CAPÍTULO 9

PATRONES Y REFACTORIZACIONES EN LA CALIDAD SOFTWARE

Javier Garzás
Mario G. Piattini Velthuis

INTRODUCCIÓN

Históricamente, la Ingeniería del Software ha ido introduciendo diferentes técnicas con el fin de mejorar la calidad de los sistemas informáticos: técnicas estructuradas, lenguajes de cuarta generación, herramientas CASE, Orientación a Objetos (OO), etc. Todas estas técnicas persiguen, entre otros objetivos, aumentar la flexibilidad del software frente a los cambios, ya que, como estableció Lehman (1980), "los grandes programas no llegan nunca a completarse y están en constante evolución", "ley" que confirmó el mismo autor casi veinte años después (Lehman *et al.*, 1998). Por tanto, el grado de flexibilidad de un producto software se relaciona directamente con la calidad del mismo.

A pesar de la utilización de las técnicas de análisis, diseño y programación OO, los sistemas OO con varias líneas de código son difíciles de mantener y casi imposibles de extender con nuevas funcionalidades (Schulz *et al.*, 1998). Esto no es debido a que principios como la herencia o el encapsulamiento no resulten adecuados, sino que estos principios por sí solos no proporcionan una solución completa para construir sistemas OO de una cierta complejidad. Este tipo de técnicas deben combinarse con otras de mayor nivel, como los patrones, que pretenden conseguir software flexible, o las refactorizaciones, que intentan asegurar transformaciones correctas tanto en el diseño como en el código.

Comenzado por el siguiente apartado, repasaremos el estado actual de la tecnología de patrones. El tercer bloque trata sobre la refactorización, técnica que está adquiriendo cada vez más importancia. En el cuarto apartado se repasa el papel de estas técnicas en la

construcción de software de calidad. Por último, se presentan algunas conclusiones y líneas de trabajo.

PATRONES

A finales de los setenta aparecen dos libros escritos por Christopher Alexander acerca del planeamiento urbano y la construcción de edificios. *The timeless Way of Building* (Alexander, 1979) y *A Pattern Language* (Alexander, 1977) presentaron la particular visión del autor sobre los problemas recurrentes que existían en la arquitectura de pueblos y ciudades, y, en general, en cualquier tipo de construcción. Alexander describió estos problemas y sus soluciones usando una expresión llamada patrón: "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, describiendo el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo dos veces de la misma forma" (Alexander, 1977). Después de examinar el campo de la construcción y el trabajo de Alexander, los investigadores han observado problemas similares en el software, donde existe una clara evidencia de los patrones en todos los niveles de diseño, desde arquitecturas de alto nivel a problemas de diseño detallado.

En 1987 Ward Cunningham y Kent Beck trabajaban con Smalltalk en el diseño de interfaces de usuario y decidieron usar algunas de las ideas de Alexander, desarrollando un pequeño lenguaje con cinco patrones que guiase a los programadores noveles en Smalltalk. Presentaron este trabajo en la conferencia OOPSLA'87, bajo el título *"Using Pattern Languages for Object Oriented Programs"*. Poco tiempo después, Jim Coplien comienza a recopilar un catálogo de patrones de bajo nivel (*Idioms*) para C++ (Coplien, 1991). Entre 1990 y 1992 varios miembros de la "Banda de los Cuatro" (conocida por sus siglas en inglés GoF, *Gang of Four*)[1] se reúnen y comienzan a catalogar patrones. En la OOPSLA de 1991 se disparan las discusiones sobre patrones, especialmente en el taller organizado por Bruce Andersen, en el que participaban, entre otros, Jim Coplien, Doug Lea, Desmond D'Souza, etc. Durante los años 1995 y 1996 los patrones OO se consolidan en difusión y utilización, gracias a la aparición de dos libros esenciales: *"Design patterns: Elements of Reusable Object Oriented Software"* (Gamma *et al.*, 1995), y *"A System of Patterns: Pattern-Oriented Software Architecture"* (Buschmann *et al.*, 1996).

En los últimos años los patrones son cada vez más aceptados, y se han extendido a otras muchas áreas dentro de la construcción y el mantenimiento de sistemas software (entornos web, tiempo real, etc., como se detalla en secciones posteriores). Su utilización,

[1] Grupo formado por E. Gamma, R. Helm, R. Johnson, y J. Vlissides, creadores del más famoso catálogo de patrones (Gamma et al., 1995)

si bien aun le queda mucho
como la incorporación de e

El concepto de pat

Coad (1992) coment
enfocarse a la construcció
existe una clara similitud
patrones de bajo nivel y sus
análisis y diseño OO más e
de relación (generalizació
bloques de construcción de
alta calidad, y a estas simil
general, no existe una defini

- "Cada patrón es u
cierto contexto, un
tiempo una cosa q
crear esa cosa y c
proceso; al mismo
descripción del pro
- "Un patrón es una
para resolver un pr
et al., 1995)
- "Cada patrón es u
contexto, cierto sist
cierta configuración
(Gabriel, 1996)

Si bien, como hem
expresan un concepto simila
confundir el concepto de pa
los patrones no son en ningú

- Invenciones, teoría
- Soluciones que sólo
- Principios abstracto
- Aplicaciones univer

si bien aun le queda mucho camino por recorrer, comienza a tener suficiente madurez, así como la incorporación de esta disciplina al soporte automatizado.

El concepto de patrón

Coad (1992) comenta cómo los métodos para la creación de modelos OO tienden a enfocarse a la construcción de bloques de bajo nivel (clases y objetos), entre los que existe una clara similitud con los elementos base descritos por Alexander. Así, los patrones de bajo nivel y sus relaciones forman un bloque de construcción que permiten un análisis y diseño OO más efectivos. Los métodos OO ya enfatizan cierto tipo de patrones de relación (generalización, asociación, agregación etc.) y estas relaciones asocian bloques de construcción de bajo nivel. En síntesis, existen similitudes en los diseños de alta calidad, y a estas similitudes es a lo que se denomina patrón. Como sucede por lo general, no existe una definición estándar, así:

- "Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un problema y una solución. El patrón es, resumiendo, al mismo tiempo una cosa que tiene su lugar en el mundo y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es al mismo tiempo una cosa y un proceso; al mismo tiempo una descripción de una cosa que tiene vida y una descripción del proceso que la generó" (Alexander, 1979)

- "Un patrón es una descripción de objetos que se comunican y clases a la medida para resolver un problema de diseño general en un contexto particular" (Gamma *et al.*, 1995)

- "Cada patrón es una regla de tres partes, que expresa la relación entre cierto contexto, cierto sistema de fuerzas que ocurren repetidamente en ese contexto y cierta configuración software que permite a estas fuerzas resolverse a sí mismas" (Gabriel, 1996)

Si bien, como hemos podido observar, no existe una definición única, todas expresan un concepto similar, conjugando problema–contexto–solución. Aun así, es fácil confundir el concepto de patrón con otros similares. A modo de contraejemplo, citar que los patrones no son en ningún caso:

- Invenciones, teoría o ideas no probadas.

- Soluciones que sólo han funcionado una vez.

- Principios abstractos o heurísticos.

- Aplicaciones universales para cualquier contexto.

Descripción y agrupación de patrones

Desde el comienzo de la disciplina el número de patrones descubiertos ha aumentado de forma considerable, por lo que en la actualidad se dispone de un gran número de ellos. Éstos debieran definirse de una manera tal que su aplicación, búsqueda y comprensión fuese lo menos compleja posible. Así, el cómo describirlos y catalogarlos ha sido muy estudiado, aunque en la actualidad no existe un consenso estándar.

Cierto es que a la hora de describir un patrón existen ciertos puntos básicos a detallar, pero dependiendo de los distintos autores éstos pueden ser diferentes en número y concepto. El formato que utiliza Gamma *et al.* (1995) para describir los patrones de su catálogo es el siguiente:

- *Intención*: ¿Qué hace el patrón? ¿Cuál es su racionalidad e intención? ¿Qué problema de diseño aborda?
- *También conocido como*: Ya que los patrones son ideas probadas y usadas durante bastante tiempo, pueden haber recibido diferentes nombres.
- *Motivación*: Un escenario que ilustra el problema y cómo la estructura del patrón lo soluciona. El escenario ayuda a comprender la descripción más abstracta.
- *Aplicabilidad*: En qué situaciones es aplicable el patrón. Ejemplos de Diseños pobres solucionables con el patrón. Cómo reconocer estas situaciones.
- *Estructura*: Una representación gráfica de las clases en el patrón.
- *Participantes*: Descripción de la responsabilidad de los objetos y clases participantes.
- *Colaboraciones*: Diagramas de colaboración entre los objetos y clases participantes.
- *Consecuencias*: ¿Cómo logra el patrón sus objetivos? ¿Cuáles son los resultados de usarlo? ¿Qué partes de la estructura puede variar de manera independiente?
- *Implementación*: ¿Qué técnicas utilizar para implementar el patrón? ¿Hay características específicas del lenguaje?
- *Código de ejemplo*: Fragmentos de código que ilustran cómo implementar el patrón.
- *Usos conocidos*: Ejemplos de uso del patrón en sistemas reales.

- *Patrones relaci*
  son las diferenci

Si es importante la
En la actualidad existe
problemas derivan a l
principalmente (Appleto

- **Catálogos de P**
  pobre o informa
  alguna referenci
  organización a la

- **Sistemas de P**
  trabajan juntos p
  su totalidad. Añ
  y uniformidad a

- **Lenguajes de**
  problemas en ur
  entre patrones. S
  comprender y c
  construyen uno s
  una arquitectura"

Lenguajes y Sis
estrechamente para resol
Patrones añade robuste
Lenguajes son computa
posibles de patrones y
práctica la diferencia ent
diferencia más important
sino que evolucionan c
crecimiento poco sistemá
desde un catálogo).

Tipos de patrones

Si bien los patro
(implementación y diseñ
áreas relacionadas con la
los patrones se clasifican
et al. (1996) define tres ni

- *Patrones relacionados*: ¿Qué patrones están fuertemente relacionados? ¿Cuáles son las diferencias importantes? ¿Con qué otros patrones se debería usar éste?

Si es importante la descripción de un patrón no lo es menos la forma de agruparlos. En la actualidad existen varias alternativas, siendo este punto uno de los que más problemas derivan a la hora de explotar el uso de patrones. Así, se distinguen principalmente (Appleton, 2001):

- **Catálogos de Patrones**: Colecciones de patrones relacionados (quizá de manera pobre o informal). Se suelen subdividir en alguna categoría y pueden incluir alguna referencia entre ellos. Añaden una cantidad muy pequeña de estructura y organización a la colección de patrones (Buschmann *et al.*, 1996)

- **Sistemas de Patrones**: Conjuntos cohesivos de patrones relacionados que trabajan juntos para soportar la construcción y evolución de una arquitectura en su totalidad. Añaden una profunda estructura, una rica interacción entre patrones y uniformidad a la colección de patrones (Buschmann *et al.*, 1996)

- **Lenguajes de Patrones**: Proporcionan un conjunto de patrones que resuelven problemas en un dominio específico. Los lenguajes documentan las relaciones entre patrones. Son una colección de patrones que forman un vocabulario para comprender y comunicar ideas. "Colección estructurada de patrones que se construyen uno sobre otro para transformar necesidades y restricciones dentro de una arquitectura" (Coplien, 1998)

Lenguajes y Sistemas forman un conjunto de patrones que interactúan estrechamente para resolver problemas de un dominio particular. Pero un Lenguaje de Patrones añade robustez, comprensividad e integridad a un sistema de patrones. Los Lenguajes son computacionalmente completos, y muestran todas las combinaciones posibles de patrones y sus variaciones para producir arquitecturas completas. En la práctica la diferencia entre sistemas y lenguajes puede ser difícil de apreciar. Quizá la diferencia más importante es que los lenguajes de patrones no son creados de una vez, sino que evolucionan desde los sistemas de patrones a través de un proceso de crecimiento poco sistemático (un sistema de patrones puede evolucionar de igual forma desde un catálogo).

Tipos de patrones

Si bien los patrones nacieron principalmente en niveles de abstracción bajos (implementación y diseño), con el tiempo su evolución se ha extendido a casi todas las áreas relacionadas con la construcción software. Entrando en aspectos más específicos, los patrones se clasifican generalmente en función de su nivel de abstracción. Buschmann *et al.* (1996) define tres niveles de abstracción:

- **Patrones arquitectónicos**: Se centran en la estructura del sistema, en la definición de subsistemas, sus responsabilidades y reglas sobre las relaciones entre ellos. Proporcionan un conjunto predefinido de subsistemas, sus responsabilidades y las líneas guía para organizar sus relaciones. Existen trabajos muy conocidos y prestigiosos, como los de Buschmann *et al.* (1996) o Shaw y Garlan (1996)

- **Patrones de Diseño**: Esquemas para refinar los subsistemas o componentes de sistema software o sus relaciones. Describen una estructura recurrente y común de componentes comunicantes que resuelven un problema de diseño dentro de un contexto (como los descritos por Gamma *et al*, 1995).

- **Patrones de Codificación o Idiomas** (***Idioms***): Patrones que ayudan a implementar aspectos particulares del diseño en un lenguaje de programación específico (como los descritos por Coplien, 1991).

Si bien la anterior clasificación es la más conocida puede llegar a ocultar otros tipos de patrones. Así, autores como Riehle y Zllighoven (1996) realizan una división en Patrones Conceptuales o de Análisis (por ejemplo, Fowler, 1997), Patrones de Diseño y Patrones de Implementación. No obstante, no podemos olvidar otras categorías, como, por ejemplo:

- **Anti patrones** (***Antipatterns***), si un patrón es "una buena práctica" un antipatrón es una "lección aprendida". Fueron inicialmente propuestos por Koenig (1995). Por lo general, existen dos variedades: los que describen una mala solución a un problema, y los que describen cómo salir de una mala situación y cómo llegar a una buena.

- **Patrones para dominios y tecnologías específicas**, centrados en entornos web, tiempo real y sistemas embebidos (OOPSLA, 2001), requisitos (Creel, 1999), relacionados con la tecnología Java (Marinescu, 2002; Grand, 2001), etc.

- **Patrones organizacionales y de procesos**, para la planificación de proyectos a gran escala, gestión, etc. (Ambler, 1998)

Debido a la gran aceptación del libro de Gamma *et al.* (1995), los patrones de diseño son los más estudiados y evolucionados en la actualidad, de ahí que el siguiente epígrafe se centre en su descripción.

Patrones de diseño

Los patrones de diseñ
un sistema software. Descri
resolver problemas de dise
patrones de escala media c
muchas veces esta fronter
dirigiendo la cooperación efe

El principal catálogo c
clasifica a los patrones de dis

- **Patrones Creacion**
proceso de instanci
crean, componen o
diseño bajo estrategi
creación, independiz

- **Patrones Estructu**
objetos y clases par
comunicar dos clases

- **Patrones de Com**
algoritmos, del repar
clases. Algunos ejen
cooperaciones entre
dependencias o el asc

La tabla 9.1 enumer
catalogados según las divisior
clase o de objeto.

ra del sistema, en la
as sobre las relaciones
de subsistemas, sus
iones. Existen trabajos
et al. (1996) o Shaw y

mas o componentes de
ura recurrente y común
ı de diseño dentro de un

ıtrones que ayudan a
guaje de programación

gar a ocultar otros tipos
alizan una división en
), Patrones de Diseño y
otras categorías, como,

ı práctica" un antipatrón
stos por Koenig (1995).
una mala solución a un
tuación y cómo llegar a

trados en entornos web,
equisitos (Creel, 1999),
and, 2001), etc.

ificación de proyectos a

(1995), los patrones de
, de ahí que el siguiente

Patrones de diseño

Los patrones de diseño se caracterizan por refinar un subsistema o componente de un sistema software. Describen una estructura a la cual muchas veces recurrimos para resolver problemas de diseño. Además, como ya se vio en la sección anterior, son patrones de escala media que no dependen del lenguaje de implementación (aunque muchas veces esta frontera es difusa) y permiten resolver problemas complejos, dirigiendo la cooperación efectiva entre componentes.

El principal catálogo de patrones de diseño es Gamma *et al.* (1995). Este catálogo clasifica a los patrones de diseño en tres grupos:

- **Patrones Creacionales (*Creational Patterns*)**: Este tipo de patrones abstraen el proceso de instanciación creando el sistema independientemente de cómo se crean, componen o representan los objetos. Por lo general, son alternativas de diseño bajo estrategias de herencia o delegación que encapsulan el mecanismo de creación, independizando los tipos de objetos "producto" que se manejan.

- **Patrones Estructurales (*Structural Pattern*)**: Determinan cómo combinar objetos y clases para definir estructuras complejas. Ejemplos típicos son cómo comunicar dos clases incompatibles, cómo añadir funcionalidad a objetos, etc.

- **Patrones de Comportamiento (*Behavioral Patterns*)**: Se ocupan de los algoritmos, del reparto de responsabilidades y la comunicación entre objetos y clases. Algunos ejemplos son la definición de abstracciones de algoritmos, la cooperaciones entre objetos para realizar tareas complejas reduciendo las dependencias o el asociar comportamiento a objetos e invocar su ejecución.

La tabla 9.1 enumera los 23 patrones definidos en Gamma *et al.* (1995), catalogados según las divisiones anteriores y dependiendo de si su aplicación es a nivel de clase o de objeto.

Alcance	Propósito: Creacional	Propósito: Estructural	Propósito: Comportamiento
Clase	Factory Method	Adapter (Class)	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype Singleton	Adapter (Object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabla 9.1. Impacto de la web en las actividades comerciales

EJEMPLO DE APLICACIÓN DE PATRONES DE DISEÑO

Imaginemos que se pretende diseñar el software referente a un videoclub. Este software debe contemplar las películas, los clientes y los alquileres que estos últimos realizan. Existen dos tipos de películas, las de estreno y las regulares, y el coste del alquiler varía en función de dichos tipos.

Un primera solución de diseño que podríamos considerar es la que se muestra en la figura 9.1, que está, a primera vista, bastante en línea con los anteriores requisitos.

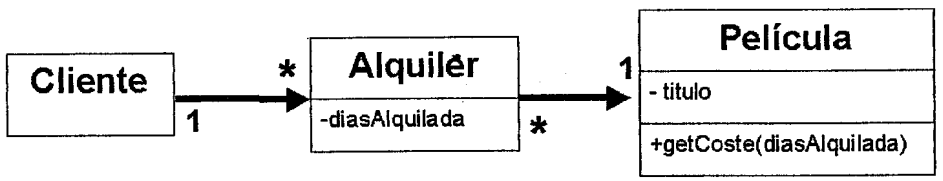

Figura 9.1. Primera solución de diseño para el ejemplo del videoclub

Realicemos el método *getCoste()* de la clase Película. Sabemos que dependiendo de la categoría de la película (Estreno o Regular) se aplicarán tarifas distintas... ¿cómo podemos implementar esto? Una forma es implementar el método *getCoste* con un conjunto de sentencias *case* o *switch* (o agrupación estructurada de sentencias condicionales) que dependiendo del tipo de película aplique un coste u otro.

Efectivamente, lo an
suficiente calidad al diseñ
requisitos cambian, ahora a
80" y posteriormente "en c
nos obligan a modificar las
tipo *switch* cada vez que ap
peor, ya que la utilización d
el diseño, así imaginemos u
que devuelva los días que se
ya dispone de la película y e
sentencia tipo *switch* con la
Finalmente, llegamos al peo
duplicación.

En síntesis observar
problemas:

- Sentencias tipo *swi*
  estado. Frecuentem
  condicional, al añadi
- Duplicación de códi
  de código.

En esencia, el proble
misma: el rol de *película est*
podríamos pensar es crear
subclase sería la correspondie
La solución anterior puede
película debe cambiar su cate
de *PeliculaEstreno* pasará co
último problema es crear un
objeto de *PeliculaRegular*,
apuntadores, destruir el obje
objeto lo apuntase, etc.; pero
otro objeto de otra clase, ya
esta última solución no es ope

Como se ha podido obs
encajando las anteriores dedu
ser bastante costoso en tiempo
comprobar que el diseño no f
veces que el mantenimiento s
nuevo.

Comportamiento
Interpreter Template Method
Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

es comerciales

DISEÑO

ferente a un videoclub. Este
alquileres que estos últimos
las regulares, y el coste del

iderar es la que se muestra en
los anteriores requisitos.

Película
- titulo
+getCoste(diasAlquilada)

l ejemplo del videoclub

a. Sabemos que dependiendo
arán tarifas distintas... ¿cómo
el método *getCoste* con un
estructurada de sentencias
un coste u otro.

Efectivamente, lo anterior funcionaría, pero... ¿sería mantenible y aportaría la suficiente calidad al diseño?: A los dos meses de estar operativa la aplicación los requisitos cambian, ahora aparece la categoría "clásicos", y dos meses después "de los 80" y posteriormente "en cartelera". Los nuevos requisitos, según nuestro diseño actual, nos obligan a modificar las sentencias de lógica condicional, añadiendo nuevas sentencias tipo *switch* cada vez que aparece una nueva categoría de película. El problema suele ser peor, ya que la utilización de lógica condicional de este tipo tiende a extenderse por todo el diseño, así imaginemos un método "*getAmpliacionDiasAlquiler*" de la clase Alquiler que devuelva los días que se puede ampliar cierto alquiler para cierto tipo de cliente que ya dispone de la película y en función del tipo de película... volveríamos a introducir una sentencia tipo *switch* con la misma estructura que la implementada en la clase Película. Finalmente, llegamos al peor de los problemas que puede presentar un diseño o código: la duplicación.

En síntesis observamos que esta solución de diseño describe dos grandes problemas:

- Sentencias tipo *switch*. Existe una gran lógica condicional que depende del estado. Frecuentemente, varias operaciones repetirán la misma estructura condicional, al añadir más tipos tendremos que buscar todas estas sentencias.

- Duplicación de código. La lógica condicional casi siempre produce duplicación de código.

En esencia, el problema es que la clase *Película* desempeña varios roles en sí misma: el rol de *película estreno*, el de *película regular*, etc. Una primera solución que podríamos pensar es crear subclases de *Película*, una por cada tipo de película (una subclase sería la correspondiente a *película Regular*, otra a *Estreno*, otra a *Clásicas*, etc.). La solución anterior puede parecer correcta, pero más tarde observaríamos que una película debe cambiar su categoría en tiempo de ejecución, ya que, por ejemplo, un objeto de *PeliculaEstreno* pasará con el tiempo a ser una *PeliculaRegular*. Una solución a este último problema es crear un nuevo objeto de la clase *PeliculaEstreno*, sacar el estado del objeto de *PeliculaRegular*, cargar todo en el objeto de *PeliculaEstreno*, cambiar los apuntadores, destruir el objeto de *PeliculaRegular*, tener cuidado de que ningún otro objeto lo apuntase, etc.; pero de todas formas, el objeto no cambiaría de clase, tendríamos otro objeto de otra clase, ya aparecería un nuevo OID (*Object Identifier*). Obviamente, esta última solución no es operativa.

Como se ha podido observar, hemos ido pasando por multitud de problemas, y el ir encajando las anteriores deducciones, si no se dispone de la suficiente experiencia, puede ser bastante costoso en tiempo, mucho más si se observa después de la implementación, al comprobar que el diseño no funciona. Multitud de problemas de este tipo hacen muchas veces que el mantenimiento sea imposible y que los sistemas tengan que reconstruirse de nuevo.

Los patrones nos ayudan a acortar el tiempo necesario para encontrar la solución correcta a este tipo de problemas de forma rápida y a anticiparlos en el diseño.

El patrón *State* (Estado) (ver Gamma *et al.* 1995 para ampliar información) es un patrón de comportamiento que permite simular la variación de comportamiento de un objeto si se producen cambios internos en su estado, con lo que se puede simular que el objeto cambie de clase. El patrón *State* posee diversas extensiones y variaciones que integran a este patrón con otros, para construir estructuras de diseño más complejas. Tras aplicar este patrón a nuestro diseño obtenemos el esquema de la figura 9.2.

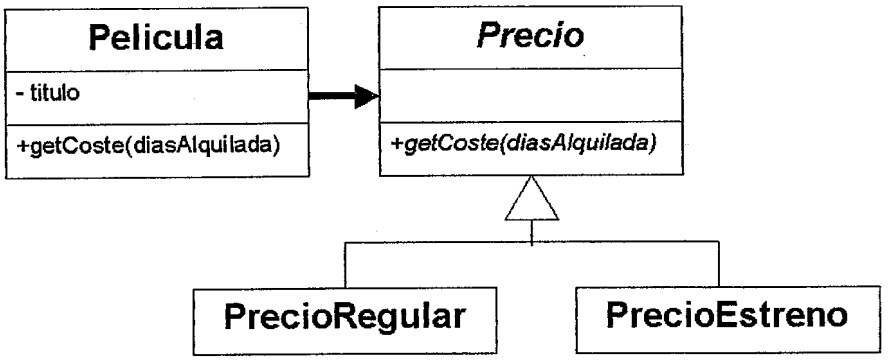

Figura 9.2. Solución de Diseño óptima para el problema del videoclub, obtenida tras la aplicación del patrón State

Ahora se añaden fácilmente nuevos estados y sus transiciones, las transiciones son explícitas, los estados están localizados y con posibilidad de ser compartidos, no hay sentencias tipo *switch*, no hay código repetido, etc.

Los métodos para la aplicación de patrones

Las técnicas para diseñar con patrones se pueden clasificar en (Yacoub y Ammar, 2000):

1. *Ad hoc.* No existe un proceso que guíe el desarrollo e integración de patrones con otros artefactos de diseño.

2. Sistemático. Va más allá de la aplicación de un cierto patrón y utiliza un mecanismo de composición para unir patrones. Pueden clasificarse en:

   a. Lenguajes de Patrones.

b. Procesos de Desarrollo. Aproximación a la composición de patrones, los pasos de análisis y diseño, modelos para soportar el desarrollo y herramientas para automatizar los pasos de desarrollo.

Mientras que los mecanismos de composición de patrones pueden clasificarse en (Yacoub y Ammar, 2000):

- Composición por Comportamiento. Se relacionan viendo los objetos como elementos que desempeñan varios roles en varios patrones. El modelado de un patrón de diseño usando roles integra tanto aspectos estáticos como dinámicos del patrón en un diagrama.

- Composición Estructural. Unión de estructuras de patrones modelados como diagramas de clases

No obstante, debido a los fallos en la aplicación de patrones, diversos autores han elaborado distintos métodos para su aplicación. Por ejemplo, Zimmer (1995) utiliza una aproximación que considera a los patrones como operadores que transforman un diseño existente en un diseño objetivo, en un proceso sistemático de cinco pasos:

- Identificación de la estructura del problema.

- Evaluación de precondiciones

- Transformación parametrizada de la estructura del problema en la estructura objetivo.

- Reorganización del contexto

- Evaluación chequeo de poscondiciones

Beneficios por el uso de patrones

Podemos destacar los siguientes beneficios del uso de patrones:

- Mejoran la productividad del programador y la calidad del programa. Incrementan la destreza en programadores noveles y animan a mejores prácticas en diseñadores experimentados. (Prechelt *et al.*, 1997)

- Definen una terminología que mejora la comunicación, tanto entre desarrolladores como entre desarrolladores y personal de mantenimiento, como comentan Prechelt *et al.* (1997), Cline (1996) y Schmidt (1995). Cuando los

equipos pueden mantener discusiones de diseño desde alto nivel, la productividad y la calidad son inmediatas.

- Clasifican diseños, más fáciles de comprender para los nuevos diseñadores, en la construcción más robusta de sistemas (Cline, 1996). Así, proporcionan transparencia para describir un problema–solución, además del contexto de limitaciones o advertencias de su uso. (Seen *et al.*, 2000)

- Reducen el acoplamiento e incrementan la flexibilidad del código. (Prechelt *et al.*, 2000). Pueden ser usados como "bisagras software", la OO permite construir de forma arbitraria software adaptable, la adaptabilidad debe ser explícitamente diseñada y colocada en ciertos lugares, complejidad que queda disminuida por el conocimiento de patrones (Cline, 1996), que, además, posibilitan una alta reutilización de las arquitecturas software (Schmidt, 1995), proporcionando respuesta a los problemas más comunes de diseño, eliminando el coste de la reinvención. (Seen *et al.*, 2000)

- Se adoptan por un bajo coste monetario y son una buena estrategia comercial a ofrecer frente a la competencia. (Seen *et al.*, 2000)

Problemas del uso de patrones

Cuando se usan patrones se observan ciertas carencias y pueden ocurrir varios tipos de problemas (Schulz *et al.* (1998), Wendorff (2001) y Schmidt (1995)):

- Ausencia de una clasificación formal de los patrones de diseño y sus relaciones mutuas, lo cual es uno de los mayores problemas.

- Compleja integración de los patrones de diseño en los sistemas existentes, ya que la descripción de los patrones está principalmente enfocada al modelo resultante de su aplicación. El proceso de aplicar el patrón no se suele tomar en consideración, lo que significa que hasta ahora los patrones han sido bloques de construcción y no operadores. Esto permite construir de manera fácil nuevos sistemas pero resulta difícil introducirlos en sistemas existentes.

- Sobrecarga de patrones y tentación a modelar todo como patrón. Dada la popularidad de libros como Gamma *et al.* (1995) muchas veces son usados en situaciones donde su flexibilidad no es necesaria, resolviendo el problema pero de forma más potente a la requerida, lo que provoca un exceso de complejidad y de código en el diseño.

- El uso exclusivo de los patrones no es suficiente para guiar un diseño de manera formal, siendo (de nuevo) la experiencia del diseñador necesaria para evitar

sobrecarga (*overload*), no-aplicación o el uso equivocado de los patrones debido a la ignorancia, o cualquier otro problema.

- Difícil aprendizaje, ya que la curva de aprendizaje suele ser muy costosa.
- Deficiencias en catálogos: aplicación y búsqueda compleja, alta dependencia del lenguaje de programación, falta de comparativas, etc.

REFACTORIZACIONES

El término refactorización (*refactoring*) se atribuye a Opdyke, quien lo introdujo por primera vez en 1992 (Opdyke, 1992). Una refactorización es una transformación parametrizada de un programa preservando su comportamiento, que automáticamente modifica el diseño de la aplicación y el código fuente subyacente (Tokuda y Batory, 2001). Sólo los cambios realizados en el software para hacerlo más fácil de modificar y comprender son refactorizaciones, por lo que no es una optimización del código, ya que esto en ocasiones lo hace menos comprensible (Fowler, 2000), ni tampoco el solucionar errores o mejorar algoritmos (Wampler, 2002). Típicamente, una refactorización es una transformación muy simple que tiene un fácil (pero no necesariamente trivial) impacto en el código fuente de la aplicación.

La refactorización posibilita la transformación de programas preservando su comportamiento, modificando sólo su estructura interna (como cambiar el nombre de una función, su interfaz para hacerla más reutilizable o eliminar código duplicado). Una técnica que permite realizar un desarrollo iterativo de manera controlable, un proceso potencialmente automatizable y controlado para alterar el software existente y que comienza con un programa que funciona pero que no está muy bien diseñado. La refactorización reduce el riesgo de hacer cambios a programas, algo habitual, típicamente derivado de la variación en los requisitos o del contexto en el cual existe el software. Tokuda y Batory (2001) afirman que, al igual que los editores de Interfaces Gráficas de Usuario revolucionaron su diseño, los editores de diagramas de clases (donde cambios en el diagrama disparan automáticamente los cambios correspondientes en el código subyacente) revolucionan la evolución del diseño software, y una de las principales tecnologías para lograrlo es la refactorización.

Por su parte Crespo y Marqués (2001) destacan cómo la refactorización es una forma especial de **reestructuración**. Se denomina reestructuración a la modificación directa de los elementos software. Una reestructuración que involucra un conjunto de elementos relacionados y transforma la forma en que éstos se relacionan se denomina reorganización. Resumiendo las ideas de Opdyke, las refactorizaciones son transformaciones a elementos software orientados a objetos que, realizando reestructuración y reorganización, preservan el comportamiento. La diferencia de nombre destaca que en sistemas OO el propósito fundamental de una reestructuración no es dotar de estructura a sistemas pobremente estructurados, porque alguna información estructural estará explícita

mediante clases, herencia, etc., sino que el objetivo es más bien refinarlos (Crespo y Marqués, 2001).

Las refactorizaciones pueden verse como una forma de mantenimiento preventivo cuyo objetivo es disminuir la complejidad del software en anticipación a los incrementos de complejidad que los cambios pudieran traer (Stroulia y Kapoor, 2001).

Beneficios de la refactorización

Entre otros, la refactorización aporta los siguientes beneficios:

- **Facilita la comprensión del software**, sobre todo para aquellas personas que no han participado en su creación. Hay que recordar la importancia que tiene el mantenimiento del software, que, sin embargo, no se suele tener en cuenta a la hora de escribir el código. La falta de comprensión del código dispara el tiempo que se necesita para cambiarlo.

- **Ayuda a encontrar errores**. Al posibilitar la comprensión del código se ayuda a visualizar errores.

- **Ayuda a programar más rápidamente.**

Problemas de la refactorización

Entre los problemas detectados en el ámbito de la refactorización, podemos destacar:

- **Cambio de las interfaces**. Muchas refactorizaciones modifican la interfaz. Lo que no es problema si se tiene acceso a todo el código, ya que se cambiaría de nombre al servicio y se renombrarían todas las llamadas a ese método. El problema aparece cuando las interfaces son usadas por código que no se puede encontrar y/o cambiar.

- **Bases de Datos**. La mayoría de las aplicaciones están fuertemente acopladas al esquema de la base de datos. Ésta es una de las razones por la que la base de datos es difícil de cambiar. Otra razón es la migración de los datos de una base de datos a otra, algo bastante costoso.

Catálogos de

En el formato
2000):

1. *Nombre de*
   forma que l
   deben contr
2. *Resumen de*
3. *Motivación.*
   las cuales no
4. *Mecanismo.*
5. *Ejemplos.* U

Por otro lado,
otros diagramas UM

En la actua
refactorizaciones a n
(2000), Opdyke (199
un listado de conoc
Fowler (2000), y a
refactorizaciones de

Add Parameter
Change Bidirecti Unidirectional
Change Reference to
Change Unidirect Bidirectional
Change Value to Ref
Collapse Hierarchy
Consolidate Conditic
Consolidate Duplicat
Decompose Conditio
Duplicate Observed I
Encapsulate Collecti
Encapsulate Downca
Encapsulate Field
Extract Class
Extract Interface

Catálogos de refactorizaciones

En el formato de una refactorización se pueden distinguir cinco partes (Fowler, 2000):

1. *Nombre de la refactorización.* Importante para construir un vocabulario, de igual forma que los patrones de diseño han creado un vocabulario las refactorizaciones deben contribuir a ampliarlo.

2. *Resumen de lo que hace y de la situación en la cual se puede necesitar.*

3. *Motivación.* Por qué la refactorización debería aplicarse y las circunstancias en las cuales no debería usarse.

4. *Mecanismo.* Descripción de cómo llevar a cabo la refactorización.

5. *Ejemplos.* Un uso de la refactorización.

Por otro lado, hay veces que se utiliza código para describir la refactorización y otros diagramas UML a nivel de implementación.

En la actualidad existen varios trabajos que describen conjuntos de refactorizaciones a mayor o menor nivel de abstracción. Ejemplos conocidos son Fowler (2000), Opdyke (1992) o Tokuda y Batory (2001), entre otros. En la tabla 9.2 puede verse un listado de conocidas refactorizaciones, extracto de las descritas en el catálogo de Fowler (2000), y a continuación se comenta, a modo de ejemplo, una de las refactorizaciones de dicho catálogo.

Add Parameter	Pull Up Constructor Body
Change Bidirectional Association to Unidirectional	Pull Up Field
Change Reference to Value	Pull Up Method
Change Unidirectional Association to Bidirectional	Push Down Field
Change Value to Reference	Push Down Method
Collapse Hierarchy	Remove Assignments to Parameters
Consolidate Conditional Expression	Remove Control Flag
Consolidate Duplicate Conditional Fragments	Remove Middle Man
Decompose Conditional	Remove Parameter
Duplicate Observed Data	Remove Setting Method
Encapsulate Collection	Rename Method
Encapsulate Downcast	Replace Array with Object
Encapsulate Field	Replace Conditional with Polymorphism
Extract Class	Replace Constructor with Factory Method
Extract Interface	Replace Data Value with Object

Extract Method	Replace Delegation with Inheritance
Extract Subclass	Replace Error Code with Exception
Extract Superclass	Replace Exception with Test
Form Template Method	Replace Inheritance with Delegation
Hide Delegate	Replace Magic Number with Symbolic Constant
Hide Method	Replace Method with Method Object
Inline Class	Replace Nested Conditional with Guard Clauses
Inline Method	Replace Parameter with Explicit Methods
Inline Temp	Replace Parameter with Method
Introduce Assertion	Replace Record with Data Class
Introduce Explaining Variable	Replace Subclass with Fields
Introduce Foreign Method	Replace Temp with Query
Introduce Local Extension	Replace Type Code with Class
Introduce Null Object	Replace Type Code with State/Strategy
Introduce Parameter Object	Replace Type Code with Subclasses
Move Field	Self Encapsulate Field
Move Method	Separate Query from Modifier
Parameterize Method	Split Loop
Preserve Whole Object	Split Temporary Variable

Tabla 9.2. Refactorizaciones descritas en el catálogo de Fowler (2000)

EJEMPLO DE REFACTORIZACIÓN: *REPLACE INHERITANCE WITH DELEGATION*

La herencia es muy útil, pero hay veces que no es lo más apropiado. Con frecuencia muchas de las operaciones de la superclase no son realmente ciertas en la subclase, y la interfaz no refleja lo que realmente hace la clase. También se heredan datos no apropiados o muchos métodos sin sentido en la subclase. La delegación deja claro que sólo se hará un uso parcial de la clase delegada, controlando qué aspectos de la interfaz se usan.

La herencia es muy útil, pero hay veces que no es lo más apropiado. Con frecuencia muchas de las operaciones de la superclase no son realmente ciertas en la subclase, y la interfaz no refleja lo que realmente hace la clase. También se heredan datos no apropiados o muchos métodos sin sentido en la subclase. La delegación deja claro que sólo se hará un uso parcial de la clase delegada, controlando qué aspectos de la interfaz se usan.

> Enunciado: Una subclase usa sólo parte de la interfaz de la superclase o no quiere heredar sus datos... *Crear un atributo para la superclase, ajustar los métodos para delegar en la superclase y eliminar la herencia.*

La figura 9.3 muestra un ejemplo en el que se aplica esta refactorización.

Figura 9.3. Ejemplo d
with deleg

Bases de un pro

La refactorizació
de desarrollo software.
una manera concreta (
general e intrínseca a la

1° Revisar cód

2° Aplicar una

3° Aplicar pru

4° Repetir los

Además, existen
proceso:

- **No añadir fu**
refactorizació
observable ex
misma forma
cambia enton
estropeado lo

ith Inheritance
ith Exception
h Test
ith Delegation
er with Symbolic Constant
Method Object
tional with Guard Clauses
h Explicit Methods
h Method
Data Class
Fields
uery
ith Class
ith State/Strategy
ith Subclasses
Modifier
able

ogo de Fowler (2000)

NHERITANCE WITH

es lo más apropiado. Con
son realmente ciertas en la
e. También se heredan datos
La delegación deja claro que
qué aspectos de la interfaz se

es lo más apropiado. Con
son realmente ciertas en la
e. También se heredan datos
La delegación deja claro que
qué aspectos de la interfaz se

de la superclase o no quiere
erclase, ajustar los métodos

ta refactorización.

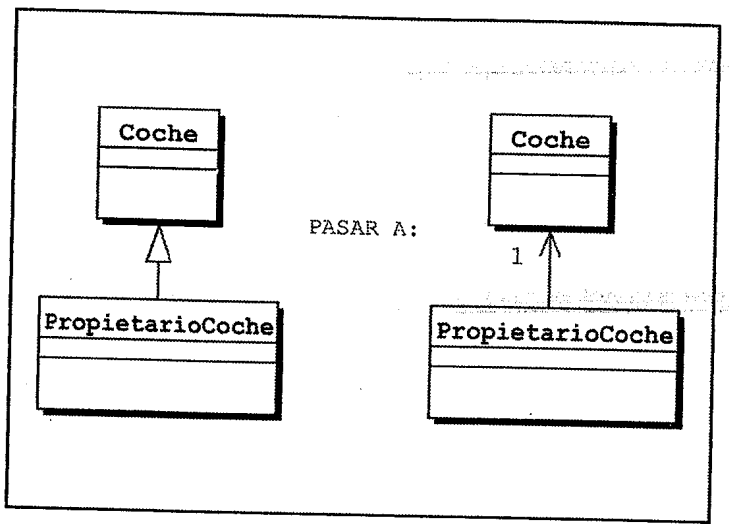

Figura 9.3. Ejemplo de una mala aplicación de la herencia y cómo Replace Inheritance with delegation *soluciona el problema mediante la delegación*

Bases de un proceso de refactorización

La refactorización como tal es una técnica aplicable a casi cualquier metodología de desarrollo software. Si bien es cierto que cada metodología particular puede tratarla de una manera concreta (generalmente marcando cuándo y cuánto), existe una secuencia general e intrínseca a la hora de refactorizar, similar a la siguiente (Wampler, 2002):

1° Revisar código y diseño para identificar refactorizaciones.

2° Aplicar una refactorización cada vez, sin cambiar la funcionalidad.

3° Aplicar pruebas unitarias, después de cada refactorización sin excepción.

4° Repetir los pasos anteriores para encontrar más refactorizaciones que aplicar.

Además, existen algunas importantes heurísticas a tener en cuenta durante el proceso:

- **No añadir funcionalidad a la vez que se refactoriza.** La regla básica de la refactorización es no cambiar la funcionalidad del código o su comportamiento observable externamente. El programa debería comportarse exactamente de la misma forma antes y después de la refactorización. Si el comportamiento cambia entonces será imposible asegurar que la refactorización no ha estropeado lo que antes ya funcionaba (Wampler, 2002).

- **Uso estricto de las pruebas.** Al realizar pruebas en cada paso se reduce el riesgo del cambio, siendo éste un requisito obligatorio tras la aplicación de cada refactorización individual.

- **Refactorizar es aplicar muchas refactorizaciones simples.** Cada refactorización individual puede realizar un pequeño progreso, pero el efecto acumulativo de aplicar muchas refactorizaciones resulta en una gran mejora de la calidad, legibilidad y diseño del código.

Respecto a cuándo se debería refactorizar hay que decir que la refactorización es algo que debe hacerse de forma constante y en pequeños pasos (Fowler, 2000), sin tener por qué dedicar tiempo adicional o exclusivo. Hay una regla que aconseja refactorizar principalmente en tres ocasiones: cuando se añade una función (haciendo más fácil añadirla o simplificándola una vez introducida), cuando se necesita reparar un error, o al revisar el código.

Los bad smells ("malos olores")

Beck y Fowler (2000) han desarrollado una lista de lo que llaman *bad smells* ("malos olores"). Son síntomas a identificar en un código y que indican qué refactorizaciones aplicar, por lo que se pueden considerar como una idea similar a la de antipatrón. Los *bad smells* pretenden localizar el problema de forma parecida a la "sensación" que los desarrolladores con un alto conocimiento y experiencia poseen sobre los buenos diseños.

La tabla 9.3 presenta los principales *bad smells* y las refactorizaciones que por lo general los resuelven. A continuación se enumeran algunos ejemplos de los principales y más conocidos.

- **Método largo.** Herencia de la programación estructurada. Los programas que viven más y mejor son aquellos con métodos cortos. Los métodos cortos son más reutilizables y aportan mayor semántica al software.

- **Lista de parámetros larga.** No es necesaria cuando se trabaja en OO, donde no se pasa al objeto todo lo que necesita sino sólo lo necesario para que él lo consiga. Los métodos con muchos parámetros elevan el acoplamiento, son difíciles de comprender y cambian con frecuencia.

- **Clase de datos.** Clases que sólo tienen atributos y métodos tipo "get" y "set". Las clases siempre deben disponer de algún comportamiento no trivial, por lo que este tipo de clases deberían cuestionarse.

- **Legado rechaza**
superclases. Si l
padre les hacen
solucionar por de
no usados de la
generalmente ab
recomendar que l

- **Envidia de carac**
clase que de la su
que toma la mayo

- **Duplicación de c**
código en más de

Bad Smell	
Atributo temporal	Ex
Cadena de mensajes	Hi
Cambio divergente	Ex
Cambios en cadena	Mo
Clase de datos	Mo
Clase grande	Ex Ob
Clase perezosa	Inl
Comentarios	Ex
Duplicación de código	Ex
Envidiade características	Mo
Estructuras de agrupación condicional	Re Re Me
Generalidad especulativa	Co
Grupos de datos	Ext
Intermediario	Re
Intimidad inadecuada	Mo Un
Jerarquías paralelas	Mo
Legado rechazado	Re
Lista de parámetros Larga	Re Obj
Método largo	Ext Obj
Obsesión primitiva	Re Re Co

Tabla

- **Legado rechazado**. Subclases que usan sólo pocas características de sus superclases. Si las clases hijas no necesitan o no quieren todo lo que sus clases padre les hacen heredar, generalmente la herencia está mal aplicada. Se suele solucionar por delegación o creando una clase hermana y colocando los métodos no usados de la clase padre en la nueva clase hermana. Ahora el padre, generalmente abstracto, posee sólo lo común a los hijos. Por ello se suele recomendar que las superclases sean abstractas.

- **Envidia de características**. Un método que utiliza más cantidad de cosas de otra clase que de la suya. Generalmente, la solución es pasar el método a la clase de la que toma la mayor parte de las cosas que necesita para trabajar.

- **Duplicación de código**. Principal razón para refactorizar. Si se observa el mismo código en más de un lugar se debe encontrar una manera de unificarlo.

Bad Smell	Refactorizaciones más utilizadas
Atributo temporal	Extract Class, Introduce Null Object
Cadena de mensajes	Hide Delegate
Cambio divergente	Extract Class
Cambios en cadena	Move Method, Move Field, Inline Class
Clase de datos	Move Method, Encapsulate Field, Encapsulate Collection
Clase grande	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Clase perezosa	Inline Class, Collapse Hierarchy
Comentarios	Extract Method, Introduce Assertion
Duplicación de código	Extract Method, Extract Class, Pull Up Method, Form Template Method
Envidiade características	Move Method, Move Field, Extract Method
Estructuras de agrupación condicional	Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object
Generalidad especulativa	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method
Grupos de datos	Extract Class, Introduce Parameter Object, Preserve Whole Object
Intermediario	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Intimidad inadecuada	Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate
Jerarquías paralelas	Move Method, Move Field
Legado rechazado	Replace Inheritance with Delegation
Lista de parámetros Larga	Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object
Método largo	Extract Method, Replace Temp with Query, Replace Method with Method Object, Decompose Conditional
Obsesión primitiva	Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy

Tabla 9.3. Bad Smells *y sus refactorizaciones más típicas*

ADOPCIÓN DE PATRONES Y REFACTORIZACIONES

El Diseño OO representa un papel determinante en el desarrollo software y en su posterior mantenimiento, porque determina la estructura de la solución software. Una vez que el diseño ha sido implementado es difícil y costoso de cambiar, aunque técnicas como la refactorización están haciendo que el cambio cada vez sea menos impactante. Por tanto, una alta calidad en el diseño es vital para reducir el coste del software. Así, Reibing (2001) comenta cómo la fase de diseño supone sólo entre un 5-10% del esfuerzo total del ciclo de vida, pero una gran parte (alrededor de un 80%) del esfuerzo total se gasta en corregir malas decisiones de diseño. Si una mala decisión no se repara en la fase de diseño su coste después de la entrega del software sería entre 5 y 100 veces mayor. En relación a lo anterior, y como comenta Opdyke (2000), existen desde hace tiempo requisitos mínimos de calidad, cualidades estructurales y características que un diseño debe mostrar para soportar extensibilidad y reutilización, como, por ejemplo, (Johnson y Foote, 1988), (Lieberherr y Holland, 1989) o los *bad smells* presentados en (Fowler, 2000). Patrones, refactorizaciones y otras tecnologías similares suponen una evolución en la forma de aplicar, estructurar, etc. el conocimiento en diseño, por lo que el valorar su incorporación en los procesos de ingeniería del software supone un paso en su perfeccionamiento y evolución. No obstante, es muy importante comprender este tipo de tecnologías más allá de su descripción técnica, observando cómo éstas pueden integrarse en una organización para la que son novedosas, el impacto que pueden tener o de qué forma pueden afectar dentro de los procesos de ingeniería del software establecidos en la organización, aspectos que se tratan en los siguientes epígrafes.

Impactos de uso y relación entre patrones y refactorizaciones

Las ventajas e inconvenientes que describen el uso patrones (ver secciones anteriores) no son excluyentes y muchas veces pueden venir combinadas. Existen situaciones donde el abuso o la no correcta aplicación de los patrones puede incluso provocar efectos negativos en el diseño, restado calidad y dificultando el mantenimiento. Generalmente, en un proceso de aplicación de patrones aparecen dos fuerzas opuestas. Por un lado, aplicar el patrón es aconsejable porque se flexibiliza el código frente a los cambios (aparte de las ya citadas ventajas de terminología común, soluciones probadas y mejores prácticas). Pero, por otro lado, existe el inconveniente de que la solución aplicada pudiera llegar a ser más compleja de lo necesario, haciendo difícil comprender el código o diseño (Prechelt *et al.*, 2000). El poco conocimiento de los programadores de cómo implementar ciertas soluciones de diseño complejas es algo que ocurre con excesiva frecuencia, como hemos podido observar en nuestra experiencia. La figura 9.4 muestra la relación que aparece entre la flexibilidad y la comprensión del diseño al aplicar patrones (Garzás, 2002).

:ACIONES

rrollo software y en su
ción software. Una vez
, aunque técnicas como
s impactante. Por tanto,
software. Así, Reibing
% del esfuerzo total del
uerzo total se gasta en
ara en la fase de diseño
:s mayor. En relación a
iace tiempo requisitos
un diseño debe mostrar
ohnson y Foote, 1988),
owler, 2000). Patrones,
lución en la forma de
alorar su incorporación
u perfeccionamiento y
le tecnologías más allá
se en una organización
é forma pueden afectar
organización, aspectos

efactorizaciones

atrones (ver secciones
combinadas. Existen
patrones puede incluso
ando el mantenimiento.
a dos fuerzas opuestas.
a el código frente a los
, soluciones probadas y
que la solución aplicada
comprender el código o
ogramadores de cómo
le ocurre con excesiva
La figura 9.4 muestra la
seño al aplicar patrones

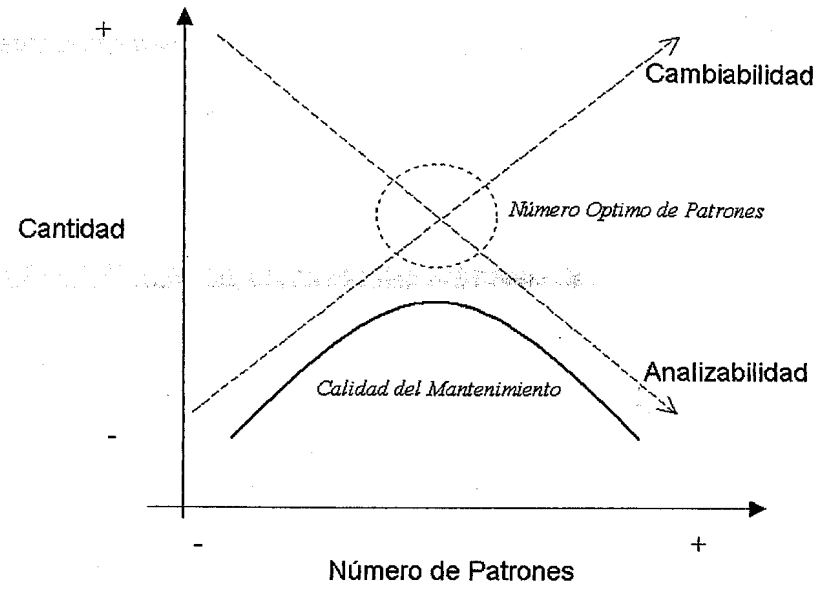

Figura 9.4. Relación entre la flexibilidad del diseño y su comprensión al aplicar patrones de diseño

Este posible efecto negativo de los patrones, que pudiera hacer complejo en exceso a un diseño, es la raíz del énfasis de ciertas tendencias metodológicas hacia la simplicidad del diseño, tendencia que tanta popularidad ha tomado desde la aparición de las metodologías denominadas "ágiles". Antes de usar refactorizaciones siempre se buscaba la solución más flexible (por ejemplo, aplicando el mayor número de patrones), siendo éstas, por lo general, más complejas y, por tanto, el software resultante más difícil de comprender y por ello de mantener. Además, ocurre que muchas veces toda esa flexibilidad no es necesaria, ya que es imposible predecir qué piezas cambiarán y dónde se necesitará la flexibilidad, por lo que se pone mucha más flexibilidad de la necesaria (síntoma de esto es la sobrecarga de patrones) (Wendorff, 2001). Con la refactorización, en lugar de implantar el máximo de soluciones flexibles antes de codificar se puede incorporar cierta flexibilidad después, tratando con diseños más simples.

Con todo lo anterior, actualmente están apareciendo metodologías que incluso llegan a rechazar el uso de patrones antes de codificar (en la fase de diseño en un proceso de ingeniería directa), en pro de un poderoso proceso de refactorización, con el propósito de mantener el diseño o código lo más semántico posible. Posturas como la anterior son las que abogan en mayor o menor grado metodologías ágiles como XP (eXtreme Programming) (Beck, 1999), que afirman que la refactorización puede ser una alternativa al diseño planificado, sin diseñar, codificando y refactorizando directamente. Este tipo de posturas produce controversia por muchas razones, pero una de las principales es que son partidarias del diseño evolutivo (donde el diseño del sistema crece conforme el sistema se

va implementado, mejorando su diseño de manera continua), oposición al clásico diseño planificado (*up-front*) (donde se usan procesos más pesados y burocráticos como el proceso Unificado) (Novatica, 2002). Normalmente, el diseño evolutivo desemboca en algo incontrolable (se convierte en un "codifica y prueba como puedas") y hace a los sistemas difíciles de mantener debido al crecimiento de la entropía. La asunción fundamental que subyace en la mayoría de las metodologías ágiles, y en concreto en XP, es que es posible un mantenimiento correcto y allanar la curva del cambio. Las prácticas que posibilitan esto son, principalmente, las pruebas, la integración continua (para mantener sincronizado al equipo) y la refactorización. Así una de las Prácticas comunes en XP es la Refactorización "extrema".

Sin llegar a extremos radicales y poco recomendables, sí es cierto que una buena refactorización cambia el rol del tradicional diseño planificado, donde ahora se relaja la presión por conseguir un diseño totalmente correcto en fases tempranas, buscando sólo una solución razonable. Se aprovecha el que cuanto más se trabaja en construir la solución más se comprende el problema. Nuestra opinión es que un uso correcto de los patrones, en su justa medida, junto con un balanceado proceso de refactorización, es la clave que posibilita un eficiente mantenimiento y dota de mayor calidad al diseño, más cuando aún la refactorización es una tecnología poco madura y que describe muchas limitaciones a la hora de reestructurar software de gran tamaño y con exceso de acoplamiento. Como muestra la figura 9.4, los ingenieros de software deben ser conscientes de estas problemáticas e intentar buscar el punto de inflexión entre flexibilidad y cambio. Este justo equilibrio aporta el beneficio más potencial de estas técnicas.

Obstáculos para la adopción industrial de patrones y refactorizaciones

Desde el punto de vista industrial ¿qué barreras se oponen a la introducción de tecnologías como patrones o refactorización para la mejora de la calidad en la construcción y mantenimiento de sistemas software? Existen ciertas características que determinan la velocidad y el grado de introducción de técnicas innovadoras en la empresa (Seen *et al.*, 2000), tales como patrones y refactorizaciones, las cuales serán adoptadas en una organización en función de cuestiones como las siguientes, que pueden, además, mostrarnos dónde se debe prestar más atención a la hora de introducir estas técnicas:

- **Beneficio relativo**. Grado en el que la innovación se percibe como mejor que la idea a la que reemplaza, con una mayor funcionalidad, calidad y productividad que la tecnología o forma de trabajar que actualmente se usa. Los patrones y las refactorizaciones, ciertamente, proporcionan un mejor método para llegar sistemáticamente a mejores prácticas. Pero pueden existir ciertos aspectos negativos a la hora de cumplimentar positivamente esta característica, destacando la creencia de que por su naturaleza, patrones y refactorizaciones formalizan

conocimiento p
describir este tip

- **Compatibilidad**
los valores exist
van a adoptar l
Diseño OO los
compatibles, ya
casi por comple

- **Complejidad.**
usar. Ciertamer
difíciles de com
para ser efectiv
con la reducció
tecnologías con
herramientas C.
pero la aún poc
considerableme
organización.

- **Experimentab**
De nuevo, este
paradigma O
refactorizacion
trabaja en OO
más difícil expe

- **Observabilida**
para otros. En
principalmente
diseño y la mej
período de tien
concluir el proy

La siguiente tabl
adopción de patrones y

conocimiento preexistente sin presentar nada nuevo, aunado a la dificultad de describir este tipo de técnicas.

- **Compatibilidad**. Grado en el que la innovación se percibe como consistente con los valores existentes, experiencias pasadas y necesidades potenciales de quienes van a adoptar la nueva tecnología. En organizaciones que utilizan Análisis y Diseño OO los patrones y las refactorizaciones de diseño pueden ser altamente compatibles, ya que la mayor potenciación de estas técnicas se ha establecido casi por completo en este paradigma.

- **Complejidad**. Grado en el que la innovación se muestra difícil de comprender y usar. Ciertamente, tanto patrones como refactorizaciones pueden llegar a ser difíciles de comprender y usar, requiriendo un nivel significativo de experiencia para ser efectivos. La sobrecarga de trabajo que pueden producir es compensada con la reducción de esfuerzo en otras fases del ciclo de vida. Por otro lado, tecnologías como la refactorización empiezan a ser eficientemente asistidas por herramientas CASE, que disminuyen el tiempo de su aplicación (Opdyke, 2000), pero la aún poca madurez de este tipo de herramientas es un elemento que frena considerablemente la introducción de patrones y refactorizaciones en una organización.

- **Experimentabilidad**. Grado en el que se puede experimentar con la innovación. De nuevo, este parámetro será positivo en organizaciones que trabajan bajo el paradigma OO, donde es menos complejo probar los patrones y refactorizaciones, introduciéndolos en proyectos piloto. Si la organización no trabaja en OO la adopción pasa primero por migrar esta tecnología, sin la cual es más difícil experimentar patrones y refactorizaciones.

- **Observabilidad**. Grado en el que los resultados de la innovación son visibles para otros. En el caso de patrones y refactorizaciones de diseño existen principalmente dos resultados observables: la creación de un vocabulario de diseño y la mejora de la mantenibilidad. El vocabulario es observable en un corto período de tiempo. La mantenibilidad sólo puede observarse tiempo después de concluir el proyecto.

La siguiente tabla resume, basándose en los parámetros anteriores, el impacto de la adopción de patrones y refactorizaciones de diseño por parte de una organización.

Características que afectan al ratio de adopción	En la organización se usa Análisis y Diseño OO	En la organización NO se usa Análisis y Diseño OO
Beneficio relativo	Moderado	Bajo
Compatibilidad	Alto	Bajo
Complejidad	Bajo	Bajo
Experimentabilidad	Alto	Bajo
Observabilidad	Neutral	Neutral

Tabla 9.4. Adopción de la tecnología de patrones y refactorizaciones

AUTOMATIZACIÓN DE PATRONES Y REFACTORIZACIONES

Cualquier técnica de ingeniería del software seria y aplicable al desarrollo y mantenimiento precisa de un proceso e, idealmente, de un soporte automatizado para su eficiente aplicación, requisitos imprescindibles en cualquier tecnología que pretenda ser pieza clave en la construcción de software de calidad, y de lo que no quedan excluidos patrones y refactorizaciones.

Como destaca Ó Cinnéide (2000), respecto a la automatización en patrones existe una carencia en el desarrollo de herramientas, y dado el tamaño de los actuales sistemas software su introducción en un proceso de mantenimiento puede suponer un altísimo coste en trabajo manual. Por otro lado, la detección de patrones usados en la construcción de un sistema es un gran valor añadido para el departamento software de una empresa a la hora de volver a crear sistemas, y esta detección es compleja sin un soporte automatizado. Por último, aun no centrando la automatización de patrones en procesos de mantenimiento, el soporte y asesoramiento a la hora de introducir patrones en nuevos diseños puede mermar notoriamente los problemas detectados en los patrones.

Ó Cinnéide (2000) afirma que automatizar la aplicación de patrones de diseño a un programa existente preservando el comportamiento es factible, y para ello basa su aproximación en el uso de refactorizaciones software. Para esto, aporta un prototipo software basado en esta filosofía. Existen otros autores que también han aportado prototipos similares, como Florijn *et al.* (1997), los cuales han desarrollado una herramienta que asiste en la generación de los elementos necesarios para introducir un patrón, integrar instancias de patrones con programas existentes y chequear la consistencia de los patrones integrados, y para lo cual muchas de las acciones que realiza esta herramienta se basan en un motor de refactorización. En Budinsky *et al.* (1996) se describe una herramienta centrada en la generación de código desde patrones de diseño, creando las declaraciones de clase y las definiciones que implementan un patrón de diseño. Bansiya (1998) presenta una herramienta, llamada DP++, que automatiza la detección, identificación y clasificación de algunos de los patrones de diseño descritos en Gamma *et al.* (1995) para C++.

Existen, por lo
uso de patrones debe
decisiones clave es qu
principalmente, puede
trabajan sólo al nivel
patrones y dependie
aproximación es que
manera de representa
describen otro tipo de
estos importantes pa
conceptos de compor
puede ser importante
clases de un patrón,
transformación del pa
cada caso la mejor
debieran trabajar desc
diseño; y ascendentes
patrón.

No obstante, si
los procesos relacio
herramientas CASE
poco contemplando el
es el caso de la conoc
el diseño de ciertos pa

La automatiz
Evidentemente, por e
larga y tediosa que s
Tokuda y Batory (2
programa de 500k lín
respecto a dos días
refactorización aún n
disciplinar y de la c
necesitan analizar la e
y que incluso pueden
una completa refacto
Extract Method que,
proceso complejo (an
abordar la extracción

ganización NO se lisis y Diseño OO
Bajo
Bajo
Bajo
Bajo
Neutral

factorizaciones

ORIZACIONES

icable al desarrollo y
: automatizado para su
logía que pretenda ser
e no quedan excluidos

ción en patrones existe
le los actuales sistemas
le suponer un altísimo
ados en la construcción
are de una empresa a la
ı soporte automatizado.
ones en procesos de
cir patrones en nuevos
os patrones.

patrones de diseño a un
:, y para ello basa su
to, aporta un prototipo
también han aportado
han desarrollado una
arios para introducir un
stentes y chequear la
las acciones que realiza
udinsky *et al.* (1996) se
esde patrones de diseño,
lementan un patrón de
?++, que automatiza la
es de diseño descritos en

Existen, por lo general, ciertos elementos que una herramienta que automatice el uso de patrones debería considerar. Florijn *et al.* (1997) comentan cómo una de las decisiones clave es qué rol representarán los patrones en el entorno de desarrollo, donde, principalmente, pueden usarse dos modelos. Por un lado tendríamos herramientas que trabajan sólo al nivel de patrón, por lo que deben generar esqueletos de código para los patrones y dependiendo de la disponibilidad de un catálogo de patrones. Otra aproximación es que la herramienta trabaje con niveles de abstracción de patrones y la manera de representar los patrones. Aparte de las vistas estructurales, los catálogos describen otro tipo de diagramas como pueden ser los de secuencia, estados, etc., todos estos importantes para resaltar características especiales del patrón como pueden ser conceptos de comportamiento dinámico del mismo. El modo de representación también puede ser importante a la hora de, por ejemplo, describir emplazamientos físicos de las clases de un patrón, por ejemplo en la parte servidor. Otro requisito a considerar es la transformación del patrón a distintos tipos de lenguaje de programación, obteniendo en cada caso la mejor aproximación en función del lenguaje. Las herramientas también debieran trabajar desde distintas aproximaciones: descendentes, insertando patrones en el diseño; y ascendentes, dado un conjunto de elementos en el programa, vincularlos a un patrón.

No obstante, si bien la mayoría de las herramientas que tratan la automatización de los procesos relacionados con patrones son prototipos, la creciente evolución de herramientas CASE comerciales para el diseño y codificación del software está poco a poco contemplando el tratamiento de patrones de diseño en mayor o menor medida, como es el caso de la conocida herramienta comercial *Together*, que contempla la inserción en el diseño de ciertos patrones.

La automatización de la refactorización es si cabe más determinante. Evidentemente, por ejemplo, llevar a cabo la refactorización manual supone una tarea larga y tediosa que se ve beneficiada de forma considerable por la automatización. En Tokuda y Batory (2001) se presenta un caso de 800 refactorizaciones aplicadas a un programa de 500k líneas de código que manualmente supusieron dos semanas de trabajo respecto a dos días de manera automatizada. El problema es que las herramientas de refactorización aún no están muy evolucionadas y esto resulta de la falta de evolución disciplinar y de la complejidad del proceso, ya que si bien hay algunas que apenas necesitan analizar la estructura del software, como aquellas que se encargan de renombrar y que incluso pueden realizarse con un editor corriente ("buscar-reemplazar"), para lograr una completa refactorización se debe analizar y manipular el árbol del programa, como *Extract Method* que, aunque pueda parecer un simple "cortar" y "pegar", requiere un proceso complejo (analizar el método, encontrar variables locales y comprender así cómo abordar la extracción).

Las herramientas de refactorización están aún en sus primeros días, pero no son desconocidas. "*Smalltalk Refactoring Browser*" (Roberts, 1999) es quizá la más antigua, evolucionada y pionera, siendo punto de referencia para todo interesado en la construcción de este tipo de software. Actualmente muchas personas y empresas están perfeccionando o evolucionando herramientas de refactorización centradas en entornos distintos a Smalltalk (especialmente Java y C++), tanto desde el plano comercial como del software libre. A continuación se enumeran las más conocidas, donde nuestro objetivo no es realizar una comparativa ni una lista completa, sino ofrecer una visión lo más aproximada del estado y madurez de la tecnología. En la actualidad, las más conocidas comercialmente son:

- *Xrefactory* (también conocida como *X-ref* y *Xref-Speller*, desarrollada por Marian Vittek, software para Emacs y Xemacs que hace refactorizaciones para Java y C).
- *Instantiations* ha producido jFactor, refactorización para Visual Age y JBuilder.
- *IntelliJ*, por su parte, ha desarrollado IDEA, IDE para Java, con una buena capacidad para encontrar referencias y con varias refactorizaciones.
- *TogetherSoft* ha comenzado añadiendo refactorizaciones a la versión 5.5 de su *control center tool.*
- Borland ha añadido soporte a la refactorización en *Jbuilder*, pero aún es bastante pobre.
- Chive Software ha puesto en marcha *Retool*, refactorización para *Jbuilder*.
- *Refactorit* (para Java e instalable en *Netbeans, Forte, JDeveloper* y *JBuilder*).

Como software libre también un buen número de desarrollos: Transmogrify (puede usarse para JBuilder y Forte), Eclipse, JRefactory (para Java), JavaRefactor (para Jedit), Guru (para programas en SELF), Bicycle Repair Man (para Python), etc.

Debemos ser conscientes de que muchas de las anteriores no podemos considerarlas estrictamente como herramientas de refactorización, ya que no contemplan las pruebas. Para ello, una herramienta muy útil y gratuita para organizar pruebas es la familia xUnit, disponible para muchos lenguajes incluyendo Java (JUnit), C++, y Visual Basic. Por último, no debe olvidarse que las herramientas son sólo un apoyo y que la decisión final sobre qué y cuándo refactorizar está siempre en manos del diseñador o del desarrollador.

CONCLUSIONE

Los expertos har
años cuando estas ideas
alcanzado su mayor po

De entre todos
refactorizaciones, la n
productividad y calida
discusiones de diseño d
se reducirá y el produc
éxito probado. Ambo
simplemente principios
conceptos probados, nc
no encuentra en los asp
el componente human
patrones o refactorizaci
están por las restriccic
afirmaciones.

No podemos de
técnicas en lo que con
Aspectos como, por e
aplicabilidad, la dificult
la refactorización explíc
aún no del todo resuelta
ejemplo, Crespo y Mar
etc. comienzan a tratar e

Con todo, patron
evolucionando con muc
convertirán en element
desarrollo o mantenimie
comienza a carecer de s
sobre las bases ya afia
(1995) "una cosa que lo
el principio [...]. Cuanc
experiencia es lo que les

AGRADECIMIEI

Esta investigació
Subdirección General
Tecnología (TIC 2000-1

ros días, pero no son
quizá la más antigua,
lo interesado en la
nas y empresas están
entradas en entornos
lano comercial como
cidas, donde nuestro
ofrecer una visión lo
actualidad, las más

sarrollada por Marian
ciones para Java y C).

sual Age y JBuilder.

Java, con una buena
aciones.

a la versión 5.5 de su

r, pero aún es bastante

n para *Jbuilder*.

eloper y *JBuilder*).

: Transmogrify (puede
aRefactor (para Jedit),
), etc.

teriores no podemos
ya que no contemplan
organizar pruebas es la
(JUnit), C++, y Visual
ólo un apoyo y que la
nos del diseñador o del

CONCLUSIONES

Los expertos han usado ideas de eficacia probada desde siempre. Es en los últimos años cuando estas ideas, materializadas en conceptos como patrón o refactorización, están alcanzado su mayor popularidad y difusión.

De entre todos los beneficios descubiertos mediante el uso de patrones y refactorizaciones, la mejora de la comunicación es el más potente. Mejoras en la productividad y calidad deberían ser inmediatas cuando los equipos puedan sostener discusiones de diseño desde un alto nivel, porque el tiempo empleado en estas discusiones se reducirá y el producto mejorará como resultado de la incorporación de soluciones de éxito probado. Ambos conceptos resuelven problemas, capturan soluciones y no simplemente principios abstractos o estrategias. Además, no debe olvidarse que son conceptos probados, no teorías o especulaciones. Si bien, aún hoy, este tipo de técnicas, no encuentra en los aspectos técnicos su mayor problema a la hora de implantarse sino en el componente humano, ya que muchas personas, con relación a técnicas como los patrones o refactorizaciones, dicen estar concienciadas por la calidad pero que más aún lo están por las restricciones temporales, sin comprender lo paradójico de este tipo de afirmaciones.

No podemos dejar de destacar la poca madurez que aún rodea a este tipo de técnicas en lo que concierne a su aplicación metodológica y su soporte automatizado. Aspectos como, por ejemplo, la propia clasificación de estas técnicas, su nivel de aplicabilidad, la dificultad de saber cuándo y dónde aplicarlas, metodologías que traten a la refactorización explícitamente (aparte de las de carácter "extremo"), etc. son cuestiones aún no del todo resueltas, aunque en la actualidad existen muchos autores que, como por ejemplo, Crespo y Marqués (2001), Garzás y Piattini (2001a y 2001b), Garzás (2002), etc. comienzan a tratar esta problemática.

Con todo, patrones y refactorizaciones se presentan como tecnologías que están evolucionando con mucha rapidez, a tener muy en cuenta y que con total seguridad se convertirán en elementos obligados para asegurar la calidad de cualquier proyecto de desarrollo ó mantenimiento software, en un futuro muy próximo, en el que la reinvención comienza a carecer de sentido y en el que la única manera de evolucionar pasa por crecer sobre las bases ya afianzadas en ingeniería del software, como afirma Gamma *et al.* (1995) "una cosa que los diseñadores expertos no hacen es resolver cada problema desde el principio [...]. Cuando encuentran una buena solución la usan una y otra vez. Esta experiencia es lo que les hace expertos".

AGRADECIMIENTOS

Esta investigación forma parte del proyecto DOLMEN-MEDEO, financiado por la Subdirección General de Proyectos de Investigación del Ministerio de Ciencia y Tecnología (TIC 2000-1673-C06-06).

BIBLIOGRAFÍA

Alexander C. (1977). *A pattern language.* Nueva York: Oxford University Press.

Alexander C. (1979). *The timeless way for building.* Nueva York: Oxford University Press.

Ambler S. W. (1998). *Process Patterns Building Large-Scale Systems Using Object Technology.* Cambridge University Press/SIGS Books.

Appleton B. (2001). *Patterns and Software: Essential Concepts and Terminology.* En *http://www.enteract.com/~bradapp/docs/patterns-intro.html*

Bansiya J. (1998, junio). "Automating Design-Pattern Identification". *Dr. Dobb's Journal.*

Beck K. (1999). *Extreme Programming Explained.* Addison-Wesley.

Budinsky F. J., Finnie M. A., Vlissides J. M. y Yu P. S. (1996). "Automatic code generation from design patterns". *IBM Systems Journal,* 35(2).

Buschmann F., Meunier R., Rohnert H., Sommerlad P. y Stal M. (1996). *A System of Patterns: Pattern-Oriented Software Architecture.* Addison-Wesley.

Cline M. P. (1996, octubre). "The Pros and Cons of Adopting and Applying Design Patterns in the Real World". *Communications of the ACM,* 39 (10), pp. 47-49

Coad P. (1992, septiembre). "Object-Oriented Patterns". *Communications of the* ACM, 35 (9), 152-159.

Coplien J. (1991). *Advanced C++ Programming Styles and Idioms.* Addison-Wesley

Coplien J. (1998). *Software Design Patterns: Common Questions and Answers.* En Linda Rising (Ed.), *The Patterns Handbook: Techniques, Strategies and Applications* (pp. 311-320). Nueva York: Cambridge University Press.

Creel C. (1999, diciembre). "Requirements by Pattern". *Software Development Magazine.* Visitado en abril, 1999, en www.sdmagazine.com

Crespo Y. y Marqués J. M. (2001). "Definición de un marco de trabajo para el análisis de refactorizaciones de software". *JISBD 2001, VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD01).* Almagro, Ciudad Real, España.

rd University Press.

Nueva York: Oxford

-Scale Systems Using

epts and Terminology.

ification". *Dr. Dobb's*

-Wesley.

996). "Automatic code

al M. (1996). *A System*
ley.

dopting and Applying
39 (10), pp. 47-49

Communications of the

and Idioms. Addison-

stions and Answers. En
es and Applications (pp.

Software Development

narco de trabajo para el
nadas de Ingeniería del
paña.

Florijn G., Meijers M. y van Winsen P. (1997). "Tool Support for Object-Oriented Patterns". En Askit M., S. Matsuoka (Eds.), *ECOOP'97. Lecture Notes in Computer Science,* 1241, Berlín, Springer-Verlag.

Fowler M. (1997) *Analysis Patterns: Reusable Object Models.* Addison-Wesley.

Fowler M. (2000). *Refactoring. Improving the Existing Code.* Addison – Wesley.

Gabriel R. P. (1996). *Patterns of Software: Tales from the Software Community.* Oxford University Press.

Gamma E., Helm R., Johnson R. y Vlissides J. (1995). *Design patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley.

Garzás J. (2002). Caracterización y aplicación del conocimiento en diseño orientado a objetos. *Trabajo de investigación. Programa de doctorado: Tecnologías Informáticas Avanzadas.* Universidad de Castilla-La Mancha, Escuela Superior de Informática de Ciudad Real.

Garzás J. y Piattini M. (2001a, octubre). "From the OO Design Principles to the Formal Understanding of the OO Design Patterns". *OOPSLA 2001, Workshop Beyond Design: Patterns (mis)used,* Bahía Tampa, Florida, EEUU.

Garzás J. y Piattini M. (2001b, agosto). "Principles and Patterns in the Object Oriented Design". En Wang Y., Patel S. y Johnston R.H. (Eds.), *OOIS 2001, 7th International Conference on Object-Oriented Information Systems* (pp. 15-24). Universidad de Calgary, Calgary, Canadá: Springer.

Grand M. (2001). *Java Enterprise Design Patterns. Patterns in Java Volume 3.* Wiley Computer.

Johnson R. y Foote B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming* 1(2), 22-35.

Koenig A. (1995, marzo-abril). "Patterns and Antipatterns". *Journal of Object-Oriented Programming (JOOP),* 8(1), 46-48.

Lehman M. M. (1980). "Programs, Life Cycles and Laws of Software Evolution". *Proceedings of the IEEE, 68(9),* pp. 1060-1076.

Lehman M. M., Perry, D. E. y Ramil, J. F. (1998). "Implications of Evolution Metrics on Software Maintenance". En Khoshgoftaar y Bennet (Eds.), *Proceedings of the International Conference on Software Maintenance,* pp. 208-217. Maryland, EE.UU: IEEE Computer Society.

Lieberherr K. J y Holland I. M (1989, septiembre-octubre). "Assuring Good Style For Object-Oriented Programs". *IEEE Software*, 6(5), pp. 38-48.

Marinescu F. (2002). *EJB Design Patterns Advanced Patterns, Processes, and Idioms.* Wiley Computer.

Novatica. (2002, marzo-abril). "Extreme Programming. Programación Extrema". *Revista de la Asociación de Técnicos en Informática.*

Ó Cinnéide, M. (2000). *Automated Application of Design Patterns: a Refactoring Approach.* Tesis Doctoral. Universidad de Dublin, Trinity College.

OOPSLA (2001). "Workshop, Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems". *OOPSLA 2001, Workshop Beyond Design: Patterns (mis)used.* Bahía Tampa, Florida, EEUU.

Opdyke W. (1992). *Refactoring Object-Oriented Frameworks.* Tesis Doctoral, Departamento de Computer Science. University de Illinois at Urbana-Champaign.

Opdyke W. (2000). "Refactoring, reuse and reality". En Fowler M. (Ed.), *Refactoring. Improving The Existing Code.* Addison–Wesley.

Prechelt L., Unger B., Philippsen M. y Tichy W. (1997). "Two controlled Experiments Assessing the Usefulness of Design Patterns Information During Program Maintenance". *Empirical Software Engineering.*

Prechelt L., Unger B., Tichy W. y Bossler P. (2000, Septiembre). "A controlled Experiments in Maintenance Comparing Design Patterns to Simpler Solutions". *IEEE Transactions on Software Engineering.*

Reibing R. (2001, octubre). "Assessing the Quality of OO Designs". *OOPSLA 2001.* Bahía Tampa, Florida, EEUU

Riehle D. y Zllighoven H. (1996). Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2 (1), pp.3-13.

Roberts D. B. (1999). *Practical Analysis For Refactoring.* PhD Thesis, Department of Computer Science. Universdad de Illinois.

Schmidt D. C. (1995, octubre). Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM,* 38,10, pp. 65-74.

Schulz B., Gen
Introduction of Desig
Oriented Languages a

Seen M., Taylo
Adoption". *TOOLS Eu*

Shaw M. y Gar
Discipline. Prentice Ha

Stroulia E. y K
Experience Report". I
International Confere
Calgary, Calgary, Can

Tokuda L. y
Refactorings". *Kluwer*
Nº 1, pp. 89-120

Wampler B. E.
and UML. Addison-We

Wendorff P.
Software Reengineeri
CSMR 2001-Europea
pp. 77-84.

Yacoub S. M.
(POAD): A Structural
2000, Technology of Ob

Zimmer W. (19
Schmidt D. C. (Eds.), P

Schulz B., Genssler T., Mohr B. y Zimmer W. (1998). "On the Computer Aided Introduction of Design Patterns into Object Oriented Systems". *Technology of the Object Oriented Languages and Systems, TOOLS 1998*

Seen M., Taylor, P. y Dick M. (2000). "Applying a Crystal Ball to Design Pattern Adoption". *TOOLS Europe,* 33, pp. 443-454

Shaw M. y Garlan D. (1996). *Software Architecture. Perspectives on an Emerging Discipline.* Prentice Hall.

Stroulia E. y Kapoor R (2001). "Metrics for Refactoring-based Development: An Experience Report". En Wang Y., Patel S. y Johnston R.H. (Eds.), *OOIS 2001, 7th International Conference on Object-Oriented Information Systems.* Universidad de Calgary, Calgary, Canadá: Springer.

Tokuda L. y Batory D. (2001). "Envolving Object-Oriented Designs with Refactorings". *Kluwer Academic Publishers - Automated Software Engineering*, vol. 8, Nº 1, pp. 89-120

Wampler B. E. (2002). *The Essence of Object-Oriented Programming with Java and UML.* Addison-Wesley.

Wendorff P. (2001). "Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project". *CSMR 2001-European Conference On Software Maintenance And Reengineering,* pp. 77-84.

Yacoub S. M. y Ammar H. (2000). "Pattern–Oriented Analysis and Design (POAD): A Structural Composition Approach to Glue Design Patterns". En *TOOLS 2000, Technology of Object-Oriented Languages and Systems.*

Zimmer W. (1995). "Relationships between design Patterns". En Coplien J. y Schmidt D. C. (Eds.), PLOP, *Pattern Languages of Program Design.* Addison Wesley.