**INTRODUCCIÓN**

Uff... Otro blog de un tipo que cree que tiene la verdad sobre la programación. El verdadero dueño del [*Tao*](http://www.variablenotfound.com/2011/10/el-tao-de-la-programacion.html).  
Bueno, no es mi intención.  
Como todo blog, simplemente pretendo volcar acá, en la medida de lo posible, mi opinión sobre algunos temas que atañen al diseño y desarrollo de software.  
Trataré sobre metodologías ágiles y otras yerbas visto siempre desde la óptica de los principios [S.O.L.I.D.](http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29)  
Espero que esto nos sirva a todos para tratar de mejorar día a día en esta profesión que nos une.  
Y como el debate amplía las ideas, pues eso, que todos podamos aportar opiniones y sacar cada quien su propia conclusión.  
Nos leemos, hasta la próxima entrada.

## [POO vs POP](http://programacionsolida.com.ar/2011/11/poo-vs-pop.html)

Este es un dilema que se nos presenta a diario cuando se requiere extender la funcionalidad de una clase o modificar el diseño para agregar funcionalidad. Para muchos, el tiempo apremia y lo más evidente es resolver las cosas de una manera que, a priori, parece la más sencilla: un parche (\*).  
Pero cuidado, resulta muy fácil caer en una trampa que nos armamos nosotros mismos. Muchas veces, la solución más sencilla en tiempo, resulta una contra,  ya que la misma, no es escalable y el tiempo que no gastamos hoy haciendo un refactoring “complejo”, lo gastamos más adelante (multiplicado por N) debido a que esa lógica ya esta muy arraigada en la aplicación.  
Un caso común, resulta del agregado de métodos o propiedades en clases a las que no les corresponde esa responsabilidad. Esto puede pasar por varios motivos:

* **Desconocimiento del framework propio**: Se desconoce la existencia de otra clase que ya posee esa responsabilidad.
* **Falta de tiempo**: Para no buscar en todo el framework si ya existe una clase a quién agregarle esta funcionalidad, se agrega en otra que podría soportarla.
* **Poco tiempo para refactorizar**: agregarle la funcionalidad a una clase que no tengo al alcance en tiempo de ejecución, desemboca en la creación de un nuevo servicio para poder utilizarla. Pero como llevar mucho tiempo hacerlo, se descarta la idea.
* **Por simple inexperiencia de diseño**: por falta de experiencia desarrollando objetos, no nos damos cuenta que lo que estamos haciendo, simplemente está “mal”. (Aquellos que venimos de programar estructurado, se nos complica un poco más pensar de manera abstracta en objetos y responsabilidades)
* **U otros casos más que ahora no se me ocurren.**

Esto, multiplicado por la cantidad de personas programando en un equipo de esta misma manera, provoca que luego, introducir un simple cambio, lleve mucho más tiempo del esperado porque se rompen más funcionalidades de las que se arreglan.  
Ahora, si este cambio es temporal, para salir del paso. Pues entonces debe tomarse como tal y arreglarlo (o cambiarlo) ni bien haya pasado la tormenta y evitar de esta manera que la deuda técnica crezca (que programando sólo de esta manera, podríamos empeñar hasta los anillos de la abuela).  
Tengamos muy en cuenta que ésta deuda técnica va totalmente en contra nuestra, los desarrolladores, no solo de la empresa. ¿Quién prefiere romperse la cabeza durante una semana para introducir un cambio en lugar de poder hacerlo en un par de horas? Yo creo que nadie. Entonces, se debe trabajar en un diseño que permita introducir cambios fácilmente y para ahorrar, a nosotros mismos y a nuestros compañeros de equipo, el mal humor y los dolores de cabeza. Porque al fin y al cabo, el cerebro es nuestra herramienta de trabajo y de esta manera, lo maltratamos.  
Se debe erradicar a los parches como una solución permanente y que sean solo eso, un parche provisorio, que va a quitarse a la brevedad.  
Se debe priorizar la calidad del código por sobre la cantidad y rapidez  
  
(\*)Entiéndase en este caso por parche a aquella funcionalidad que se introduce al diseño para ahorrar tiempo.  
  
Ejemplos:

* Poner un IF para bifurcar la ejecución del código en lugar de crear dos objetos diferentes.
* Mezclar responsabilidades en una misma clase.
* Repetir código en lugar de extraerlo de donde está y crear una nueva clase.

## [Principio K.I.S.S. ( parte I )](http://programacionsolida.com.ar/2011/11/principio-kiss-parte-i.html)

[K.I.S.S.](http://es.wikipedia.org/wiki/Principio_KISS) es la abreviatura de Keep It Simple, Stupid (mantenlo simple, estúpido).  
Un problema común entre los ingenieros y desarrolladores de software hoy en día, es que tendemos a complicar más los problemas.  
Normalmente, cuando nos enfrentamos a un problema, dividimos en trozos más pequeños que pensamos que entendemos y luego tratamos de implementar la solución en el código. Yo diría que 8 o 9 de cada 10 desarrolladores cometemos el error de no descomponer el problema en trozos suficientemente pequeños o suficientementes comprensibles. Esto se traduce en implementaciones muy complejas de los problemas más simples.  
Otro efecto secundario es el llamado: código spagetthi. Algo que creíamos que sólo en BASIC podía hacerse con sus sentencias GOTO. Pero no, en lenguajes modernos como C# o Java, se traduce en clases de 500-1000 líneas de código (decenas de métodos con decenas de líneas).  
Este desorden de código es el resultado de la realización de casos de excepción, por parte nuestra, a la solución original mientras que se está escribiendo el código.  
Dichos casos se hubieran resuelto si hubiéramos descompuesto aún más el problema.  
  
**Beneficios de aplicar este principio:**

* Resolver más problemas, más rápido.
* Resolver problemas en pocas líneas de código.
* Producir código de mejor calidad.
* Construir sistemas muy grandes, fáciles de mantener.
* Lograr un framework propio más flexible y más fácil de ampliar, modificar o refactorizar cuando lleguen nuevos requerimientos.
* Se podrá trabajar en grandes equipos de desarrollo y grandes proyectos ya que el código será simple y estúpido.

## [Principio K.I.S.S. ( parte II )](http://programacionsolida.com.ar/2011/11/principio-kiss-parte-ii.html)

Habiendo desarrollado en la última entrada de este blog el concepto del [principio K.I.S.S. y sus beneficios](http://programacionsolida.blogspot.com/2011/11/principio-kiss-parte-i.html), veremos a continuación la segunda y última entrega de este principio.  
  
  
**Cómo aplicarlo al trabajo diario**  
Son muchos pasos los que se deben llevar a cabo, muy simples, pero pueden resultar un reto para algunos.  
Tan fácil como suena, mantenerlo simple, es sólo cuestión de paciencia. Sobretodo, con uno mismo.  
A continuación, veremos los puntos a tener en cuenta para llevarlo a cabo:

* Ser humilde. No pensar en uno mismo como un super genio. Este es el primer error. Siendo humilde, es muy posible alcanzar el estatus de super genio. Pero, aunque así no se lograse, a quién le importa?! El código es simple y estúpido, por lo que no hay que ser un genio para trabajar con él.
* Dividir las tareas en subtareas que, en principio, no deben llevar más de 4-12 horas de codificación.
* Dividir los problemas en pequeños problemas. Cada problema se debe poder resolver dentro de una o muy pocas clases.
* Mantener los métodos pequeños. Cada método no debe tener nunca más de 10-20 líneas y sólo debe resolver un (y sólo un) problema. Si contiene muchas condiciones, hay que dividirlo en métodos más pequeños. (Esto no sólo los hace más legibles y mantenibles, sino que además, encontrar errores es mucho más rápido.
* Mantener las clases pequeñas. Aplicar la misma metodología que con los métodos.
* Hay que resolver los problemas y luego codificarlos, no al revés. Resolver los problemas a medida que se codifica, no tiene nada de malo. Se puede hacer esto y aún seguir alineado a lo sugerido. Pero no hay que temer de refactorizar una y otra, y otra y otra y otra vez el código. Lo que importa es el resultado final y el número de líneas no es una medición, a menos que la medida sea que menos, es mejor, por supuesto.
* No temer a desechar lo codificado. Refactorizar y recodificar, son dos cuestiones muy importantes. Al llegar nuevos requerimientos (que no existían cuando se escribió el código o no se estaba al tanto), se debe poder resolver los viejos y los nuevos problemas con una solución aún mejor. Si se ha seguido los consejos anteriores, la cantidad de código a reescribir, es mínimo. Y si no se ha seguido los consejos anteriores, es muy probable que igualmente deba ser reescrito.
* Para todos los demás escenarios, se debe tratar de mantenerlo lo más simple posible. Este es el patrón de comportamiento más difícil de aplicar, pero una vez que se logra adquirir, se puede mirar hacia atrás y decir "No puedo imaginar cómo trabajaba antes".

**Conclusión**  
Algunos de los mejores algoritmos del mundo, son siempre los que tienen la menor cantidad de líneas de código. Y al analizarlas, son fácilmente entendibles. Sus creadores, dividieron el problema hasta que fue tan fácil, que pudo implementarse.  
**Muchos grandes solucionadores de problemas, no fueron grandes codificadores, pero aún así, produjeron un gran código.**

## [Los principios S.O.L.I.D.](http://programacionsolida.com.ar/2011/11/los-principios-solid.html)

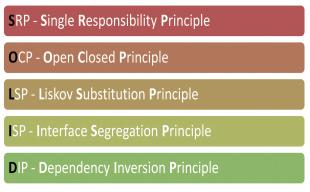
Toda aplicación nace de los requisitos de una persona que los realiza. Si dichos requisitos fueran inamovibles e inmutables a lo largo del tiempo, las modificaciones de software nunca serían necesarias.

Lamentablemente, esto es sólo una utopía para los programadores y con el correr del tiempo hemos tratado de no caer en las trampas que nosotros mismos nos creamos cuando programamos sin pensar en los cambios que los requerimientos pueden sufrir a lo largo del tiempo.

Entra tantas mejoras que se han ido agregando al desarrollo de software, la programación orientada a objetos ([P.O.O.](http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos)) se ha convertido en un conocimiento que hoy se considera muy importante para esta tarea.

Implementar los principios [S.O.L.I.D.](http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29), nos redundará en una mejora sustancial del diseño y arquitectura de nuestras aplicaciones, haciendo que las mismas sean mucho más flexibles y extensibles.

Este acrónimo mnemotécnico surge de la unión de varios principios básicos de la programación orientada a objetos, explicados en el libro "[Agile software development: Principles, Patterns and Practices](http://www.amazon.co.uk/gp/product/0135974445/ref=as_li_ss_tl?ie=UTF8&tag=mundogeek-21&linkCode=as2&camp=1634&creative=19450&creativeASIN=0135974445)" por uno de los grandes exponentes de la Artesanía del software: el famoso [Uncle Bob (Robert Cecil Martin)](http://en.wikipedia.org/wiki/Robert_C._Martin).

[](http://3.bp.blogspot.com/-lcnKDEECO7Y/TsGAKSJL8-I/AAAAAAAAAGE/hVOtJ9swagk/s1600/SOLID1.JPG)

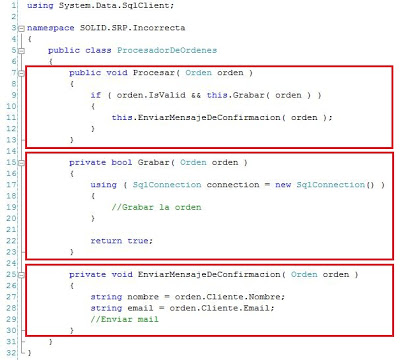
* Principio de responsabilidad única: Cada clase debe tener un y sólo un motivo para cambiar.
* Principio abierto/cerrado: Los objetos deben estar abiertos a extensión, pero cerrados a la modificación.
* Principio de sustitución de Liskov: Los objetos de una clase deben poder ser sustituidos por instancias de clases derivadas.
* Principio de segregación de interfaces: Crear pequeñas interfaces específicas para los clientes.
* Principio de inversión de dependencias: Depender de abstracciones, no de implementaciones concretas.

A lo largo de sucesivos posteos iré tratando de desentrañar cada uno de estos principios con la esperanza de que les pueda servir a todos tanto como a mí me han servido.

Nota: Si bien no es excluyente, se recomienda tener experiencia en [P.O.O.](http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos) para hacer una buena implementación de ellos y así obtener el mayor beneficio posible.

## [Principio de responsabilidad única (SRP)](http://programacionsolida.com.ar/2011/11/principio-de-responsabilidad-unica-srp.html)

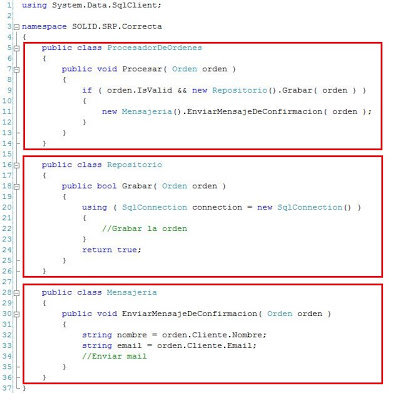
El principio de responsabilidad única ([SRP](http://en.wikipedia.org/wiki/Single_responsibility_principle)) es el primer principio del acrónimo SOLID y establece que "Nunca debe haber más de una razón para que una clase cambie".  
Algo con un propósito, una responsabilidad, siempre va a resultar mucho más fácil de cambiar. Esto es simplemente porque se puede ver mucho más claramente lo que hace y lo que no. Además, como tiene una única responsabilidad, no hay chance de que el cambio en el código pueda afectar a otro código, comportamiento quizás, pero código, no. Y mejora en gran medida la reutilización de código.  
Finalmente, el código se hace mucho más fácil de probar ya que se puede realizar el test de una responsabilidad en forma aislada de las demás.  
  
Un ejemplo claro de esto, son las navajas suizas. Tienen un cuchillo, un destapador, un abrelatas, etc. Y cuando uno de ellos se rompe, hay que cambiar la navaja completa.  
Yendo al código, que es nuestro campo, podemos ver a continuación un pequeño ejemplo:

[](http://4.bp.blogspot.com/-3X_1HlpgsCE/TsQAGGUqDkI/AAAAAAAAAHI/WipUP-a-gp4/s1600/SOLID.SRP.Incorrecta.JPG)

Veamos las diferentes responsabilidades que posee la clase ProcesadorDeOrdenes, ellas son:

* El procesamiento de las ordenes.
* La grabación en la base de datos.
* El envío de un mensaje de confirmación.

Rápidamente podemos ver que hay tres responsabilidades bien separadas. Por este motivo, hay tres razones por las cuales ésta clase podría cambiar.

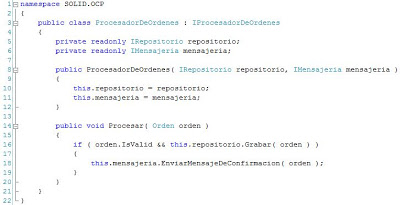
[](http://2.bp.blogspot.com/-dch7VchDc2Q/TsqNFCxp-AI/AAAAAAAAAI8/FPQchBxGffs/s1600/SOLID.SRP.Correcta.JPG)

Como se puede observar, hemos sido capaces de separar las diferentes responsabilidades en tres clases separadas (ProcesadorDeOrdenes, Repositorio y Mensajeria). Una mirada a cada una de ellas, nos permite identificar claramente lo que debe estar haciendo. Esto también nos habilita la reutilización de las diferentes responsabilidades.

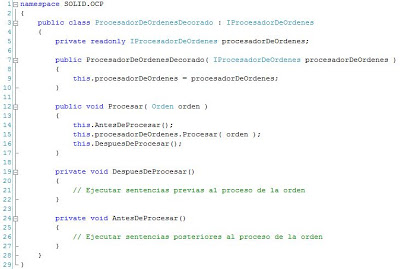
Nota: [SRP](http://en.wikipedia.org/wiki/Single_responsibility_principle) es uno de los principios más difíciles de hacer cumplir, porque siempre hay espacio para la refactorización de una clase en varias para que cada cual tenga una sola responsabilidad. Es una preferencia personal, porque la explosión de clases hace que algunas personas se conviertan en fanáticos del código.

## [Principio Abierto Cerrado (OCP)](http://programacionsolida.com.ar/2011/11/principio-abierto-cerrado-ocp.html)

La segunda letra del acrónimo S.O.L.I.D., representa al [principio Abierto Cerrado](http://en.wikipedia.org/wiki/Open/closed_principle). El cuál establece que **"Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión pero cerradas para modificación"**. Esto básicamente significa que debemos ser capaces de cambiar el comportamiento externo o las dependencias externas de una clase, sin tener que cambiar físicamente la clase en sí misma. Este tipo de comportamiento nos habilita a cambiar una parte del código sin estar obligado a cambiar otras partes del mismo, siempre y cuando nos mantengamos dentro de los límites del contrato existente.  
Suena fácil, pero muchos desarrolladores podemos perder el norte durante la implementación de este enfoque de extensibilidad. No se trata necesariamente por una cuestión de habilidades, sino más bien, que no se nos ha enseñado a abordar este principio en el diseño de clases.  
Retomando el ejemplo de código visto anteriormente, podemos ver a la clase ProcesadorDeOrdenes ([SRP](http://programacionsolida.blogspot.com/2011/11/principio-de-responsabilidad-unica-srp.html)) algo cambiada (veremos otros detalles de la implementación más adelante con el principio de inversión de dependencias).

[](http://2.bp.blogspot.com/-c_gumvmkV0k/TsqNqgvEUZI/AAAAAAAAAJI/j_nEDHdNWm4/s1600/SOLID.OCP.1.JPG)

Esta clase ya está preparada para el principio Abierto cerrado. Como se puede ver en la imagen anterior, ahora la clase ProcesadorDeOrdenes implementa la interfaz IProcesadorDeOrdenes. Ahora podemos crear otra implementación con el mismo contrato.

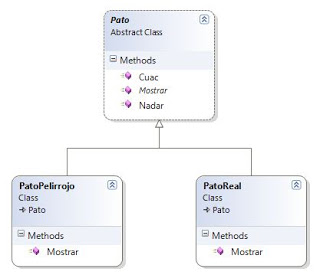
[](http://2.bp.blogspot.com/-in7dHcAIVdo/TsqSmCmlWEI/AAAAAAAAAJU/34fkIHotyJc/s1600/SOLID.OCP.2.JPG)

Con la clase ProcesadorDeOrdenesDecorado (que implementa la interfaz IProcesadorDeOrdenes) somos capaces de agregarle funcionalidad a la clase original sin modificar el código de la clase ProcesadorDeOrdenes.

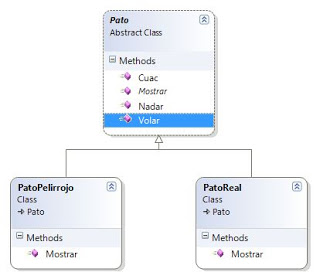
## [Principio abierto cerrado (OCP) Parte II](http://programacionsolida.com.ar/2011/11/principio-abierto-cerrado-ocp-parte-ii.html)

Teniendo en cuenta los comentarios recibidos en la [entrada anterior](http://programacionsolida.blogspot.com/2011/11/principio-abierto-cerrado-ocp.html) y en forma personal, paso a comentar otra forma de poder cumplir con este principio abierto y cerrado (OCP): la implementación del [patrón de diseño strategy](http://en.wikipedia.org/wiki/Strategy_pattern).  
Este patrón de diseño, establece que los comportamientos no deben heredarse. En su lugar, se deben encapsular usando interfaces.  
Para dar un ejemplo, me remitiré a uno ya existente en el libro [Head first design patterns](http://www.amazon.com/First-Design-Patterns-Elisabeth-Freeman/dp/0596007124).

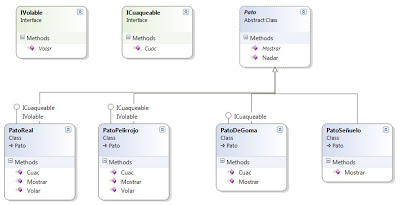
Debemos programar una simulación de un pato, para ello, creamos una clase abstracta denominada Pato, la cual posee los métodos (o comportamientos) Cuac(), Nadar() y Mostrar(). Todos los patos hacen cuac y nadan, por lo que esta clase implementa dichos métodos. Pero, como todos los patos lucen diferentes, el método Mostrar será abstracto, es decir, se implementará en cada subclase.

[](http://4.bp.blogspot.com/-X9LsiOADP1k/TtUaFMVpZnI/AAAAAAAAALs/9dY0G73VMms/s1600/SOLID.OCP.Pato1.JPG)

En un segundo análisis, notamos el hecho de que nos ha faltado un comportamiento: Volar. Por este motivo, incluimos e implementamos el método con el mismo nombre en nuestra clase abstracta Pato. De esta manera, las subclases de Pato ya pueden volar.

[](http://3.bp.blogspot.com/-HXNfqeNVpZs/TtUbTXleMaI/AAAAAAAAAL0/Fuf2B5VAFFc/s1600/SOLID.OCP.Pato2.JPG)

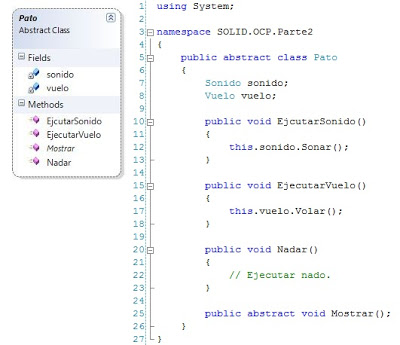
Pues bien, qué pasa si agregamos una nueva clase de pato, el pato de goma. Tendría, equivocadamente, la posibilidad de volar. Si resolviéramos esto a través de la herencia, sólo tendríamos que sobrescribir el método Volar y listo, todo arreglado. Pero no, porque si además agregáramos otra clase, por ejemplo, el pato señuelo, además de no ser capaz de volar, tampoco tiene la capacidad de hacer cuac.  
Es evidente que la herencia no resuelve este problema. Que tal si implementamos interfaces. Extraemos el método Volar y Cuac fuera de la clase Pato y creamos dos interfaces para ellos. De esta manera, sólo implementarían la clase que corresponda para aquellos patos que pueden volar o hacer cuac.

[](http://3.bp.blogspot.com/-As3Hxr2vaPc/TtUjyeTIGiI/AAAAAAAAAMI/urRFn-tEZUo/s1600/SOLID.OCP.Pato3.JPG)

En primera instancia, parece una buena idea, salvo que tiene una pequeña contra: el código repetido. Tanto el pato real como pato pelirrojo, hacen cuac y vuelan del mismo modo. Si nos pareció mal pisar métodos heredados para remover comportamientos, duplicar código es casi tan malo como eso. Sobre todo, si en lugar de tener sólo dos subclases que los implementan, tuviéramos 48 y hubiera que modificar todos solamente para introducir un cambio.  
Como la única constante en software, es el cambio, nos conviene aplicar los patrones de diseño. Para ello, separemos aquellos aspectos de nuestro código que varían de los que permanecen iguales.  
Para este caso, lo que varía entre las distintas clases de patos, son los comportamientos de volar y hacer cuac. Claramente podemos notar en el caso de volar dos implementaciones diferentes: volar y no volar. Mientras que en el caso del sonido, tenemos tres: el cuac propiamente dicho, el chillido y sin sonido.

[](http://3.bp.blogspot.com/-S1WTvz2us-A/TtUvHfcsVqI/AAAAAAAAAMQ/rQJOGBMQi-E/s1600/SOLID.OCP.Pato4.JPG)

Con este diseño, los comportamientos extraídos ahora pueden ser reutilizados por otras clases ya que no están limitados solamente a las clases a las que pertenecían.  
Teniendo en cuenta la creación de estos comportamientos, sólo nos resta modificar la clase Pato para que implemente estos cambios.

[](http://1.bp.blogspot.com/--m5B37dmN-8/TtUvHZojw2I/AAAAAAAAAMY/skyc5r8NzMg/s1600/SOLID.OCP.Pato5.JPG)

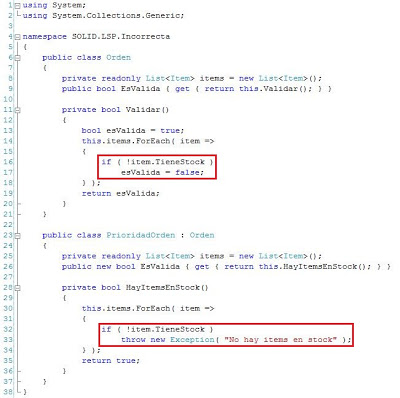
Las implementaciones de PatoReal, PatoPelirrojo, PatoDeGoma y PatoSeñuelo instanciarán las clases de comportamiento según corresponda.

## [Principio de sustitución de Liskov (LSP)](http://programacionsolida.com.ar/2011/11/principio-de-sustitucion-de-liskov-lsp.html)

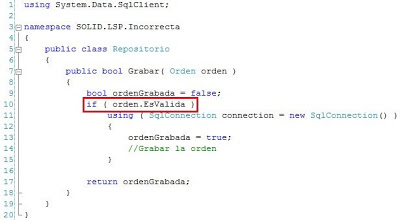
|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://3.bp.blogspot.com/-RLRLrpOBEoY/TspzxLJ9b4I/AAAAAAAAAIQ/OjI97R9bp50/s1600/LSP.jpg)

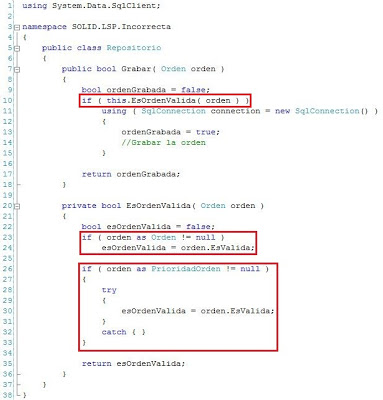
La tercera letra del acrónimo S.O.L.I.D., representa al [Principio de sustitución de Liskov (LSP)](http://en.wikipedia.org/wiki/Liskov_substitution_principle). Establece que **"Las funciones que usan punteros o referencias a clases base, deben ser capaces de utilizar objetos o clases derivadas sin necesidad de saberlo"**. Esto significa que cuando nuestro código usa una clase específica (o una interfaz), éste debería ser capaz de usar otra clase que herede de la específica (o diferentes implementaciones de la interfaz) sin tener que cambiar su comportamiento interno.  
Esto, nuevamente, es para minimizar el impacto de los cambios que puedan haber sobre nuestro código.  
Los problemas que resuelve LSP casi siempre son fácilmente evitables. Hay algunos habituales signos reveladores de que una violación a LSP aparece en el código. He aquí un escenario que muestra cómo puede ocurrir una violación a LSP.

[](http://2.bp.blogspot.com/-cc9Cyij2Nxw/TsvSISoZzjI/AAAAAAAAAJ0/VTjF4AH2vTE/s1600/SOLID.LSP.Incorrecta.JPG)

Si miramos la clase PrioridadOrden, veremos como la misma viola éste principio porque cuando un cliente de la clase Orden chequea la propiedad EsValida, espera un retorno de tipo *bool*.

[](http://3.bp.blogspot.com/-FlB-AbHqBDE/TsvxeJs9VnI/AAAAAAAAAKY/3-_4qzqtNPc/s1600/SOLID.LSP.Incorrecta.2.JPG)

Pero cuando en lugar de tener un instancia de Orden, el cliente tiene una instancia de PrioridadOrden, súbitamente el cliente necesita capturar una posible excepción (*try-catch*) que se lanza cuando la orden no es válida.

[](http://4.bp.blogspot.com/-dYRjBOQYsTo/TsvzGpTjJ8I/AAAAAAAAAK0/I-nEaX54RwY/s1600/SOLID.LSP.Incorrecta.3.JPG)

De pronto, el cliente necesita conocer con qué tipo de clase específica esta tratando, lo que representa una clara violación al principio de sustitución de Liskov.

Este principio fue introducido inicialmente como *comportamiento*por [Barbara Liskov](http://en.wikipedia.org/wiki/Barbara_Liskov" \t "_blank) en un discurso en una conferencia de 1987 titulada: [Abstracción de datos y jerarquía](http://www.sr.cefetes.br/%7Emcosta/disciplinas/20091/tpa/recursos/p17-liskov.pdf).

## [Principio de segregación de interfaces (ISP)](http://programacionsolida.com.ar/2011/12/principio-de-segregacion-de-interfaces.html)

|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://4.bp.blogspot.com/-wVMJeV__VUw/TspzxG0VCrI/AAAAAAAAAIM/U-aJkbzVBck/s1600/ISP.jpg)

La cuarta letra del acrónimo S.O.L.I.D., representa al [Principio de segregación de interfaces (ISP)](http://en.wikipedia.org/wiki/Interface_segregation_principle) y afirma que **"Los clientes no deben ser forzados a depender de interfaces que no utilizan"**. Esto significa que cuando un objeto tiene diferentes usos (no confundir con distintas responsabilidades) debe haber entonces una interfaz específica por cada uno de esos usos.  
Esto se torna repetitivo, pero esto es para minimizar el impacto de los cambios que puedan haber sobre nuestro código.  
En otras palabras: si tenemos una clase abstracta o una interfaz, entonces, quienes la implementen, no deben ser forzados a implementar partes (métodos o propiedades) que no les importen.

Consideremos un tienda web en la cual los clientes pueden agregar o quitar items de su carrito de compras y, por supuesto, consultar los items que actualmente contiene. Luego, ellos van a cerrar su pedido y allí se les pide que completen sus datos personales. Como paso final, se hace una revisión completa de los items pedidos y los datos personales ingresados para ahí enviar la orden de pedido.

[](http://2.bp.blogspot.com/-TymYRniZeIo/TswKvtlXsxI/AAAAAAAAALE/cFT6IqFZhEk/s1600/SOLID.ISP.Incorrecta.JPG)

Aquí hay tres diferentes usos de la orden. Cada paso (Carrito, Caja y Proceso) no necesita o no debe tener toda la información que la orden puede proveer.

[](http://2.bp.blogspot.com/-R18oujGDV1E/TswKv7Ute-I/AAAAAAAAALI/mdnIny_4Yd4/s1600/SOLID.ISP.Correcta.JPG)

Ahora, hemos creado interfaces específicas para los pasos específicos en el proceso de la orden. La clase Orden aun se ve como la habíamos codificado, pero los distintos clientes tienen una visión completamente diferente de ella. Esto nos permite también, crear diferentes objetos, uno para cada paso implementando sólo la interfaz necesaria.

Un ejemplo más conocido es el ejemplo de transacciones de un cajero automático dado en [Desarrollo de Software Ágil: principios, patrones y prácticas](http://www.amazon.co.uk/gp/product/0135974445/ref=as_li_ss_tl?ie=UTF8&tag=mundogeek-21&linkCode=as2&camp=1634&creative=19450&creativeASIN=0135974445) y también en un [artículo escrito](http://www.objectmentor.com/resources/articles/isp.pdf) por [Robert C. Martin](http://en.wikipedia.org/wiki/Robert_C._Martin) específicamente [acerca del ISP](http://www.objectmentor.com/resources/articles/isp.pdf). En este ejemplo se trata de una interfaz para la interfaz de usuario de un cajero automático, que se encarga de todas las solicitudes tanto una solicitud de depósito, o una solicitud de retiro, y cómo esta interfaz tiene que ser segregada en interfaces individuales y específicas.

## [Principio de inversión de dependencia (DIP)](http://programacionsolida.com.ar/2011/12/principio-de-inversion-de-dependencia.html)

15

dic

|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://2.bp.blogspot.com/-rFcG1L9IMD4/Tspw8mh1H4I/AAAAAAAAAIA/8MAnJdH5x9o/s1600/DIP.jpg)

El [Principio de inversión de dependencia (DIP)](http://en.wikipedia.org/wiki/Dependency_inversion_principle) es el quinto y último del acrónimo SOLID y establece que: "**Los módulos de alto nivel no deberían depender de módulos de bajo nivel, ambos deberían depender de abstracciones**" y "**Las abstracciones no deberían depender de detalles sino que los detalles deberían depender de las abstracciones**". Si esto no es un buen trabalenguas, los trabalenguas dónde están...  
Esto básicamente significa que en ningún lugar de nuestro código deberíamos depender de una implementación actual, sino que en su lugar debemos depender de interfaces o clases abstractas. ¿Y por qué deberíamos hacer esto? Porque esto nos habilita a cambiar cualquier implementación por otra y ninguna otra clase que hayamos codificado debe saberlo o importarle. Si, es así, a ninguna otra clase debe importarle que el cambiemos un colaborador por otro.

[](http://2.bp.blogspot.com/-uefiT37_-g4/TuY4MKoRP7I/AAAAAAAAANw/CqvWdlMEFys/s1600/SOLID.DIP.Incorrecta.JPG)

En este ejemplo podemos ver que la clase ProcesadorDeOrdenes depende de la implementación actual (detalle) de Repositorio y NotificadorPorMail. Entonces, si quisiéramos cambiar la manera en que se envían las notificaciones al cliente, por ejemplo por SMS, deberíamos crear la clase NotificadorPorSMS y deberíamos cambiar la clase ProcesadorDeOrdenes para que use tanto NotificadorPorMail como NotificadorPorSMS, lo cual, **está mal**, porque estamos haciendo que ProcesadorDeOrdenes sea consciente de la forma en que se envían las notificaciones.

[](http://3.bp.blogspot.com/-P8T3JMo5DCI/TuY4MViIElI/AAAAAAAAAN0/3jqeY898P4I/s1600/SOLID.DIP.Correcta.JPG)

Ahora, ProcesadorDeOrdenes ya no depende más de las implementaciones actuales, sólo se basa en interfaces: INotificadorPorMail y IRepositorio. Entonces, si ahora queremos cambiar la manera en que se envían las notificaciones simplemente debemos crear una nueva implementación de INotificadorPorMail y proveerle dicha implementación al ProcesadorDeOrdenes. Ésta, simplemente debe seguir trabajando como hasta el momento, su comportamiento ha cambiado, pero esto no le importa, sólo le importa su propia lógica.

Este principio fue postulado por [Robert C. Martin](http://en.wikipedia.org/wiki/Robert_C._Martin) y descrito en varias publicaciones incluyendo: [Object Oriented Design Quality Metrics: an analysis of dependencies](http://www.objectmentor.com/resources/articles/oodmetrc.pdf" \t "_blank): un artículo publicado en el C++ Report en mayo de 1996 titulado [El principio de inversión de dependencia](http://www.objectmentor.com/publications/dip.pdf). Y los libros [*Agile Software Development, Principles, Patterns, and Practices*](http://www.amazon.com/Software-Development-Principles-Patterns-Practices/dp/0135974445/ref=sr_1_3?s=books&ie=UTF8&qid=1323478887&sr=1-3) y [*Agile Principles, Patterns, and Practices in C#*](http://www.amazon.com/Agile-Principles-Patterns-Practices-C/dp/0131857258/ref=sr_1_1?ie=UTF8&qid=1323478799&sr=8-1).

## [Inversión de control (IoC)](http://programacionsolida.com.ar/2011/12/inversion-de-control-ioc.html)

19

dic

|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://3.bp.blogspot.com/-oS4-nI4QOtY/Tu-NanQY66I/AAAAAAAAAPo/ooqFQVRQBSI/s1600/inverted-bookshelf.jpg)

Implementar el principio de inversión de dependencia (DIP) sin realizar [inversión de control](http://en.wikipedia.org/wiki/Inversion_of_control) puede resultar en un trabajo incompleto. Vamos a terminar con las dependencias reales en nuestro código mediante la inyección de clases colaboradoras pero solamente en el código de alto nivel. Esto aun nos deja en un mal lugar, aquí es donde la inversión de control tiene lugar.  
Echemos una mirada al siguiente código:

[](http://3.bp.blogspot.com/--dmAPHXUjeU/Tu-RtO5z_zI/AAAAAAAAAP0/Ndzli3FsMl8/s1600/SOLID.IOC.SinIoC.JPG)

Como podemos ver, las diferentes sentencias *new* aun son necesarias en este nivel.

[](http://1.bp.blogspot.com/-b40DV4B7kpk/Tu-XW14xp0I/AAAAAAAAAQA/Jq-sAa6N55I/s1600/SOLID.IOC.ConIoC.1.JPG)

Ahora, este otro luce mucho más limpio. Lo que estamos viendo aquí es el uso del [*Common Service Locator*](http://commonservicelocator.codeplex.com/), que es una interfaz común que soporta todos los diferentes *contenedores* de inversión de control. Así que aquí tampoco hay una dependencia de un contenedor específico de Inversión de control. Un contenedor apropiado puede resolver cualquier dependencia que una clase solicite e inyectárselas al crear una nueva instancia de ella. En otras palabras, ha realizado automáticamente lo que hubiéramos tenido que hacer manualmente.

[](http://1.bp.blogspot.com/-5KP9Cb38Svs/Tu-dTAWpqAI/AAAAAAAAAQM/RnJnBe9RYnc/s1600/SOLID.IOC.ConIoC.2.JPG)

Viendo este último ejemplo de código notamos que en realidad es necesario configurar el contenedor de Inversión de control. Esto es específico para el contenedor que utilicemos. En este ejemplo, vemos al [StructureMap](http://structuremap.net/structuremap/). Lo bueno de esta configuración, es que este es el único lugar del código en donde definimos nuestras dependencias actuales. Este es el único lugar en el que tenemos que conocer nuestras diferentes implementaciones.

Muchos contenedores de Inversión de control soportan también configuraciones vía XML pero hay que tener cuidado ya **se pierde la seguridad de los tipos y el apoyo del entorno para la refactorización**.

## [DRY y YAGNI, dos metodologías para no perder de vista](http://programacionsolida.com.ar/2011/12/para-cerrar-una-etapa-de-acronimos-o.html)

22

dic

|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://1.bp.blogspot.com/-K7hCOLzoY2o/TvSi4Hu2u-I/AAAAAAAAAQY/1q_Id_SRmPs/s1600/acronym-soup.gif)

Para cerrar una etapa de acrónimos o siglas relacionados al diseño de software, me resta mencionar al menos dos que no está demás tener en cuenta.  
Son metodologías en las que solemos caer muy fácilmente y para colmo de males, con bastante frecuencia.  
Estas son: DRY (*[Don't Repeat Yourself](http://en.wikipedia.org/wiki/Don%27t_Repeat_Yourself" \t "_blank)*) y YAGNI (*[You Ain't Gonna Need It](http://en.wikipedia.org/wiki/You_Ain%27t_Gonna_Need_It" \t "_blank)*).  
Intentar llevar a cabo todas y cada una de las metodologías y principios vistos hasta aquí, nos permitirá tener un código limpio; mucho más fácil de leer, comprender y expandir.

* **DRY**: lo que para nosotros en castellano significa: N*o te repitas a tí mismo*. No es ni más ni menos que eso. Es decir, que al tener la misma funcionalidad en nuestro código, una y otra vez, cuando debemos realizar un cambio, corremos el riesgo de olvidarnos de algo. Y no solamente tenemos duplicado el código, sino también las pruebas que les hayamos aplicado. Esta duplicidad, vuelve nuestro código frágil.
* **YAGNI**: traducido a nuestro idioma: *No lo vas a necesitar*. Nos sucede a menudo que cuando estamos codificando algo, pensamos "Esto es algo que vamos a necesitar después" y lo agregamos en ese mismo momento aunque no estemos trabajando en eso exactamente. Entonces, cuando llegamos al punto en que realmente tenemos una completa comprensión de lo que actualmente se necesita, nos damos cuenta que lo que creamos, **no** es lo que se requiere. Así que, en lugar de ahorrar tiempo realizándolo cuando trabajábamos en el mismo contexto, perdimos nuestro preciado tiempo porque ahora tenemos que volver a hacerlo de nuevo. Cuando hemos diseñado nuestro código apropiadamente, agregarle funcionalidad no va a ser un problema. Debemos estar preparados para los cambios, pero debemos hacerlos cuando realmente son necesarios, no antes.

Tengamos siempre en cuenta que nosotros mismos nos podemos convertir en nuestros propios *psicópatas violentos*. Entonces, seamos benevolentes con nosotros mismos y produzcamos código de calidad.

## [Patrones de diseño](http://programacionsolida.com.ar/2012/01/patrones-de-diseno.html)

05

ene

|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://2.bp.blogspot.com/-pCm4UcP-3is/TvszqU5VBzI/AAAAAAAAAQ8/lFOU8fGcxRo/s1600/puzzle.jpg)

En la ingeniería de software, un patrón de diseño es una solución genérica y repetitiva aplicable a problemas comunes que ocurren en el diseño de software. Un patrón de diseño no es un diseño definitivo que puede ser transformado en código directamente. Es mas bien una descripción o plantilla que nos indica el *cómo* resolver el problema y que puede usarse en muchas diferentes situaciones.  
Los patrones de diseño pueden acelerar el proceso de desarrollo proporcionando paradigmas comprobados. El diseño eficaz de software requiere considerar cuestiones que pueden no ser visibles hasta después de la implementación.  
El uso de patrones de diseño ayuda a prevenir problemas sutiles que pueden causar problemas mayores y mejora la legibilidad del código para los programadores y arquitectos familiarizados con los patrones.  
A menudo, la gente sólo comprende la manera de aplicar ciertas técnicas de diseño de software para ciertos problemas. Estas técnicas son difíciles de aplicar a una amplia gama de problemas. Los patrones de diseño proveen soluciones generales, documentadas de una forma que no requieren detalles específicos vinculados a un problema particular.  
  
Además, lo patrones le permiten a los desarrolladores comunicarse con nombres conocidos para las interacciones. Los patrones de diseño habituales se pueden mejorar con el tiempo haciéndolos más robustos que los diseños ad-hoc.  
  
***Patrones de diseño de creación***  
Son patrones de instanciación de clases. Estos pueden ser divididos en patrones de creación de clases y patrones de creación de objetos. Mientras que los primeros usan la herencia de manera efectiva en el proceso de instanciación, los segundos usan la delegación para hacer su trabajo.

* [*Abstract Factory*](http://programacionsolida.blogspot.com/2012/01/patrones-de-diseno-de-creacion-abstract.html): Crea una instancia de varias familias de clases.
* [*Builder*](http://programacionsolida.blogspot.com/2012/01/patrones-de-diseno-de-creacion-builder.html): Separa la construcción de objetos de su representación.
* [*Factory Method*](http://programacionsolida.blogspot.com/2012/02/patrones-de-diseno-de-creacion-factory.html): Crea una instancia de varias clases derivadas.
* [*Object Pool*](http://programacionsolida.blogspot.com/2012/02/patrones-de-diseno-de-creacion-object.html): Evita la costosa adquisición y liberación de recursos mediante el reciclado de objetos que ya no están en uso.
* [*Prototype*](http://programacionsolida.blogspot.com/2012/02/patrones-de-diseno-de-creacion.html): Una instancia completamente inicializada que puede ser copiada o clonada.
* [*Singleton*](http://programacionsolida.blogspot.com/2012/02/patrones-de-diseno-creacionales.html): Una clase de la cual sólo puede existir una sola instancia.

***Patrones de diseño estructurales***  
Estos tratan a cerca de composición de Clases y Objetos. El patrón estructural de creación de clases usa la herencia para componer interfaces. El patrón estructural de objetos define maneras de componer objetos para obtener nuevas funcionalidades.

* [*Adapter*](http://programacionsolida.com.ar/2012/02/patrones-de-diseno-estructurales.html): Hace que coincidan interfaces de diferentes clases.
* [*Bridge*](http://programacionsolida.com.ar/2012/02/patrones-de-diseno-estructurales-bridge.html): Separa la interfaz de un objeto de su implementación.
* [*Composite*](http://programacionsolida.com.ar/2012/03/patrones-de-diseno-estructurales.html): Una estructura de tipo árbol de objetos simples y compuestos.
* [*Decorator*](http://programacionsolida.com.ar/2012/03/patrones-de-diseno-estructurales_08.html): Agrega responsabilidades a objetos dinámicamente.
* [*Facade*](http://programacionsolida.com.ar/2012/03/patrones-de-diseno-estructurales-facade.html): una única clase que representa todo un subsistema.
* [*Flyweight*](http://programacionsolida.com.ar/2012/03/patrones-de-diseno-estructurales_22.html): Una instancia granulada para compartir de manera eficiente.
* [*Private class data*](http://programacionsolida.com.ar/2012/03/patrones-de-diseno-estructurales_29.html): Restringe el acceso / muta el acceso.
* [*Proxy*](http://programacionsolida.com.ar/2012/04/patrones-de-diseno-estructurales-proxy.html): Un objeto que representa a otro objeto.

***Patrones de diseño de comportamiento***  
Estos patrones son los que se ocupan específicamente de la comunicación entre los objetos.

* [*Chain of responsability*](http://programacionsolida.com.ar/2012/04/patrones-de-diseno-de-comportamiento.html): Una forma de pasar una solicitud a través de una cadena de objetos.
* [*Command*](http://programacionsolida.com.ar/2012/04/patrones-de-diseno-de-comportamiento_23.html): Encapsula una solicitud como un objeto.
* [*Interpreter*](http://programacionsolida.com.ar/2012/04/patrones-de-diseno-de-comportamiento_26.html): Una manera de incluir elementos de un lenguaje a un programa.
* [*Iterator*](http://programacionsolida.com.ar/2012/05/patrones-de-diseno-de-comportamiento.html): Accede secuencialmente los elementos de una colección.
* [*Mediator*](http://programacionsolida.com.ar/2012/05/patrones-de-diseno-de-comportamiento_31.html): Define una comunicación simplificada entre clases.
* [*Memento*](http://programacionsolida.com.ar/2012/06/patrones-de-diseno-de-comportamiento.html): Captura y restaura el estado interno de un objeto.
* [*Null Object*](http://programacionsolida.com.ar/2012/06/patrones-de-diseno-de-comportamiento_14.html): Diseñado para actuar como valor por defecto de un objeto.
* [*Observer*](http://programacionsolida.com.ar/2012/06/patrones-de-diseno-de-comportamiento_28.html): Una forma de notificar cambios a un número de clases.
* [*State*](http://programacionsolida.com.ar/2012/07/patrones-de-diseno-de-comportamiento.html): Altera el comportamiento de un objeto cuando cambia su estado.
* [*Strategy*](http://programacionsolida.com.ar/2012/07/patrones-de-diseno-de-comportamiento_05.html): Encapsula un algoritmo dentro de una clase.
* [*Template Method*](http://programacionsolida.com.ar/2012/07/patrones-de-diseno-de-comportamiento_09.html): Difiere los pasos exactos de un algoritmo hacia una subclase.
* [*Visitor*](http://programacionsolida.com.ar/2012/07/patrones-de-diseno-de-comportamiento_12.html): Define una nueva operación a una clase sin cambio.

***Críticas***

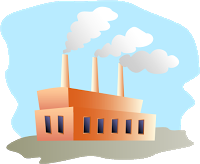
El concepto de patrones de diseño ha sido criticado por algunos en el campo de la informática. Sostienen que el uso de patrones de diseño es en realidad el resultado del uso de lenguajes o técnicas con insuficiente habilidad de abstracción. Como escribe Paul Graham en su ensayo [La venganza de los nerds](http://www.paulgraham.com/icad.html). O Peter Norvig, quien demuestra que 16 de 23 de los patrones presentados en el libro [Design Patterns](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612) (enfocado a C++) están simplificado o incluso eliminados directamente por lenguajes como [Lisp](http://en.wikipedia.org/wiki/Lisp_%28programming_language%29) o [Dylan](http://en.wikipedia.org/wiki/Dylan_%28programming_language%29).

## [Patrones de diseño de creación: Abstract Factory](http://programacionsolida.com.ar/2012/01/patrones-de-diseno-de-creacion-abstract.html)

09

ene

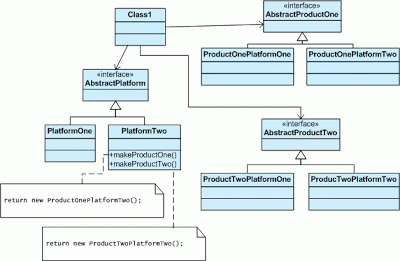
|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://1.bp.blogspot.com/-jbsSi14RmNU/TwZjvKHCU9I/AAAAAAAAASY/U_GZTqAwZgI/s1600/factory_T.png)

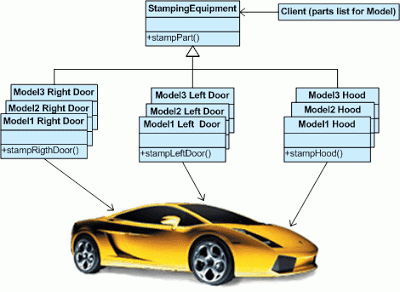
***Intención del patrón***

* Proveer una interfaz para la creación de familias de objetos, relacionados o dependientes, sin especificar su clase concreta.
* Una jerarquía que encapsula: muchas posibles plataformas y la construcción de una suite de productos.
* Que el operador *new* sea considerado como perjudicial.

***Ejemplo de problema***  
Si una aplicación está diseñada para ser portátil, ésta necesita encapsular sus dependencias de la plataforma en la que se ejecute. Dichas plataformas podrían incluir: sistema operativo, interfaces de usuario, bases de datos, etc. Muy a menudo, éste encapsulamiento no ha sido diseñado de antemano y un montón de declaraciones del tipo #ifdef con opciones para todas las plataformas soportadas comienzan a reproducirse como conejos a través del código.  
***Discusión***  
La factoría de objetos tiene la responsabilidad de proveer servicios de creación para toda la familia de plataformas. Los clientes nunca crean directamente los objetos de una plataforma; ellos le solicitan a la factoría que les proporcione la instancia que corresponda.  
Este mecanismo hace que el intercambio de familia de productos sea fácil, porque la la clase específica de la factoría de objetos aparece sólo una vez en la aplicación (cuando se crea la instancia). La aplicación puede reemplazar indiscriminadamente toda la familia de productos simplemente instanciando una diferente clase concreta de la factoría.  
Debido a que el servicio provisto por la factoría de objetos es tan difundido, que por lo general es implementado como un *Singleton*.  
***Estructura***  
La factoría define un *Factory Method* por producto. Cada *Factory Method*encapsula el operador *new* y la clase, plataforma específica o producto concreto. Cada plataforma es modelada con una clase *Factory* derivada.

[](http://4.bp.blogspot.com/-C-UqMvymWt4/TwZsOTNlGCI/AAAAAAAAASg/POKDqSwNj0A/s1600/Abstract_Factory.gif)

***Ejemplo***  
Recordemos que el propósito de una Abstract Factory es proveer una interfaz capaz de crear una familia de objetos relacionados **sin especificar clases concretas**. Este patrón se encuentra en el equipo de estampado de láminas de metal usado en la fabricación de automóviles japoneses. El equipo de estampado es una *Abstract Factory* que se encarga de crear las partes de un auto. La misma maquinaria es utilizada para hacer las puertas derecha e izquierda, los guardabarros delanteros derecho e izquierdo, capós, etc. para diferentes modelos de autos. Mediante el uso de rodillos para cambiar las matrices de estampado, las clases concretas producidas por la máquina pueden ser cambiadas en tres minutos.

[](http://1.bp.blogspot.com/-jEj86qFurgc/TwZsUqOz2iI/AAAAAAAAASo/6k4cGwsgCJI/s1600/Abstract_Factory_example1.gif)

***Check list***

1. Decidir si la "independencia de plataformas" y servicios de creación son la fuente actual de problemas.
2. Trazar una matriz de plataformas y productos.
3. Definir una interfaz que consista en un Factory Method por producto.
4. Definir una clase Factory derivada por cada plataforma que encapsule toda referencia al operador *new*.
5. El cliente debe retirar toda referencia al operador *new* y usar los *Factory Method* para crear los objetos.

***Reglas de oro***

* A veces los patrones creacionales compiten entre ellos: existen casos en los que tanto el patrón *Abstract Factory* como el *Prototype* pueden ser usados de forma productiva. En otras ocasiones son complementarios: *Abstract Factory* puede almacenar un conjunto de Prototypes de los cuales clonar y retornar los objetos, el patrón *Builder* puede usar uno de los otros patrones para implementar cual componente debe ser instanciado. *Abstract Factory*, *Prototype* y *Builder*, pueden usar *Simgleton* en sus implementaciones.
* Las clases *Abstract Factory* son a menudo implementadas con *Factory Methods*. Pero éstos también pueden ser implementados usando *Prototype*.
* *Abstract Factory* puede ser usado como una alternativa a *Facade* para ocultar las clases específicas de las plataformas.
* *Builder* hace incapié en la construcción de objetos complejos paso a paso, mientras que *Abstract Factory* hace énfasis en una familia de objetos (sean simples o complejos).
* A menudo, el diseño nace como un *Factory Method* (menos complicado, más personalizable, proliferan las subclases) y evolucionan hacia un *Abstract Factory*, *Prototype* o *Builder* (más flexible, más complejo) como el diseñador descubra cuánta flexibilidad es necesaria.

***Ejemplo en código C#***

1. using System;
2. class MainApp
3. {
4. public static void Main()
5. {
6. // Abstract factory #1
7. AbstractFactory factory1 = new ConcreteFactory1();
8. Client c1 = new Client(factory1);
9. c1.Run();
10. // Abstract factory #2
11. AbstractFactory factory2 = new ConcreteFactory2();
12. Client c2 = new Client(factory2);
13. c2.Run();
14. // Wait for user input
15. Console.Read();
16. }
17. }
18. // "AbstractFactory"
19. abstract class AbstractFactory
20. {
21. public abstract AbstractProductA CreateProductA();
22. public abstract AbstractProductB CreateProductB();
23. }
24. // "ConcreteFactory1"
25. class ConcreteFactory1 : AbstractFactory
26. {
27. public override AbstractProductA CreateProductA()
28. {
29. return new ProductA1();
30. }
31. public override AbstractProductB CreateProductB()
32. {
33. return new ProductB1();
34. }
35. }
36. // "ConcreteFactory2"
37. class ConcreteFactory2 : AbstractFactory
38. {
39. public override AbstractProductA CreateProductA()
40. {
41. return new ProductA2();
42. }
43. public override AbstractProductB CreateProductB()
44. {
45. return new ProductB2();
46. }
47. }
48. // "AbstractProductA"
49. abstract class AbstractProductA
50. {
51. }
52. // "AbstractProductB"
53. abstract class AbstractProductB
54. {
55. public abstract void Interact(AbstractProductA a);
56. }
57. // "ProductA1"
58. class ProductA1 : AbstractProductA
59. {
60. }
61. // "ProductB1"
62. class ProductB1 : AbstractProductB
63. {
64. public override void Interact(AbstractProductA a)
65. {
66. Console.WriteLine(this.GetType().Name + " interacts with " + a.GetType().Name);
67. }
68. }
69. // "ProductA2"
70. class ProductA2 : AbstractProductA
71. {
72. }
73. // "ProductB2"
74. class ProductB2 : AbstractProductB
75. {
76. public override void Interact(AbstractProductA a)
77. {
78. Console.WriteLine(this.GetType().Name + " interacts with " + a.GetType().Name);
79. }
80. }
81. // "Client" - the interaction environment of the products
82. class Client
83. {
84. private AbstractProductA AbstractProductA;
85. private AbstractProductB AbstractProductB;
86. // Constructor
87. public Client(AbstractFactory factory)
88. {
89. AbstractProductB = factory.CreateProductB();
90. AbstractProductA = factory.CreateProductA();
91. }
92. public void Run()
93. {
94. AbstractProductB.Interact(AbstractProductA);
95. }
96. }

## [Patrones de diseño de creación: Builder](http://programacionsolida.com.ar/2012/01/patrones-de-diseno-de-creacion-builder.html)

12

ene

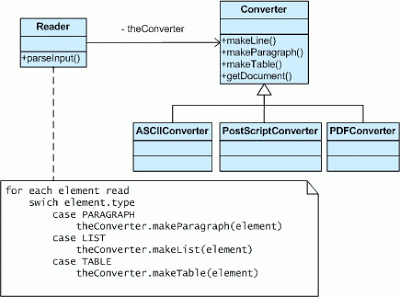
|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://4.bp.blogspot.com/-UE19zXSN1Gs/TwdYwFEnvII/AAAAAAAAAS8/oDMZFDUlFHI/s1600/Builder.JPG)

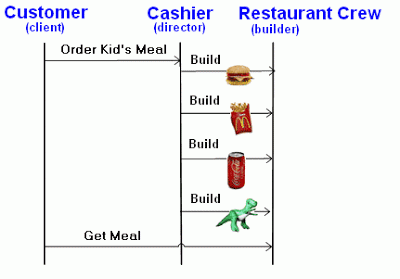
***Intención del patrón***

* Separar la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones.
* Procesar una representación compleja y crear uno de los diversos tipos de objeto.

***Ejemplo de problema***  
Una aplicación necesita crear los elementos de un conjunto complejo. La especificación para el conjunto existe en un almacenamiento secundario y una de las representaciones debe ser creada en el almacenamiento primario.  
  
***Discusión***  
Separar el algoritmo de interpretación (por ejemplo: leer y analizar) de un mecanismo de almacenamiento (ej. un archivo RTF) del algoritmo de construcción y representación de uno de muchos otros formatos (Ej. ASCII, TeX,etc.). El foco / diferencia está en la creación de los conjuntos complejos.  
El "director" invoca al servicio "*Builder*" para interpretar el formato externo. El *Builder* crea parte del objeto complejo cada vez que es llamado y mantiene todos los estados intermedios. Cuando el "producto" es terminado, el cliente recibe el resultado por parte del "*Builder*".  
***Estrcutura***  
El Lector encapsula el análisis de la entrada común. La jerarquía del Builder hace posible la creación polimórfica de muchas representaciones específicas.

[](http://4.bp.blogspot.com/-5RW36E3exPk/Twjtl4Zx2YI/AAAAAAAAATI/vjYaL32LSfg/s1600/BuilderEstructura.gif)

***Ejemplo***  
Recordemos que el patrón *Builder* separa la construcción de un objeto complejo de su representación y que el mismo proceso de construcción puede crear diferentes representaciones. Este patrón es usado por cadenas de Fast Food para hacer la comida de los niños. Dicha comida consiste típicamente en un ítem principal, una guarnición, una bebida y un juguete (Ej. Hamburguesa, papas fritas, una gaseosa y un dinosaurio de juguete). Puede haber variaciones en este conjunto que conforma una "cajita" para niños, pero el proceso que las "arma" es el mismo. Cuando un cliente solicita una hamburguesa, una hamburguesa con queso o bocaditos de pollo, el proceso es el mismo. El empleado del mostrador ordena a la cocina el ensamblado de un ítem principal, una guarnición y un juguete. Éstos elementos son puestos en una caja. La bebida es puesta en un vaso y permanece fuera de la caja. Este mismo proceso es usado en muchas cadenas de este tipo.

[](http://2.bp.blogspot.com/-JcJPcf7p42M/TwjtuPCQXlI/AAAAAAAAATU/ZrRC4d7MR4Y/s1600/Builder_example1.gif)

***Check list***

1. Decidir si una entrada común y muchas representaciones (salidas) posibles, es el problema en cuestión.
2. Encapsular el parseo de la entrada común en una clase *Reader*.
3. Diseñar un protocolo estándar para crear todas las posibles representaciones de salida. Capturar los pasos de dicho protocolo en una interfaz *Builder*.
4. Definir una clase *Builder* derivada por cada representación.
5. El cliente debe poseer un objeto *Reader* (director)y otro *Builder*. Y registrar el último con el primero.
6. El cliente le pide al *Reader* que "construya" al *Buidler*.
7. El cliente le solicita al Builder que le devuelva el producto que corresponda.

***Reglas de oro***

* Algunos patrones creacionales son complementarios: *Builder* puede utilizar otro de los otros patrones para implementar cuál producto debe construir.
* *Builder* hace foco en construir objetos complejos paso a paso y los retorna como paso final.
* *Builder*, a menudo, construye un *Composite*.
* A menudo, el diseño nace como un *Factory Method* (menos complicado, más personalizable, proliferan las subclases) y evolucionan hacia un *Abstract Factory*, *Prototype*o *Builder*(más flexible, más complejo) como el diseñador descubra cuánta flexibilidad es necesaria.

***Ejemplo en código C#***

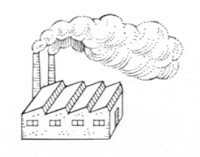
1. using System;
2. using System.Collections;
3. public class MainApp
4. {
5. public static void Main()
6. {
7. // Create director and builders
8. Director director = new Director();
9. Builder b1 = new ConcreteBuilder1();
10. Builder b2 = new ConcreteBuilder2();
11. // Construct two products
12. director.Construct(b1);
13. Product p1 = b1.GetResult();
14. p1.Show();
15. director.Construct(b2);
16. Product p2 = b2.GetResult();
17. p2.Show();
18. // Wait for user
19. Console.Read();
20. }
21. }
22. // "Director"
23. class Director
24. {
25. // Builder uses a complex series of steps
26. public void Construct(Builder builder)
27. {
28. builder.BuildPartA();
29. builder.BuildPartB();
30. }
31. }
32. // "Builder"
33. abstract class Builder
34. {
35. public abstract void BuildPartA();
36. public abstract void BuildPartB();
37. public abstract Product GetResult();
38. }
39. // "ConcreteBuilder1"
40. class ConcreteBuilder1 : Builder
41. {
42. private Product product = new Product();
43. public override void BuildPartA()
44. {
45. product.Add("PartA");
46. }
47. public override void BuildPartB()
48. {
49. product.Add("PartB");
50. }
51. public override Product GetResult()
52. {
53. return product;
54. }
55. }
56. // "ConcreteBuilder2"
57. class ConcreteBuilder2 : Builder
58. {
59. private Product product = new Product();
60. public override void BuildPartA()
61. {
62. product.Add("PartX");
63. }
64. public override void BuildPartB()
65. {
66. product.Add("PartY");
67. }
68. public override Product GetResult()
69. {
70. return product;
71. }
72. }
73. // "Product"
74. class Product
75. {
76. ArrayList parts = new ArrayList();
77. public void Add(string part)
78. {
79. parts.Add(part);
80. }
81. public void Show()
82. {
83. Console.WriteLine("\nProduct Parts -------");
84. foreach (string part in parts)
85. Console.WriteLine(part);
86. }
87. }

## [Patrones de diseño de creación: Factory Method](http://programacionsolida.com.ar/2012/02/patrones-de-diseno-de-creacion-factory.html)

06

feb

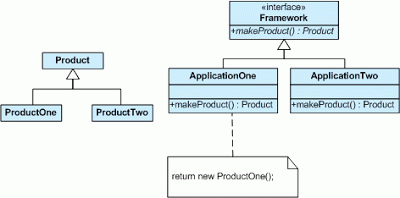
|  |  |  |
| --- | --- | --- |
|  |  |  |

[](http://3.bp.blogspot.com/-sHvSiaO1OPY/TwscgzleJ1I/AAAAAAAAAUM/5tHpbA4sjWM/s1600/FactoryMetho.JPG)

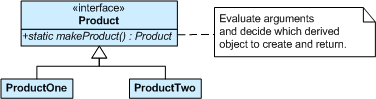
***Intención del patrón***

* Definir una interfaz para crear un objeto, pero dejar que las subclases decidan cuál clase instanciar. *Factory Method* permite a una clase diferir la instanciación a subclases.
* La definición de un constructor "virtual".
* Que el operador new sea considerado dañino.

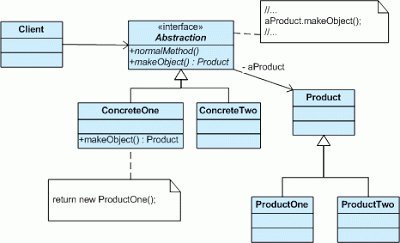
***Ejemplo de problema***  
Un *framework* necesita estandarizar el modelo de arquitectura para una variedad de aplicaciones, pero permitirle las aplicaciones definir de manera individual sus propios objetos de dominio y asegurar su instanciación.  
  
***Discusión***  
*Factory Method* sirve para crear objetos así como *Template Method* se utiliza para implementar un algoritmo. Una superclase especifica todos los comportamientos estándares y genéricos (usando "*placeholders*" puros para los pasos de la creación), y delega los detalles de la creación a las subclases que son suministradas por el cliente.  
*Factory Method* permite un diseño más personalizable y sólo un poco más complicado. Otros patrones de diseño requieren nuevas clases, mientras que éste sólo requiere una nueva operación.  
A menudo se usa éste patrón como la manera estándar de crear objetos; pero no sería necesario si: la clase que se instancia nunca cambia, o la instanciación toma lugar en un método que una subclase puede fácilmente sobrescribir (como un método de inicialización).  
Éste patrón es similar al patrón *Abstract Factory*, pero sin el énfasis en las familias de objetos.  
Los *Factory Methods* son especificados de manera rutinaria por un *framework* arquitectónico, y entonces es implementado por el usuario de dicho *framework*.  
***Estructura***  
La implementación de *Factory Method* discutida por el GoF, coincide en gran medida con el patrón *Abstract Factory*. Por esta razón, la presentación en este post se centra en el enfoque que lo ha vuelto popular desde entonces.

[](http://1.bp.blogspot.com/-SAuBQMqq_uw/TwsUyHnkdVI/AAAAAAAAATs/HROqLKTigLE/s1600/Factory_Method_Estructura1.gif)

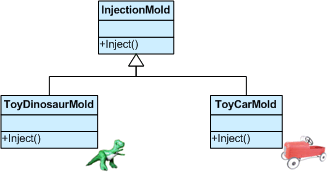
Una definición cada vez más popular de éste patrón es: un método *static* de una clase que retorna un objeto del mismo tipo. Pero a diferencia de un constructor, el objeto real que retorna puede ser la instancia de una subclase. Otra diferencia con un constructor, es que un objeto existente puede ser reutilizado en lugar de crear un nuevo objeto. También, pueden tener nombres diferentes y más descriptivos (Ej. Color.make\_RGB\_color(float red, float green, float blue) y Color.make\_HSB\_color(float hue, float saturation, float brightness))

[](http://1.bp.blogspot.com/-tMh-womEAW0/TwsUyq5aR_I/AAAAAAAAAT0/YNwAbmL_L-w/s1600/Factory_Method_Estructura2.gif)

El cliente queda totalmente desacoplado de los detalles de la implementación de clases derivadas. La creación polimórfica es ahora posible.

[](http://4.bp.blogspot.com/-r5Hx6XT1Hoc/TwsUz9qhVII/AAAAAAAAAT8/Es21R8-auI8/s1600/Factory_Method_Estructura3.gif)

***Ejemplo***  
El patrón *Factory Method* define una interfaz para la creación de objetos, pero le permite a las subclases decidir cuales clases instanciar. Las prensas de inyección de matrices demuestran este patrón. Los fabricantes de juguetes de plástico procesando el moldeado de polvo de plástico, y lo inyecta en los moldes con la forma deseada. La clase de juguete (Auto, muñeco, etc.) es determinado por el molde.

[](http://3.bp.blogspot.com/-MvfjCDjjjYc/TwsU0lHLuEI/AAAAAAAAAUE/f51MHPpd5p8/s1600/Factory_Method_example1.gif)

***Check list***

1. Si se tiene una jerarquía de herencia que posee polimorfismo, se debe considerar la capacidad polimórfica de creación mediante la definición de un *Factory Method* *static* en la clase base.
2. Diseñar los argumentos para el *Factory Method*. ¿Qué cualidades o características son necesarias y suficientes para identificar la clase derivada correcta para instanciar?
3. Considerar el diseño de un *Object Pool* interno que permita que los objetos sean reutilizados en lugar de crearlos desde cero.
4. Considerar que todos los constructores sean *private* o *protected*.

***Reglas de oro***

* Las *Abstract Factory* son a menudo implementadas con *Factory Methods*, pero ellos pueden ser implementados usando *Prototype*.
* Los Factory Method suelen ser llamados dentro de los *Template Method*.
* Factory Method: creación a través de la herencia. *Template Method*: creación a través de la delegación.
* A menudo, el diseño nace como un *Factory Method* (menos complicado, más personalizable, proliferan las subclases) y evolucionan hacia un *Abstract Factory*, *Prototype*o *Builder*(más flexible, más complejo) como el diseñador descubra cuánta flexibilidad es necesaria.
* *Prototype* no requiere subclases, pero necesita una operación de inicialización. *Factory Method* requiere subclases, pero no necesita ser inicializado.
* La ventaja de un Factory Method es que puede retornar la misma instancia varias veces, o puede retornar una subclase en lugar de un objeto del mismo tipo.
* Algunos defensores del *Factory Method* recomiendan como una cuestión del lenguaje de diseño (o en su defecto, como una cuestión de estilo), absolutamente todos los constructores deberían ser *private* o *protected*. No es incumbencia de nadie cuándo una clase crea un objeto o si un objeto viejo reciclado.
* El operador *new* se considera como algo dañino. Es diferente solicitar un objeto que crear uno. El operador *new* siempre crea un objeto y falla la encapsulación de la creación del mismo. Un *Factory Method* fuerza dicha encapsulación y le permite a un objeto ser solicitado evitando el acoplamiento en el acto de la creación del mismo.

***Ejemplo de código en C#***

1. using System;
2. using System.Collections;
3. class MainApp
4. {
5. static void Main()
6. {
7. // An array of creators
8. Creator[] creators = new Creator[2];
9. creators[0] = new ConcreteCreatorA();
10. creators[1] = new ConcreteCreatorB();
11. // Iterate over creators and create products
12. foreach(Creator creator in creators)
13. {
14. Product product = creator.FactoryMethod();
15. Console.WriteLine("Created {0}",
16. product.GetType().Name);
17. }
18. // Wait for user
19. Console.Read();
20. }
21. }
22. // "Product"
23. abstract class Product
24. {
25. }
26. // "ConcreteProductA"
27. class ConcreteProductA : Product
28. {
29. }
30. // "ConcreteProductB"
31. class ConcreteProductB : Product
32. {
33. }
34. // "Creator"
35. abstract class Creator
36. {
37. public abstract Product FactoryMethod();
38. }
39. // "ConcreteCreator"
40. class ConcreteCreatorA : Creator
41. {
42. public override Product FactoryMethod()
43. {
44. return new ConcreteProductA();
45. }
46. }
47. // "ConcreteCreator"
48. class ConcreteCreatorB : Creator
49. {
50. public override Product FactoryMethod()
51. {
52. return new ConcreteProductB();
53. }
54. }