2

(I)

Patrones de diseño

Introducción

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2017-2018

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal

Resumen

El «libro de patrones» comienza precisamente describiendo algunos de los principios de diseño orientado a objetos que ya hemos estudiado, en los que se basan muchos de los patrones presentados en el catálogo. Recordemos algunos de los más importantes.

Herencia de clases frente a herencia de interfaces

¡Hay que distinguir entre la clase de un objeto y su tipo!

- **La clase define la implementación del objeto**
- **El tipo de un objeto es su interfaz**
  - El conjunto de peticiones a las que puede responder
- **Un objeto puede tener muchos tipos, y objetos de distintas clases pueden tener el mismo tipo**

Herencia de clases frente a herencia de interfaces

- **En tanto una clase define las operaciones que puede realizar un objeto, también define su tipo**
  - Al decir que un objeto es una instancia de una clase, estamos diciendo que se adhiere a la interfaz definida por la clase
- **Lenguajes como C++ usan clases para especificar tanto el tipo de un objeto como su implementación**
  - Java incorpora el concepto de interfaz en el propio lenguaje
- **En Smalltalk no se declaran los tipos de las variables, por lo que cuando se envía una petición a un objeto se comprueba en tiempo de ejecución si éste implementa el mensaje (y no si es de una determinada clase)**

Herencia de clases frente a herencia de interfaces

- **La herencia de clases es un mecanismo para reutilizar código**
- **La herencia de interfaces (*subtipificado*) describe cuándo se puede utilizar un objeto en lugar de otro**
- **La mayoría de lenguajes no hacen la distinción explícita**
  - Java sí lo hace (`interface`)

Programar para una interfaz, no para una implementación

Programar para una interfaz, no para una implementación

- **Ventaja de manipular los objetos sólo en términos de su interfaz:**
  - A los clientes les trae sin cuidado el tipo concreto de objeto, con tal de que cumpla una determinada interfaz
  - Sólo conocen su interfaz, no las clases que implementan esa interfaz
  - Esto reduce de tal forma las dependencias de implementación entre subsistemas que da lugar a este principio de la orientación a objetos

Favorecer la composición de objetos sobre la herencia de clases

Herencia frente a composición

- **La herencia de clases permite definir la implementación de una clase en términos de la de otra**
  - Reutilización de caja blanca
    - Las interioridades de la clase padre son visibles para la subclase
- **La composición de objetos es una alternativa a la herencia**
  - La nueva funcionalidad se obtiene ensamblando o componiendo objetos
  - Reutilización de caja negra

Herencia frente a composición

- **La herencia se define estáticamente, en tiempo de compilación**
- **Es fácil de utilizar**
  - La incorpora directamente el lenguaje de programación
  - Normalmente codificamos menos
- **La subclase es muy dependiente de la implementación de la clase padre**
  - Hace muy difícil la reutilización en otros dominios

Herencia frente a composición

- **La composición de objetos se define dinámicamente en tiempo de ejecución a través de objetos que guardan referencias a otros objetos**
- **A estos objetos sólo se accede a través de sus interfaces**
  - No se rompe el encapsulamiento, como con la herencia
  - No hay dependencias de implementación
- **Se puede cambiar cualquier objeto por otro de su mismo tipo en tiempo de ejecución**

Herencia frente a composición

- **La composición suele tener otro efecto sobre el diseño del sistema:**
  - Ayuda a que cada clase se centre en una única tarea
  - Las clases y jerarquías de clases permanecerán relativamente pequeñas, es más improbable que se conviertan en monstruos inmanejables

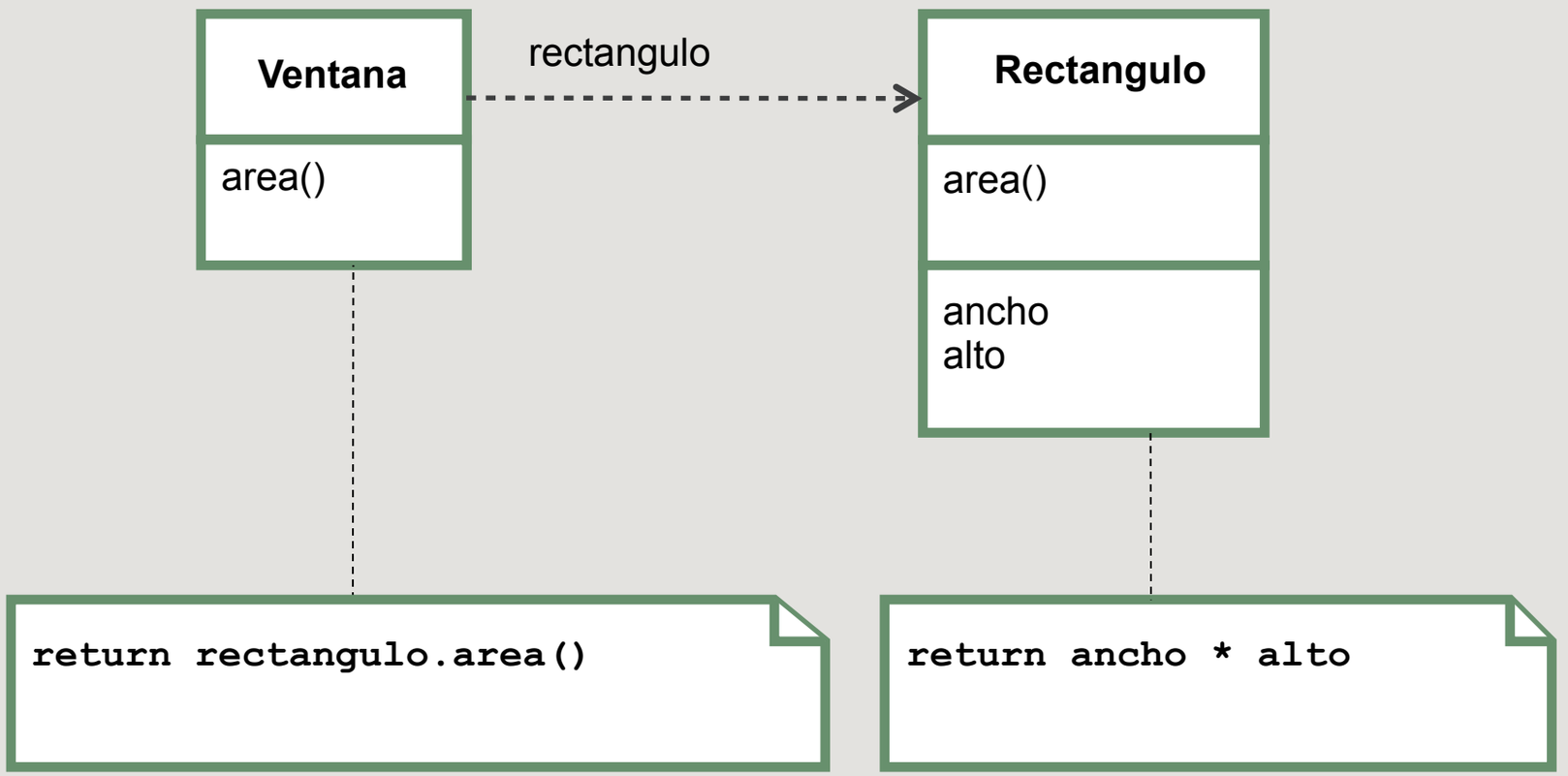
Herencia frente a composición

- **En resumen:**
  - ¡Ojo con la herencia!
  - Hay que ser extremadamente cautos al utilizarla
    - Sólo cuando sea obvio su necesidad
  - Lo importante de la OO no es la herencia, sino el **polimorfismo**
  - Esto nos lleva a otro principio del DOO:

> *Favorecer la composición de objetos sobre la herencia de clases.*

Delegación

- **Mediante la delegación son dos los objetos encargados de responder a una petición:**
  - Un objeto recibe la petición y la reenvía a un objeto delegado



Delegación

- **La principal ventaja es que permite variar muy fácilmente el comportamiento en tiempo de ejecución**
  - Por ejemplo, hacer que las ventanas pasen a ser circulares
- **Inconvenientes:**
  - El software dinámico y muy parametrizado suele ser más difícil de entender
  - Rendimiento

Delegación

- **En resumen:**
  - Utilizarla cuando simplifica más que complica
  - Funciona mejor cuando se usa forma estilizada
    - Esto es, en los patrones estándar
- **Muchos patrones se basan en la delegación**
  - *State*, *Strategy*, *Visitor*...
- **Es un caso particular de composición de objetos**
  - Y una prueba de cómo es posible siempre reemplazar la herencia por la composición de objetos como mecanismo de reutilización

Estructuras en tiempo de compilación y en tiempo de ejecución

Estructuras en tiempo de compilación y en tiempo de ejecución

- **La estructura en tiempo de ejecución de un programa a menudo tiene poco que ver con la estructura de su código**
  - Ésta es estática: clases con una serie de relaciones entre ellas
  - La estructura en tiempo de ejecución consiste en una red de objetos que cooperan y se comunican entre sí

- **Ejemplo: diferencia entre agregación y composición**

  Es fácil confundir ambos tipos de relaciones porque normalmente se implementan igual (en Java, por ejemplo, todo son referencias a objetos). Se distinguen más por su intención que por mecanismos explícitos del lenguaje La estructura en tiempo de ejecución normalmente la impone el diseñador más que el propio lenguaje.

Estructuras en tiempo de compilación y en tiempo de ejecución

- Muchos patrones de diseño (en especial los de objetos) representan explícitamente la distinción entre la estructura dinámica y estática

Composite, Decorator, Observer y Chain of Responsibility son ejemplos de patrones que involucran estructuras en tiempo de ejecución que a menudo no se pueden apreciar en sus diagramas de clases y que no están claras en el código si no se comprende el patrón.

Introducción

Problemas repetidos

- Los ingenieros o arquitectos software se enfrentan cada día a multitud de problemas de distinto calibre
- La efectividad de un ingeniero se mide por su rapidez y acierto en la diagnosis, identificación y resolución de tales problemas
- El mejor ingeniero es el que más reutiliza la misma solución —matizada— para resolver problemas similares

Reinventar la rueda

- **La orientación a objetos propugna no reinventar la rueda en la pura codificación respecto de la resolución de problemas**
  - ¿Por qué, entonces, reinventarla para el ataque genérico a problemas comunes de análisis, diseño e implementación?
- **Debe existir alguna forma de comunicar al resto de los arquitectos software los resultados encontrados tras mucho esfuerzo por algunos de ellos**
- **Se necesita, en definitiva, algún esquema de documentación que permita tal comunicación**

Adaptación al cambio

- **Si diseñar software orientado a objetos es difícil, diseñar software orientado a objetos *reutilizable* lo es aún más**
- **El diseño debe ser específico del problema a resolver, pero también lo suficientemente general**
  - Como para adaptarse a futuros cambios en los requisitos
  - Evitando o, al menos, minimizando el rediseño

Diseños flexibles

- **En definitiva, perseguimos un diseño flexible**
  - Es casi imposible dar con él la primera vez
- **¿Cómo se arreglan los buenos diseñadores orientados a objetos?**
  - Suelen ser diseñadores experimentados
    - Lleva bastante tiempo aprender en qué consiste el buen diseño orientado a objetos
    - Los diseñadores novatos tienden a caer en las viejas técnicas no orientadas a objetos

¿Qué diferencia a los diseñadores experimentados de los novatos?

- **Que tienden a reutilizar soluciones que se han probado útiles en el pasado**
  - Y que es lo que les hace ser expertos
- **Así, se encuentran los mismos patrones recurrentes de clases una y otra vez en muchos sistemas orientados a objetos**
  - Que los hacen más flexibles, elegantes y, en definitiva, reutilizables
- **Un arquitecto software familiarizado con ellos es capaz de aplicarlos inmediatamente sin tener que redescubrirlos cada vez**

Ejemplos de problemas típicos

- ¿Cómo puedo representar estados mediante objetos?
- ¿Cómo garantizar que una clase tenga una única instancia?
- ¿Cómo añadir responsabilidades dinámicamente (sin emplear la herencia) a un objeto?

Ejemplos de problemas típicos

- **Hay diseños OO que resuelven esos y otros muchos problemas similares**
  - Que son genéricos, no específicos de una aplicación concreta
- **¿No hay forma de documentar ese conocimiento de alguna forma?**
  - Para eso surgen los patrones de diseño

¿Qué es un patrón de diseño?

Cada patrón describe un problema que se repite una y otra vez en nuestro entorno, y describe el núcleo de la solución a dicho problema, de modo que pueda usarse esta solución millones de veces, aunque siempre de forma ligeramente distinta.

— Christopher Alexander (1977)

Christopher Alexander (1977). *A Pattern Language*

El Libro AIS

- **El arquitecto Cristopher Alexander, en su libro *A Pattern Language. Towns, Buildings, Construction* (Oxford University Press, 1977) presenta 253 patrones referidos al modo de diseñar edificios y ciudades de forma que tenga en cuenta las necesidades de sus habitantes**
  - Se le conoce como el libro AIS (las iniciales de los tres primeros autores)
- **Los patrones de diseño software se consideran deudores de los trabajos de Alexander**

La calidad sin nombre

- ¿Existe en verdad una parte común en los buenos diseños, a veces tan dispares entre sí?
- Christopher Alexander así lo afirma, y da a esta parte la elusiva calificación de «la calidad que no se puede nombrar»
- Alexander sostiene que existe un «algo innombrable» que no puede ser modelado únicamente por medio de un conjunto arbitrario de requisitos
- Los sistemas poseerían, así, una esencia cualitativa que les otorgaría verdadera identidad y equilibraría sus fuerzas internas

Un ejemplo rural

www.flickr.com/photos/acebal/3309710619

Si nos fijamos en las construcciones de una determinada zona rural observaremos que todas ellas poseen apariencias parejas (por ejemplo, tejados de pizarra con gran pendiente, una determinada orientación, etcétera), pese a que los requisitos personales por fuerza han debido ser distintos. De alguna manera la esencia del diseño se ha copiado de una construcción a otra, y a esta esencia se pliegan de forma natural los diversos requisitos. Diríase aquí que existe un «patrón» que soluciona de forma simple y efectiva los problemas de construcción en tal zona.

Elementos de un patrón

- **Alexander, en su otro libro *The Timeless Way of Building* (1979) define el término patrón como sigue:**

> *Cada patrón es una regla de tres partes, que expresa una relación entre un contexto determinado, un problema y una solución.*

El contexto

- Describe las situaciones en las que se da el problema

El problema

- **Una descripción general que represente bien la esencia de éste**
  - ¿Cuál es la cuestión concreta de diseño que debemos resolver?
    - Por ejemplo, el patrón Modelo-Vista-Controlador permite que las interfaces de usuario varíen independientemente de los datos que representan
- **Ese enunciado general se completa con un conjunto de fuerzas (*force*) implicadas:**
  - Requisitos que debe cumplir la solución
  - Restricciones a considerar
  - Propiedades deseables

La solución

- **Cómo resolver el problema**
  - Es decir, cómo equilibrar ese conjunto de fuerzas
- **Dicha solución incluye dos aspectos:**
  - La estructura estática del patrón
    - Las clases participantes y sus relaciones
  - Su comportamiento, los aspectos dinámicos
    - ¿Cómo colaboran los participantes? ¿Cómo se reparten el trabajo? ¿Cómo se comunican entre ellos?

Los patrones son un esquema

- **Un patrón software proporciona un esquema general**
- **No es un artefacto implementado, un prototipo para ser usado tal cual**
- **Es un bloque de construcción mental para ser implementado en un sinfín de aplicaciones y contextos diferentes**
  - No habrá dos implementaciones idénticas
  - Después de aplicar el patrón, se identificará su estructura y los papeles (roles)[†] desempeñados por las clases participantes, pero éstas estarán ajustadas a las necesidades de cada problema concreto

---

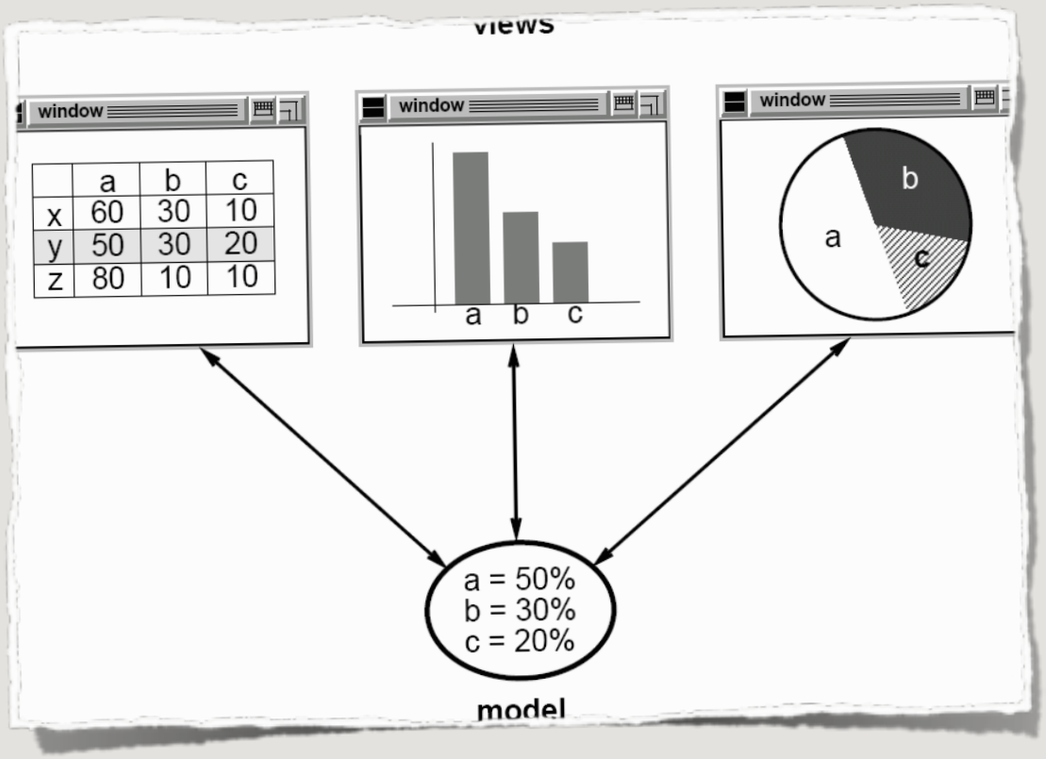
† Aunque «rol» con ese mismo significado que en inglés se introdujo en el español ya a finales del s. XIX, no deja de ser eso, un anglicismo innecesario, frente a los equivalentes en español «papel» o «función». ¡Ah!, y, desde luego, lo que no se puede decir jamás es «jugar un papel» (http://buscon.rae.es/dpdi/SrvltConsulta?lema=rol)

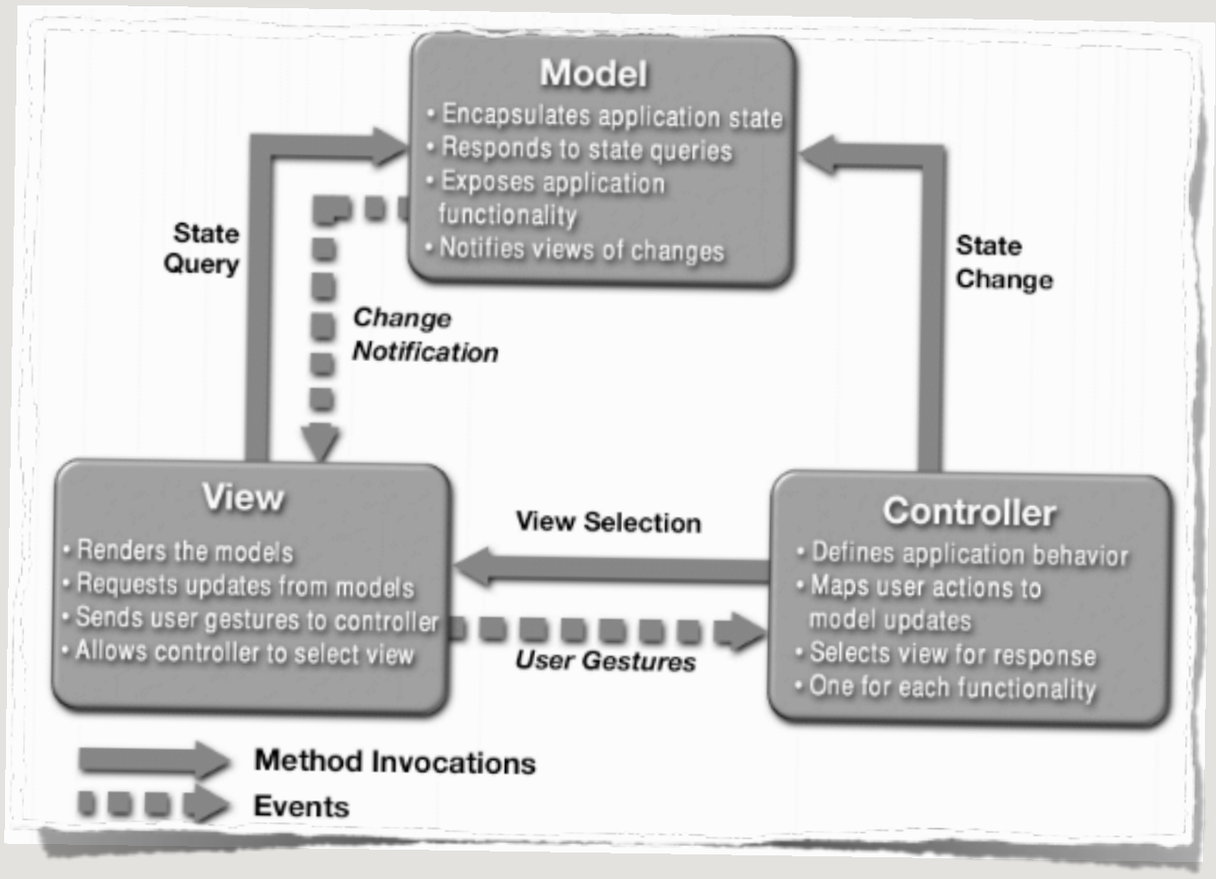
Categorías de patrones

- **Podemos hacer una primera clasificación de los patrones software atendiendo al nivel de abstracción de éstos:**
  - Arquitectónicos
  - De diseño
  - Centrados en el código
    - En un lenguaje de programación concreto (modismos, patrones de implementación, «idioms»)

Patrones arquitectónicos

- **Situados a un nivel de abstracción más alto, describen la arquitectura, la estructura de un sistema en torno a subsistemas y las relaciones entre ellos**
  - Ejemplo: Modelo/Vista/Controlador (MVC)

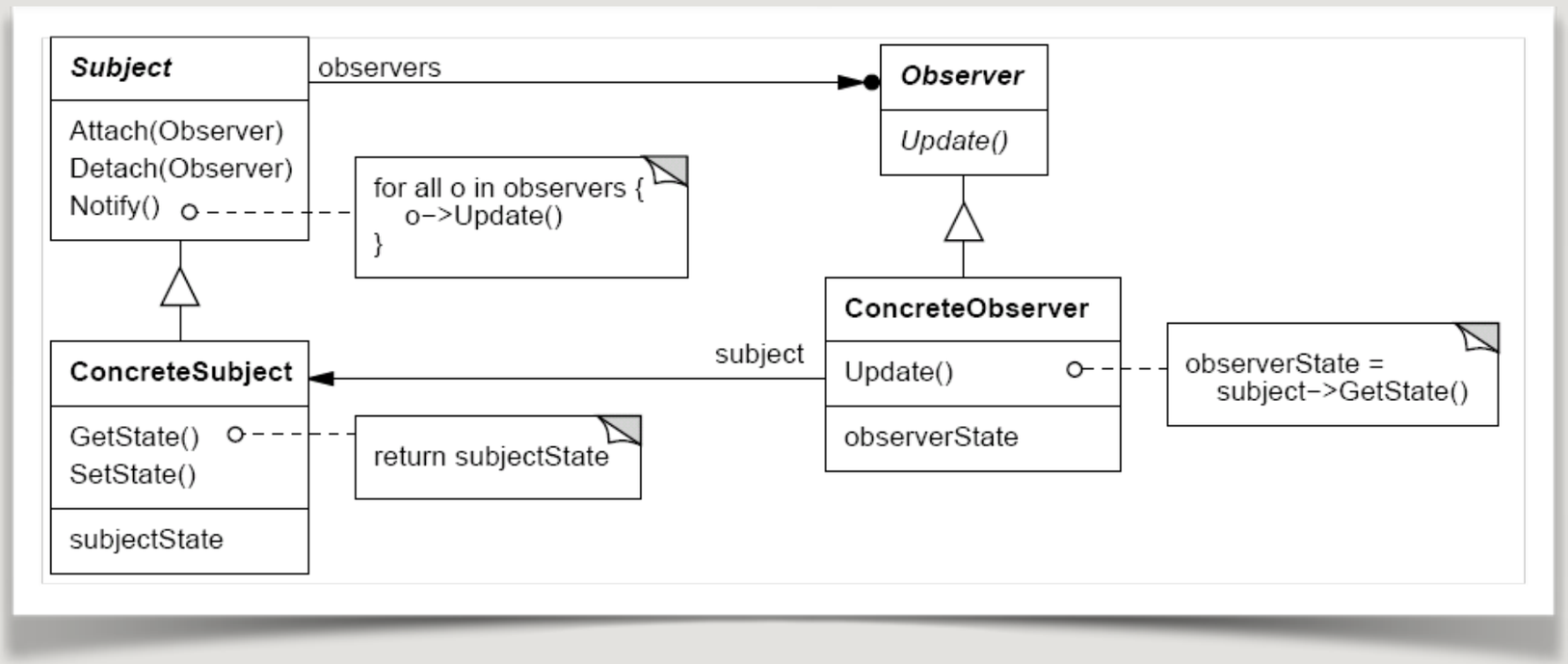




Patrones de diseño

- **Serán el grueso de la asignatura**
- **Se sitúan a un nivel de abstracción medio, de diseño**
  - Una escala inferior a los arquitectónicos pero independientes del lenguaje de programación

- **Ejemplo: Observer**



Modismos (*idioms*)

- **Son patrones de muy bajo nivel (codificación)**
- **La mayoría son específicos de un lenguaje de programación determinado**
- **Contribuyen a dar un estilo uniforme, consistente, a los programas escritos en ese lenguaje**
  - Guías de Estilo
  - Importantísimos
  - Al final, su conocimiento es lo que distingue a los buenos programadores en un determinado lenguaje de aquéllos que simplemente conocen su sintaxis

Smalltalk Best Practice Patterns (Kent Beck, 1997)
Effective C++ (Scott Meyers, 1997)
Java Style. Patterns for Implementation (Jeff Langr, 2000)
Effective Java (Joshua Bloch, 2008)
etc.

Modismos

- **Ejemplo: recorrer una colección, en Java**

```
Iterator iterator = list.iterator();
while (iterator.hasNext())
{
    Element object = (Element) iterator.next();
    // hacer lo que sea con el objeto
}
```

- **Realmente es un enfoque bastante procedimental, más que OO, pero es perfectamente válido en Java desde el momento en cualquier programador en ese lenguaje reconoce estas líneas de código nada más verlas, sabe qué es lo que hacen**

Revelan su intención

Modismos

- **Algunos son mucho más simples aún**
  - Ejemplo: Indented Control Flow
    - Cómo sangrar las líneas en Smalltalk (Beck, 1997)

En el caso de mensajes sin argumentos o con un solo argumento, poner éstos en la misma línea que el receptor:

```
a < b
  ifTrue:[...]
  ifFalse:[...]
```

Para aquéllos con dos o más argumentos, poner cada uno en su propia línea, sangradas con un tabulador:

```
foo isNil
  2 + 3
  a < b ifTrue:[...]
```

Propiedades de los patrones

Algunas de las propiedades más características de los patrones (no pretende ser un listado exhaustivo, sino ofrecer una visión general de éstos)

Propiedades (I)

Un patrón responde a un problema de diseño que se repite frecuentemente en determinadas situaciones, y presenta una solución a dicho problema.

Propiedades (II)

Los patrones sirven para documentar la experiencia previa, los diseños que ya se probaron útiles.

- **¡No se inventan artificialmente!**
- **Al contrario, destilan el conocimiento de diseño adquirido por los profesionales experimentados y proporcionan un medio para representar dicho conocimiento y facilitar su reutilización**
- **Deben haber sido utilizados en ámbitos muy distintos de aplicación**

Propiedades (III)

Identifican y especifican abstracciones que están por encima de clases y objetos o componentes individuales.

- **Típicamente, un patrón describe varios componentes, clases u objetos, así como sus responsabilidades y relaciones, y el modo en que cooperan**

Propiedades (IV)

Proporcionan un vocabulario de diseño común.

- **Tal vez sea su principal aportación**
- **Los nombres de los patrones pasan a formar parte de nuestro vocabulario de diseño, facilitando la comunicación**

—«Aquí deberías usar un Strategy»
—«¿Qué arquitectura utilizaremos? ¿MVC?»

Propiedades (V)

Permiten documentar nuestros diseños.

- **Ayudan a describir la visión que teníamos en mente cuando diseñamos el software, disminuyendo así el riesgo de que sea malinterpretado y se viole esa visión**
  - Al implementar el sistema por parte de los programadores, modificarlo en un futuro, etcétera

Propiedades (VI)

Ayudan a construir arquitecturas complejas y heterogéneas.

Catálogos de patrones

Si aceptamos que los patrones pueden resultar útiles en el desarrollo de software, el siguiente paso es reunirlos en catálogos de forma que resulten accesibles mediante distintos criterios, pues lo que necesitamos no es tan sólo la completa descripción de cada uno de los patrones sino, esencialmente, la correspondencia entre un problema real y un patrón (o conjunto de patrones) determinado.

La curva de aprendizaje

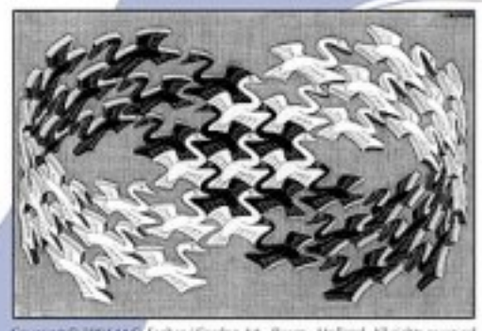
- Lo que se pretende con un catálogo de patrones no es favorecer al diseñador experto (que quizás no necesite en absoluto de los patrones), sino más bien ayudar al diseñador novel a adquirir con cierta rapidez las habilidades de aquél, como también comunicar al posible cliente, si es el caso, las decisiones de diseño de forma clara
- Un catálogo de patrones es un medio para comunicar la experiencia de forma efectiva, reduciendo lo que se conoce como «curva de aprendizaje» del diseño

El GoF

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Sobre el GoF

- **De Gang of Four o Banda de los cuatro, en alusión a sus autores**
  - Y tomado, a su vez, de la revolución cultural china
- **Libro clásico por excelencia de patrones de diseño**
- **Puede ser un poco duro, al principio, pero es un libro que cualquier programador, diseñador, arquitecto... debería leer**
  - ¡Y que debería existir en cualquier empresa!
- **Catálogo de 23 patrones divididos en tres áreas: de creación, estructurales y de comportamiento**

Elementos de un patrón de diseño

- **Hay cuatro fundamentales:**
  - El **nombre** del patrón
    - Incrementa nuestro vocabulario de diseño, y facilita pensar en esos términos al hablar con otros miembros del equipo
  - El **problema**
    - Cuándo tiene sentido aplicar el patrón
  - La **solución**
    - La estructura de clases, relaciones, responsabilidades y colaboraciones que resuelven el problema general
  - Las **consecuencias**
    - Impacto de aplicar el patrón en un sistema (ventajas e inconvenientes que permiten evaluarlo)

Punto de vista de cada uno

- **Qué es y qué no es un patrón, en última instancia depende del punto de vista de cada uno**
  - Lo que para una persona para otra puede ser un bloque de construcción primitivo
  - Los patrones de diseño que veremos son «descripciones de objetos y clases que se comunican entre sí y que hay que adaptar para resolver un problema general de diseño en un contexto particular»

Patrones de diseño en el GoF

- Un patrón de diseño da nombre a una abstracción de diseño, identifica las clases y objetos participantes, sus papeles (*roles*) y colaboraciones, y la distribución de responsablidades
- Cada patrón se centra en un problema de diseño orientado a objetos concreto, y describe cuándo es aplicable, así como las consecuencias y ventajas e inconvenientes de su uso
- Por último, también se proporciona código de ejemplo (en C++ y Smalltalk) para ilustrar una implementación concreta

Sobre el lenguaje de programación

- **La elección del lenguaje de programación es importante**
  - Las propias características del lenguaje influye sobre nuestro punto de vista
  - Por ejemplo, en el GoF no hay patrones como «Herencia», «Encapsulamiento» o «Polimorfismo»
  - De igual modo, algunos de los patrones del GoF no harían falta en lenguajes más dinámicos, como CLOS, Dylan o Self

Descripción de un patrón de diseño

- Las notaciones gráficas no son suficientes
- Los patrones de diseño proporcionan un esquema documental consistente que se repite en todos ellos
- Cada patrón se divide en las siguientes secciones

Secciones de un patrón de diseño

- **Nombre del patrón (y clasificación)**
  - La importancia del nombre es fundamental
  - La clasificación se refiere a su propósito (si son de creación, de estructura o de comportamiento) y al ámbito del patrón (de clases o de objetos)
- **Propósito**
  - ¿Qué hace este patrón de diseño? ¿Para qué sirve?
- **También conocido como**
  - Otros nombres por los que se conozca al patrón
- **Motivación**
  - Un escenario que ilustra un problema de diseño concreto y cómo el patrón puede ayudar a resolverlo

Secciones de un patrón de diseño

- **Aplicabilidad**
  - ¿En qué situaciones tiene sentido aplicar el patrón? Ejemplos de malos diseños que el patrón puede mejorar y cómo podemos reconocer estas situaciones
- **Estructura**
  - Diagrama de clases del patrón (en algunos casos, también un diagrama de secuencia para ilustrar las colaboraciones entre los objetos del patrón)
- **Participantes**
  - Las clases y objetos participantes junto con sus responsabilidades
- **Colaboraciones**
  - Cómo colaboran los participantes entre sí para llevar a cabo dichas responsabilidades

Secciones de un patrón de diseño

- **Consecuencias**
  - ¿Cómo cumple el patrón sus objetivos? ¿Cuáles son las ventajas e inconvenientes de sus uso? Al aplicarlo, ¿cuál es el aspecto del sistema que permite variar independientemente?
- **Implementación**
  - Detalles de bajo nivel a tener en cuenta a la hora de implementar el patrón (trucos, técnicas, cuestiones específicas de tal o cual lenguaje...)
- **Código de ejemplo**
- **Patrones relacionados**
  - Patrones relacionados con éste, en qué se diferencian, junto con qué otros patrones suele aparecer...

El catálogo

- Un total de 23 patrones de diseño

Organización del catálogo

- **Según su ámbito**
  - De clases
    - Patrones que tratan sobre todo con relaciones entre clases y sus subclases a través de la herencia
    - O sea, estáticas, fijadas en tiempo de compilación
  - De objetos
    - Trabajan con relaciones entre objetos, que pueden cambiarse en tiempo de ejecución (dinámicas)
    - Es aquí donde caen la mayoría de los patrones

Organización del catálogo

- **Según su propósito:**
  - De creación
    - Permiten postergar la creación de un objeto a las subclases o a otros objetos (dependiendo del ámbito del patrón)
  - Estructurales
    - Usan la herencia (los de clase) para componer clases o describen modos para ensamblar objetos (los de objetos)
  - De comportamiento
    - Los de clase usan la herencia para describir algoritmos y flujos de control, en tanto que los de objetos describen cómo cooperan un grupo de objetos entre sí para llevar a cabo una determinada tarea

Organización del catálogo

		Propósito		
		Creación	**Estructural**	**Comportamiento**
Ámbito	**Clases**	Factory Method	Adapter	Interpreter Template Method
	Objetos	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Cómo seleccionar un patrón de diseño

- **Tener en cuenta lo que acabamos de ver acerca de cómo los patrones ayudan a resolver problemas de diseño**
- **Examinar la sección *Intent* (*Propósito*)**
- **Estudiar cómo se interrelacionan los patrones**
  - **Estudiar los patrones de propósito similar**
    - Secciones introductorias de los tres tipos de patrones de diseño (creación, estructurales y de comportamiento)
  - **Tener claras las causas de rediseño vistas**
  - **Considerar qué es lo más probable que cambie en nuestro diseño**
    - Y encapsular el concepto que varía

Cómo usar un patrón de diseño

- **Leer el patrón entero la primera vez**
- **Estudiar en detalle las secciones de *Estructura*, *Participantes* y *Colaboraciones***
- **Echar un vistazo al *Código de ejemplo***
- **Elegir los nombres correctos para los participantes en nuestro contexto**
  - Adaptando los del patrón, que son demasiado abstractos para ser usados directamente
    - Ejemplo: `SimpleLayoutStrategy`, `TeXLayoutStrategy`

Cómo usar un patrón de diseño

- **Definir las clases**
  - Interfaces, atributos y relaciones
- **Definir nombres específicos de la aplicación para las operaciones del patrón**
  - Por ejemplo, utilizar el prefijo `Create...` para denotar un método de fabricación (*Factory Method*)
- **Implementar las operaciones que lleven a cabo las responsabilidades del patrón**

Diseñar para el cambio

(Causas de rediseño)

Diseñar para el cambio

- La clave de la reutilización está en anticiparse a los cambios en los requisitos y diseñar los sistemas de forma que puedan evolucionar de manera acorde
- Cada patrón de diseño permite que un sistema pueda variar de una determinada forma, haciendo al sistema más robusto frente a ese cambio concreto
- A continuación veremos algunas causas frecuentes de rediseño junto con los patrones que las evitan

Herencia de clases frente a herencia de interfaces

¡Hay que distinguir entre la clase de un objeto y su tipo!

- **La clase define la implementación del objeto**
- **El tipo de un objeto es su interfaz**
  - El conjunto de peticiones a las que puede responder
- **Un objeto puede tener muchos tipos, y objetos de distintas clases pueden tener el mismo tipo**

Causas de rediseño

Crear un objeto especificando explícitamente su clase

- **Especificar el nombre de la clase en el código al crear el objeto nos liga a una implementación particular en vez de a una interfaz**
- **Es mejor crear los objetos indirectamente**
- **Patrones de diseño:**
  - *Abstract Factory*, *Factory Method*, *Prototype*

Causas de rediseño

Depender de operaciones concretas

- **Evitando llamar directamente a un método determinado en el código se facilita cambiar la forma en que se responde a una petición**
- **Patrones de diseño:**
  - *Chain of Responsibility, Command*

Causas de rediseño

Dependencia de plataformas hardware o software

- **Las interfaces de los sistemas operativos y las API (*Application Programming Interface*) de muchas aplicaciones son dependientes de la plataforma**
- **El software dependiente de plataforma es difícil de llevar a otras**
- **E incluso de adaptarse a actualizaciones de su plataforma nativa**
- **Hay que limitar tales dependencias**
- **Patrones de diseño:**
  - *Abstract Factory*, *Bridge*

Causas de rediseño

Depender de implementaciones o representaciones de objetos

- **Los clientes que conocen cómo se representa internamente, cómo se almacena, dónde se localiza o cómo se implementa tendrán que cambiar cuando cambie aquél**
- **Es preciso ocultar esta información a los clientes para prevenir los cambios en cascada**
- **Patrones de diseño:**
  - Abstract Factory, Bridge, Memento, Proxy

Causas de rediseño

Dependencias de algoritmos

- **Los algoritmos que es probable que cambien a lo largo del tiempo (para optimizarlos o porque se sustituyan por otro distinto) deben estar aislados**
- **Patrones de diseño:**
  - Builder, Iterator, Strategy, Template Method, Visitor

Causas de rediseño

Fuerte acoplamiento

- **Es difícil reutilizar clases que están fuertemente acopladas a otras**
- **Esto conduce a sistemas monolíticos donde no se puede cambiar o eliminar una clase sin que afecte a muchas de las demás**
- **Patrones de diseño:**
  - Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Causas de rediseño

Extender funcionalidad mediante la herencia

- **Definir una subclase requiere un profundo conocimiento de la clase padre**
  - Redefinir una operación puede requerir redefinir otra, o que haya que llamar primero a una operación heredada
  - Puede dar lugar a una explosión de clases incluso para una extensión muy simple
  - La composición de objetos en general y la delegación en particular suelen ser mejores alternativas a la hora de combinar comportamiento
    - Aunque también suelen ser más difíciles de entender los diseños
- **Patrones de diseño:**
  - Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

Causas de rediseño

No se puede modificar las clases

- **A veces hay que modificar una clase de la que no disponemos del código fuente (por ejemplo, con una biblioteca de clases comercial) o que requeriría cambiar montones de subclases existentes**
- **Hay patrones que nos permiten modificar clases bajo tales circunstancias**
- **Patrones de diseño:**
  - Adapter, Decorator, Visitor

Un comentario final

(Acerca de lo de diseñar para el cambio...)

- **Tampoco debemos caer en el error del sobrediseño**
  - Como preconizan los métodos ágiles como Extreme Programming (XP)

Tipos de software

Tipos de software

- **Cómo de importante sea la flexibilidad también depende del tipo de software que estemos desarrollando**
  - Aplicaciones
  - Bibliotecas de clases
  - Frameworks

Aplicaciones

- Las prioridades son la reutilización interna, la mantenibilidad y la extensibilidad

Bibliotecas de clases

- **Son bibliotecas de clases predefinidas**
  - Diseñadas para algún tipo de funcionalidad de propósito general
  - Por ejemplo, bibliotecas gráficas para el desarrollo de aplicaciones de ventana, la API de Java, etcétera
- **Se basan en la reutilización de código (no nos imponen ningún diseño concreto)**
- **Es más importante aún evitar toda posible dependencia: no sabemos qué aplicaciones las van a utilizar**

Frameworks

- **Conjunto de clases cooperantes que constituyen un diseño reutilizable para un determinado tipo de aplicaciones**
  - Ejemplo: HotDraw para editores gráficos, JUnit para pruebas, Struts para aplicaciones web...
- **El framework debe ser adaptado a cada aplicación concreta**
  - Creando subclases específicas de la aplicación a partir de las clases abstractas proporcionadas por aquél

Frameworks

- **El framework dictamina cómo será la arquitectura de nuestra aplicación**
  - Estructura general, partición en clases y objetos, cómo colaboran entre sí, el hilo de control...
  - Contiene las decisiones de diseño que son comunes a ese dominio de aplicación
    - Enfatizan, así, la reutilización del diseño más que la de código

Frameworks

- **Cuando reutilizamos una biblioteca de clases escribimos el código principal de la aplicación y llamamos al código que queremos reutilizar**
- **En el caso de los frameworks, escribimos el código específico de la aplicación que será llamado por el framework**
  - Habrá que ajustarse a una serie de convenios impuestos por el framework, pero se minimizan las decisiones de diseño a tomar

Frameworks

- **Ni que decir tiene que es el tipo de software más difícil de diseñar bien para que sea reutilizable**
- **Es donde más sentido tienen los patrones de diseño**
  - Además, si la documentación también se expresa en términos de patrones, es más fácil de asimilar
    - (Suelen tener una curva de aprendizaje que hay que superar antes de empezar a usarlo)

Diferencias entre los frameworks y los patrones de diseño

- **Los patrones de diseño son más abstractos que los frameworks**
  - Los patrones de diseño no se codifican más que en los ejemplos de uso
  - El framework se escribe en un lenguaje de programación y se puede ejecutar directamente
- **Los patrones de diseño son elementos arquitectónicos más pequeños**
- **Los patrones de diseño son generales; los frameworks, especializados**