Patrones de Diseño

Elementos de software orientado a objetos reutilizable

ERICH GAMMA

RICHARD HELM

RALPH JOHNSON

JOHN VLISSIDES

Traducción:

César Fernández Acebal
Universidad de Oviedo

Revisión técnica:

Juan Manuel Cueva Lovelle
Universidad de Oviedo

Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • São Paulo • Reading, Massachusetts • Harlow, England

Diseñar software orientado a objetos es difícil, y aún lo es más diseñar software orientado a objetos reutilizable. Hay que encontrar los objetos pertinentes, factorizarlos en clases con la granularidad adecuada, definir interfaces de clases y jerarquías de herencia y establecer las principales relaciones entre esas clases y objetos. Nuestro diseño debe ser específico del problema que estamos manejando, pero también lo suficientemente general para adecuarse a futuros requisitos y problemas. También queremos evitar el rediseño, o al menos minimizarlo. Los diseñadores experimentados de software orientado a objetos nos dirán que es difícil, sino imposible, lograr un diseño flexible y reutilizable a la primera y que, antes de terminar un diseño, es frecuente intentar reutilizarlo varias veces, modificándolo cada una de ellas.

Sin embargo, estos diseñadores experimentados realmente consiguen hacer buenos diseños, mientras que los diseñadores novatos se ven abrumados por las opciones disponibles y tienden a recurrir a las técnicas no orientadas a objetos que ya usaron antes. A un principiante le lleva bastante tiempo aprender en qué consiste un buen diseño orientado a objetos. Es evidente que los diseñadores experimentados saben algo que los principiantes no. ¿Qué es?

Algo que los expertos saben que *no* hay que hacer es resolver cada problema partiendo de cero. Por el contrario, reutilizan soluciones que ya les han sido útiles en el pasado. Cuando encuentran una solución buena, la usan una y otra vez. Esa experiencia es parte de lo que les convierte en expertos. Por tanto, nos encontraremos con patrones recurrentes de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos. Estos patrones resuelven problemas concretos de diseño y hacen que los diseños orientados a objetos sean más flexibles, elegantes y reutilizables. Los patrones ayudan a los diseñadores a reutilizar buenos diseños al basar los nuevos diseños en la experiencia previa. Un diseñador familiarizado con dichos patrones puede aplicarlos inmediatamente en los problemas de diseño sin tener que redescubrirlos.

Ilustremos este punto con una analogía. Los novelistas y escritores rara vez diseñan las tramas de sus obras desde cero, sino que siguen patrones como el del *héroe trágico* (Macbeth, Hamlet, etc.) o *la novela romántica* (innumerables novelas de amor). Del mismo modo, los diseñadores orientados a objetos siguen patrones como "representar estados con objetos" o "decorar objetos de manera que se puedan añadir y borrar funcionalidades fácilmente". Una vez que conocemos el patrón, hay muchas decisiones de diseño que se derivan de manera natural.

Todos sabemos el valor de la experiencia en el diseño. ¿Cuántas veces hemos tenido un *déjà-vu* de diseño —esa sensación de que ya hemos resuelto ese problema antes, pero no sabemos exactamente dónde ni cómo—? Si pudiéramos recordar los detalles del problema anterior y de cómo lo resolvimos podríamos valernos de esa experiencia sin tener que reinventar la solución. Sin embargo, no solemos dedicarnos a dejar constancia de nuestra experiencia en el diseño de software para que la usen otros.

El propósito de este libro es documentar la experiencia en el diseño de software orientado a objetos en forma de **patrones de diseño**. Cada patrón nomina, explica y evalúa un diseño importante y recurrente en los sistemas orientados a objetos. Nuestro objetivo es representar esa experiencia de diseño de forma que pueda ser reutilizada de manera efectiva por otras personas. Para lograrlo, hemos documentado algunos de los patrones de diseño más importantes y los presentamos como un catálogo.

Los patrones de diseño hacen que sea más fácil reutilizar buenos diseños y arquitecturas. Al expresar como patrones de diseño técnicas que ya han sido probadas, las estamos haciendo más accesibles para los desarrolladores de nuevos sistemas. Los patrones de diseño nos ayudan a elegir las alternativas de diseño que hacen que un sistema sea reutilizable, y a evitar aquéllas que dificultan dicha reutilización. Pueden incluso mejorar la documentación y el mantenimiento de los sistemas existentes al proporcionar una especificación explícita de las interacciones entre clases y objetos y de cuál es su intención. En definitiva, los patrones de diseño ayudan a un diseñador a lograr un buen diseño más rápidamente.

Ninguno de los patrones de diseño de este libro describe diseños nuevos o que no hayan sido probados. Se han incluido sólo aquellos que se han aplicado más de una vez en diferentes sistemas. La mayoría de ellos estaba sin documentar, y existían bien como parte del repertorio de la comunidad de la orientación a objetos, bien como elementos de algunos buenos sistemas orientados a objetos —y de ninguna de ambas formas resultan fáciles de aprender por los diseñadores novatos—. Así que, aunque estos diseños no son nuevos, los hemos expresado de una forma nueva y accesible: como un catálogo de patrones de diseño que tienen un formato consistente.

A pesar del tamaño del libro, los patrones de diseño que hay en él representan sólo una parte de lo que puede saber un experto. No contiene patrones que tengan que ver con concurrencia o programación distribuida o en tiempo real. Tampoco hay patrones de dominios específicos. No se cuenta cómo construir interfaces de usuario, cómo escribir controladores de dispositivos o cómo usar una base de datos orientada a objetos. Cada una de estas áreas tiene sus propios patrones, y sería bueno que alguien los catalogase también.

1.1. ¿QUÉ ES UN PATRÓN DE DISEÑO?

Según Christopher Alexander, "cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces" [AIS+77, página *x*]. Aunque Alexander se refería a patrones en ciudades y edificios, lo que dice también es válido para patrones de diseño orientados a objetos. Nuestras soluciones se expresan en términos de objetos e interfaces, en vez de paredes y puertas, pero en la esencia de ambos tipos de patrones se encuentra una solución a un problema dentro de un contexto.

En general, un patrón tiene cuatro elementos esenciales:

1. El **nombre del patrón** permite describir, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias. Al dar nombre a un patrón inmediatamente estamos incrementado nuestro vocabulario de diseño, lo que nos permite diseñar con mayor abstracción. Tener un vocabulario de patrones nos permite hablar de ellos con otros colegas, mencionarlos en nuestra documentación y tenerlos nosotros mismos en cuenta. De esta manera, resulta más fácil pensar en nuestros diseños y transmitirlos a otros, junto con sus ventajas e inconvenientes. Encontrar buenos nombres ha sido una de las partes más difíciles al desarrollar nuestro catálogo.
2. El **problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Puede describir problemas concretos de diseño (por ejemplo, cómo representar algoritmos como objetos), así como las estructuras de clases u objetos que son sintomáticas de un diseño inflexible. A veces el problema incluye una serie de condiciones que deben darse para que tenga sentido aplicar el patrón.
3. La **solución** describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o una implementación en concreto, sino que un patrón es más bien como una plantilla que puede aplicarse en muchas situaciones diferentes. El patrón proporciona una descripción abstracta de un problema de diseño y cómo lo resuelve una disposición general de elementos (en nuestro caso, clases y objetos).
4. Las **consecuencias** son los resultados así como las ventajas e inconvenientes de aplicar el patrón. Aunque cuando se describen decisiones de diseño muchas veces no se reflejan sus consecuencias, éstas son fundamentales para evaluar las alternativas de diseño y comprender los costes y beneficios de aplicar el patrón. Las consecuencias en el software suelen referirse al equilibrio entre espacio y tiempo. También pueden tratar cuestiones de lenguaje e implementación. Por otro lado, puesto que la reutilización suele ser uno de lo factores de los diseños orientados a objetos, las consecuencias de un patrón incluyen su impacto sobre la flexibilidad,

extensibilidad y portabilidad de un sistema. Incluir estas consecuencias de un modo explícito nos ayudará a comprenderlas y evaluarlas.

Qué es y qué no es un patrón de diseño es una cuestión que depende del punto de vista de cada uno. Lo que para una persona es un patrón puede ser un bloque primitivo de construcción para otra. En este libro nos hemos centrado en patrones situados en un cierto nivel de abstracción. *Patrones de Diseño* no se ocupa de diseños como listas enlazadas y tablas de dispersión (*hash*) que pueden codificarse en clases y reutilizarse como tales. Tampoco se trata de complicados diseños específicos de un dominio para una aplicación o subsistema completo. Los patrones de diseño de este libro son *descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto.*

Un patrón de diseño nomina, abstrae e identifica los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizable. El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se centra en un problema concreto, describiendo cuándo aplicarlo y si tiene sentido hacerlo teniendo en cuenta otras restricciones de diseño, así como las consecuencias y las ventajas e inconvenientes de su uso. Por otro lado, como normalmente tendremos que implementar nuestros diseños, un patrón también proporciona código de ejemplo en C++, y a veces en Smalltalk, para ilustrar una implementación.

Aunque los patrones describen diseños orientados a objetos, están basados en soluciones prácticas que han sido implementadas en los lenguajes de programación orientados a objetos más usuales, como Smalltalk y C++, en vez de mediante lenguajes procedimentales (Pascal, C, Ada) u otros lenguajes orientados a objetos más dinámicos (CLOS, Dylan, Self). Nosotros hemos elegido Smalltalk y C++ por una cuestión pragmática: nuestra experiencia diaria ha sido con estos lenguajes, y éstos cada vez son más populares.

La elección del lenguaje de programación es importante, ya que influye en el punto de vista. Nuestros patrones presuponen características de los lenguajes Smalltalk y C++, y esa elección determina lo que puede implementarse o no fácilmente. Si hubiéramos supuesto lenguajes procedimentales, tal vez hubiéramos incluido patrones llamados "Herencia", "Encapsulación" y "Polimorfismo". De manera similar, algunos de nuestros patrones están incluidos directamente en lenguajes orientados a objetos menos corrientes. CLOS, por ejemplo, tiene multi-métodos que reducen la necesidad de patrones como el Visitor (página 305). De hecho, hay suficientes diferencias entre Smalltalk y C++ como para que algunos patrones puedan expresarse más fácilmente en un lenguaje que en otro (por ejemplo, el Iterator (237)).

2. PATRONES DE DISEÑO EN EL MVC DE SMALLTALK

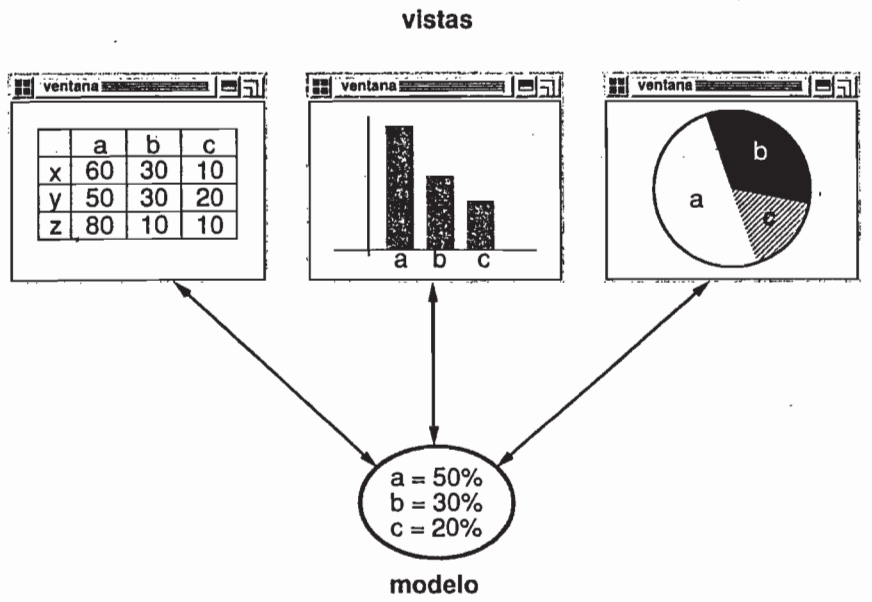
La tríada de clases Modelo/Vista/Controlador (MVC) [KP88] se usa para construir interfaces de usuario en Smalltalk-80. Observar los patrones de diseño que hay en MVC debería ayudar a entender qué queremos decir con el término "patrón".

MVC consiste en tres tipos de objetos. El Modelo es el objeto de aplicación, la Vista es su representación en pantalla y el Controlador define el modo en que la interfaz reacciona a la entrada del usuario. Antes de MVC, los diseños de interfaces de usuario tendían a agrupar estos objetos en uno solo. MVC los separa para incrementar la flexibilidad y reutilización.

MVC desacopla las vistas de los modelos estableciendo entre ellos un protocolo de suscripción/notificación. Una vista debe asegurarse de que su apariencia refleja el estado del modelo. Cada vez que cambian los datos del modelo, éste se encarga de avisar a las vistas que dependen de él. Como respuesta a dicha notificación, cada vista tiene la oportunidad de actualizarse a sí misma. Este enfoque

permite asignar varias vistas a un modelo para ofrecer diferentes presentaciones. También se pueden crear nuevas vistas de un modelo sin necesidad de volver a escribir éste.

El siguiente diagrama muestra un modelo y tres vistas (hemos dejado fuera los controladores para simplificar). El modelo contiene algunos valores de datos y las vistas, consistentes en una hoja de cálculo, un histograma y un gráfico de tarta que muestran estos datos de varias formas. El modelo se comunica con sus vistas cuando cambian sus valores, y las vistas se comunican con el modelo para acceder a éstos.



Si nos fiamos de él, este ejemplo refleja un diseño que desacopla las vistas de los modelos. Pero el diseño es aplicable a un problema más general: desacoplar objetos de manera que los cambios en uno puedan afectar a otros sin necesidad de que el objeto que cambia conozca detalles de los otros. Dicho diseño más general se describe en el patrón Observer (página 269).

Otra característica de MVC es que las vistas se pueden anidar. Por ejemplo, un panel de control con botones se puede implementar como una vista compleja que contiene varias vistas de botones anidadas. La interfaz de usuario de un inspector de objetos puede consistir en vistas anidadas que pueden ser reutilizadas en un depurador. MVC permite vistas anidadas gracias a la clase VistaCompuesta, una subclase de Vista. Los objetos VistaCompuesta pueden actuar simplemente como objetos Vista, es decir, una vista compuesta puede usarse en cualquier lugar donde pudiera usarse una vista, pero también contiene y gestiona vistas anidadas.

De nuevo, podríamos pensar en él como un diseño que nos permite tratar a una vista compuesta exactamente igual que a uno de sus componentes. Pero el diseño es aplicable a un problema más general, que ocurre cada vez que queremos agrupar objetos y tratar al grupo como a un objeto individual. El patrón Composite (151) describe este diseño más general. Dicho patrón permite crear una jerarquía

en la que algunas subclases definen objetos primitivos (por ejemplo, Boton) y otras, objetos compuestos (VistaCompuesta), que ensamblan los objetos primitivos en otros más complejos.

MVC también permite cambiar el modo en que una vista responde a la entrada de usuario sin cambiar su representación visual. En este sentido, tal vez queramos cambiar cómo responde al teclado, por ejemplo, o hacer que use un menú contextual en vez de atajos de teclado. MVC encapsula el mecanismo de respuesta en un objeto Controlador. Hay una jerarquía de controladores y es fácil crear un nuevo controlador como una variación de uno existente.

Una vista usa una instancia de una subclase de Controlador para implementar una determinada estrategia de respuesta; para implementar una estrategia diferente, simplemente basta con sustituir la instancia por otra clase de controlador. Incluso es posible cambiar el controlador de una vista en tiempo de ejecución, para hacer que la vista cambie el modo en que responde a la entrada de usuario. Por ejemplo, para desactivar una vista y que no acepte entradas basta con asignarle un controlador que haga caso omiso de los eventos de entrada.

La relación entre Vista y Controlador es un ejemplo del patrón Strategy (289). Una Estrategia es un objeto que representa un algoritmo. Es útil cuando queremos reemplazar el algoritmo, ya sea estática o dinámicamente, cuando existen muchas variantes del mismo o cuando tiene estructuras de datos complejas que queremos encapsular.

MVC usa otros patrones de diseño, tales como el Factory Method (99) para especificar la clase controlador predeterminada de una vista, y el Decorator (161) para añadir capacidad de desplazamiento a una vista. Pero las principales relaciones en MVC se dan entre los patrones de diseño Observer, Composite y Strategy.

1.3. DESCRIPCIÓN DE LOS PATRONES DE DISEÑO

¿Cómo describimos los patrones de diseño? Las notaciones gráficas, aunque importantes, no son suficientes. Simplemente representan el producto final del proceso de diseño, como las relaciones entre clases y objetos. Para reutilizar el diseño, debemos hacer constar las decisiones, alternativas y ventajas e inconvenientes que dieron lugar a él. También son importantes los ejemplos concretos, porque nos ayudan a ver el diseño en acción.

Describimos los patrones de diseño empleando un formato consistente. Cada patrón se divide en secciones de acuerdo a la siguiente plantilla. Ésta da una estructura uniforme a la información, haciendo que los patrones de diseño sean más fáciles de aprender, comparar y usar.

Nombre del patrón y clasificación
El nombre del patrón transmite sucintamente su esencia. Un buen nombre es vital, porque pasará a formar parte de nuestro vocabulario de diseño. La clasificación del patrón refleja el esquema que se presenta en la Sección 1.5.

Propósito
Una frase breve que responde a las siguientes cuestiones: ¿Qué hace este patrón de diseño? ¿En qué se basa? ¿Cuál es el problema concreto de diseño que resuelve?

También conocido como
Otros nombres, si existen, por los que se conoce al patrón.

Motivación
Un escenario que ilustra un problema de diseño y cómo las estructuras de clases y objetos del patrón resuelven el problema. El escenario ayudará a entender la descripción que sigue.

Aplicabilidad
¿En qué situaciones se puede aplicar el patrón de diseño? ¿Qué ejemplos hay de malos diseños que el patrón puede resolver? ¿Cómo se puede reconocer dichas situaciones?

Estructura
Una representación gráfica de las clases del patrón usando una notación basada en la Técnica de Modelado de Objetos (OMT) [RBP+91]. También hacemos uso de diagramas de interacción [JCJO92, Boo94] para mostrar secuencias de peticiones y colaboraciones entre objetos. El Apéndice B describe estas notaciones en detalle.

Participantes
Las clases y objetos participantes en el patrón de diseño, junto con sus responsabilidades.

Colaboraciones
Cómo colaboran los participantes para llevar a cabo sus responsabilidades.

Consecuencias
¿Cómo consigue el patrón sus objetivos? ¿Cuáles son las ventajas e inconvenientes y los resultados de usar el patrón? ¿Qué aspectos de la estructura del sistema se pueden modificar de forma independiente?

Implementación
¿Cuáles son las dificultades, trucos o técnicas que deberíamos tener en cuenta a la hora de aplicar el patrón? ¿Hay cuestiones específicas del lenguaje?

Código de ejemplo
Fragmentos de código que muestran cómo se puede implementar el patrón en C++ o en Smalltalk.

Usos conocidos
Ejemplos del patrón en sistemas reales. Incluimos al menos dos ejemplos de diferentes dominios.

Patrones relacionados
¿Qué patrones de diseño están estrechamente relacionados con éste? ¿Cuáles son las principales diferencias? ¿Con qué otros patrones debería usarse?

Los apéndices proporcionan información adicional que le ayudará a entender los patrones y las discusiones que los rodean. El Apéndice A es un glosario de la terminología empleada en el libro. En el Apéndice B que ya hemos mencionado se presentan las diferentes notaciones. También describiremos aspectos de las notaciones a medida que las vayamos introduciendo en las discusiones venideras. Finalmente, el Apéndice C contiene el código fuente de las clases básicas que usamos en los ejemplos.

1.4. EL CATÁLOGO DE PATRONES DE DISEÑO

El catálogo que comienza en la página 71 contiene 23 patrones de diseño. A continuación mostraremos el nombre y el propósito de cada uno de ellos para ofrecerle una perspectiva general. El número entre paréntesis que sigue a cada patrón representa el número de página de éste (una convención que seguiremos a lo largo de todo el libro).*

Abstract Factory **(Fábrica Abstracta) (79)**
Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

Adapter **(Adaptador) (131)**
Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.

Bridge **(Puente) (141)**
Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

* También se proporciona una traducción del nombre del patrón, si bien éste se deja en lengua inglesa por compatibilidad con toda la notación existente sobre patrones. *(N. del T.)*

Builder **(Constructor) (89)**
Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Chain of Responsibility **(Cadena de Responsabilidad) (205)**
Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.

Command **(Orden) (215)**
Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

Composite **(Compuesto) (151)**
Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Decorator **(Decorador) (161)**
Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una altemativa flexible a la herencia para extender la funcionalidad.

Facade **(Fachada) (171)**
Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

Factory Method **(Método de Fabricación) (99)**
Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Flyweight **(Peso Ligero)* (179)**
Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

Interpreter **(Intérprete) (225)**
Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

Iterator **(Iterador) (237)**
Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

Mediator **(Mediador) (251)**
Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

Memento **(Recuerdo) (261)**
Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

Observer **(Observador) (269)**
Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él.

Prototype **(Prototipo) (109)**
Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de este prototipo.

Proxy **(Apoderado) (191)**
Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Singleton **(Único) (119)**
Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

* Literalmente, "peso mosca". *(N. del T.)*

State **(Estado) (279)**
Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Strategy **(Estrategia) (289)**
Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Template Method **(Método Plantilla) (299)**
Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

Visitor **(Visitante) (305)**
Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

1.5. ORGANIZACIÓN DEL CATÁLOGO

Los patrones de diseño varían en su granularidad y nivel de abstracción. Dado que existen muchos patrones de diseño, es necesario un modo de organizarlos. Esta sección clasifica los patrones de diseño de manera que podamos referirnos a familias de patrones relacionados. La clasificación ayuda a aprender más rápidamente los patrones del catálogo, y también puede encauzar los esfuerzos para descubrir nuevos patrones.

Tabla 1.1: Patrones de diseño

		Propósito		
		De Creación	**Estructurales**	**De comportamiento**
Ámbito	**Clase**	Factory Method (99)	Adapter (de clases) (131)	Interpreter (225) Template Method (299)
	Objeto	Abstract Factory (79) Builder (89) Prototype (109) Singleton (119)	Adapter (de objetos) (131) Bridge (141) Composite (151) Decorator (161) Facade (171) Flyweight (179) Proxy (191)	Chain of Responsibility (205) Command (215) Iterator (237) Mediator (251) Memento (261) Observer (269) State (279) Strategy (289) Visitor (305)

Nosotros clasificamos los patrones de diseño siguiendo dos criterios (Tabla 1.1). El primero de ellos, denominado **propósito**, refleja qué hace un patrón. Los patrones pueden tener un propósito **de creación, estructural** o **de comportamiento**. Los patrones de creación tienen que ver con el proceso de creación de objetos. Los patrones estructurales tratan con la composición de clases u objetos. Los de