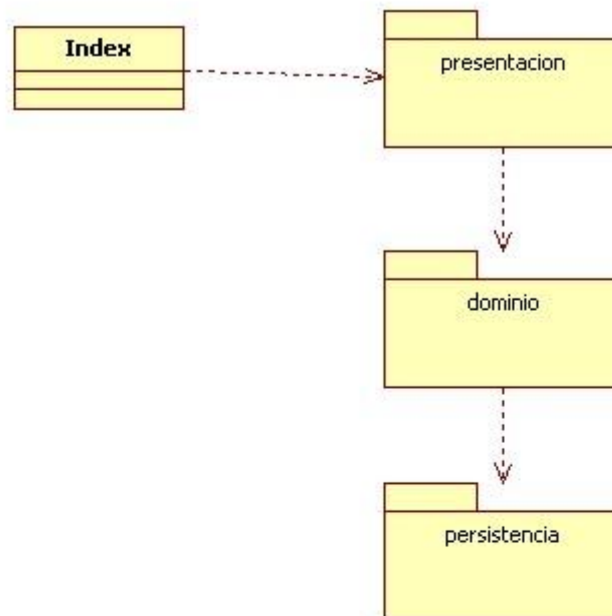
Este fue el último ejercicio de los cursos a distancia que integraban todos los conceptos vertidos en los capítulos de este libro.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Requerimientos

"Ustedes están a prueba por la empresa SURFORCE y a modo de probar sus conocimientos para obtener el puesto de PHP Senior ***recibirán un sub-sistema que deberán concluir con todo éxito****.*

Lo que hay desarrollado hasta el momento es ***el esqueleto de un sistema en 3 capas*** *que cumple con el siguiente UML:*



Como se puede observar, todo parte de un único archivo inicial "index.php*" y posteriormente accede a la capa de presentación, luego a la capa de dominio y finalmente a la capa de persistencia. Se entiende que* ***no se pueden saltear las capas propuestas ni el sentido de las dependencias entre paquetes****.*

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Lo que se deberá desarrollar en primera instancia es un sistema de ABM de Usuarios (Altas/Bajas/Modificaciones) con un diseño de interfaz similar a entornos como Ruby On Rails (a continuación un ejemplo de cómo se deberá ver la interfaz):

Cliente was successfully created.

Listing clientes

Nombre	Telefono			
Enrique Place	123-45679	Show	Edit	Destroy
Ernesto Climent	123-34234	Show	Edit	Destroy
Carlos Madrigal	123-5555	Show	Edit	Destroy
Roberto Ambrosoni	6969696-1234	Show	Edit	Destroy
Fabian Clavijo	34123-452345	Show	Edit	Destroy
Mauro Castro	14514235-3	Show	Edit	Destroy
Mr. Cohen	1234-12345	Show	Edit	Destroy
Armando Gervaz	34513-313	Show	Edit	Destroy
Oscar Lopez	09999-123	Show	Edit	Destroy
Laura Recto	52345245	Show	Edit	Destroy

New cliente

La primer pantalla es un listado de los registros existentes en la tabla, posteriormente en cada línea existen las opciones de mostrar toda la información, editarla o eliminarla. Al final del listado se puede observar la opción de dar de alta un usuario.

Podrán bajar un sistema base de ejemplo (sin terminar) y ustedes deberán:

1. *Aplicar todos los conocimientos vistos hasta el momento (Kiss es fundamental).*
2. *Relevar todo lo existente en el sistema entregado (archivo de configuración, entender cómo funciona, cual es la forma de trabajo, qué falta, etc).*
3. *Implementar todo lo necesario para cumplir el ejemplo de diseño anterior, logrando hacer un listado de usuarios, detalle, alta, baja y modificación de datos.*
4. *Presentar el diagrama UML final con todos los agregados realizados.*

Se sugiere la lectura complementaria de los siguientes materiales:

- *PHP5: Diseño en 3 capas y problemas con subdirectorios*
- *Conceptos: "Separar el código de la capa de presentación"*
- *"Petición para soporte de Name Spaces en PHP5"*

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Se entregará un archivo comprimido con el código fuente del sistema con todas las capas y hasta una carpeta llamada "sql" que contendrá una base de datos y una tabla usuarios para poder iniciar su desarrollo.

¡Ese puesto en la compañía puede ser tuyo!"

Nota: El diseño general de 3 capas del sistema base que les entrego fue usado en un sistema real de aplicaciones SMS para celulares (no es meramente "otro problema teórico para resolver").

Para bajar el archivo comprimido con el código fuente

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Solución

La idea de esta tarea es presentar un problema completo a partir de un template de un sistema existente, el cual deberían usar de apoyo para cumplir con los requerimientos solicitados.

A pesar que el ejercicio permite aplicar algunos conceptos aprendidos en este libro, **no deja de ser un ejercicio y como tal, desborda de programación "artesanal".** Hay muchos detalles que no son tenidos en cuenta, como puede ser la seguridad del sistema o cómo procesar más eficientemente la capa de presentación en conjunto al armado de html, estilos, etc.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Cambios que se aplicaron en la resolución

A grandes rasgos se hicieron las siguientes modificaciones

General

- **Se usó el criterio de definir estático todo lo que era genérico o general de la clase**, y lo opuesto para referirse a los métodos que se aplican a una instancia de un objeto. Es decir, si la clase usuario se usaba para traer todos los usuarios del sistema, debía ser Usuario::getAll(), y los métodos estándar se aplicaban a una instancia única que representaba a "él" usuario.
- **Se creó un método load de la clase dominio/Usuario** para una vez obtenido el id de usuario por parámetro, este cargue el usuario que se encuentre en la persistencia. Se puede decir que hace el efecto de una "fábrica" (patrón de diseño).

Persistencia

- Modificamos la persistencia para que retorne los datos en un array
- Modificamos la clase Sql para contemplas los demás casos del ABM

Dominio

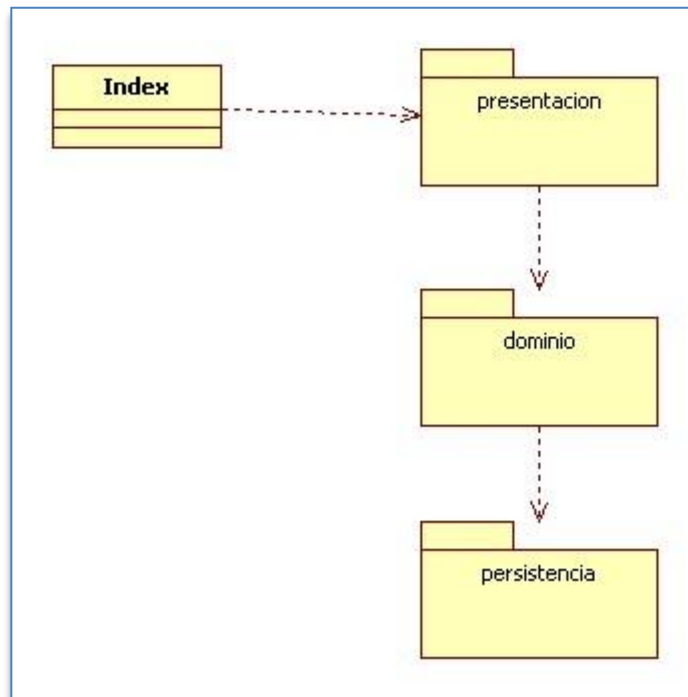
- Agregamos el constructor para inicializar los datos que consideramos básicos del objeto
- Modificamos el getAll para que pueda retornar una colección de objetos array
- Creo el toString para que cada usuario sepa cómo imprimirse cuando lo use la capa de presentación

Presentación

- **Se armaron todos los formularios requeridos para el ABM**
- **Se agregaron redirecciones para** luego de procesar regrese a la página inicial.

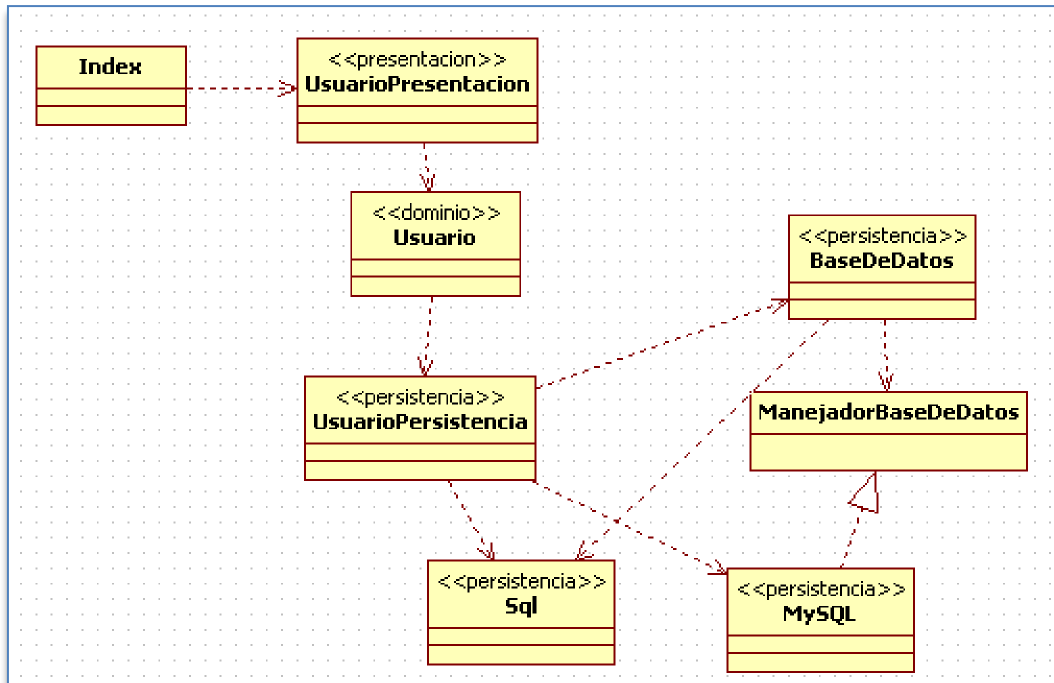
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diagrama tradicional de paquetes



Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diagramas de clases y sus paquetes de origen

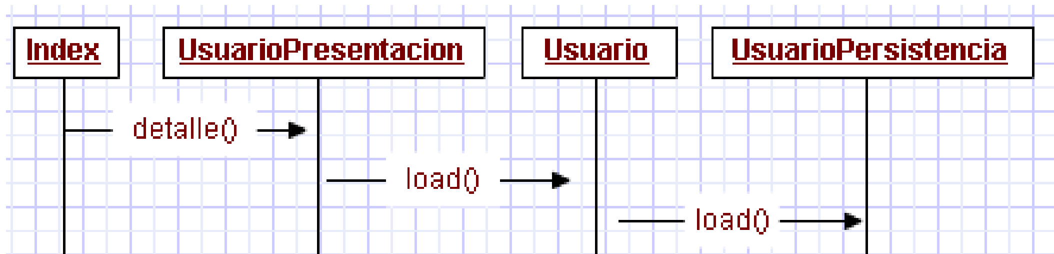


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Diagramas de Secuencia

Se sabe que **los diagramas de clases son una "foto estática" del diseño general de una solución**, pero no siempre es suficiente para entender cómo un diseño de clases debe trabajar, qué se invoca primero y qué después, para ello existen los "diagramas de secuencia" que nos permiten tener una visión más dinámica y temporal de cómo interactúan los objetos una vez que se inicia una acción determinada.

Caso: Ver detalle de un usuario (ejemplo)



Aquí se pude observar cómo se van sucediendo las invocaciones de los métodos, y cómo es que una clase se relaciona con otra invocando los métodos de la clase siguiente. La línea temporal se define de arriba – abajo, el primer método que se invoca es el que hace Index al ejecutar **UsuarioPresentacion::detalle()** y sigue bajando hasta llegar a la última ejecución.

Por ejemplo,

1. Index invoca el detalle() de UsuarioPresentación,
2. posteriormente este invoca el método load() de la clase Usuario y
3. finalmente este invoca el load de la clase UsuarioPersistencia (todo un "pasamanos").

Aunque generalmente no se representa, al final de la cadena lo que sucede es que la última invocación retorna información, que recibe la clase siguiente al invocación, hasta llegar otra vez a la clase inicial (secuencia inversa):

1. UsuarioPersistencia retorna datos,
2. Usuario lo recibe y arma los objetos para luego retornarlos a
3. la clase UsuarioPresentación para luego esta retornar a
4. Index, recibe lo que tiene que mostrar (html) y así ejecuta un "echo"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Toda documentación UML debería contar con los correspondientes diagramas de secuencia que documenten cómo se usan las clases entre ellas y en qué momento una invoca métodos de la otra.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Partes esenciales del código de la solución

Index.php

```
<?php
require_once 'configuracion.php';
require_once PRE . DIRECTORY_SEPARATOR . 'UsuarioPresentacion.php';

abstract class Index
{
    const SIN_PARAMETROS = 0;

    static public function run($get)
    {
        DEBUG ? var_dump($get) : null;

        if(count($get) != self::SIN_PARAMETROS){
            self::_procesarModulo();
        }else{
            self::_porDefecto();
        }
    }
    static private function _porDefecto()
    {
        echo 'Pagina por Defecto';
        echo '<ul>';
        echo '<li><a href="?modulo=listado">listado</li>';
        echo '</ul>';
    }
    static private function _moduloNoExiste()
    {
        echo 'Modulo no Existe';
    }
```

Nota

Para facilitar el entendimiento del código el método a continuación se pasa entero a la siguiente página (todo es parte de la misma clase)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

```
    static private function _procesarModulo()
    {
        switch ($_GET['modulo']){
            case 'listado':
                if (isset($_GET['mensage']) && $_GET['mensaje'] != "") {
                    echo "El Usuario ha sido ".$_GET['mensaje']." correctamente";
                }
                echo UsuarioPresentacion::listadoUsuarios();
                break;
            case 'detalle':
                echo UsuarioPresentacion::detalle($_GET['id']);
                break;
            case 'nuevousuario':
                echo UsuarioPresentacion::mostrarFormNuevoUsuario();
                break;
            case 'insertar':
                if(UsuarioPresentacion::guardarUsuario(
                        $_POST['nombre'],
                        $_POST['apellido'])) {

                    header("Location:index.php?modulo=listado&mensaje=guardado");
                }
                break;
            case 'modificarusuario':
                echo UsuarioPresentacion::mostrarFormModificarUsuario($_GET['id']);
                break;
            case 'modificar':
                if(UsuarioPresentacion::modificarUsuario(
                        $_POST['id'],
                        $_POST['nombre'],
                        $_POST['apellido'])){

                    header("Location:index.php?modulo=listado&mensaje=modificado");
                }
                break;
            case 'eliminar':
                if(UsuarioPresentacion::eliminarUsuario($_GET['id'])) {
                    header("Location:index.php?modulo=listado&mensaje=eliminado");
                }
                break;
            default:
                self::_moduloNoExiste();
                break;
        }
    }
}
Index::run($_GET);
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

UsuarioPresentacion.php

```
<?php
require_once 'configuracion.php';
require_once DOM . DIRECTORY_SEPARATOR . 'Usuario.php';

abstract class UsuarioPresentacion
{
    static public function listadoUsuarios()
    {
        $usuarios_arr = Usuario::getAll();

        $retorno = '<ul>';
        foreach($usuarios_arr as $objetoUsuario){
            $retorno .= '<li>'.$objetoUsuario;
            $retorno .= " <a href='?modulo=detalle&id=".$objetoUsuario-
>getId() ."'>Mostrar</a> | ";
            $retorno .= " <a href='?modulo=modificarusuario&id=". $objetoUsuario-
>getId() ."'>Modificar</a> | ";
            $retorno .= " <a href='?modulo=eliminar&id=". $objetoUsuario-
>getId() ."'>Eliminar</a> | ";
            $retorno .='</li>';
        }
        $retorno .= "<li><a href='?modulo=nuevousuario'>Nuevo Usuario</a>";
        $retorno .= '</ul>';
        return $retorno;
    }
    static public function detalle($id)
    {
        return Usuario::load($id);
    }
    static public function mostrarFormNuevoUsuario()
    {
        return self::_mostrarFormulario();
    }
    static public function mostrarFormModificarUsuario($id)
    {
        $usuario = Usuario::load($id);

        $form = self::_mostrarFormulario(
            $id,
            $usuario->getNombre(),
            $usuario->getApellido(),
            "modificar"
        );
        return $form;
    }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

```
    static public function modificarUsuario($id, $nombre, $apellido)
    {
        $usuario = new Usuario($id, $nombre, $apellido);
        return $usuario->modificarUsuario();
    }
    static public function eliminarUsuario($id)
    {
        $usuario = new Usuario($id);
        return $usuario->eliminarUsuario();
    }
    static private function _mostrarFormulario($id = "", $nombre = "", $apellido = "",
 $accion = "insertar")
    {
        $retorno = "";
        $retorno = "<form action='?modulo=".$accion."' method='post'>";
        $retorno .= "<input type='hidden' name='id' value='".$id."' />";
        $retorno .= "Nombre:<input type='text' name='nombre' value='". $nombre."' /> "
;
        $retorno .= "Apellido:<input type='text' name='apellido' value='".$apellido."'
 />";
        $retorno .= "<input type='submit' name='submit' value='".ucwords($accion)."' /
>";
        $retorno .= "</form>";

        return $retorno;
    }
    static public function guardarUsuario($nombre, $apellido)
    {
        $usuarioNuevo = new Usuario(NULL,$nombre, $apellido);
        return $usuarioNuevo->guardarUsuario();
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Usuario.php

```
<?php
require_once 'configuracion.php';
require_once PER . DIRECTORY_SEPARATOR . 'UsuarioPersistencia.php';

/**
 * Description of Usuario
 *
 * @author Pc
 */
class Usuario
{
    private $_id;
    private $_nombre;
    private $_apellido;

    public function __construct($id = "", $nombre = "", $apellido = "")
    {
        $this->_id = $id;
        $this->_nombre = $nombre;
        $this->_apellido = $apellido;
    }

    public function getId()
    {
        return $this->_id;
    }
    public function getNombre()
    {
        return $this->_nombre;
    }
    public function getApellido()
    {
        return $this->_apellido;
    }
    public static function getAll()
    {
        $usuarioPersistencia = new UsuarioPersistencia();
        $datos_array = $usuarioPersistencia->getAll();

        foreach($datos_array as $usuario_array){
            $id         = $usuario_array['id'];
            $nombre     = $usuario_array['nombre'];
            $apellido   = $usuario_array['apellido'];

            $retorno[] = new Usuario($id, $nombre, $apellido);
        }
        return $retorno;
    }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

```
    public static function load($id)
    {
        $usuarioPersistencia = new UsuarioPersistencia();
        $datos_array = $usuarioPersistencia->load($id);

        foreach($datos_array as $usuario_array){
            $usuario = new Usuario(
                $id,
                $usuario_array['nombre'],
                $usuario_array['apellido']
            );
        }
        return $usuario;
    }
    public function guardarUsuario()
    {
        $usuarioPersistencia = new UsuarioPersistencia();
        $guardo = $usuarioPersistencia->guardarUsuario(
            $this->_nombre,
            $this->_apellido
        );

        return $guardo;
    }
    public function modificarUsuario()
    {
        $usuarioPersistencia = new UsuarioPersistencia();
        $modificar = $usuarioPersistencia->modificarUsuario(
            $this->_id,
            $this->_nombre,
            $this->_apellido
        );

        return $modificar;
    }
    public function eliminarUsuario()
    {
        $usuarioPersistencia = new UsuarioPersistencia();
        $eliminar = $usuarioPersistencia->eliminarUsuario($this->_id);

        return $eliminar;
    }
    public function __toString()
    {
        return $this->_id." ".$this->_nombre." ".$this->_apellido;
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

UsuarioPersistencia.php

```
<?php
require_once 'configuracion.php';

require_once PER . DIRECTORY_SEPARATOR . 'BaseDeDatos.php';
require_once PER . DIRECTORY_SEPARATOR . 'MySQL.php';
require_once PER . DIRECTORY_SEPARATOR . 'Sql.php';

class UsuarioPersistencia
{
    public function getAll()
    {
        $bd = new BaseDeDatos(new MySQL());
        $sql = new Sql();

        $sql->addTable('usuarios');
        return $bd->ejecutar($sql);
    }

    public static function load($id)
    {
        $bd = new BaseDeDatos(new MySQL());
        $sql = new Sql();

        $sql->addTable('usuarios');
        $sql->addWhere("id = ".$id);

        return $bd->ejecutar($sql);
    }
    public function guardarUsuario($nombre, $apellido)
    {

        $bd = new BaseDeDatos(new MySQL());
        $sql = new Sql();

        $sql->addFuncion("insert");
        $sql->addTable("usuarios");
        $sql->addSelect("nombre");
        $sql->addSelect("apellido");
        $sql->addValue($nombre);
        $sql->addValue($apellido);

        return $bd->ejecutar($sql);
    }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

```
    public function modificarUsuario($id, $nombre, $apellido)
    {
        $bd = new BaseDeDatos (new MySQL());
        $sql = new SQL();

        $sql->addFuncion("update");
        $sql->addTable("usuarios");
        $sql->addSelect("nombre='".$nombre."'");
        $sql->addSelect("apellido='".$apellido."'");
        $sql->addWhere("id=".$id);

        return $bd->ejecutar($sql);
    }
    public function eliminarUsuario($id)
    {
        $bd = new BaseDeDatos (new MySQL());
        $sql = new SQL();

        $sql->addFuncion("delete");
        $sql->addTable("usuarios");
        $sql->addWhere("id=".$id);

        return $bd->ejecutar($sql);
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Resumen

Quiero felicitarte si llegaste hasta el final del libro y completaste todos los ejercicios.

Honestamente, el gusto de escribir todo este material fue mío y espero que sigamos en contacto a través del sistema para usuarios, cursos a distancia, libros o simplemente un comentario en el blog consultando alguna duda :-)

Baja el ejercicio completo con todo el código comentado

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

CAPÍTULO 21 - ANEXO: "MANEJO DE EXCEPCIONES"

Un tema no menos importante en la POO es el manejo de errores, algo que todos los lenguajes modernos soportan y dan la posibilidad de unificar la forma de manejarlos en el sistema.

Esta funcionalidad se agregar a partir de PHP5.

Manual oficial:

http://www.php.net/manual/es/language.exceptions.php

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Introducción

Una excepción es una situación anormal que ocurre durante la ejecución de un sistema que no está contemplada en el flujo esperado de nuestro código.

Por ejemplo, una parte de nuestro sistema tiene la responsabilidad de listar los usuarios actualmente existentes. Esa es la funcionalidad esperada y así lo implementamos. Necesariamente a bajo nivel vamos a requerir operaciones de persistencia, es decir, conectarnos a la base de datos, requerir los datos y desconectarnos. **Ese es el flujo normal y se espera que todo salga bien.**

Lamentablemente debemos contemplar todas las situaciones que "no son esperadas / no son normales" para que nuestro sistema sepa cómo proceder en caso de fallas:

- el servidor de base de datos está fuera de servicio,
- cambió la clave del usuario que usamos para conectarnos,
- no existe la tabla de usuarios,
- algún campo de la tabla cambió de nombre,
- etc.

En la programación tradicional, haríamos algo como esto (extrato del manual oficial):

```
<?php
$link = mysql_connect(
      'localhost',
      'mysql_user',
      'mysql_password'
);

if (!$link) {
    die(
      'Could not connect: '
      . mysql_error()
    );
}

echo 'Connected successfully';
mysql_close($link);
?>
```

Aquí solo estamos controlando que se pudo hacer la conexión a la base de datos, pero no los demás casos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Podríamos agregarle entonces:

```
<?php
$link = mysql_connect(
      'localhost',
      'mysql_user',
      'mysql_password'
);

if (!$link) {
    die(
      'Could not connect: '
      . mysql_error()
    );
}

echo 'Connected successfully';

$result = mysql_query("SELECT id,email FROM people WHERE id = '42'");

if (!$result) {
    echo 'Could not run query: ' . mysql_error();
    exit;
}
$row = mysql_fetch_row($result);

echo $row[0]; // 42
echo $row[1]; // the email value

mysql_close($link);

?>
```

Si somos observadores nos iremos dando cuenta que **el código para validar todas las situaciones anómales empieza a crecer y superar en extensión al verdadero código funcional**, donde muy probablemente entraremos a anidar “if/else/elseif” y/o “switchs”, y así seguirá creciendo en complejidad y aumentando la posibilidad de fallos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Básicamente cómo funcionan las excepciones

Existen dos claras zonas, la primera de "try" que se traduce cómo "intentar ejecutar el código que está entre llaves y que es posible que pueda fallar" y el "catch", el código previsto para tratar el fallo que pueda ocurrir.

```
try{

   /* Aquí va el código
   que podría fallar*/

}catch(Exception $e){

   /* Si alguna línea
   dentro del try falló,
   podremos tomar el
   control aquí */
}
```

Si quisiéramos emular una situación donde si falla simplemente envíe un mensaje en pantalla, podríamos hacer lo siguiente:

```
try{

   /* Aquí va el código
   que podría fallar */

}catch(Exception $e){

   echo $e;

}
```

En esta situación no logramos mucho más avance, ya que al fallar alguna rutina dentro del "try" simplemente pasa el controlo al "catch" y toda la información queda dentro de la instancia "$e" de tipo clase Exception.

Aclaración

Si vienes prestando atención, el **echo** $e retorna información porque la clase Exception implementa **el método toString()** con el mensaje básico de error.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Estructura interna de una clase Exception

La estructura de una clase Exception es (según el manual oficial):

```
class Exception
{
    protected $message = 'Unknown exception';   // exception message
    protected $code = 0;                        // user defined exception code
    protected $file;                            // source filename of exception
    protected $line;                            // source line of exception

    function __construct($message = null, $code = 0);

    final function getMessage();                // message of exception
    final function getCode();                   // code of exception
    final function getFile();                   // source filename
    final function getLine();                   // source line
    final function getTrace();                  // an array of the backtrace()
    final function getTraceAsString();          // formated string of trace

    /* Overrideable */
    function __toString();                      // formated string for display
}
```

Esto significa que podemos hacer uso de los métodos listados para poder, si así lo deseamos, generar un mensaje a medida para el usuario de acuerdo a nuestras necesidades.

Si hacemos un **echo $e;** obtendremos el mensaje estándar de la Excepción, que generalmente podría ser bastante extenso e informativo, tal vez ideal para ambientes de desarrollo pero no tanto para producción, ya que nuestra seguridad se podría compromenter al darle tantos detalles a un “extraño”.

Un ejemplo de un mensaje más breve y sin menos datos:

```
try{

   /* Aquí va el código que podría fallar*/

}catch(Exception $e){

   echo "Ocurrió un error inesperado, "
       ."por favor consultar a soporte técnico";
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Tal ver preferimos dar una referencia, un código para poder completar en un formulario de asistencia técnica.

```
try{

   /* Aquí va el código que podría fallar*/

}catch(Exception $e){

   echo "Ocurrió un error inesperado, "
      ."por favor consultar a soporte técnico "
      " (Código de Error ".$e->getCode().")";

}
```

Pero tal vez no queremos que el usuario sepa tanto, somos un poco más proactivos y preferimos dejarles un mensaje genérico y que se genere un registro en el log, o si es muy grave, nos envíe una notificación por email o hasta un SMS a nuestro celular.

```
try{

   /* Aquí va el código que podría fallar*/

}catch(Exception $e){

   echo "Ocurrió un error inesperado, "
      ."ya se envió una notificación al departamento de sistemas "
      " En un momento se estarán contactando con usted";

   Mail::send("sistemas@surforce.com", "Error:".$e );
}
```

Esto es lo básico de cómo responder a un error, pero casi de la misma forma que podríamos hacerlo con un if/else.

Ahora veremos las diferencias.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Importante: PHP no tiene excepciones por defecto

Una de las ventajas viene directamente por el paradigma POO, como las excepciones se representan con objetos, podemos extender la clase Exception y crear nuestras propias excepciones de acuerdo a nuestras necesidades.

En lenguajes como Java ya existen por defecto cientos de clases agrupadas por temas como base de datos, manejo de archivos, para problemas matemáticos, arrays, etc. Y lo más importante, **todo retorna por defecto una excepción, algo que no sucede en PHP5!**

¿Cuan grave es no tener Excepciones predefinidas y por defecto?

Que es muy normal que lo primero que probemos sea una sentencia de conexión a una base de datos (como vimos al principio de este capítulo) y que las excepciones no funcionen, dejandonos algo confusos.

Por lo tanto tenemos dos problemas:

- **En PHP5 debemos crearnos nuestras propias excepciones** a partir de la única clase que nos provee el lenguaje: Exception.
- **Todo lo que atrapemos debemos asegurarnos que retorne una Excepción**, y como el lenguaje no lo hace por defecto, debemos hacerlo a nivel de clases propias, validando internamente de la forma tradicional (if/else/switch), pero al subir de nivel el resto podrá hacer uso normal de los try/catch.

Sí, lamentablemente aún estamos en pañales con las excepciones,pero se espera que por lo menos en PHP7 se modifiquen todas las rutinas del lenguaje para que así lo hagan.

Zend Framework tiene Excepciones Predefinidas y por Defecto

Una de las ventajas de este framework es que todos sus componentes tiene manejo de excepciones, es decir, incorpora desde **Zend_Exception** (algo más completo que Exception estándar de PHP) y muchas más clases para cada situación, muy similar a Java.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Ejemplo funcional de excepciones en PHP5

Aquí veremos cómo hacerlas funcionar sin excepciones por defecto creando una clase genérica que tendrá todo el código del primer ejemplo donde retornaba los mensajes de error clásicos y los cambiaremos ahora por excepciones (comentaré las líneas que cambian y agrego a continuación las nuevas):

```
<?php
class BaseDeDatosEjemplo
{
    private $_link;

    public function __construct()
    {
        $this->_link = mysql_connect(
            'localhost',
            'mysql_user',
            'mysql_password'
        );

        if (!$link) {
            // die('Could not connect: ' . mysql_error());
            throw new Exception('Could not connect: ' . mysql_error());
        }
    }
    public function traerDatos()
    {
        $result = mysql_query("SELECT id,email FROM people WHERE id = '42'");

        if (!$result) {
            //echo 'Could not run query: ' . mysql_error();
            throw new Exception('Could not run query: ' . mysql_error());
            //exit;
        }
        $row = mysql_fetch_row($result);

        mysql_close($link);

        return $row;
    }
}
```

Se podría decir que el "throw" es similar a "return algo", lo que hace es "lanzar la excepción" a un nivel más arriba, donde se invocó originalmente la rutina, para que tomen el control de la situación anómala y procedan en consecuencia.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Por lo tanto, si todo el código anterior queremos "atraparlo" en un "try" para tenerlo controlado en caso de fallas, deberíamos hacer lo siguiente:

```
try{
    $bd = new BaseDeDatosEjemplo();
    $datos = $bd->traerDatos();

}catch(Exception $e){
    echo
      "Falla al recuperar "
      ."los datos";
}
```

Esta es la forma de trabajo más elemental, posteriormente habría que crear distintos tipos de Excepciones de acuerdo al tipo de falla, como por ejemplo:

```
try{

   /* Aquí va el código que podría fallar*/

}catch(DbException $e){
  /* Aquí va el código para responder
   a un error de Base de Datos */

}catch(Exception $e){
  /* Aquí va el código para responder
   a un error Genérico */
}
```

Importante: el orden de las excepciones

hay que tener en cuenta **el orden de las excepciones dentro del catch**, ya que debe ir primero la excepción más específica e ir bajando a la excepción más genérica. Si colocamos la excepción genérica primero, todos los fallos entrarán ahí y no a la excepción acorde al tipo de error.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Beneficios de las Excepciones

- **Permiten separar el código de manejo de errores del código que debe cumplir con los requerimientos de funcionalidad de la aplicación**

- **Permite un manejo homogéneo de los errores:** evitará que tengamos distintas estructuras para contener los errores, como ser: en algunos casos una cascada de if, en otros un switch y tal vez en otros algún objetos de manejo de errores.

- **No solo transfiere el control del sistema de un lugar a otro, sino que también trasmite información sobre la situación anómala:** como unidad todas las excepciones manejan objetos de tipo Excepción y como todo objeto, tendrá atributos y métodos relacionados con el manejo de errores.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

En Resumen

Las excepciones son herramientas que cuentan todos los lenguajes POO modernos y no se discute su utilidad. Es fundamental entender los conceptos base, qué es una Excepción, que no todo debería considerarse como tal y solo las "situaciones anómalas / no esperadas" de nuestro sistema.

Como desarrolladores siempre tendremos dos caminos para decidir: si ocurre una excepción, es responsabilidad de esa parte del código resolverla o deberá "relanzarla" al nivel superior para que este se encargue de hacer algo al respecto.

Es muy habitual que si tenemos un sistema de 3 capas y falla en la capa de persistencia, esta relance la excepción al dominio y este a la capa de presentación para que se transforme en una ventana de error que será entregada elegantemente al usuario como:

"Ha ocurrido un error al intentar guardar los datos, por favor pruebe nuevamente dentro de unos minutos" ***;-)***

Quieres saber si hay alguna actualización de este capítulo?

Ingresa a http://usuarios.surforce.com

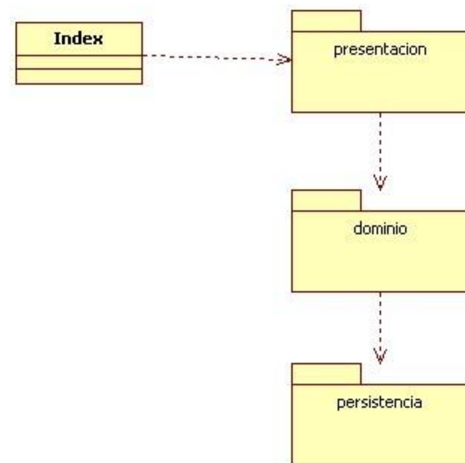
Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Capítulo 22 - Cierre del libro y reflexiones finales

¿No pasa que en algunas películas, cuando parece que todo terminó, al final se muestra una secuencia que nos adelanta el inicio de una continuación? **¿la revelación de una nueva trama?**

Bien, aquí viene ;-)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/



Aquí hay algo que está mal.. ¿no se dieron cuenta?

Si dijimos durante casi la mitad del taller que **lo más importante es el "dominio"** y que las flechas evidencian que si hay una relación de A -> B significa que todo cambio de B afecta a A, qué pasa con este modelo tradicional de "3 capas"?

Index -> presentación -> dominio -> persistencia

Mmmm… todo bien que presentación dependa de dominio que es lo más importante, si este cambia, es lógico que cambie las pantallas… pero está bien que dominio dependa de la persistencia?! eh?! ah?!

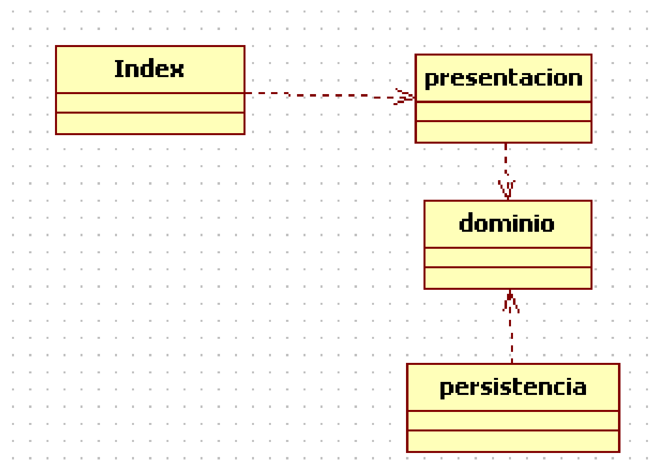
¿Entonces, todo lo que aprendimos está mal? ¿equivocado? ¿erróneo?

¡AGHHHH!

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

...No del todo! ;-)

Lo que significa que el diseño tradicional de "3 capas" con el mismo sentido de flechas desde la "presentación" pasando por "dominio" y terminando en "persistencia", por más aceptado que esté, el diseño no cumple con todos los conceptos que vimos, por lo tanto, **el verdadero diseño de 3 capas bien implementado debería ser como en el diagrama de la derecha** (*"persistencia apuntando a dominio"*).



Es verdaderamente interesante el mundo más allá de la ***Programación Orientada a Objetos*** y que entra en lo que es el ***Análisis y Diseño Orientado a Objetos***, un nivel más, *el arte de los druidas*, el camino del *"Arquitecto de la Matrix",* poder detectar la estabilidad de los paquetes, los principios que afectan a todos los diseños, cómo mejorar nuestro diseño o detectar sus fallas, que... vendrá en **un futuro no muy lejano!**

Ahora sí, hasta el próximo libro, curso, taller o post en el blog! **;-)**

FIN.

Confirma que estás leyendo la última versión de este libro

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Enrique Place

http://phpsenior.blogspot.com

http://enriqueplace.blogspot.com

http://www.surforce.com

¿Qué te pareció el libro? ¡Envíame tus comentarios!

Ingresa a http://usuarios.surforce.com

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Anexo I: "Qué es lo nuevo en PHP5?"

Este anexo está basado en el artículo What's new in PHP5 (18/3/2004) publicado ***por Zend Developer Zone***, al cual le agrego un poco más de información y comentarios extras sobre mi opinión de cada uno de ellos.

Veamos un resumen de "lo nuevo" en PHP5.

> ***"La mejor manera de estar preparados para el futuro es inventarlo" (John Sculley)***

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Características del Lenguaje

PHP5 es la versión que hace un salto importante con respecto a la POO, incorporando funcionalidades que cualquier lenguaje OO necesita como mínimo (algo que PHP4 carecía completamente). Así que la mayoría de lo que veremos a continuación no son más que "una puesta al día" de PHP con respecto a los demás lenguajes POO (antes no se le podía decir que lo era por carecer de todo lo que veremos).

1.Modificadores de acceso "public/private/protected"

Se incorpora el uso de modificadores de acceso "public / private / protected" para atributos y métodos.

Definir el **"alcance o visibilidad" (scope)** es importante para poder aplicar el *"principio de ocultación"*, es decir, proteger al objeto del exterior y solo permitir acceso a los datos y comportamientos que nosotros especifiquemos. **PHP4 carecía completamente de esta posibilidad y antes "todo era público"**.

Aunque por defecto cualquier método que no diga nada al principio de su firma signifca que es "public", para unificar el criterio y claridad de la codificación **se sugiere que todos los método públicos inicien siempre con el "modificador de acceso" correspondiente**.

```php
<?php
class MyClass
{
    private $_id = 18;

    public function getId()
    {
        return $this->_id;
    }
    private function _calcularValorInternamente()
    {
        /* código */
    }
    protected function usarValor()
    {
        $this->_calcularValorInternamente();
    }
    public function __toString()
    {
        return $this->_id;
    }
}
```

¡Dile "no" a los atributos públicos!

Aunque *"tecnicamente lo permita"*, esto no significa que se deban usar *"atributos públicos"*.

Recuerda, por defecto en POO se considera que todos los atributos deben ser "no públicos" y solo en algunos casos muy especiales (0.001 %) se justificaría su uso, pero serían contextos muy "raros" (tal vez en el desarrollo de un generador de código o un framework, pero no en un sistema desarrollado típicamente en POO).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

2. El método reservado __construct()

En PHP4 para definir un constructor había que escribir un método con el mismo nombre de la clase (al igual que lo hace Java)

```
1  <?php
2  /* PHP4 */
3  class MiClase
4  {
5      public function MiClase()
6      {
7          echo "dentro del constructor";
8      }
9  }
```

PHP5 incorpora ahora un método exclusivo, a través del métod reservado **__construct**

```
1  <?php
2  /* PHP5 */
3  class MiClase
4  {
5      public function __construct()
6      {
7          echo "dentro del constructor";
8      }
9  }
```

3. El método reservado __destructor()

De la misma forma que el constructor, se agrega un método reservado que permite agregar funcionalidad cuando el objeto se destruya (por ejemplo, desconectarnos de una base de datos en el momento que se haga un **unset** de la instancia).

```
1  <?php
2  /* PHP5 */
3  class MiClase
4  {
5      public function __destruct()
6      {
7          echo "dentro del constructor";
8      }
9  }
10
11 $miClase = new MiClase();
12
13 unset($miClase);
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

4. Interfaces

Se incorpora por primera vez la posibilidad de crear interfaces y poder agrupar clases *"que hacen lo mismo"* o *"que tienen operaciones comunes"*.

```
1  <?php
2
3  interface Mostrable
4  {
5      public function mostrar();
6  }
7
8  class Circulo implements Mostrable
9  {
10     public function mostrar()
11     {
12         print "mostrando el círculo";
13     }
14 }
```

A diferencia de la herencia, se pueden implementar varias interfaces, por lo que podemos llegar a tener una clase que herede de una clase y que además implemente varias interfaces:

```
1  <?php
2
3  interface Mostrable
4  {
5      public function mostrar();
6  }
7
8  class Circulo extends Figura implements Mostrable, Imprimible
9  {
10     public function mostrar()
11     {
12         print "mostrando el círculo";
13     }
14 }
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

5. El operador "instance of"

Se deja de usar el operador de PHP4 is_a() y se agrega uno nuevo más especializado para objetos: "es instancia de":

```
if ($obj instance of Circle) {
    print '$obj is a Circle';
}
```

Aquí estamos preguntando si la instancia "obj" es de tipo "la clase Círculo".

¡No uses el "instanceof"!

Aunque *"tecnicamente exista y pueda usarse ",* no se recomienda estar preguntando a las instancias cual es su clase, ya que **rompemos los diseños genéricos que pueden hacer uso del polimorfismo (estrategia base del DOO).**

Cuando trabajemos con varios objetos debemos manipularlos "genéricamente" y cada objeto debe saber cómo comportarse. **Si por cada objeto vamos a preguntar de que tipo es y en base a eso hacer algo, estamos atando el comportamiento de nuestro código** y tendremos una cadena de if's/switchs para contemplar cada caso (tenemos un nuevo "foco de cambio", ya que necesitaremos seguir preguntando por cada nuevo tipo que se cree).

6. Operador "final" para los métodos

Final es otra herramienta que nos permite reforzar el diseño de nuestra clase. En este caso podemos definir qué métodos que otra clase vaya a heredar no pueden ser modificados a través de la "sobreescritura" de los mismos (los métodos deben usarse tal cual se reciben de su padre).

```
1  <?php
2
3  class MiClase
4  {
5      final function getBaseClassName()
6      {
7          return __CLASS__;
8      }
9  }
```

En este ejemplo podemos decir que **diseñamos un método (por lo simple y concreto) que no debería existir ninguna situación donde necesite que sea modificado**, por lo tanto "lo aseguramos" para que nadie que herede nuestra clase pueda modificar su comportamiento (ya que sería muy raro).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

7. Operador "final" para la clase

Este operador también se aplica a las clases, impidiendo que una clase se pueda heredar, por lo que estamos entonces obligando a usar la clase sin posibilidad de crear otra en base a la original.

```
final class FinalClass

{
}
class BogusClass extends FinalClass

{
}
```

En este caso el sistema dará un error grave porque se intentó heredar de una clase que está definida como "final".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

8. Método reservado __clone para clonado de objetos

Desde que PHP5 empieza a trabajar por referencia y no por valor (como hacía por defecto PHP4), **toda asignación con objetos representa una copia de la referencia a un mismo objeto**, **pero nunca se crea otro objeto duplicado.** Si necesitáramos hacerlo (en casos muy especiales), podremos usar el comando clone.

```
1  <?php
2  class Usuario
3  {
4
5  }
6  $usuario = new Usuario();
7  $usuario_copia = $usuario;
8
9  var_dump($usuario);
10 var_dump($usuario_copia);
11
12 /* la salida será la misma, no existe copia, es el mismo
13  * objeto pero con dos referencias (ambos objetos tienen
14  * el id #1)
15  *
16  * object(Usuario)#1 (0) { }
17  * object(Usuario)#1 (0) { }
18  */
19
20 $usuario_clonado = clone($usuario);
21
22 var_dump($usuario_clonado);
23
24 /* Si volvemos a ejecutar el scripts veremos que
25  * la tercer línea tiene un id distinto (#2), por lo tanto
26  * recién ahora tenemos un objeto "copia"
27  *
28  * object(Usuario)#1 (0) { }
29  * object(Usuario)#1 (0) { }
30  * object(Usuario)#2 (0) { }
31  */
```

Pero, **tal vez queramos especificar cómo debería clonarse el objeto**, por lo que podemos definir en el comportamiento interno del mismo a través del método reservado __clone() todas las operaciones que necesitemos.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

```php
<?php

class Usuario
{
    private $_clave;

    public function __construct($clave)
    {
        $this->_clave = $clave;
    }
    public function getClave()
    {
        return $this->_clave;
    }
    public function __clone()
    {
       unset($this->_clave);
    }
}
$usuario = new Usuario('mi clave');

$usuario_clonado = clone($usuario);

echo $usuario_clonado->getClave();
```

En este ejemplo podemos ver cómo se usa el método __clone, en el cual agregaremos que si alguien quiere clonar nuestro objeto no tenga el valor de la clave del objeto original.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

9. Atributos Constantes para las clases

Ahora se incorpora la posibilidad de incluir atributos constantes en las clases. Hay que tener en cuenta que como toda constante, estas son siempre públicas, y no podemos desde la clase esconderlas (su ámbito siempre será la clase).

```
1  <?php
2  class Bd
3  {
4      const SUCCESS = "Success";
5      const FAILURE = "Failure";
6  
7      const CONEXION_PERSISTENTE = 1;
8      const CONEXION_NO_PERSISTENTE = 1;
9  }
10 /* Ejemplo de acceso desde la clase, sin instancia */
11 
12 echo Bd::SUCCESS;
13 
14 /* Ejemplo de uso para documentar los parámetros de
15  * una clase, evitando "números mágicos"
16  */
17 
18 $bd = new Bd(Bd::CONEXION_PERSISTENTE);
19
```

En este ejemplo podemos observar el uso de las constantes como forma de documentar y evitar los "números mágicos" (técnica de Refactoring), donde para conocer la lista de parámetros (sin importar saber de memoria sus valores) se lo podemos pedir a la misma clase consultando sus constantes (que son públicas y su propia descripción auto-documentan su uso).

Esta práctica es ampliamente usada en muchos lenguajes como Java (tanto para clases como para interfaces) y **se recomienda adoptar siempre que definamos parámetros o valores en nuestro sistema**.

Nota: las constantes se consideran "elementos de clase" (ver más adelante el apartado sobre "métodos estáticos"), por lo tanto dentro de la clase deberías usar self::CONSTANTE (igual puedes usar Clase::CONSTANTE).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

10. Miembros estáticos o de clase (static)

Las clases pueden incluir elementos "estáticos" (también denominados "elementos de clase"), accesibles a través de la clase y no de la instancia.

El ejemplo más común es crear una clase Usuario que cada vez que se cree una instancia, la clase sepa cual es el último número de id y asignar dinámicamente un número más para el nuevo usuario.

El atributo debería ser "de clase / estático", ya que su valor, en oposición a los atributos, son compartidos por todas las instancias de la misma clase.

```
1  <?php
2  class Usuario
3  {
4      static private $_ultimoId = 0;
5
6      private $_id;
7
8      public function __construct()
9      {
10         self::$_ultimoId += 1;
11
12         $this->_id = self::$_ultimoId;
13     }
14     public function __toString()
15     {
16         return (string)$this->_id;
17     }
18 }
19
20 $usuario1 = new Usuario();
21 $usuario2 = new Usuario();
22
23 echo $usuario1;
24 echo " ";
25 echo $usuario2;
```

En este ejemplo podemos ver cómo a través del *"atributo de clase"* se mantiene la información entre instancias de la misma clase. Hay que tener en cuenta que este ejemplo no es del todo útil más allá de explicar cómo funciona, ya que cada vez que reiniciemos nuestra página, el contador empezará nuevamente de cero (para evitarlo habría que persistir esta información de alguna forma).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

Otro uso común de los miembros estáticos es en el patrón Singleton, el cual está pensado para retornar siempre la misma instancia de un objeto:

```
class Singleton

{
    static private $_instance = NULL;

    private function __construct() {}

    static public function getInstance()

    {
        if (self::$_instance == NULL) {
            self::$_instance = new Singleton();
        }
        return self::$_instance;
    }
}
```


Licencia: http://creativecommons.org/licenses/by-nc/3.0/

11. Métodos estáticos o de clase (static)

De la misma forma que ahora se pueden crear atributos de clase, **también se pueden crear métodos de clase o estáticos**. En esencia la lógica es la misma, son métodos que se comparten entre elementos de la misma clase y no son "de instancia" (no requieren una instancia para poder usarse).

Un uso que se le da a los miembros de clase es **crear una serie de métodos que no necesariamente están afectados a una instancia** y que bien se pueden usar como una "librería" (conjunto de funcionalidades agrupadas bajo una clase), como sucede con clases bases en Java. Una clase String en Java tendrá todo un conjunto de métodos que no necesariamente se aplican solo a una instancia, y que pueden invocarse llamando a la clase y al método correspondiente.

Imaginemos que queremos que PHP imite el comportamiento de Java, por lo tanto necesitaríamos crear clases base como Integer, String, Date, etc (que actualmente no existen), por lo que deberíamos buscar en el manual de PHP todas las funciones aisladas entre sí que tratan un mismo tema y agruparlos bajo la misma clase (espero que en algún futuro no muy lejano suceda en PHP). Así, con **todas estas funcionalidades deberíamos poder aplicarlas tanto en una instancia o a través de una clase**, para tener la libertad de no necesitar crear una instancia cada vez que queramos usar una funcionalidad que está aislada del contexto de la instancia.

Por ejemplo, PHP tiene una función strtolower(), si quisiéramos ordenarlo en un entorno 100% Orientado a Objetos, podríamos decir que debería ser un método de una clase String, y deberíamos poder usarlo en cualquier ambiente, tanto bajo una instancia como sin ella. Si creamos un método común, solo lo podremos usar con una instancia:

```
1  <?php
2  class String
3  {
4      public function strtolower($cadena)
5      {
6          return strtolower($cadena);
7      }
8  }
9
10 $string = new String();
11
12 echo $string->strtolower('PASAR A MINÚSCULAS');
```

Bien podríamos cambiar el diseño y decir que no necesitamos crear constantemente una instancia que luego no se va a usar, por lo tanto cambiemos el método a "de clase".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

```php
<?php
class String
{
    public static function strtolower($cadena)
    {
        return strtolower($cadena);
    }
}

$string = new String();

echo $string->strtolower('PASAR A MINÚSCULAS');

echo String::strtolower('PASAR A MINÚSCULAS');
```

De ahora en más podremos usarla de dos formas.

Hay que tener en cuenta que si llamamos al método internamente, **debemos cambiar el $this que se aplica a instancias por el método self que se aplica a elementos de clase**.

```php
<?php
class String
{
    public static function strtolower($cadena)
    {
        return strtolower($cadena);
    }
    public function __toString()
    {
        return self::strtolower('soy cadena');
    }
}

$string = new String();

echo $string;
```

Otro ejemplo de uso muy tradicional es la típica clase *Index*, donde no necesariamente requiere que creemos una instancia, ya que solo la usaremos para iniciar la ejecución de un sistema, por lo que acceder a un método es suficiente.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

12. Clases Abstractas

"abstract" es una forma de decir que una clase *"no se puede instanciar".*

Por ejemplo, si tenemos una clase que es demasiado "genérica" para realmente necesitar que se instancie, la aseguramos colocando un "abstract", por lo tanto solo servirá como "modelo" para poder hacer "herencia".

Ej, **la clase Persona**, en la vida real es muy poco probable que necesitemos instanciarla, pero si contamos con clases de tipo *Usuario* que heredan características de *Persona*, sí les servirá de modelo.

El otro caso, es la clase Index, que no se ejecuta como una instancia, lo hace como una clase, por lo tanto para hacer más robusto el diseño le agregamos "abstract" para que nadie la use de otra forma.

```
abstract class MyBaseClass
```

13. Métodos abstractos

Un método abstracto obliga a una clase que hereda de la clase que lo contiene (al método abstracto) a definir el contenido de ese método. Una clase que tiene métodos abstractos debe definirse como clase abstracta. De la misma forma que podemos decir que una clase abstracta puede ser usada como una clase "modelo" para que otra herede de ella, podemos decir que un método abstracto sirve de modelo para que una clase que herede tenga que implementar el método definido en la clase padre.

```
abstract class MyBaseClass

{
    abstract function display();
}
```

¿clases abstractas == interfaces?

Si hemos prestado atención, **con las clases y métodos abstractos podemos "simular" el comportamiento de las interfaces**, al crear un "contrato de implementación". Pero **no son lo mismo**, recordar que la herencia agrupa "elementos del mismo tipo / relación de parentesco" y las interfaces "elementos que hacen lo mismo sin importar si tienen o no relación de parentesco"

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

14. Validación de Tipo a través de Clases (type hints)

PHP desde sus orígenes es un lenguaje *"dinámicamente tipado"*, lo que hace que cada variable defina su tipo según el valor que se le asigne, pero en sí no necesitamos como Java que es obligatorio definir el tipo antes de usarlo. Con el tiempo nos dimos cuenta que *"tanta libertad"* (no manejar tipos) puede no ser tan buena idea, por lo que ahora en más **se habilita para los objetos la posibilidad de verificar "el tipo" del parámetro que estamos recibiendo**, permitiendo con esto hacer diseños más robustos donde *"no todo es tan dinámico"*, ya que ahora podemos aplicar un poco más de control. Si no se cumple con el tipo especificado, dará un error.

```
function expectsMyClass(MyClass $obj) {

}
```

Un detalle importante a tener en cuenta es que *la validación es "mecánica"*, es decir, cuando recibe el objeto pregunta cual es su tipo (con qué clase se creó o cuales son sus padres) y luego comparará exactamente con la cadena de texto que agregamos a la izquierda (en este caso MyClass).

De aquí se desprenden dos cosas:

1. **No es técnicamente necesario incluir la clase de filtro**, por ejemplo, para usar el filtro MyClass no hace falta hacer un require_once de MyClass, ya que con el nombre solo ya le alcanza para hacer la comprobación.

2. **Cuando se hereda, una clase es del tipo base (su clase) y la de su padre**, por lo tanto si soy un Usuario y además heredé de Persona, la validación puede ser filtro(Persona $persona) y ambas serán aceptadas, porque ambos objetos son "personas".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

15.Soporte a invocaciones anidadas de objetos retornados

En PHP4 estábamos obligados a hacer lo siguiente:

```
$dummy = $obj->method();
$dummy->method2();
```

Ahora en PHP 5 podemos hacer las invocaciones en cadena de la siguiente forma:

```
$obj->method()->method2();
```

Es una técnica muy utilizada y se le llama "Interfaces fluidas" (fluent interface), y para aplicarlas deberíamos, a cada método que queremos que se pueda anidar con la ejecución de otro, **hacer que retorne la instancia actual con "this"**. De esta forma podemos anidarlos, ya que la salida de uno es la misma instancia que se va a aplicar en siguiente método.

```
1  <?php
2
3  class Usuario
4  {
5      private $_clave;
6      private $_habilitado = false;
7
8      public function setClave($clave)
9      {
10         $this->_clave = $clave;
11         return $this;
12     }
13     public function setHabilitado($valor)
14     {
15         $this->_habilitado = $valor;
16         return $this;
17     }
18 }
19
20 $usuario = new Usuario();
21
22 $usuario->setClave('nueva')->setHabilitado(true);
23
24 var_dump($usuario);
25
26 // Retornará
27 //
28 // object(Usuario)#1 (2) {
29 //     ["_clave:private"]=> string(5) "nueva"
30 //     ["_habilitado:private"]=> bool(true) }
31
```

Retorno la misma instancia del objeto donde estoy parado dentro de un método que normalmente no retornaría nada.

Idem.

Métodos anidados, siempre aplicados a la instancia "usuario".

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

¡No abuses de las interfaces fluidas!

Aunque esta técnica de simplificación es muy usada (Java, frameworks de javascripts y PHP, etc) y puede sernos de utilidad, **el abuso de este tipo de codificación puede "oscurecer" el entendimiento del código**, ya que si anidamos 10 métodos en una sola línea, al final, no sabremos qué hace.

Sugerencias: implemerlo en métodos simples, donde antes no retornaban nada, aprovecha y retorna un this y solo lo usas cuando realmente lo necesites y veas que simplifique. En lo posible anida siempre el mismo objeto. No pases de 3 o 4 anidaciones por línea, y trata de usarlo en lugares puntuales, no hagas un sistema que predomine esta codificación o estarás condenado a pasar el resto de tu vida en el infierno de los programadores! (según me han comentado, te tienen 24 hs al día programando POO con COBOL ☺)

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

16. Iteradores

PHP5 permite ahora que las colecciones de una clase puedan ser iteradas a través de la misma clase con solo cumplir con una interface, así, con un simple foreach podremos recorrer los valores de un objeto:

```php
<?php
class Usuario implements Iterator
{
    private $_items = array(1,2,3,4);

    public function rewind()
    {
        reset($this->_items);
    }
    public function current()
    {
        return current($this->_items);
    }
    public function key()
    {
        return key($this->_items);
    }
    public function next()
    {
        return next($this->_items);
    }
    public function valid()
    {
        return $this->current() !== false;
    }
}

$usuario = new Usuario();

foreach($usuario as $id){
    echo $id;
}
```

No es algo que necesitemos hacer regularmente, pero es importante saber que si nuestra clase solo esconde una colección, tal vez esta sea una forma más simple de acceder a la misma, o al revés, darle a una colección de objetos más funcionalidad que un simple array (observar que la interfaz define muchos comportamientos).

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

17. __autoload()

Es muy común que debamos incluir al principio de nuestra clase toda una serie de require_once de clases que necesitamos para trabajar. **PHP5 ahora incorpora una función que permite definir un simple algoritmo de búsqueda de clases** que se ejecutará siempre que nuestro fuente intente hacer un "new" de una clase (automáticamente la función recibirá por parámetros el nombre de la clase que intentamos instanciar, el resto lo definimos nosotros).

```
function __autoload($class_name) {
    require_once($class_name . "php");
}

$obj  = new MyClass1();
$obj2 = new MyClass2();
```

No siempre es tan útil como parece

Aunque a veces hacer "todo dinámico" no siempre es bueno, menos para la auto-documentación del código (ya que es útil saber claramente que tiene que requerir nuestra clase), cuando las estructuras de directorios son complicadas, esta función puede no sernos útil, o hasta generar una pequeña "sobrecarga" si por cada vez que intenta encontrar una clase tiene que buscarla de forma recursiva dentro del directorio de nuestra aplicación.

Para usar, pero en situaciones concretas y muy simples.

Nota final del Anexo

En sucesivas versiones se irá ampliando esta sección con más comentarios sobre las nuevas características de PHP5 como lenguaje POO.

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

ANEXO II: RECOPILACIÓN DE FRASES

"Los programas deben ser escritos para que la gente los lea (entienda) y solo incidentalmente para que los ejecuten las maquinas"

Abelson / Sussman

Si hubiese preguntado a la gente qué es lo que quería me habrían respondido que caballos más rápidos - *Henry Ford*

(se puede aplicar cuando decimos "y por qué no le preguntamos al usuario exactamente qué quiere?")

"Pregunta: ¿Cómo se atrasa un año un proyecto grande de software? Respuesta: Un día a la vez."

Fred Brooks

"Hacer un programa a partir de unas especificaciones y caminar sobre el agua son cosas muy simples, siempre y cuando ambas estén congeladas."

"una de cada 10 personas nace con un talento innato para programar, para entenderse con las computadoras de una manera mágica e increíble.... lamentablemente los otros nueve creen tener ese talento y por eso tenemos carreras de ingeniería de software y libros como este"

"Un sistema exitoso se mide por su cantidad de usuarios, no por su supuesta calidad"

"Si quieres ganar dinero programando, solo escucha a alguien desesperado con un gran problema, necesitado por una solución y con dinero"

"Si quieres ser pobre programando, escucha a todos los que te pidan un PEQUEÑO PROGRAMITA, que es MUY FACIL de hacer"

"La historia de la programación es una carrera constante hacia mayores niveles de abstracción"

"La programación puede ser divertida, al igual que la criptografía; sin embargo, ambas no deberían combinarse"

"El software es como la entropía: difícil de atrapar, no pesa, y cumple la Segunda Ley de la Termodinámica, es decir, tiende a incrementarse"
-- *Norman Augustine*

"La imaginación es más importante que el conocimiento. El conocimiento es limitado, mientras que la imaginación no"
-- *Albert Einstein*

"Una de las cosas más fascinantes de los programadores es que no puedes saber si están trabajando o no sólo con mirarlos. A menudo están sentados aparentemente tomando café, chismorreando o mirando a las nubes. Sin embargo, es posible que estén poniendo en orden todas las ideas individuales y sin relación que pululan por su mente"
-- *Charles M. Strauss*

"Comentar el código es como limpiar el cuarto de baño; nadie quiere hacerlo, pero el resultado es siempre una experiencia más agradable para uno mismo y sus invitados"
-- *Ryan Campbell*

"Está bien investigar y resolver misteriosos asesinatos, pero no deberías necesitar hacerlo con el código. Simplemente deberías poder leerlo"
-- *Steve McConnell*

"Si automatizas un procedimiento desastroso, obtienes un procedimiento desastroso automatizado"
-- *Rod Michael*

A lo cual yo diría dado el fanatismo de los programadores novatos por priorizar "optimización de código" a funcionalidad de código

"Si optimizas un procedimiento desastroso, obtienes un procedimiento desastroso optimizado"
-- *Rod Michael*

"Ley de Alzheimer de la programación: si lees un código que escribiste hace más de dos semanas es como si lo vieras por primera vez"
-- *Via Dan Hurvitz*

"La simplicidad llevada al extremo se convierte en elegancia"
-- *Jon Franklin*

Licencia: http://creativecommons.org/licenses/by-nc/3.0/

"Todo el mundo sabe el peligro de la optimización prematura. Pienso que deberíamos estar igualmente preocupados con el diseño prematuro, es decir, el hecho de diseñar demasiado pronto lo que un programa debería hacer"
-- *Paul Graham*

(haciendo referencia al desarrollo evolutivo)

"Por norma, los sistemas software no funcionan bien hasta que han sido utilizados y han fallado repetidamente en entornos reales"
-- *Dave Parnas*

"Cuando se está depurando, el programador novato introduce código correctivo; el experto elimina el código defectuoso"
-- *Richard Pattis*

"La mayoría de ustedes están familiarizados con las virtudes del programador. Son tres, por supuesto: pereza, impaciencia y orgullo desmedido"
-- *Larry Wall*

"El buen código es su mejor documentación"
-- *Steve McConnell*

"Cualquier código tuyo que no hayas mirado en los últimos seis meses o más es como si lo hubiese escrito otro"
-- *Eagleson's Law*

"El primer 90% del código corresponde al primer 90% del tiempo de desarrollo. El 10% restante corresponde al otro 90% del desarrollo"
-- *Tom Cargill*

"Depurar es al menos dos veces más duro que escribir el código por primera vez. Por tanto, si tu escribes el código de la forma más inteligente posible no serás, por definición, lo suficientemente inteligente para depurarlo"
-- *Brian Kernighan*

Fuente:

- **101 citas célebres del mundo de la informática**
- **Otras 101 citas célebres del mundo de la informática**

Licencia: http://creativecommons.org/licenses/by-nc/3.0/