

# ¿Ha muerto el diseño?

### [Martin Fowler](http://martinfowler.com/) Chief Scientist, [ThoughtWorks](http://www.thoughtworks.com/)

Texto original: [Is Design Dead?](http://www.martinfowler.com/articles/designDead.html)

Ultima revisión significativa: [Mayo 2004](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc14)

[Traducción](http://www.programacionextrema.org/cgi-bin/wiki.pl?TraduccionIsDesignDead): [Alejandro Sierra](http://www.programacionextrema.org/cgi-bin/wiki.pl?AlejandroSierra), marzo de 2003, revisada en septiembre de 2004.

*Para algunas personas que sólo han tenido un contacto breve con la Programación Extrema, pareciera que la XP convoca a la muerte del diseño de software. No solamente se ridiculiza a la actividad de diseño como "Big Up Front Design", sino que técnicas como UML, marcos flexibles e incluso patrones son menospreciados o simplemente ignorados. De hecho la XP involucra mucho diseño, pero lo hace de una manera diferente a la de los procesos de software establecidos. La XP ha rejuvenecido la noción de diseño evolutivo con prácticas que permiten a la evolución ser una estrategia de diseño viable. También brinda nuevos retos y habilidades pues los diseñadores necesitan aprender cómo hacer diseño simple, cómo usar refactoración para mantener el diseño limpio y cómo usar patrones en un estilo evolutivo.*

(Este artículo fue mi ponencia en la conferencia XP 2000 y fue publicada como parte de las memorias.)

* [Diseño Planeado y Evolutivo](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc1)
* [Las Prácticas Habilitadoras de la XP](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc2)
* [El Valor de la Simplicidad](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc3)
* [Finalmente Qué Demonios es Simplicidad](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc4)
* [La Refactoración Viola YAGNI](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc5)
* [Patrones y XP](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc6)
* [Creciendo una Arquitectura](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc7)
* [UML y XP](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc8)
* [Sobre la Metáfora](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc9)
* [¿Quieres ser Arquitecto cuando seas grande?](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc10)
* [Reversibilidad](http://www.programacionextrema.org/articulos/designdead.es.html#N101B3)
* [El Deseo de Diseñar](http://www.programacionextrema.org/articulos/designdead.es.html#N101C5)
* [Cosas que son difíciles de refactorar](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc11)
* [¿Está Ocurriendo el Diseño?](http://www.programacionextrema.org/articulos/designdead.es.html#N101FD)
* [¿Así que ha Muerto el Diseño?](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc12)
* [Reconocimientos](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc13)
* [Revisión](http://www.programacionextrema.org/articulos/designdead.es.html#tth_sEc14)

La [Programación Extrema](http://www.extremeprogramming.org) (XP por sus siglas en inglés) desafía muchos de los presupuestos comunes acerca del desarrollo de software. La más controversial es el rechazo a un esfuerzo significativo en el diseño previo, en favor de un estilo más evolutivo. Para sus detractores, esto es un retorno al desarrollo "codificar y corregir" - usualmente denostado como *hackear*. Para sus fans esto es frecuentemente visto como un rechazo a técnicas de diseño (tal como el UML), principios y patrones. No preocuparse por el diseño, si escuchas tu código un buen diseño aparecerá.

Me encuentro en el centro de este debate. Gran parte de mi carrera ha involucrado lenguajes gráficos de diseño -el Unified Modeling Language (UML) y sus seguidores- y patrones. Realmente he escrito libros tanto sobre UML como sobre patrones. ¿Significa mi adhesión a la XP una renuncia a todo lo que he escrito sobre esos temas, limpiando mi mente de todas esas nociones contrarrevolucionarias?

Bueno, no puedo prolongar más el suspenso. La respuesta corta es no. La larga es el resto de este artículo.

## Diseño planeado y evolutivo

Voy a describir dos estilos de cómo se diseña en desarrollo de software. Quizá el más común es el diseño evolutivo. Esencialmente, evolutivo significa que el diseño del sistema crece conforme se implanta el sistema. El diseño es parte del proceso de programación y conforme el programa evoluciona el diseño cambia.

En su uso común, el diseño evolutivo es un desastre. El diseño acaba siendo la agregación de una sarta de decisiones tácticas ad-hoc, cada una de las cuales hace el código más difícil de modificar. Se podría alegar que eso no es diseño, ciertamente suele llevar a un diseño pobre. Como indica Kent, el diseño está para permitir cambiar el software fácilmente a largo plazo. Conforme el diseño se deteriora, igualmente se deteriora la capacidad de cambio. Se tiene el estado de entropía de software, conforme pasa el tiempo el diseño empeora y empeora. Esto no solo hace el software más difícil de cambiar, también facilita la generación de *bugs* y dificulta el encontrarlos y eliminarlos con seguridad. Esta es la pesadilla de "codifica y corrige", donde los bugs devienen exponencialmente más costosos de arreglar conforme el proyecto avanza.

El diseño planeado es todo lo contrario, y contiene nociones nacidas de otras ramas de la ingeniería. Si usted quiere construir la casa de su perro, puede simplemente tomar unas tablas y construir una forma ruda. Si quiere construir un rascacielos, no puede hacerlo de la misma manera - se caería antes de terminar siquiera la mitad. Así que empieza con dibujos ingenieriles, hechos en un estudio ingenieril como en el que trabaja mi esposa en el centro de Boston. Conforme hace el diseño ella se figura todo el asunto, en parte por análisis matemático pero principalmente usando códigos de construcción. Los códigos de construcción son reglas acerca de cómo diseñar estructuras con base en la experiencia de qué es lo que funciona bien (y algo de matemáticas). Una vez hecho el diseño, su compañía ingenieril puede pasar el diseño a otra empresa que lo construya.

El diseño planeado en software debería funcionar de la misma manera. Los diseñadores piensan los grandes problemas con anticipación. No necesitan programar porque no están construyendo el software, sólo lo están planeando. Así que pueden usar técnicas de diseño como el UML que deja de lado algunos de los detalles de la programación y permite a los diseñadores trabajar a un nivel más abstracto. Una vez hecho el diseño, pueden pasarlo a otro grupo (o a otra compañía) que lo construya· Ya que los diseñadores están pensando en una escala mayor, pueden evitar las series de decisiones tácticas que llevan a la entropía del software. Los programadores pueden seguir la dirección del diseño y, dado que siguen el diseño al pie de la letra, tener un sistema bien construido.

Ahora bien, el diseño planeado ha estado allí desde los 70s, y mucha gente lo ha usado. Es mejor en muchas formas que el diseño evolutivo de codificar y corregir. Pero tiene algunas fallas. La primera es que es imposible pensar en todos los problemas que se necesitan tratar cuando se programa. Es inevitable que al programar se encuentren cosas que ponen en entredicho el diseño. Si los diseñadores ya acabaron y ya están en otro proyecto, ¿qué pasa? Los programadores empiezan a codificar en torno al diseño y la entropía aparece. Aún si el diseñador no se ha ido, lleva tiempo organizar los problemas de diseño, cambiar los dibujos y alterar el código. Usualmente se hace una corrección rápida por la presión de tiempo. Entropía (otra vez).

Además frecuentemente hay un problema cultural. Los diseñadores se han formado por su habilidad y experiencia, pero están tan ocupados trabajando en diseños que ya no tienen tiempo para programar. Sin embargo las herramientas y materiales de desarrollo de software cambian rápidamente. Cuando dejas de programar no solo pierdes los cambios que ocurren en este flujo tecnológico, también pierdes el respeto de los que sí programan.

Esta tensión entre constructores y diseñadores también existe en la construcción, pero es más intensa en el software. Es intensa porque hay una diferencia clave. En construcción hay una división clara de habilidades entre los que diseñan y los que construyen, pero en software no es tanto así. Cualquier programador trabajando en ambientes de alto diseño necesita ser muy hábil. Lo suficientemente hábil para cuestionar los diseños del diseñador, especialmente cuando el diseñador sabe menos acerca de las realidades diarias de la plataforma de desarrollo.

Ahora bien, estos problemas pueden corregirse. Podríamos manejar la tensión humana. Quizá podemos conseguir diseñadores tan hábiles para tratar con la mayoría de los problemas y tener un proceso suficientemente disciplinado para cambiar los dibujos. Aún hay otro problema: los requerimientos cambiantes. Ese es el problema número uno, causante de dolores de cabeza en los proyectos de software en que he participado.

Una manera de controlar los requerimientos cambiantes es añadir flexibilidad en el diseño de modo que se pueda cambiar fácilmente conforme cambian los requerimientos. Esto requiere perspicacia en la clase de cambios esperados. Un diseño puede planearse para tratar con áreas de volatilidad, pero mientras eso ayuda con cambios de requerimientos previstos, no ayuda (y puede dañar) con cambios imprevistos. Así que los requerimientos tienen que entenderse lo suficientemente bien para separar las áreas volátiles, y en mi experiencia eso es muy difícil.

Algunos de estos problemas de requerimientos se deben a la falta de un entendimiento claro de los mismos. Mucha gente se enfoca en los procesos de ingeniería de requerimientos con la esperanza de que esto prevendrá la necesidad de cambiar el diseño más tarde. Pero aún esta receta puede no llevar a la cura. Muchos requerimientos cambiantes imprevistos ocurren debido a cambios en el negocio. Esos no pueden prevenirse, no importa cuan cuidadoso sea el proceso de ingeniería de requerimientos.

Esto hace sonar imposible al diseño planeado. Ciertamente estos son grandes retos. Pero no me inclino a afirmar que el diseño planeado es peor que el evolutivo, bajo el estilo "codifica y corrige". De hecho prefiero el diseño planeado. No obstante estoy consciente de los problemas del diseño planeado y busco una dirección nueva.

## Las Prácticas Habilitadoras de la XP

La XP es controversial por muchas razones, pero una de las banderas rojas clave de la XP es que aboga por el diseño evolutivo en lugar del diseño planeado. Como sabemos, el diseño evolutivo no puede funcionar debido a decisiones ad hoc y la entropía de software.

El meollo para entender este argumento es la curva de cambio del software. La curva del cambio dice que mientras más avanza el proyecto, es exponencialmente más caro hacer cambios. La curva del cambio se expresa usualmente en términos de las fases "un cambio hecho en el análisis por $1 cuesta miles en producción". Esto es irónico ya que la mayoría de los proyectos aún trabajan en un proceso ad-hoc que no tiene una fase de análisis, pero las exponenciaciones aún están allí. La curva exponencial del cambio significa que el diseño evolutivo no puede funcionar. También explica por qué el diseño planeado debe ser hecho cuidadosamente porque cualquier error encara la misma exponenciación.

La suposición fundamental de la XP es que es posible aplanar la curva del cambio lo suficiente como para hacer que funcione el diseño evolutivo. Este aplanamiento es a la vez permitido y explotado por la XP. Esto es parte del enganche de prácticas en XP: específicamente no puedes hacer todas las cosas que explotan la curva aplanada sin hacer las cosas que permiten aplanarla. Esta es una fuente común de controversia sobre la XP. Mucha gente critica la explotación sin entender la habilitación. Frecuentemente las críticas surgen de la propia experiencia de los críticos que no hicieron las prácticas habilitadoras que permiten funcionar a las prácticas explotadoras. Como resultado se quemaron y cuando ven la XP recuerdan el fuego.

En el centro de las prácticas habilitadoras están las de Pruebas e Integración Continua. Sin la seguridad dada por las pruebas el resto de la XP sería imposible. La integración continua es necesaria para mantener al equipo en sincronía, de modo que puedas hacer un cambio sin preocuparte de integrarla con otra gente. Juntas, estas prácticas pueden tener un gran efecto en la curva del cambio. Me acordé de esto otra vez aquí en ThoughtWorks. La introducción de Pruebas e Integración Continua ha marcado un mejoramiento en el esfuerzo de desarrollo. Ciertamente suficiente para cuestionar seriamente la afirmación de la XP de que necesitas todas las prácticas para lograr una gran mejoría.

La refactoración tuvo un efecto similar. La gente que refactora su código de la manera disciplinada sugerida por la XP encuentra una diferencia significativa en la efectividad comparada a hacer una reestructuración relajada más ad-hoc. Esa fue ciertamente mi experiencia una vez que Kent me enseñó a refactorar propiamente. Después de todo, sólo un cambio así de fuerte me habría motivado a escribir un libro entero sobre el tema.

Jim Highsmith, en su excelente [resumen de la XP](http://www.cutter.com/ead/ead0002.html), usa la analogía de un conjunto de escalas. En una bandeja está el diseño planeado, en la otra la refactoración. En propuestas más tradicionales el diseño planeado domina porque se supone que no puedes cambiar de idea más tarde. Conforme baja el costo del cambio puedes hacer más con tu diseño más tarde como refactoración. El diseño planeado no se esfuma, solo que ahora hay un balance entre los dos acercamientos. Para mi, siento que antes de la refactoración estaba diseñando con una sola mano.

Estas prácticas habilitadoras de integración continua, pruebas y refactoración, crean un nuevo ambiente que hace plausible el diseño evolutivo. Sin embargo algo que aún no imaginamos es el lugar del punto de equilibrio. Estoy seguro de que, a pesar de la impresión externa, XP no es sólo prueba, codifica y refactora. Hay espacio para el diseño antes de codificar. Algo de esto ocurre antes de que se haya código, la mayor parte ocurre en las iteraciones antes de codificar una tarea particular. Pero hay un nuevo balance entre diseño *up-front* y refactoración.

## El valor de la Simplicidad

Dos de los más grandes gritos de batalla de la XP son los eslogans "Haz la cosa más simple que pueda funcionar" y "No lo vas a necesitar" (conocido como YAGNI por sus siglas en inglés). Ambos son manifestaciones de la práctica XP del diseño simple.

La manera en que usualmente se describe YAGNI, es que no deberías añadir hoy código que sólo será usado por alguna característica que será necesaria mañana. En principio esto suena simple. El problema viene con cosas tales como marcos (*frameworks*), componentes reusables, y diseño flexible. Tales cosas son complicadas de construir. Se paga un costo up-front extra para construirlas, en la expectativa de que recuperarás ese costo más tarde. Esta idea de construir flexibilidad up-front es vista como la parte clave del diseño de software efectivo.

Sin embargo el consejo de la XP es que no construyas componentes flexibles y frameworks para el primer caso que necesite esa funcionalidad. Deja crecer esas estructuras como se necesiten. Si quiero una clase Money hoy que maneje adición pero no multiplicación entonces solo construyo adición en la clase Money. Aún si estoy seguro de que necesitaré multiplicación en la iteración siguiente, y entiendo cómo hacerlo fácilmente, y creo que sería realmente rápido hacerlo, aún así lo dejaré para la siguiente iteración.

Una razón de esto es económica. Si tengo que trabajar por una característica que sólo se necesitará mañana, estoy comprometiendo esfuerzo de características que necesitan hacerse para la iteración actual. El plan de entrega dice qué es lo que necesita trabajarse ahora, trabajar en cosas del futuro es contrario a los acuerdos de los desarrolladores con el cliente. Hay un riesgo de que las historias de la iteración pudieran no hacerse. Aún si estas historias de la iteración no están en riesgo depende del cliente decidir que trabajo extra debe hacerse -y que podría no incluir aún la multiplicación.

Este contraincentivo económico es reforzado por la posibilidad de que pudiéramos no hacerlo bien. Como sea que estemos seguros de que esta función trabaja, aún podríamos equivocarnos -especialmente si aún no tenemos requerimientos detallados. Trabajar en la solución errónea anticipadamente es aún peor despilfarro que trabajar en la función correcta con anticipación. Y los XPertos generalmente creen que es más probable que nos equivoquemos (y yo concuerdo con esa opinión).

La segunda razón por el diseño simple es que un diseño complejo es más difícil de entender que un diseño simple. Por lo tanto cualquier modificación al sistema se hace más difícil por la complejidad añadida. Esto agrega un costo en el periodo que va de cuando el diseño más complicado se añadió a cuando se necesitó.

Ahora bien, para mucha gente este consejo no tiene sentido, y tienen razón. Tienen razón si te imaginas en el mundo de desarrollo convencional donde las prácticas habilitadoras de la XP no existen. No obstante, cuando el balance entre diseño planeado y evolutivo cambia, entonces YAGNI se convierte en una buena práctica (y sólo entonces).

Para resumir, no debes gastar esfuerzo añadiendo cosas que no serán necesarias hasta una iteración futura. Aun si el costo es cero, no debes porque eso incrementa el costo de modificación. De cualquier manera sólo puedes comportarte así si estás usando XP o alguna práctica similar que baje el costo del cambio.

## Finalmente Qué Demonios es Simplicidad

Así que queremos nuestro código tan simple como sea posible. Eso no suena difícil de sostener, después de todo ¿quién quiere ser complicado? Pero desde luego esto trae la pregunta "¿qué es simple?"

En [XPE](http://www.amazon.com/exec/obidos/ASIN/0201616416) Kent da cuatro criterios para un diseño simple, en orden de importancia:

* Correr todas las pruebas
* Revelar toda la intención
* Evitar duplicación
* El menor número de clases y métodos

El que corran todas las pruebas es un criterio muy simple. No duplicación también es muy directo, aunque muchos de los desarrolladores necesitan guía para lograrlo. El truco tiene que ver con revelar la intención. ¿Qué significa eso exactamente?

El valor básico aquí es claridad de código. XP pone en alto el código fácil de leer. En XP "código astuto" es un término de abuso. Pero para algunas personas, el código que revela la intención es otra astucia.

En su artículo en XP 2000, Josh Kerievsky señala un buen ejemplo de esto. El atiende al que posiblemente es el código XP más público - [JUnit](http://www.xprogramming.com/software.htm). JUnit usa decoradores para añadir funcionalidad opcional a los casos de prueba, tales como sincronización concurrente y código batch set up. Separando este código en decoradores permite al código general ser más claro de lo que podría ser.

Pero uno se pregunta si el código resultante es realmente simple. Para mí lo es, pero estoy familiarizado con el patrón Decorator. Pero para muchos que no lo están esto es muy complicado. Similarmente JUnit usa métodos enchufables los cuales, he notado, la mayoría de la gente los encuentra cualquier cosa menos claros. ¿Así que debemos concluir que el diseño de JUnit es simple para diseñadores experimentados pero complicado para los menos experimentados?

Yo creo que el meollo en eliminar la duplicación, tanto el "Una vez y sólo una vez" de la XP y el DRY (Don't Repeat Yourself) del [Programador Pragmático](http://www.amazon.com/exec/obidos/ASIN/020161622X) es uno de esos obvios y maravillosamente poderosos buenos consejos. El solo seguirlo puede llevar muy lejos. Pero no es todo, y la simplicidad es aún algo complicado de hallar.

Recientemente me vi envuelto en hacer algo que bien podría estar sobrediseñado. Lo arreglamos refactorando y removiendo algo de su flexibilidad. Pero como uno de los desarrolladores dijo "es más fácil refactorar sobrediseño que refactorar sin diseño". Es mejor ser un poco más simple de lo que necesitas ser, pero no es un desastre ser un poco más complejo.

El mejor consejo que he escuchado sobe todo esto vino del tío Bob (Robert Martin). Su consejo fue no demorarse mucho sobre cual es el diseño más simple. Después de todo se puede, se debe y se refactorará más tarde. Al final la disposición a refactorar es más importante que saber que la cosa más simple ya está.

## La refactoración viola YAGNI

Este asunto surgió recientemente en la lista de correo XP, y vale la pena traerlo a colación, ya que estamos viendo el rol del diseño en la XP.

Básicamente la cuestión empieza con el punto de que la refactoración toma tiempo pero no añade funcionalidad. Como el punto del YAGNI es que se supone que se debe diseñar para el presente y no para el futuro, ¿es esto una violación?

El punto de YAGNI es no añadir complejidad innecesaria para las historias actuales. Esto es parte de la práctica del diseño simple. Refactorar es necesario para mantener el diseño tan simple como puedas, así que debes refactorar cuando notes que puedes simplificar las cosas.

El diseño simple explota las prácticas de la XP y es también una práctica habilitadora. Sólo si has hecho pruebas, integrado continuamente y refactorado puedes practicar el diseño simple efectivamente. Pero al mismo tiempo, mantener el diseño simple es esencial para mantener la curva del cambio plana. Cualquier complejidad innecesaria hace al sistema difícil de cambiar en toda dirección excepto la que anticipaste con la complicada flexibilidad añadida. Sin embargo la gente no es buena anticipando, así que es mejor esforzarse por la simplicidad. No se obtiene la cosa más simple la primera vez, así que necesitas refactorar para acercarte a la meta.

## Patrones y XP

El ejemplo JUnit me lleva inevitablemente a los patrones. La relación entre patrones y la XP es interesante y es una cuestión común. Joshua Kerievsky arguye que los patrones son subestimados en la XP y lo sustenta elocuentemente, así que no lo repetiré. Pero vale la pena tomar en cuenta que para mucha gente los patrones parecen estar en conflicto con la XP.

La esencia de este argumento es que los patrones frecuentemente se sobreutilizan. El mundo esta lleno del legendario buen programador, fresco de su primera lectura del [GOF](http://www.amazon.com/exec/obidos/ASIN/0201633612) que incluye 16 patrones en 32 líneas de código. Recuerdo una tarde, alimentada por una muy buena cerveza, trabajando con Kent en un artículo que sería llamado "Patrones de No Diseño: 23 trucos baratos". Estabamos pensando en cosas como usar una declaración if en lugar de una estrategia. La broma tenía una razón, los patrones son frecuentemente sobreutilizados, pero eso no los hace tan mala idea. La cuestión es cómo los usas.

Una teoría es que las fuerzas del diseño simple te llevarán a los patrones. Muchas refactoraciones hacen esto explícitamente, pero aún sin ellas siguiendo las reglas del diseño simple llegarás a los patrones aun si no los conoces previamente. Esto puede ser cierto pero, ¿es la mejor forma de hacerlo? Seguramente es mejor si sabes toscamente a dónde vas y tienes un libro que te guíe en lugar de tener que inventarlo tú mismo. Yo ciertamente aún uso el GOF cuando siento que surge un patrón. Para mí el diseño efectivo implica que necesitamos saber si vale la pena pagar el precio de un patrón -eso es su propia habilidad. Similarmente, como sugiere Joshua, necesitamos estar más familiarizados acerca de cómo llegar a un patrón gradualmente. En este respecto la XP trata la manera en que usamos patrones diferente a la manera en que algunas personas los usan, pero ciertamente no elimina su valor.

Leyendo algunas de las listas de correo tengo la sensación de que mucha gente ve a la XP como desalentadora de patrones, a pesar de la ironía de que la mayoría de los proponentes de la XP también han sido líderes del movimiento de los patrones. ¿Será que ellos han visto más alla de los patrones, o tienen a los patrones tan embebidos en su pensamiento que ellos ya no lo notan? No sé la respuesta para otros, pero para mí los patrones aún son vitalmente importantes. XP puede ser un proceso de desarrollo, pero los patrones son una columna vertebral del conocimiento de diseño, conocimiento que es valioso cualquiera que sea el proceso. Procesos diferentes pueden usar patrones de diferente manera. XP enfatiza no usar patrones hasta que sea necesario y evolucionar el camino hacia un patrón via una implementación simple. Pero los patrones son todavía una pieza clave de conocimiento a adquirir.

Mi consejo a los XPeros en el uso de patrones es:

* Invertir tiempo en aprender sobre patrones
* Concentrarse en cuándo aplicarlos (no muy pronto)
* Concentrarse en cómo implementar el patrón en su forma más simple primero, y añadir complejidad luego.
* No temer eliminar un patrón si no está valiendo su peso.

Pienso que la XP debería enfatizar aprender más sobre patrones. No estoy seguro de cómo encajaría esto en las prácticas de la XP, pero estoy seguro que Kent puede encontrar la manera.

## Creciendo una arquitectura

¿Qué queremos decir con arquitectura de software? Para mí el término arquitectura conlleva una noción de los elementos centrales del sistema, las piezas que son difíciles de cambiar. Un fundamento sobre el cual debe construirse el resto.

¿Qué rol juega la arquitectura cuando usamos diseño evolutivo? De nuevo los críticos de la XP afirman que la XP ignora la arquitectura, que la ruta de la XP es ir hacia la codificación rápida y confiar en que la refactoración resolverá todos los problemas de diseño. Interesantemente tienen razón, y eso bien puede ser una debilidad. Ciertamente los más agresivos XPeros - Kent Beck, Ron Jeffries y Bob Martin - están poniendo más y más energía en evitar cualquier diseño arquitectónico. No pongas una base de datos hasta que realmente sepas que la necesitas. Trabaja con archivos primero y refactora a la base de datos en una iteración posterior.

Yo soy conocido por ser un XPero cobarde, y como tal tengo que discordar. Yo creo que hay lugar para una amplia arquitectura inicial. Cosas tales como establecer temprano cómo separaremos la aplicación, como interaccionaremos con la base de datos (si hace falta una), qué estrategia usar para manejar el servidor web.

Esencialmente creo que muchas de estas áreas son patrones que hemos aprendido a través de los años. Conforme crece el conocimiento de patrones, se debe tener una idea razonable de cómo usarlos. De cualquier modo la diferencia clave es que no se espera poner en piedra estas decisiones arquitectónicas tempranas , o bien el equipo sabe que ellas pueden errar sus decisiones tempranas y deberán tener el valor de arreglarlas. Otros cuentan la historia de un proyecto en que, cerca de la implantación, deciden que no necesitaban más EJB y lo quitaron del sistema. Fue una refactoración considerable, se hizo tarde, pero las prácticas habilitadoras lo hicieron no sólo posible, sino valioso.

Cómo habría funcionado esto de la otra manera. Si decides no usar EJB, ¿habria sido más difícil añadirlo luego? ¿No debes pues empezar con EJB hasta que hayas tratado cosas sin él y encontrado su falta? Es una pregunta que implica varios factores. Ciertamente trabajar sin un componente complejo incrementa la simplicidad y hace las cosas ir más rápido. Sin embargo algunas veces es más fácil sacar algo que meterlo.

Mi consejo es empezar estimando la arquitectura probable. Si ves un gran montón de datos con múltiples usuarios, adelante, usa una base de datos desde el primer día. Si ves lógica de negocio compleja, pon un modelo del dominio. De cualquier modo en deferencia a los dioses del YAGNI, cuando dudes ve al lado de la simplicidad. También prepárate para simplificar tu arquitectura tan pronto veas que parte de la misma no está dando nada.

## UML y XP

De todas las preguntas que me hacen sobre mi compromiso con la XP una de las más grandes es respecto a mi asociación con el UML. ¿No son los dos incompatibles?

Hay un número de puntos de incompatibilidad. Ciertamente la XP menosprecia los diagramas en gran medida. Aunque la posición oficial es en la línea de "úsalos si son útiles", se puede leer entre líneas "los verdaderos XPeros no diagraman". Esto se refuerza con el hecho de que gente como Kent no se sienten cómodos con los diagramas, de hecho nunca he visto a Kent dibujar voluntariamente un diagrama de software en ninguna notación fija.

Yo creo que el asunto viene de dos causas separadas. Una es el hecho de que algunas personas encuentran los diagramas útiles y otras no. El peligro está en los que piensan que quienes no los usan deberían usarlos y viceversa. En lugar de eso deberíamos aceptar que algunos usen diagramas y otros no.

El otro problema es que los diagramas tienden a asociarse con procesos pesados. Tales procesos gastan mucho tiempo en dibujar diagramas que no ayudan y pueden dañar. De manera que pienso que la gente debe ser guiada en cómo usar bien los diagramas y evitar las trampas, en lugar del mensaje "solo si tienes que" de los XPertos.

Aquí esta mi consejo para usar bien los diagramas.

Primero piensa para qué los estás dibujando. El principal valor es la comunicación. Comunicación efectiva significa seleccionar las cosas más importantes y descuidar las menos. Esta selectividad es la clave para usar bien el UML. No dibujes cada clase -sólo las importantes. Para cada clase, no muestres cada atributo y operación -sólo los importantes. No dibujes diagramas de secuencia para todos los casos de uso y escenarios -sólo... ya lo entiendes. Un problema común con el uso de los diagramas es que la gente intenta hacerlos extensos. El código es la mejor fuente de información extensa ya que el código es lo más fácil de mantener en sincronía con el código. La exhaustibilidad de los diagramas es la enemiga de su comprensión.

Un uso común de los diagramas es explorar un diseño antes de empezar a programarlo. Frecuentemente se tiene la impresión de que tal actividad es ilegal en XP, pero eso no es cierto. Mucha gente dice que cuando se tiene una tarea difícil vale la pena juntarse para tener una sesión de diseño rápido. De cualquier manera cuando hagan tales sesiones:

* manténganlas cortas
* no traten de concentrarse en todos los detalles (sólo los importantes)
* traten el diseño resultante como un esbozo, no como el diseño final

El último punto merece extenderse. Cuando se hace un diseño up-front, inevitablemente encontrarás que algunos aspectos del diseño están mal, y eso sólo se descubre cuando lo programas. Eso no es un problema dado que entonces cambias el diseño. El problema viene cuando la gente piensa que el diseño está hecho y entonces no aprovechan la oportunidad de devolver al diseño el conocimiento adquirido al programar.

Cambiar el diseño no necesariamente implica cambiar los diagramas. Es perfectamente razonable dibujar diagramas que te ayuden a entender el diseño y entonces hacerlos a un lado. El dibujarlos ayudó y eso es suficiente para que valga la pena hacerlos. Pero no tienen que ser artefactos permanentes. Los mejores diagramas UML no son artefactos.

Muchos XPeros usan tarjetas CRC. Eso no entra en conflicto con el UML. Yo uso una mezcla de CRC y UML todo el tiempo, usando la técnica que sea más útil para el trabajo en mano.

* Toma mucho tiempo mantener los diagramas actualizados, así que pierden sincronía con el código.
* Están ocultos en una herramienta CASE, así que nadie los mira.

El consejo de documentación sobre la marcha viene de estos problemas observados:

* Sólo usa diagramas que puedas mantener actualizados sin esfuerzo notable.
* Pon los diagramas donde todos puedan verlos fácilmente. Me gusta pegarlos en la pared. Alienta a la gente a editar la copia en la pared para cambios simples.
* Presta atención a si alguien usa los diagramas. Si nadie lo hace, tíralos.

El último aspecto de usar UML es para documentación en una situación de relevo, tal como cuando un grupo ocupa el lugar de otro. Aquí el punto XP es que producir documentación es una historia como cualquier otra, y así su valor de negocio es determinado por el cliente. Otra vez la UML es útil aquí, los diagramas son selectivos para ayudar a la comunicación. Recuerda que el código es el repositorio de información detallada, el diagrama actúa para resumir y resaltar asuntos importantes.

## Sobre la Metáfora

Bueno, podría decirlo públicamente -no he captado el punto de esta cosa, metáfora. La he visto funcionar, y funcionó bien en el proyecto C3, pero eso no significa que yo tenga una idea de cómo hacerla, ni menos cómo explicar cómo hacerla.

La práctica XP de la metáfora se originó a partir de la propuesta de Ward Cunningham de un sistema de nombres. El punto es que vienes con un bien conocido conjunto de nombres que actúan como vocabulario para hablar acerca del dominio. Este sistema de nombres juega en el modo en que nombras las clases y métodos en el sistema.

He construido un sistema de nombres construyendo un modelo conceptual del dominio. Lo he hecho con los expertos del dominio usando UML o sus predecesores. He encontrado que hay que ser cuidadoso al hacerlo. Se necesita mantener un conjunto de notación mínimo y tienes que guardarte de dejar que cualquier asunto técnico se cuele en el modelo. Pero si lo haces, he encontrado que puedes usarlo para construir un vocabulario del dominio que los expertos del dominio puedan entender y usarlo para comunicarse con los desarrolladores. El modelo no empareja con el diseño de clases perfectamente, pero es suficiente para dar un vocabulario común de todo el dominio.

Ahora, no veo ninguna razón por la que este vocabulario no pueda ser metafórico, tal como la metáfora C3 en que la nómina se convirtió en una linea de ensamblaje de fábrica. Pero tampoco veo por qué el basar tu sistema de nombres en el vocabulario del dominio sea tan mala idea. Ni me inclino a abandonar una técnica que funciona bien para mi obteniendo los nombres del sistema.

Frecuentemente se critica a la XP sobre la base de que se necesita al menos algún esbozo de diseño de un sistema. Los XPeros frecuentemente responden con la respuesta "esa es la metáfora". Pero yo aún no creo haber visto una explicación convincente de la metáfora. Este es un hoyo real en la XP, y uno que el XPero necesita sortear.

## ¿Quieres ser arquitecto cuando seas grande?

En la última década, el término "arquitecto de software" se ha vuelto popular. Es un término que me resulta difícil de usar. Mi esposa es ingeniero estructural. La relación entre ingenieros y arquitectos es ... interesante. Mi favorito es "los arquitectos son buenos para las tres B's: bulbos, bushes [arbustos], birds [pájaros]". La idea es que los arquitectos salen con todos esos dibujos bonitos, pero son los ingenieros quienes tienen que asegurarse de que realmente puedan ponerse en pie. Como resultado he evitado el término arquitecto de software; después de todo si mi esposa no puede tratarme con respeto profesional quién puede hacerlo.

En software, el término arquitecto significa muchas cosas. (En software cualquier término significa muchas cosas.) En general, sin embargo conlleva ciertos problemas, como "no soy un mero programador -soy un arquitecto". Esto puede traducirse como "ahora soy un arquitecto -soy demasiado importante como para programar". La cuestión es si separarte del mundano esfuerzo de programar es algo que debes hacer si quieres ejercer liderazgo técnico.

Esta cuestión genera demasiada emoción. He visto gente enojarse mucho al pensar que no tienen más el rol de arquitectos. "No hay lugar en XP para arquitectos experimentados" es el grito que frecuentemente escucho.

Tanto como en el rol del diseño en sí, no creo que sea el caso que la XP no valore la experiencia de o las buenas habilidades de diseño. En realidad muchos de los proponentes de la XP - Kent Beck, Bob Martin, y desde luego Ward Cunningham - estan entre de quienes he aprendido mucho acerca de lo que es el diseño. De cualquier manera esto significa que sus roles difieren de lo que mucha gente ve como un rol de liderazgo técnico.

Como ejemplo, citaré a uno de nuestros líderes técnicos en ThoughtWorks: Dave Rice. Dave ha estado en varios ciclos de vida y se ha puesto el sombrero extraoficial de líder técnico en un proyecto de 50 personas. Su rol como líder significa gastar mucho tiempo con los programadores. El trabaja con un programador cuando éste necesita ayuda, siempre anda merodeando para ver quién necesita ayuda. Una señal significativa es cuando se sienta. Como un trabajador de mucho tiempo en ThoughtWorks, él podría muy bien tener la oficina que quisiera. Compartió una por un tiempo con Cara, la gerente de entregas. Sin embargo en los últimos meses se mudó a las bahías abiertas donde trabajan los programadores (usando el estilo abierto "war room" que la XP favorece). Esto es importante para él porque de este modo ve lo que está pasando y está disponible para dar la mano donde se necesite.

Aquellos que saben XP se darán cuenta de que estoy describiendo el rol explícito de *Coach*. En verdad uno de los varios juegos de palabras que hace la XP es que llama a la figura de primer técnico el "Coach". El significado es claro: en XP el liderazgo técnico se muestra enseñando a los programadores y ayudándolos a tomar decisiones. Es uno que requiere buena habilidad de gentes tanto como buenas habilidades técnicas. Jack Bolles en XP 2000 comentó que hay poco espacio ahora para el maestro solitario. Colaboración y enseñanza son las claves del éxito.

En una cena en una conferencia, Dave y yo hablabamos con un oponente a la XP. Conforme discutíamos lo que hacíamos, las similaridades de nuestro enfoque eran marcadas. Todos nosotros gustábamos del desarrollo adaptativo e iterativo. Las pruebas eran importantes. Así que nos tenía perplejos su oposición. Entonces vino esta declaración en la línea de "lo último que quiero es que mis programadores refactoren y manoseen el diseño". Ahora todo estaba claro. La laguna conceptual me la explicó Dave "si él no confía en sus programadores ¿por qué los contrata?". En XP la cosa más importante que puede hacer el desarrollador experimentado es pasar cuantas habilidades pueda a los desarrolladores junior. En lugar de un arquitecto que toma todas las decisiones importantes, tienes un coach que enseña a los desarrolladores a tomar decisiones importantes. Como Ward Cunningham señaló, así él amplía sus habilidades y añade más a un proyecto de lo que cualquier héroe solitario podría.

## Reversibilidad

En la XP 2000 Enrico Zaninotto dió una plática fascinante en la que discutió la relación entre los métodos ágiles y la manufactura esbelta (*lean manufacturing*). Según él, uno de los aspectos clave de ambos enfoques es que atajan la complejidad al reducir la irreversiblidad del proceso.

Desde esta vista una de las principales fuentes de complejidad es la irreversibilidad de las decisiones. Si puedes cambiar tus decisiones, no es tan importante que sean correctas - lo que hace tu vida mucho más sencilla. La consecuencia para el diseño evolutivo es que los diseñadores deben pensar en cómo evitar la irreversibilidad en sus decisiones. Más que tratar de tomar la decisión correcta ahora, busca una manera de posponerla (hasta que tengas más información) o toma la decisión de tal modo que seas capaz de revertirla después sin demasiada dificultad.

Esta determinación de mantener la reversibilidad es una de las razones de que los métodos ágiles pongan tanto énfasis en sistemas de control de código fuente y de ponerlo todo en esos sistemas. Mientras que esto no garantiza la reversibilidad, particularmente para decisiones de vida larga, proporciona un fundamento que da confianza al equipo, aun si rara vez se usa.

Diseñar para la irreversibilidad también implica un proceso que hace que los errores se muestren pronto. Uno de los valores del desarrollo iterativo es que las iteraciones rápidas permiten a los clientes ver el sistema conforme crece, y si se comete un error en los requerimientos, puede ser localizado y corregido antes de que el costo de la corrección sea prohibitivo. Esta misma localización rápida es importante para el diseño. Esto significa que tienes que poner las cosas de modo que las áreas potencialmente problemáticas sean rápidamente verificadas para ver qué problema sale. También significa que hay que hacer experimentos para ver qué tan difíciles pueden ser los cambios futuros, aun si no haces el cambio ahora - efectivamente desechando un prototipo de una rama del sistema. Varios equipos han reportado tratar un cambio futuro en modo prototipo temprano para ver qué tan difícil sería.

## El Deseo de Diseñar

Aunque me he concentrado mucho en las prácticas técnicas en este artículo, una cosa que es muy fácil de pasar por alto es el aspecto humano.

Para funcionar, el diseño evolutivo necesita una fuerza que lo lleve a converger. Esta fuerza sólo puede venir de la gente -alguien en el equipo tiene que tener la deteminación para asegurar que la calidad del diseño permanezca alta.

Este deseo no tiene que venir de todos (aunque es bueno si así pasa), usualmente sólo una o dos personas en el equipo toman la responsabilidad de mantener el diseño íntegro. Esta es una de las tareas que usualmente caen bajo el término 'arquitecto'.

Esta responsabilidad significa vigilar el código base, por si cualquier área del mismo se está enredando, y entonces actuar de inmediato para corregir el problema antes de que se salga de control. El encargado del diseño no tiene que ser el mismo que lo arregla - pero tiene que asegurarse de que alguien lo arregle.

La falta del deseo de diseñar parece ser una razón importante por la que el diseño evolutivo falla. Aun si la gente está familiarizada con las cosas de las que he hablado en este artículo, sin ese deseo no habrá diseño.

## Cosas que son difíciles de refactorar

¿Podemos usar la refactoración para tratar todas las decisiones de diseño, o hay algunos asuntos tan diseminosos que son difíciles de añadir después? Actualmente, la XP ortodoxa dice que todas las cosas son fáciles de añadir cuando las necesitas, asi que YAGNI siempre se aplica. Me pregunto si hay excepciones. Un buen ejemplo de algo que es controversial de añadir es la internacionalización. ¿Cuesta tanto añadirla después que deberías empezar con ella desde un principio?

Podría fácilmente imaginar que hay algunas cosas que caerían en esta categoría. Sin embargo la realidad es que aún tenemos muy pocos datos. Si tienes algo que añadir más tarde, como internacionalización, eres muy consciente del esfuerzo que tomará hacerlo. Eres menos consciente del esfuerzo que realmente habría tomado, semana tras semana, ponerlo y mantenerlo antes de que fuera necesario. También estás menos consciente del hecho de que bien podrías haberlo puesto mal, y así de todos modos tendrías que hacer algo de refactoración.

Parte de la justificación del YAGNI es que muchas de estas necesidades potenciales terminan no siendo necesarias, o al menos no en el modo que esperabas. El esfuerzo que ahorras no haciendo ninguna de ellas es menor al esfuerzo requerido para refactorar las que realmente necesitas.

Otra cuestión a tener en cuenta es si realmente sabes cómo hacerlo. Si has hecho internacionalización varias veces, sabes los patrones que necesitas emplear. Como tal es más probable que lo hagas bien. Añadir estructuras anticipatorias es probablemente mejor si estás en esa posición, que si eres nuevo en el problema. Mi consejo sería que si sabes cómo hacerlo, estás en la posición de juzgar el costo de hacerlo o no hacerlo después. No obstante si no lo has hecho antes, no sólo no eres capaz de calcular el costo lo suficientemente bien, es menos probable que lo hagas bien. En ese caso deberías dejarlo para después. Si lo añades entonces, y lo encuentras doloroso, probablemente aún así estarás mejor de lo que estarías si lo hubieras instalado antes. Tu equipo es más experimentado, conoces mejor el dominio y entiendes mejor los requerimientos. A menudo en esta posición ves lo fácil que hubiera sido con una retrospectiva 20/20. Añadirlo antes puede ser mucho más difícil de lo que crees.

Esto trae a colación la cuestión del orden de las historias. En [Planning](http://www.amazon.com/exec/obidos/ASIN/0201710919), Kent y yo abiertamente manifestamos nuestro desacuerdo. Kent está a favor de dejar que los valores de negocio sean el único factor de orden de las historias. Después de un desacuerdo inicial, Ron Jeffries ahora coincide. Yo todavía no estoy seguro. Yo creo que hay un balance entre valores de negocio y riesgo técnico. Esto me llevaría a proveer al menos algo de internacionalización pronto para mitigar este riesgo. No obstante esto sólo es cierto si la internacionalización fuera necesaria para la primera entrega. Llegar a la entrega tan rápido como sea posible es vitalmente importante. Cualquier complejidad adicional debe hacerse después de la primera entrega si no se necesita para la primera entrega. El poder del código empacado y funcionando es enorme. Esto enfoca la atención del cliente, aumenta la credibilidad y es una fuente masiva de aprendizaje. Haz todo lo que puedas para adelantar esa fecha. Aún si es más esfuerzo añadir algo después de la primera entrega, es mejor entregar antes.

## ¿Está ocurriendo el diseño?

Una de las dificultades del diseño evolutivo es que es muy difícil decir si el diseño está realmente ocurriendo. El peligro de entremezclar diseño con programación es que puede haber programación sin diseño - esta es la situación donde el Diseño Evolutivo diverge y falla.

Si estás en el equipo de desarrollo, entonces sabes si el diseño está ocurriendo por la calidad del código base. Si el código base se vuelve más complejo y difícil de trabajar, no se está haciendo suficiente diseño. Pero este es tristemente un punto de vista subjetivo. No tenemos métricas confiables que nos permitan medir objetivamente la calidad del diseño.

Si esta falta de visiblidad es dura para personas técnicas, es más alarmante para miembros del equipo no técnicos. Si usted es un gerente o cliente ¿cómo puede decir si el software está bien diseñado? A usted le interesa porque un software pobremente diseñado será más caro de modificar en el futuro. No hay una respuesta fácil para esto, pero aquí van unas pocas sugerencias:

* Escuche a la gente técnica. Si se quejan de la dificultad de hacer cambios, tome esas quejas seriamente y deles tiempo de arreglar las cosas.
* Vigile cuanto código se desecha. Un proyecto que hace refactoración saludable estará constantemente eliminando código malo. Si no se elimina nada, entonces es casi seguro que no se está refactorando lo suficiente - lo cual llevará a la degradación del diseño. Sin embargo, como cualquier métrica esta puede ser abusada, la opinión de gente técnica buena, por subjetiva que sea, triunfa sobre cualquier métrica.

## ¿Así que ha muerto el diseño?

De ninguna manera, pero la naturaleza del diseño ha cambiado. El diseño XP busca las siguientes habilidades:

* Un deseo constante de mantener el código tan claro y simple como sea posible.
* Habilidades de refactoración, de modo que puedas confiadamente hacer mejoras cuando veas la necesidad.
* Un buen conocimiento de patrones: no sólo las soluciones sino también apreciar cuando usarlos y cuando evolucionar hacia ellos.
* Saber cómo comunicar el diseño a la gente que necesita entenderlo, usando código, diagramas y sobre todo conversación.

Esta es una selección pusilánime de habilidades, pero siempre ha sido difícil ser un buen diseñador. La XP no lo hace realmente más fácil, al menos para mí. Pero creo que la XP nos da una nueva forma de pensar acerca del diseño efectivo porque ha hecho el diseño evolutivo una estrategia plausible otra vez. Y yo soy un gran fan de la evolución -¿de otro modo quién sabe qué podría ser yo?

## Reconocimientos

En los últimos años he recogido y robado muchas buenas ideas de mucha buena gente. La mayoría de ellas están perdidas en la turbiedad de mi memoria. Pero recuerdo haber pinchado buenas ideas de Joshua Kerievski. También recuerdo muchos comentarios útiles de Fred George y Ron Jeffries. Tampoco puedo olvidar cuantas buenas ideas siguen viniendo de Ward y Kent.

Siempre estoy agradecido de todos aquellos que hacen preguntas y señalan errores de tecleo. Me ha dado flojera mantener una lista de estos dos areconocimientos, pero incluyen a Craig Jones, Nigel Thorne, Sven Gorts, Hilary Nelson, Terry Camerlengo.

## Revisión

He aquí una lista de las principales actualizaciones de este artículo

* *Mayo 2004:* Se añadieron las secciones 'Reversibilidad', 'El Deseo de Diseñar' y '¿Está Ocurriendo el Diseño?'.
* *Febrero 2001:* Articulo actualizado con secciones sobre madurando una arquitectura, el rol del arquitecto, y cosas difíciles de refactorar.
* *Julio 2000:* Artículo original enviado a XP 2000 y publicado en martinfowler.com

©Copyright [Martin Fowler](http://www.martinfowler.com), all rights reserved   
© Derechos reservados de la traducción [Alejandro Aguilar Sierra](http://www.programacionextrema.org/cgi-bin/wiki.pl?AlejandroSierra).

Si desea comentar sobre el artículo o la traducción, oprima [aquí](http://www.programacionextrema.org/cgi-bin/wiki.pl?TraduccionIsDesignDead).