Introducción
a Design Patterns

Por
Fernando Dodino
Nicolás Passerini

Versión 2.1
Mayo 2007

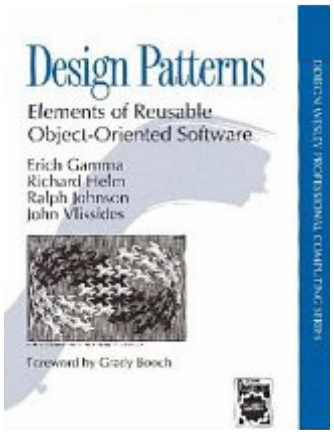
Indice

Y Él le dijo al Programador: «Yo soy el Gamma y el Omega.
Sube a encontrarte conmigo en el monte, y quédate allí.
Voy a darte las tablas con la ley
y los patterns que he escrito para guiarlos en la vida.»
Éxodo 24:11-13

¿Qué es un Design Pattern?

Un Design Pattern:

- "es una regla que expresa la relación entre un contexto, un problema y una solución" (Christopher Alexander, creador de Patterns para la Ingeniería Civil)



- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (nuevamente Christopher Alexander, según el libro *Design Patterns del Gang of Four –algo así como los cuatro fantásticos: Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides*-). A partir de aquí nos referiremos a esta bibliografía como *el libro de Gamma*.

- "Una solución (probada) a un problema en un determinado contexto" (Erich Gamma)
- "A Design Pattern names, asbtracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design." (nuevamente, Erich Gamma dixit)

Partiendo de estas definiciones, definimos qué debe contener un pattern:

Un **nombre** que describe el problema. Esto nos permite:

- Tener un vocabulario de diseño común con otras personas. *Un pattern se transforma en una **herramienta de comunicación** con gran poder de simplificación*[1]

- Por otra parte, **logramos un nivel de abstracción mucho mayor**. De la misma manera que una lista doblemente enlazada define cómo se estructura el tipo de dato y qué operaciones podemos pedirle, el pattern trabaja una capa más arriba: define un conjunto de objetos/clases y cómo se relacionarán entre sí (resumiéndolo en una palabra). Ya existían conceptos similares en la programación estructurada: el *apareo* y el *corte de control* (definían la forma de encarar la solución).

El **problema** define cuándo aplicar el pattern, *siempre que el contexto lo haga relevante*.

La **solución** contiene un *template* genérico de los elementos que componen el diseño, sus responsabilidades, relaciones y colaboraciones. Un pattern no es instanciable per se, la abstracción representada en el problema debe aplicarse a nuestro dominio.

En las **consecuencias** se analiza el impacto de aplicar un pattern en la solución, tanto a favor como en contra.

[1] Se puede estudiar el gato "El explicado", de Les Luthiers para recordar la importancia de los nombres: http://www.atame.org/l/les_luthiers/el_explicado.shtml

¿Qué no es un Design Pattern?

Un Design Pattern:

- **No** es garantía de un sistema bien diseñado. Tengo las respuestas, pero me falta saber si hice la pregunta correcta.

- Es un buen punto de partida para pensar una solución, **no la** solución. Al ser una herramienta, no puede reemplazar al diseñador, que es quien maneja la herramienta.

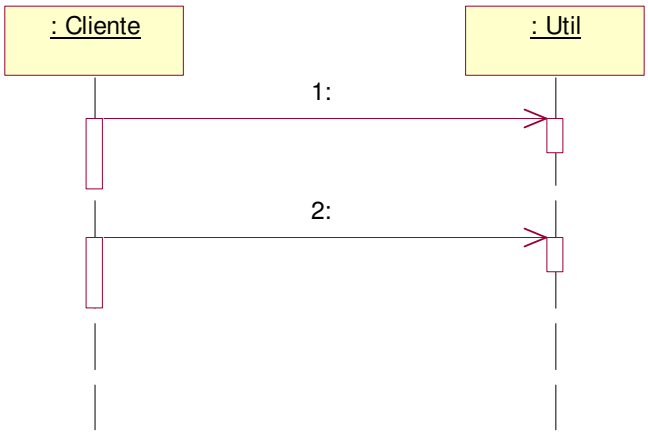
- De la misma manera que los estudiantes de psicología encuentran rasgos de neurosis obsesiva entre sus parientes, amigos e incluso en sí mismos, el estudiante de patterns suele querer "descubrir" patrones en su solución aún cuando no siempre se justifique. Entonces aparecen signos de sobrediseño: abuso por prever todo tipo de escenarios. El mismo Erich Gamma nos brindará consejos más adelante sobre cómo **no** usar los DP.

- Está en la página anterior, pero vale la pena recalcar que un pattern **no es instanciable**, y **no es dependiente de un dominio**. Es el bosquejo de una idea que ha servido en otras ocasiones, para una problemática similar. Representa una unidad de abstracción mayor a simples líneas de código.

Librería, Framework y Pattern

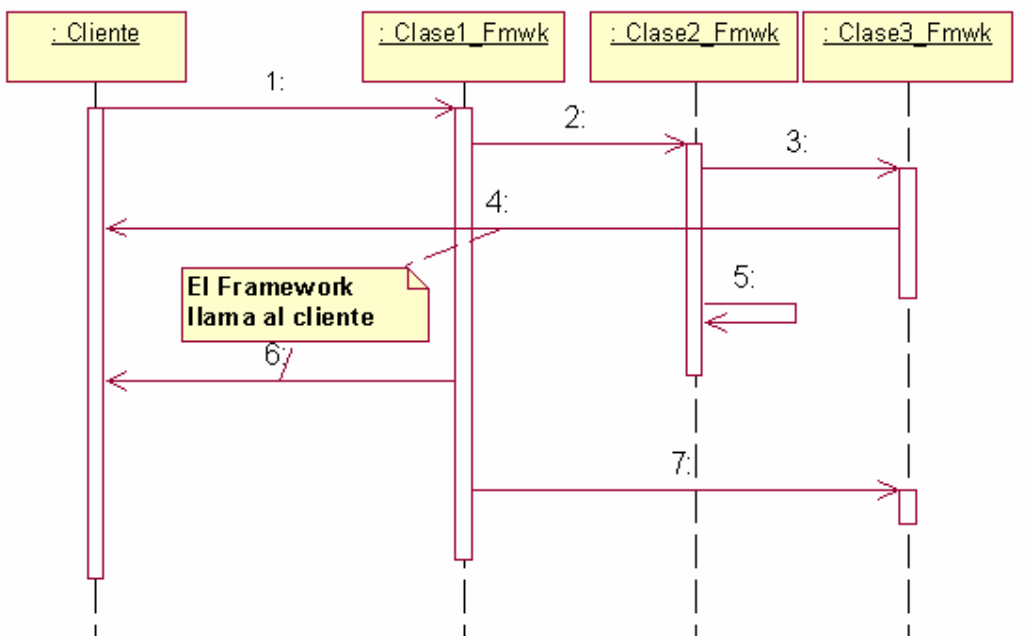
Antes de continuar pasamos en limpio los tres términos[2]:

- **Librería:** es un conjunto de funciones llamadas desde "afuera" por un cliente. Dentro del POO, esto es una clase que recibe un mensaje, lo ejecuta y devuelve luego el control al cliente. Podemos agregar que la instanciación de una librería es relativamente sencilla.



- **Framework:** representa una abstracción de diseño y tiene un comportamiento en sí. No es solamente una clase, sino que es un conjunto de objetos que se relacionan para servir a un dominio específico. El cliente puede usar el framework subclasificando o componiendo sus propias clases con las clases del framework y entonces *el código del framework es el que llama al código cliente*. La instanciación del framework no es tan sencilla, ya que requiere un conocimiento del mismo.

[2] La definición de Librería vs. Framework está auspiciada por Martin Fowler en http://martinfowler.com/bliki/InversionOfControl.html



- Por último, los **Patterns** tienen un nivel de abstracción superior al framework, como dijimos antes no tienen un dominio específico sino que son soluciones generales, aunque por ese mismo motivo necesitan aplicarse en un contexto para poder ser implementados.

¿Por qué Design Patterns?

Desde tiempos inmemoriales la industria del software ha tratado de lidiar con la complejidad intrínseca del problema de construir sistemas intentando llegar a una perspectiva ingenieril del problema.

En determinado punto de la historia la mayoría de los componentes de la industria estaban de acuerdo con esa perspectiva, vinculada a un desarrollo en cascada, haciendo énfasis en que los procesos están por encima de las personas y viendo al desarrollo de software como una cadena de montaje.

Tarde o temprano, las personas vinculadas al desarrollo fueron descubriendo (o descubrirán algún día) que ese esquema no daba los resultados esperados. Que la visión de la cadena de montaje no aplica a la producción de software y principalmente que los ciclos de vida en cascada no aplican a nuestra industria. La metáfora de la ingeniería civil tampoco fue suficiente.

Ante ese panorama, se abrieron dos caminos, el primero sugiere hacer algunas correcciones al modelo de ingeniería original (diríamos que correcciones menos radicales de lo que a veces se pretende hacernos creer, pero esa es otra discusión) y se propusieron otros modelos más o menos ingenieriles.

El segundo camino, diametralmente opuesto, sugiere que el software sólo puede construirse de manera artesanal y propone no confiar en los procesos, dando una importancia mucho mayor a las personas en el desarrollo.

Existiendo argumentos para ambas líneas de pensamiento, entendemos que esa discusión sólo es mantenible a futuro. Es decir, que la esperanza de que la ingeniería nos brinde un método (una 'receta') que nos diga de una vez y para siempre cómo se hace para construir software no tiene asidero para el estado actual del arte de construir sistemas de software.

Desde nuestro punto de vista, la gran mayoría de los proyectos exitosos hoy en día depende -para el horror de los ingenieros en software- en un altísimo porcentaje de la capacidad de las personas y no de la metodología que se utilice y tampoco se puede decir que esas personas tampoco sigan un método definido sino que sus decisiones se basan principalmente en la experiencia y en la intuición.

Por lo tanto, en la situación actual el trabajo del ingeniero consiste en formalizar la forma en que se toman esas decisiones e ir generando herramientas a partir de ese proceso de formalización. En todos los aspectos de la ingeniería se puede ver ese mecanismo, en lugar de tener un 'proceso' o 'metodología' lo que tenemos son 'patterns' y 'best practices'.

Pero, ¿qué es una best practice? Una best practice es un esfuerzo por formalizar una actividad que anteriormente se dio en forma natural, describir su utilidad, sus características y debilidades, con el objetivo de poder reutilizarla donde sea conveniente. Cuando descubrimos algo que nos da resultado, tratamos de repetirlo. Cuando descubrimos que se puede además dar resultado en múltiples ocasiones, encontramos la oportunidad de ponerle un nombre y formalizar la idea.

Un design pattern es entonces la intención por formalizar una práctica de diseño, una herramienta para diseñar. No tenemos hoy un mecanismo que nos permita pasar de un conjunto de requerimientos a un diagrama de clases o de secuencia ni nada que se le parezca. Cualquiera que haya tratado de utilizar los mecanismos pseudoautomáticos propuestos por las grandes consultoras del tema, habrá descubierto que no tienen nada de automático.

Tal vez en algún momento la industria madure lo suficiente para encontrar un proceso totalmente ingenieril que nos permita realizar esa transformación en 7 simples pasos. Hasta ahora eso no ha ocurrido y tal como lo predijera Fred Brooks[3] en 1986, seguimos sin encontrar 'silver bullets': el diseño es un proceso arduo, manual, artesanal e iterativo. Ante la ausencia de ese proceso mágico, lo que nos queda es ayudar a nuestras pruebas y errores (iterativo tiene mucho de "a prueba y error") con algunas 'ideas piolas que vemos que pueden calzar en muchos lugares'.

A esas ideas piolas le ponemos nombres para extender nuestro lenguaje. A veces nos sirven 'out of the box' del libro a nuestro diagrama de clases, en otras oportunidades sólo sirven como disparadoras de otras ideas nuevas.

¿Por qué damos Design Patterns en TADP?

- Porque abre la mente, nos da la capacidad de explorar mejoras a nivel de diseño (no sólo estamos tirando código sino que nos paramos una capa más arriba).

- Porque al adaptarnos al vocabulario de la comunidad ganamos tiempo. No es lo mismo
    - Hacer un diseño donde nos damos cuenta que hay diferentes criterios para calcular punitorios por no pagar un crédito, tener que comunicar esa misma idea al equipo de desarrollo diciendo “Debemos subclasificar los criterios, pero me parece que sería bueno despegarlo del cliente para que no queden acoplados el cliente y el criterio que tenemos para calcularle punitorios por no pago, etc. etc.”
    - Decir “El cliente tiene un strategy para calcular punitorios”. Pudimos transmitir la misma idea con menos palabras: logramos mayor expresividad llevando la discusión a un nivel mucho más alto y conceptual.

- Nos obliga a ir adelante y atrás en el diseño y no pensar exclusivamente en una solución de compromiso para hoy. Esto no sólo produce diseños más elegantes, sino que así también aprendo a equilibrar hasta cuándo dejo la puerta abierta para cambios y hasta cuándo cierro una solución para poder implementarla.

- Es una forma de documentación. Muchas veces nos ha pasado tener un dejà vu y no recordar en qué ocasión hicimos algo parecido. Ponerle un nombre también ayuda a documentar nuestro trabajo y a comunicarlo a los demás.

[3] Frederick P.Brooks Jr., ingeniero de software que escribió el ensayo “***No Silver Bullet - essence and accidents of software engineering***” donde pronosticó las limitaciones de las tecnologías a la hora de resolver las complejidades intrínsecas de un problema. Recomendamos su lectura en:
http://www-inst.eecs.berkeley.edu/~maratb/readings/NoSilverBullet.html
Profundo investigador del proceso de desarrollo de software, es autor de una frase conocida como La Ley de Brooks: “Agregar gente a un proyecto retrasado retrasa el proyecto”. La justificación no podría ser más clara: “No podemos esperar que nueve mujeres puedan tener un bebé en un mes”.

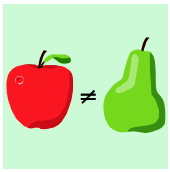
¿Cómo aplicar los Design Patterns?

En http://hillside.net/patterns/onlinepatterncatalog.htm tenemos un catálogo actualizado de patterns, además de los 23 originales del libro del Gang of Four. Evitando caer en recetas metodológicas[4], hay dos maneras de abordar los patrones de diseño:

- Teniendo un problema concreto que resolver (la mejor motivación)
- Como lectura de interés para situaciones futuras (me sirve para guardar en la caja de herramientas del diseñador).

En el caso de diseñar alguna aplicación, nos puede pasar que:

1. necesitemos resolver algún caso no trivial y no sabemos cómo



2. sabemos cómo resolverlo pero queremos contrastar nuestra solución con otra que ya funcionó en otros casos (y aun así nos podemos seguir quedando con nuestra idea original, solo que ahora con más fundamento*). Lo más importante del diseño son las preguntas,* y aquí se abren muchos caminos posibles: ¿mi solución encaja con algún pattern existente? ¿o son más de uno relacionados? ¿mi solución está bien para hoy pero no contempla situaciones que se van a dar a corto plazo? etc. etc.

Estudiar los Design Patterns es fácil. Lo difícil es saber qué patterns calzan en mi solución para darle al sistema robustez y flexibilidad, y para eso como decíamos antes no existe ninguna herramienta automática. Sólo podemos tener en cuenta algunos consejos:

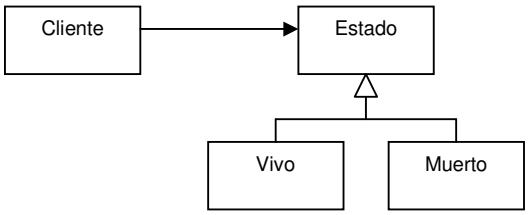
- Cada Pattern tiene tres secciones que describen el problema que resuelve: Intent, Motivation y Applicability. Leer estas secciones nos ayudará a comprender si la naturaleza del problema encaja con la solución propuesta por el/los pattern/s.

- Los patterns se agrupan en tres grandes categorías:
  i. Creacionales: abstraen el proceso de instanciación,
  ii. Estructurales: se ocupan de generar estructuras entre clases y objetos, se estudian con los diagramas de clases/objetos y
  iii. De Comportamiento: se encargan de la asignación de responsabilidades entre objetos y cómo se comunican entre sí, se estudian con diagramas de secuencia/colaboración.

  Además, muchos patterns están relacionados entre sí, lo que afortunadamente está documentado y nos ayuda en la elección.

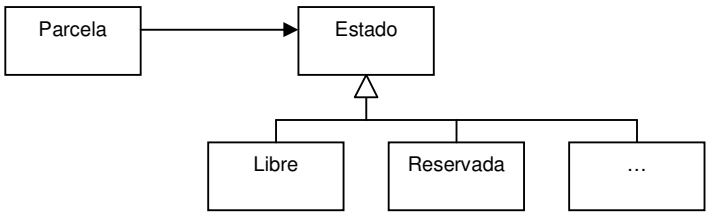
- **No todo puede ser variable en el diseño**. De ser así, generaríamos una pieza de software tan abstracta que no serviría para ningún negocio. *¿Qué nos interesa que el día de mañana pueda cambiar?* Gamma recomienda concentrarse en *encapsular la parte más variable* de nuestra aplicación.
  Ejemplo: "En una funeraria viene un cliente y solicita una parcela…"

[4] El lector interesado en propuestas metodológicas puede abordar "Instrucciones para dar cuerda a un reloj" e "Instrucciones para subir una escalera" del libro *Historia de Cronopios y de Famas*, de Julio Cortázar



Va a ser difícil probar que el State Pattern calce en esta aplicación, teniendo en cuenta que es difícil que varíe el estado de un cliente (vivo, muerto, ¿y el día de mañana?).

Distinto puede ser si quiero trabajar el estado de una parcela:



siempre que el dominio lo justifique.

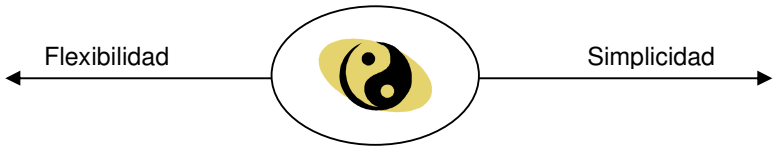
Una vez elegido el pattern, lo estudiamos y generamos el diagrama de clases dándole nombres representativos a las clases en base al contexto. Para elegir los nombres, a veces conviene utilizar como sufijo el nombre del rol que cumple la clase dentro del pattern: `DepositoObserver`, es el objeto encargado de ser notificado ante cualquier cambio que sufra el depósito.

De todas maneras no siempre es recomendable ensuciar la semántica de una clase cuando el nombre de ésta es lo suficientemente representativo: `DepositoSubject` no parece tan buena elección como Depósito (si al fin de cuentas es el objeto de negocio representado). Además, una clase puede estar participando en más de un pattern a la vez.

- En las páginas 24 y 25 del libro de Gamma se enumeran ocho causas comunes de rediseño, junto con los patterns que trabajan sobre esos problemas; *recomendamos ampliamente su lectura*.

Cómo no usarlos (Consejo de Gamma)

Cada vez que decidimos flexibilizar una parte de nuestro sistema, estamos agregando complejidad al diseño y esto redunda en un mayor costo de implementación. Aparecen fuerzas contrapuestas:

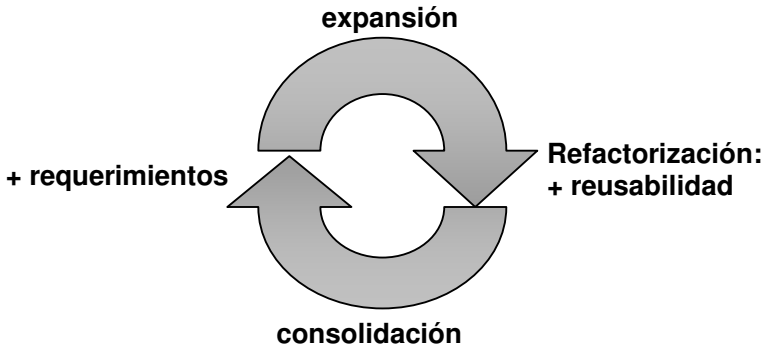


Los Design Patterns no deben ser usados indiscriminadamente. Si bien logramos mayor flexibilidad, también agregamos niveles de indirección que pueden complicar el diseño y/o bajar la performance. Un patrón de diseño sólo debería usarse cuando la flexibilidad pague su costo.

¿Cuándo aplicar Design Patterns?

¿En qué ciclo de desarrollo debería aplicarlos?

- En esa sutil diferencia entre el qué y el cómo, el Análisis queda fuera de juego (todavía estoy conociendo el dominio y definiendo la funcionalidad del sistema con mi cliente).
- A favor de la etapa de Diseño es que son "Design Patterns". En la fase de Diseño todavía me falta experiencia en el dominio, pero puedo buscar lo que puede cambiar en el sistema con ayuda de algún usuario experto.
- ¿En la programación? Mmm, si al leer el código que vamos escribiendo notamos que aparece un pattern no diseñado es que tenemos que actualizar el diseño.
- Veamos qué sucede en la etapa de Mantenimiento[5]…



Como concepto novedoso, la **refactorización** plantea que para que un sistema pueda evolucionar, necesita dos fases:

1. Aceptar los requerimientos de los usuarios. Esto lleva un período de expansión del software.

2. En algún momento, el sistema necesita *detener todos los requerimientos* y "acomodarse", volverse más flexible para poder convertirse en una pieza de software lo suficientemente maleable para volver a aceptar nuevos requerimientos. Este proceso se conoce como "refactorización".

Cada espiral de refactorización de un sistema es un buen momento para hacer mi diseño más general; como ya tengo cierto know-how del dominio, aquí es donde más fácil puedo determinar si aplicar (o descartar) patterns.

- Siempre… que sea necesario.

[5] Extractado de Erich Gamma et.al, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley, pág. 354