Tema 6

Patrones de diseño   
de software

Ingeniería del Software Avanzada

Índice

[Esquema 3](#_Toc532468171)

[Ideas clave 4](#_Toc532468172)

[6.1. Introducción y objetivos 4](#_Toc532468173)

[6.2. Los patrones de diseño 4](#_Toc532468174)

[6.3. Patrones de Gamma 6](#_Toc532468175)

[6.4. Patrones de diseño de interacción 22](#_Toc532468176)

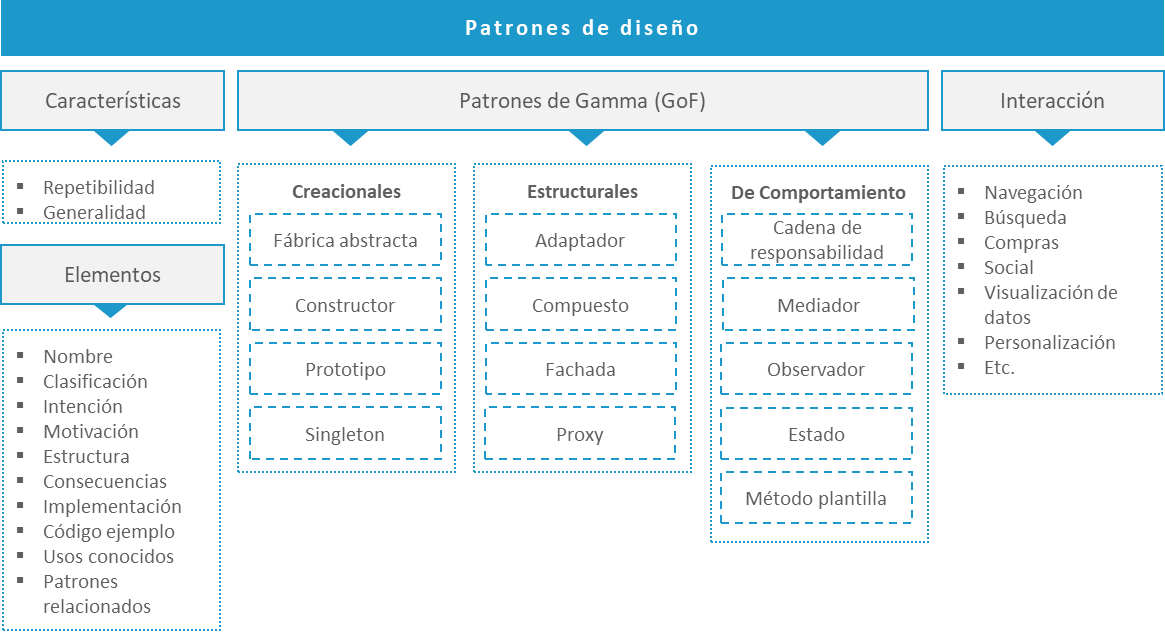
[6.5. Referencias bibliográficas 26](#_Toc532468177)

[A fondo 27](#_Toc532468178)

[Actividades 29](#_Toc532468179)

[Test 30](#_Toc532468180)

Esquema



Ideas clave

6.1. Introducción y objetivos

En este tema vamos a estudiar un conjunto de patrones específicos que permiten resolver problemas habituales en el diseño de aplicaciones.

Los patrones de diseño son herramientas que permiten aplicar la reutilización de conocimiento en nuestros proyectos.

La experiencia acumulada a lo largo de décadas de proyectos de desarrollo software ha ido mostrando **ciertas arquitecturas de componentes a nivel de clase que permiten resolver problemas comunes**, relacionados con la creación de objetos, su organización dentro del sistema, o la manera en que deben implementarse sus comportamientos.

Mediante el estudio de este tema alcanzaremos los siguientes objetivos:

* **Comprender la importancia de los patrones de diseño** en el desarrollo de aplicaciones y sus características fundamentales.
* **Conocer los elementos** que suelen acompañar a la descripción de un patrón de diseño.
* **Saber utilizar los principales patrones de diseño** para la creación, estructuración y comunicación entre objetos.
* **Comprender la importancia** de los patrones de diseño de interacción en el desarrollo de aplicaciones interactivas.

6.2. Los patrones de diseño

En el tema anterior nos centramos en el estudio de los patrones de arquitectura, que permiten modelar al más alto nivel los componentes principales de un sistema, así como las relaciones que se establecen entre ellos. Aquí, nos centraremos en soluciones más específicas a problemas de desarrollo habituales, mucho más cercanas al dominio de implementación del sistema.

Para poder considerar una **solución como un patrón de diseño** deben darse al menos **dos características** (Polo Usaola, 2012):

* **Debe ser repetible**, habiendo mostrado su efectividad en más de una ocasión. Por tanto, la solución no solo debe resultar válida para una tecnología o problema de un dominio de aplicación concreto.
* **Su descripción debe ser los suficientemente genérica** como para que pueda ser aplicado en contextos tecnológicos diversos.

En algunos casos, los **patrones de diseño** ofrecen soluciones que en apariencia complican excesivamente el diseño del sistema —por ejemplo, introduciendo elementos adicionales que *a priori* no parecen indispensables—. Por este motivo, el abuso de su empleo (o su utilización incorrecta) puede hacer que el sistema se vuelva más difícil de comprender y mantener. Sin embargo, en términos generales, los patrones de diseño **persiguen aumentar la cohesión y reducir el acoplamiento de los sistemas, favoreciendo prácticas deseables como la reutilización de software.**

Tal es la importancia que cobran los patrones de diseño, como técnica de reutilización de software y resolución rápida de problemas habituales, que este tipo de conocimiento es recopilado y publicado en diferentes manuales y catálogos.

Estos catálogos utilizan una estructura estandarizada y generalmente definen, entre otros, **los siguientes elementos para cada uno de los patrones**:

1. **Nombre del patrón**, generalmente en inglés.
2. **Clasificación del patrón** (de creación, de estructura o de comportamiento).
3. Intención o **descripción del problema** que intenta resolver.
4. **Motivación** o explicación de escenarios prácticos que justifican la posible aplicación del patrón.
5. **Estructura**, empleando diagramas de clases que representan los objetos que intervienen en el patrón y sus relaciones.
6. **Consecuencias positivas y negativas** de su empleo.
7. **Implementación**, ofreciendo indicaciones para ello.
8. **Código de ejemplo**, a veces en diferentes lenguajes como demostración.
9. **Usos conocidos**, dando ejemplos de sistemas reales que lo aplican.
10. **Patrones relacionados**, indicando los nombres de otros patrones con los que presenta similitudes, o con los que es posible combinarlo.

El más conocido y antiguo de estos catálogos es el de Gamma, Helm, Johson y Vlissides (*Patrones de Gamma*), que publicaron una primera versión de su libro, *Design Patterns,* en 1994, y desde entonces han aparecido múltiples ediciones (Gamma, Helm, Johnson y Vlissides, 2003). El trabajo de estos cuatro autores ha sido tan influyente que generalmente son conocidos como «La banda de los cuatro» —*Gang of Four* (GoF)—.

Además, periódicamente, se celebran conferencias en todo el mundo sobre esta temática. Las más importantes son las *Pattern Languages of Programs* (PLoP), un grupo de conferencias anuales que se celebran en todo el mundo. La principal de ellas se celebra anualmente en Estados Unidos, pero también existen otras ediciones, ya sea centradas en áreas geográficas concretas (AsianPLoP, EuroPLoP o KoalaPLoP) o en temáticas específicas (ScrumPLoP o ChilliPLoP).

6.3. Patrones de Gamma

En su libro, Gamma et al. (2003) identifican **13 patrones de diseño agrupados en tres categorías fundamentales**:

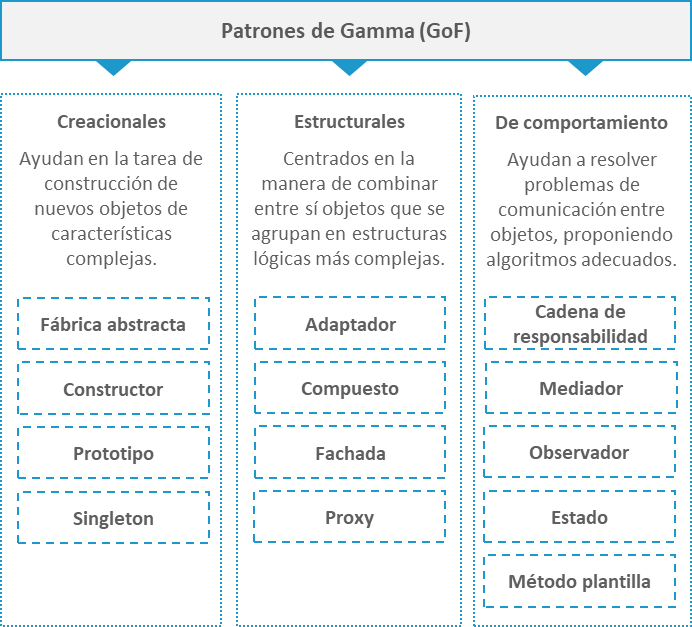


Figura 1. Patrones de diseño

Patrones creacionales

* **Fábrica abstracta (*Abstract Factory*)**

**Problema**: queremos trabajar con familias de objetos relacionados o dependientes entre sí, pero no nos preocupa la clase concreta, sino la manera de manipularlos.

**Este patrón permite crear una interfaz de manipulación común para un conjunto de objetos**. **No interesan las características concretas de cada objeto, al menos desde el punto de vista de su utilización.**

Pensemos en una aplicación con interfaz gráfica, que podemos generar empleando dos posibles alternativas de familias de controles gráficos, Tipo1 y Tipo2 (cada una de ellas compuesta por botones, listas desplegables, etc.). El mismo tipo de control dentro de cada familia se utiliza empleando el mismo conjunto de métodos. El diagrama mostrado en la figura 2 permite resolver este tipo de problema. Al cliente que crea y manipula los controles no le interesa el tipo concreto de botón que se instancia, sino las interfaces que ofrece para su manipulación. La clase encargada de crear el objeto del tipo correcto en cada caso es una factoría que, a su vez, implementa unos métodos de creación conocidos por el cliente, independientemente del tipo de factoría concreta que estemos utilizando.

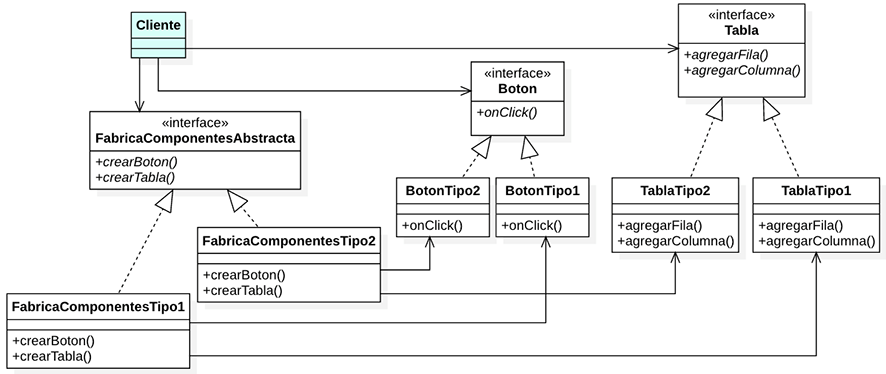


Figura 2. Ejemplo de utilización del patrón de fábrica abstracta

Una aplicación que manipula diferentes tipos de documentos o un videojuego con varios tipos de personaje son otros ejemplos de aplicaciones que podrían beneficiarse de este patrón, siempre y cuando **estas familias de objetos puedan compartir una interfaz de utilización común, de manera que no importe la clase concreta que estemos empleando en cada caso.**

* **Constructor (*Builder*)**

**Problema:** deseamos separar la construcción de un objeto (el conjunto de pasos necesarios) de su representación (la información que contiene), de manera que un mismo proceso de construcción pueda dar lugar a representaciones diferentes.

**El patrón *builder* nos ayuda a encapsular el procedimiento de construcción de objetos complejos en una clase especializada.** El cliente que desea crear el objeto simplemente utiliza los métodos ofrecidos por el constructor, que devuelve una instancia concreta con las características deseadas.

En la figura 3 tenemos una posible manera de modelar este patrón. De manera más concreta, supongamos que estamos desarrollando una aplicación capaz de conversión de vídeo. El resultado puede ser ofrecido en diferentes formatos contendores (AVI, MPG, MOV, etc.) siendo posible a la vez emplear diferentes códecs para la compresión de audio y vídeo, y realizar otro tipo de operaciones durante el procesamiento (recortado, escalado, etc.).

En este caso, podríamos implementar un constructor concreto para cada uno de los tipos de contenedor soportados, devolviendo cada uno de ellos un conversor específico para ese formato. La configuración de cada uno de los conversores puede ser similar; su construcción requiere de los mismos pasos (selección de códec de vídeo, selección de códec de audio, etc.), de manera que el proceso de construcción, o al menos los pasos necesarios, quedan modelados en la clase abstracta «Constructor», de la que todos los constructores concretos heredan.

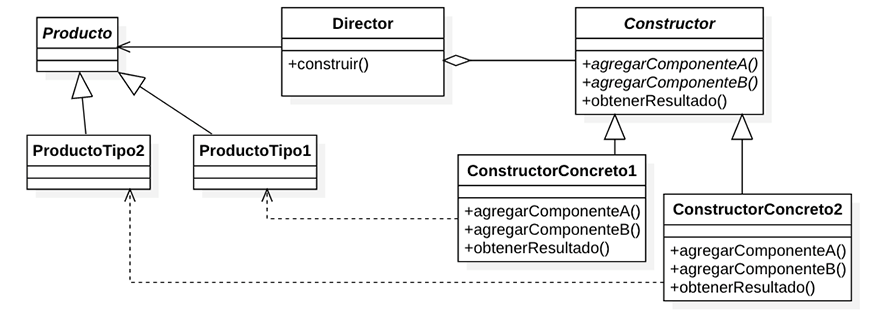


Figura 3. Diagrama de clases para un patrón constructor

Un ejemplo claro de utilización de este patrón lo encontramos en el proceso de construcción de una URI en código Java, como se muestra en la figura 4.



Figura 4. Ejemplo de utilización del patrón *builder* en código Java

* **Prototipo (*Prototype*)**

**Problema**: necesitamos crear en tiempo de ejecución nuevas instancias de objetos similares o idénticos a otros que ya existen en el programa.

**Este patrón permite crear copias de otros objetos en tiempo de ejecución, de manera que las copias sean idénticas o con ciertas modificaciones.**

La instancia prototípica implementa una interfaz que contiene el método clonar(), de manera que el cliente sabe que es posible obtener copias de manera sencilla de ese objeto. La figura 5 muestra un esquema de esta solución.

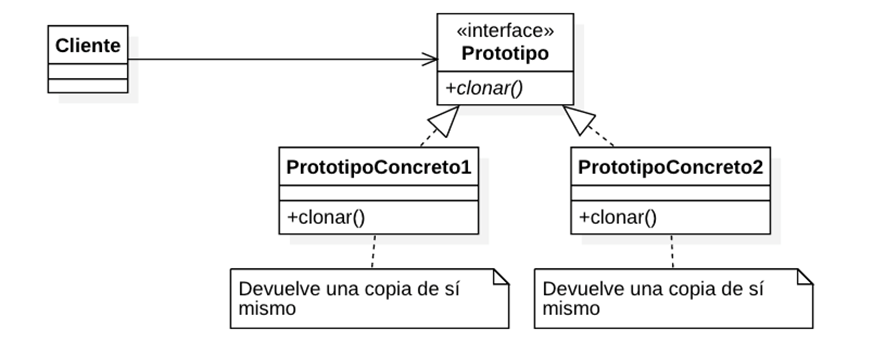


Figura 5. Ejemplo de diagrama de clases para el patrón prototipo

Pensemos, como ejemplos de aplicación, en **un juego** en el que es necesario crear muchas copias de un personaje —por ejemplo, un enemigo— en un momento dado, o en **un editor de gráficos vectoriales**, donde debemos crear habitualmente diferentes formas que comparten muchas características comunes.

* **Singleton**

**Problema**: necesitamos garantizar que en todo momento sólo existe una instancia de un determinado tipo de objeto.

**Este patrón define una clase con un único punto de acceso global y es la propia clase quien garantiza que el objeto que devuelve es la única instancia de esa clase.**

La **solución** consiste en ocultar el constructor de la clase y definir un método estático que devuelve la única instancia existente, como vemos en la figura 6.

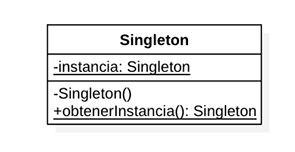
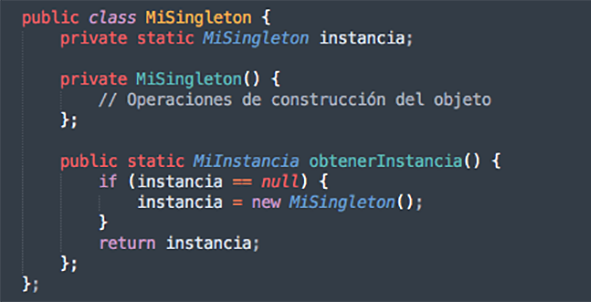


Figura 6. Modelo de clase y código de ejemplo Java del patrón singleton

Patrones estructurales

* **Adaptador (*Adapter*)**

**Problema**: necesitamos adaptar la interfaz que ofrece un componente para que los clientes puedan utilizarla. Este es un problema habitual cuando empleamos un **enfoque de reutilización de software**. Puede que necesitemos comunicar entre sí clases que ofrecen servicios con otras que pueden consumirlos, pero las interfaces de estos objetos son incompatibles.

**La solución**, que permite la colaboración, consiste en intercalar entre ambas una nueva clase que hace el trabajo de traducción.

Pensemos en un **editor de dibujo** **que utiliza una librería** que permite la creación de objetos gráficos. Las clases propias del dominio de aplicación aparecen representadas en amarillo en la figura 7, mientras que las clases que proceden de la librería reutilizada se muestran en azul.

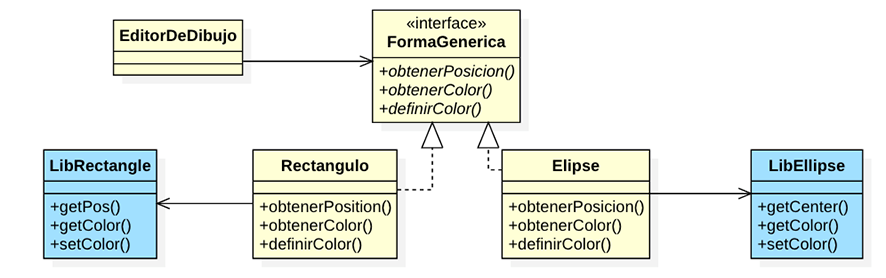


Figura 7. Ejemplo de diagrama de clases para el patrón adaptador

Otro ejemplo podría ser el de **una aplicación que consume información de diferentes API**, todas ellas con un tipo de información semejante (por ejemplo, comentarios de usuarios en noticias de diferentes medios especializados en un tema), pero que desde el punto de vista de la aplicación es necesario uniformizar su uso estableciendo una nomenclatura de variables uniforme.

* **Compuesto (*Composite*)**

**Problema**: tenemos un conjunto de objetos similares que se agrupan jerárquicamente, siendo unos contenedores de otros, y queremos uniformizar y simplificar el trabajo con los diferentes elementos, individuales o compuestos.

Hay aplicaciones que de manera natural requieren de la agrupación recursiva de elementos. Por ejemplo, un editor de dibujo que permite agrupar un conjunto de formas básicas, o grupos de ellas, entre sí, o una aplicación que muestra una conversación entre usuarios con respuestas anidadas.

La **solución** consiste en definir «una clase abstracta que representa tanto a primitivas como a sus contendores» (Gamma et al., 2003, p. 151). Esta idea se muestra en el diagrama de la figura 8.

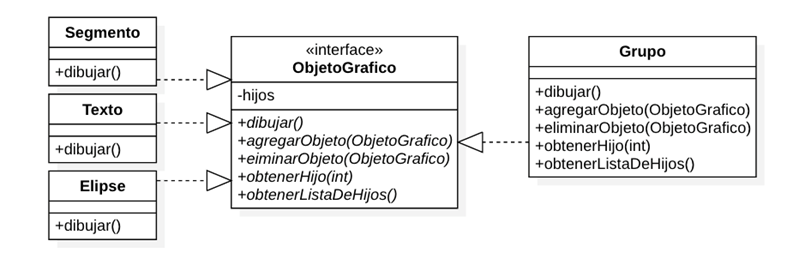


Figura 8. Ejemplo de diseño para el patrón composite

* **Fachada (*Facade*)**

**Problema**: nuestros objetos necesitan interactuar con un conjunto de interfaces de objetos de un subsistema, lo cual lleva a un elevado acoplamiento.

**El patrón de fachada proporciona una interfaz unificada y un punto de acceso común para un conjunto de interfaces de un sistema, simplificando su empleo.**

Supongamos una aplicación con interfaz de usuario en la que existen varias ventanas, diferentes en la capa de presentación, y diferentes clases, en la capa de negocio especializadas en operaciones concretas. Puede ser útil insertar una capa intermedia entre la capa de presentación y la de negocio, o fachada, que ofrezca un único punto de acceso a las funcionalidades ofrecidas. De esta manera, todas las peticiones son gestionadas por la fachada, que luego las encamina al objeto de dominio apropiado. También es una manera de simplificar la utilización de un conjunto de objetos que ofrecen servicios, y de disminuir el acoplamiento entre nuestras clases y las del sistema que ofrece los servicios que estamos consumiendo.

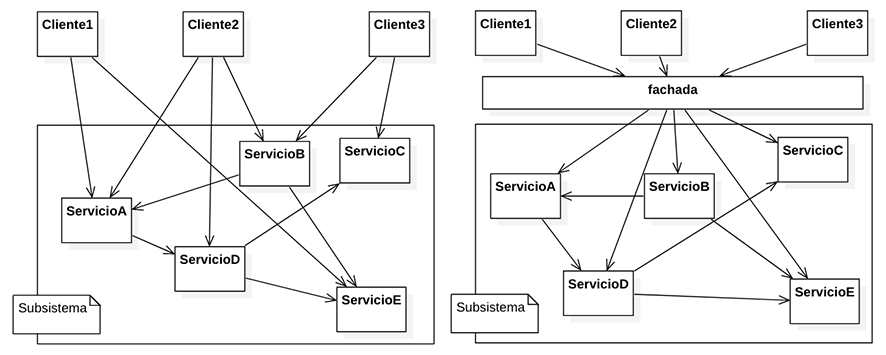


Figura 9. Diagrama de ejemplo para el patrón de fachada. A la izquierda se muestra el problema original, y a la derecha la solución que ofrece el patrón

* **Proxy**

**Problema**: necesitamos acceder a operaciones de un objeto, pero queremos añadir alguna función extra, o el acceso directo resulta demasiado complejo o costoso.

La **solución** consiste en definir un objeto proxy (o apoderado) que es un representante o sustituto del objeto real (al que llamaremos sujeto), permitiendo su manipulación indirecta. Se consigue de esta manera un mecanismo de manipulación más sofisticado de un objeto, que permite agregar funcionalidad.

En muchos casos, el objeto sustituido es un sistema o un dispositivo externo, siendo habitual su empleo en **operaciones de acceso a la red**. En este caso, el patrón de fachada es equivalente al patrón proxy.

Otra aplicación habitual es la **sustitución de un objeto cuya carga en memoria puede resultar muy lenta o costosa**. Por ejemplo, en un procesador de textos cada uno de los elementos del documento está asociado a un objeto que existe en la memoria del computador. Interesa que la carga de los documentos sea rápida, pero algunos elementos, como las imágenes, pueden resultar voluminosos, ocupar gran espacio en memoria, y ralentizar la apertura del documento. En este caso, se pueden sustituir las imágenes por objetos proxy en el momento de la carga del documento, de manera que se aplaza la carga de la imagen del disco a memoria hasta el momento en el que debe ser representada al usuario en la pantalla.

Encontramos, así, **diferentes tipos de proxy** (Gamma et al., 2003, p. 192):

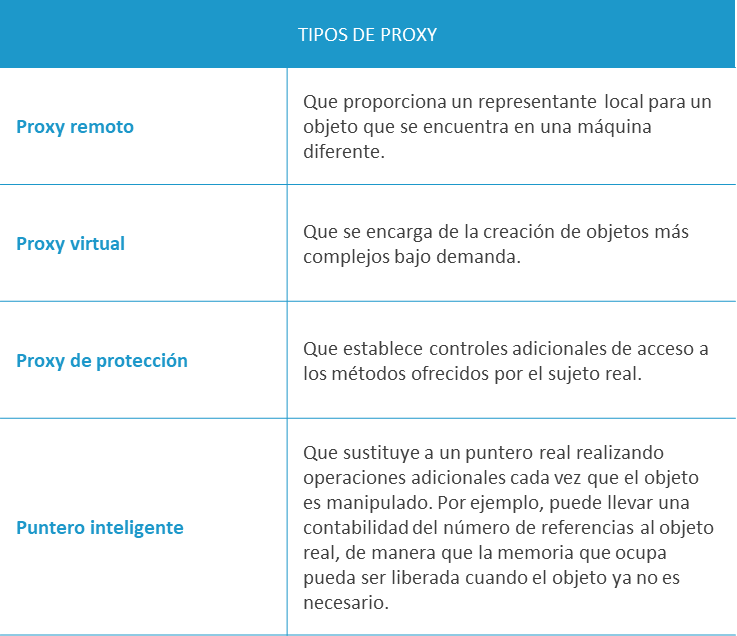


Figura 10. Tipos de proxy

En las figuras 11 y 12 se muestra un diseño simplificado de este tipo de solución para un problema genérico. Tanto el objeto proxy como el sujeto real implementan la misma interfaz. Sin embargo, el cliente únicamente colabora con el objeto proxy, que es el único que conoce al sujeto real que ofrece la funcionalidad solicitada.

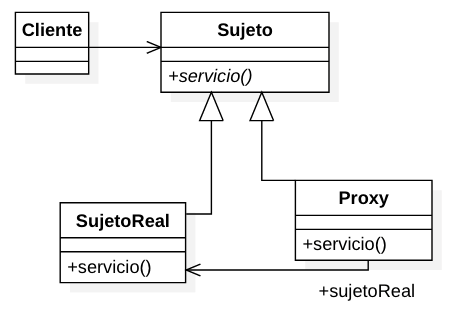


Figura 11. Diagrama de clases como ejemplo de un patrón proxy

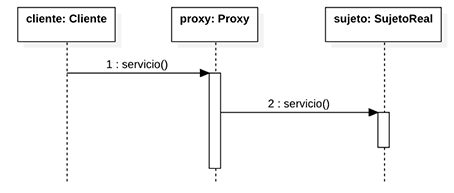


Figura 12. Diagrama de secuencia como ejemplo de un patrón proxy

Patrones de comportamiento

* **Cadena de responsabilidad (*Chain of responsibility*)**

**Problema**: se genera un evento en el sistema que puede ser procesado por uno o varios objetos diferentes y se busca la mejor manera de dirigir el flujo de procesamiento del evento.

**Este patrón establece una cadena de objetos que pasan de unos a otros una petición, hasta que se encuentra un objeto capaz de procesarla. De esta manera el emisor y el receptor de la petición quedan desacoplados.**

Una **solución** consiste en definir una clase abstracta con un método capaz de responder a esa petición, y un objeto sucesor al cuál pasar la petición en caso de ser necesario. Todas las instancias concretas implementan esta interfaz; las que pueden responder a la petición lo hacen y, en caso contrario, la derivan al sucesor asignado.

Supongamos que un procesador de texto ofrece un sistema de ayuda contextual al usuario, simplemente pulsando sobre un elemento de la interfaz. La ayuda ofrecida debería ser diferente en función del elemento seleccionado. Puede que algunos elementos no ofrezcan un mensaje de ayuda concreto, pero en este caso sería bueno que delegaran esta funcionalidad en algún otro objeto de jerarquía superior, que sea capaz de procesar el evento, buscando siempre una información de ayuda de lo más específico a lo más general, como se representa en la   
figura 13.

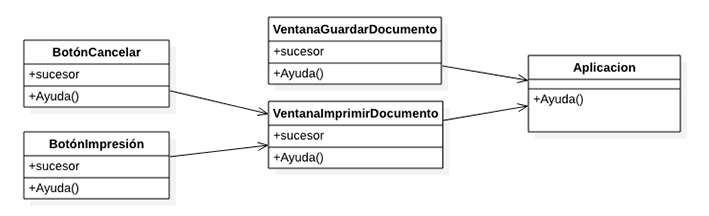


Figura 13. Ejemplo de patrón de cadena de responsabilidad en un procesador de textos

* **Patrón mediador (*Mediator*)**

**Problema**: existe un conjunto de objetos que colaboran habitualmente entre sí intercambiando información o invocando métodos ajenos, y se desea reducir el acoplamiento entre ellos.

En muchos sistemas encontramos conjuntos de objetos que deben colaborar o comunicarse entre sí. Una **solución** simple podría conducir a que todos los objetos conocieran a todos los restantes, aumentando innecesariamente el número de conexiones y, por tanto, el acoplamiento general del sistema.

**La solución de este patrón consiste en definir un objeto mediador que es «responsable de controlar y coordinar las interacciones entre un grupo de objetos»** (Gamma et al., 2003, p. 252).

Supongamos una interfaz de usuario en la que modificaciones en el estado de algún componente afectan a otros (por ejemplo, si un campo de texto está vacío se deshabilita un botón, o si se supera un número de caracteres se muestra un mensaje de alerta). Podría ser una ventana de *login* a un sistema, como se muestra en la figura 14.

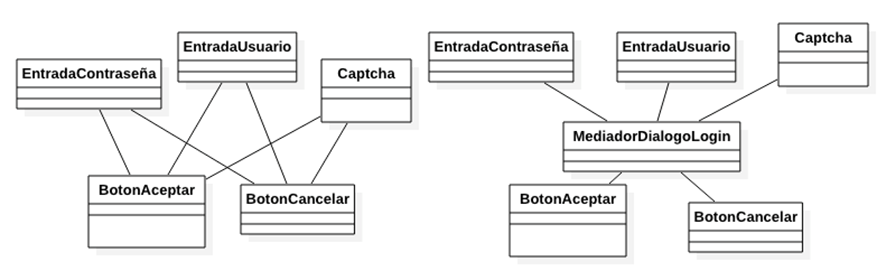


Figura 14. Ejemplo de patrón mediador en una ventana de *login* a un sistema. A la izquierda encontramos un sistema fuertemente acoplado, con muchas dependencias entre sus clases. El patrón mediador empleado en el ejemplo de la izquierda reduce el acoplamiento, introduciendo un objeto central de coordinación de la comunicación entre el resto de los participantes

El estado de los diferentes elementos de entrada de información puede afectar a la habilitación o deshabilitación de otros elementos de la interfaz. Mientras que a la izquierda de la figura tenemos un conjunto de objetos muy acoplados entre sí, la solución propuesta, introduciendo un mediador, reduce el acoplamiento y facilita la mantenibilidad.

* **Observador (*Observer*)**

**Problema:** un conjunto de objetos debe ser notificado acerca de los cambios que se producen en otro objeto del sistema.

**El patrón que resuelve este problema es similar al mediador, pero en este caso se trata únicamente de notificar sobre los cambios en el estado de un objeto a otro conjunto de objetos.**

La **solución** consiste en definir dos tipos de objeto fundamentales:

* + El **sujeto**, donde se producen los cambios en la información.
  + El **observador**, que es notificado por el sujeto sobre estos cambios.

Se define así un modelo de publicación–suscripción, en el que el sujeto notifica a sus observadores cuando se produce un cambio, como se muestra en el ejemplo de la figura 15.

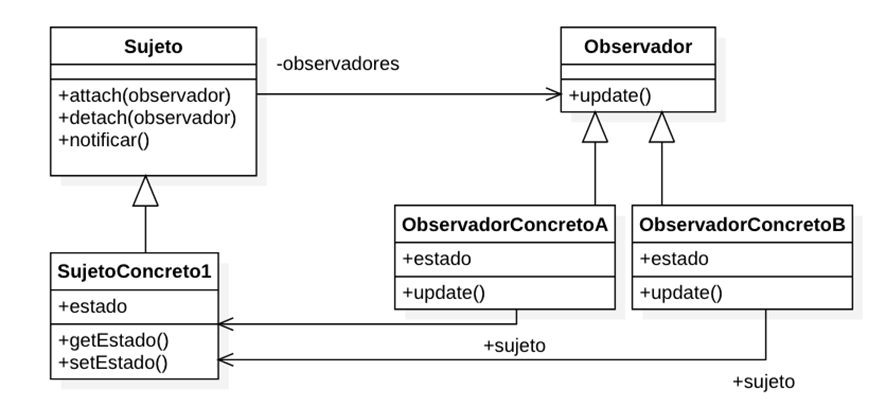


Figura 15. Ejemplo de utilización del patrón observador

La secuencia de interacciones se representa en el diagrama de la figura 16.

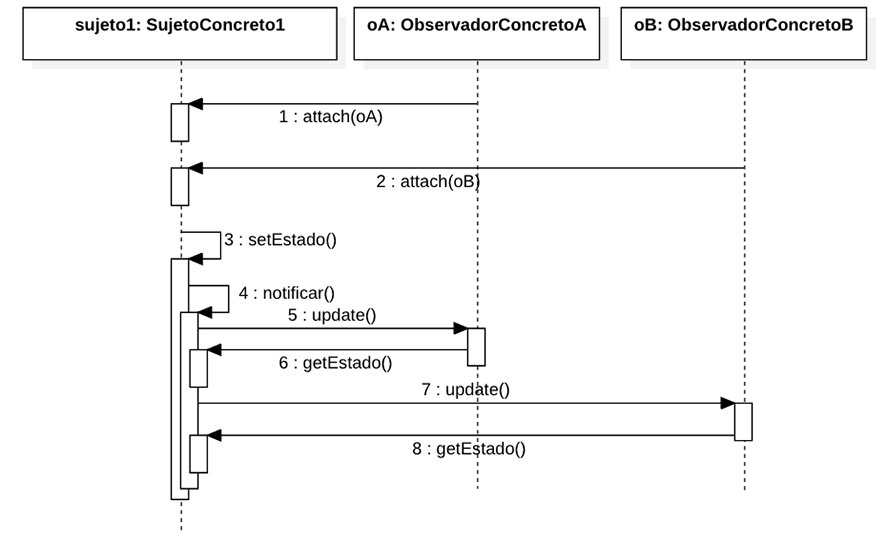


Figura 16. Diagrama de secuencia mostrando las interacciones entre clases empleando un patrón observador

Como podemos ver, el sujeto mantiene una lista de todos los observadores que se han registrado con él. Cuando se produce un cambio en su estado interno, lo notifica a todos ellos, de manera que puedan sincronizarse y adquirir la nueva información de estado.

* **Patrón Estado (*State*)**

**Problema**: el comportamiento de un objeto depende de su estado interno.

En muchas aplicaciones el comportamiento de un objeto viene afectado por su estado interno, o depende de ciertas variables. En estos casos, es posible delegar la implementación de esa funcionalidad correspondiente en una clase asociada.

Supongamos, por ejemplo, un videojuego en el que el comportamiento de un personaje depende de ciertas variables, como su nivel de energía, salud, o experiencia. En este caso, el personaje es un contexto que mantiene una instancia de una clase concreta que representa su estado actual (ver figura 17). El contexto ofrece una interfaz que permite a sus clientes realizar peticiones, pero cuando estas llegan el contexto ejecuta el método de gestión de la instancia concreta, que representa el estado actual, y es capaz de responder a esa petición.

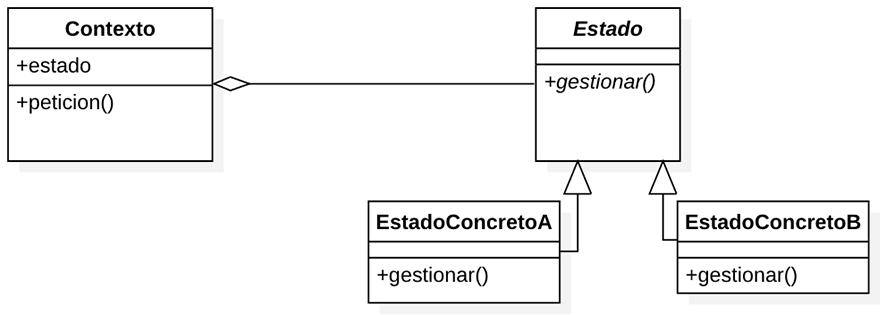


Figura 17. Ejemplo de utilización del patrón de estado.

* **Método plantilla (*Template–method*)**

**Problema**: existe un conjunto de objetos que realizan un tipo de operaciones similares, con una estructura algorítmica general similar, pero se diferencian en algunos pasos del proceso. Se desea construir una jerarquía de clases que evite la duplicación de código.

La **solución** consiste en crear en una clase base un método que implementa el esqueleto del algoritmo (método plantilla), haciendo uso de métodos abstractos que deberán ser concretados en las clases derivadas, donde se puede agregar funcionalidad específica. De esta manera, la secuencia de pasos lógicos necesarios queda fija, y solo es preciso agregar el comportamiento específico en las clases concretas que utiliza la aplicación.

En la figura 18 vemos un ejemplo de ello. Supongamos que estamos desarrollando un visor de documentos que soporta varios formatos de archivo y tipos de documento (hoja de cálculo, imagen, documentos de texto, etc.). El proceso de carga es básicamente el mismo en cuanto al procedimiento, pero se diferencia en función del tipo de documento específico que debe ser cargado y representado de manera diferente. En la clase base Aplicación se define el método plantilla **abrirDocumento(),** que define los pasos necesarios para cargar y representar un documento cuyo nombre es recibido como parámetro. Se encarga de realizar la comprobación de que el formato es soportado empleando un método virtual **(documentoCompatible())**, y a partir de ahí define la secuencia de pasos necesaria para mostrar finalmente el contenido del archivo, como vemos en la figura 19.

Cabe resaltar que los métodos **Aplicacion::crearDocumento(), Aplicacion::crearVisor()**, **Documento::analizar()** y **Visor::mostrar()** son todos ellos métodos virtuales, que deberán ser concretados, respectivamente, en las clases **MiAplicacion, MiDocumento** y **MiVisor**.

Figura 18. Ejemplo de aplicación del patrón de método plantilla

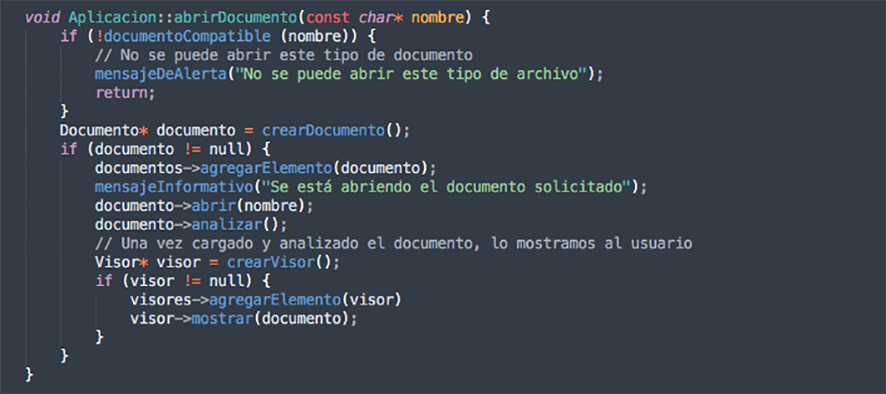
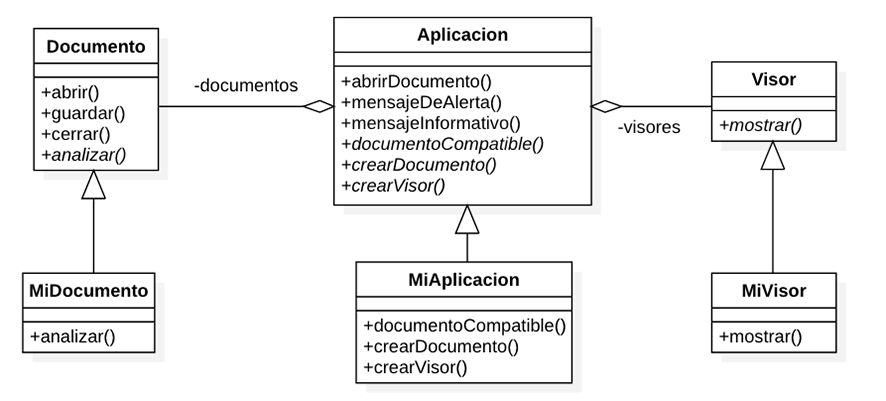


Figura 19. Código de ejemplo de implementación de un patrón de método plantilla.

6.4. Patrones de diseño de interacción

**Los patrones de diseño de interfaz de usuario son buenas prácticas que permiten resolver problemas comunes desde el punto de vista de la interacción del usuario con el software.**

Mientras que los patrones de diseño software, que hemos venido estudiando, están destinados a resolver problemas de funcionalidad, **los patrones de diseño de interacción responden a problemas de usabilidad** (Hassan, 2012). Sin embargo, ambos enfoques son complementarios, y a menudo se utilizan conjuntamente en el contexto de las aplicaciones interactivas, pues la funcionalidad afecta a la usabilidad, y viceversa.

En el caso de los patrones de diseño de interacción encontramos **mayores problemas de normalización en cuanto a su clasificación**, y **la dificultad de que no existe un lenguaje de etiquetado** que permita definirlos precisamente facilitando su reutilización. Aun así, podemos encontrar algunas colecciones de patrones distribuidas por la red, que se resumen en la tabla 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **TÍTULO DEL CATÁLOGO** | **URL** | **CATEGORÍAS DEFINIDAS EN EL CATÁLOGO** | |
| *User Interface Design Patterns* | <https://goo.gl/HEsFn5> | * Búsqueda * Visualización de datos * Almacenamiento * Selección y manipulación de objetos | * Tiempo * Jerarquías y conjuntos * Guardar y deshacer |
| *Patterns in Interaction Design* | <http://www.welie.com/> | * Navegación * Interacción básica * Búsqueda * Trabajo con datos * Personalización | * Compras * Tomar decisiones * Entrada de datos * Miscelánea |
| *UI Patterns* | <http://ui-patterns.com/> | * Entrada de datos * Navegación * Trabajo con datos | * Social * Incorporación * Miscelánea |

Tabla 1. Selección de catálogos de diseño de interacción

El número de patrones de diseño de interfaz es enorme. Como muestra, vamos a ver a continuación algunos de ellos, obtenidos a partir de la recopilación de Toxboe (2018).

Indicador de la fuerza de una contraseña

**Categoría**: entrada de información/formularios.

**Problema**: deseamos asegurarnos de que la contraseña elegida tiene la fuerza suficiente para prevenir posibles ataques.

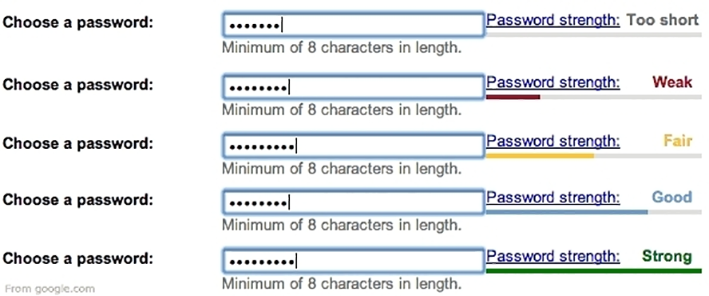


Figura 20. Ejemplo de utilización de un indicador de fuerza de contraseña. Fuente: Toxboe (2018)

Carrusel

**Categoría**: navegación/contenido.

**Problema**: el usuario desea navegar a lo largo de un conjunto de elementos y posiblemente seleccionar uno de ellos.

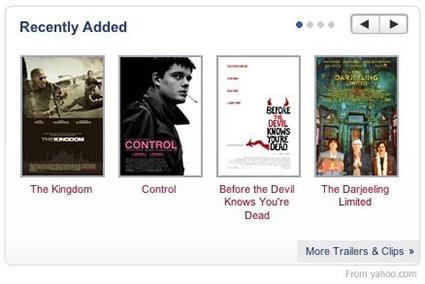


Figura 21. Ejemplo de utilización de un carrusel

Tabla de precios

**Categoría**: miscelánea/compras.

**Problema**: el usuario necesita tener una visión general de los planes de precios ofrecidos y sus diferencias (Toxboe, 2018)

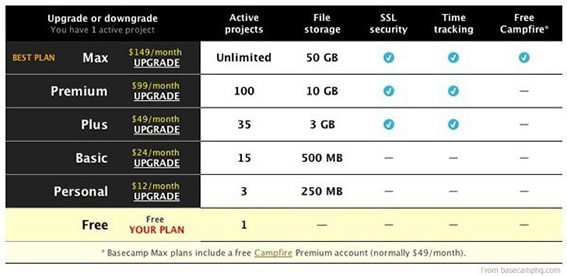


Figura 22. Ejemplo de aplicación del patrón de tabla de precios. Fuente: Toxboe (2018)

Reacción

**Categoría:** social/interacciones sociales.

**Problema:** el usuario quiere expresar sus emociones sobre un contenido de manera sencilla.

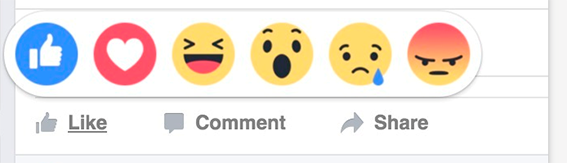


Figura 23. Ejemplo de utilización del patrón de reacción. Fuente: Toxboe (2018)

6.5. Referencias bibliográficas

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (2003). *Patrones de diseño: elementos de software orientado a objetos reutilizables*. Madrid: Pearson Educación.

Hassan, Y. (2012). Patrones de diseño de interacción [Blog personal]. Recuperado de <http://yusef.es/blog/2012/11/patrones-de-diseno-de-interaccion/>

Polo, M. (2012). *Desarrollo de software basado en reutilización* [Material docente]. Barcelona: Universitat Oberta de Catalunya. Recuperado de <http://hdl.handle.net/10609/63466>

Toxboe, A. (2018). User Interface Design Patters. Recuperado de <http://ui-patterns.com/>

A fondo

**Lección magistral: Patrones de Gamma**

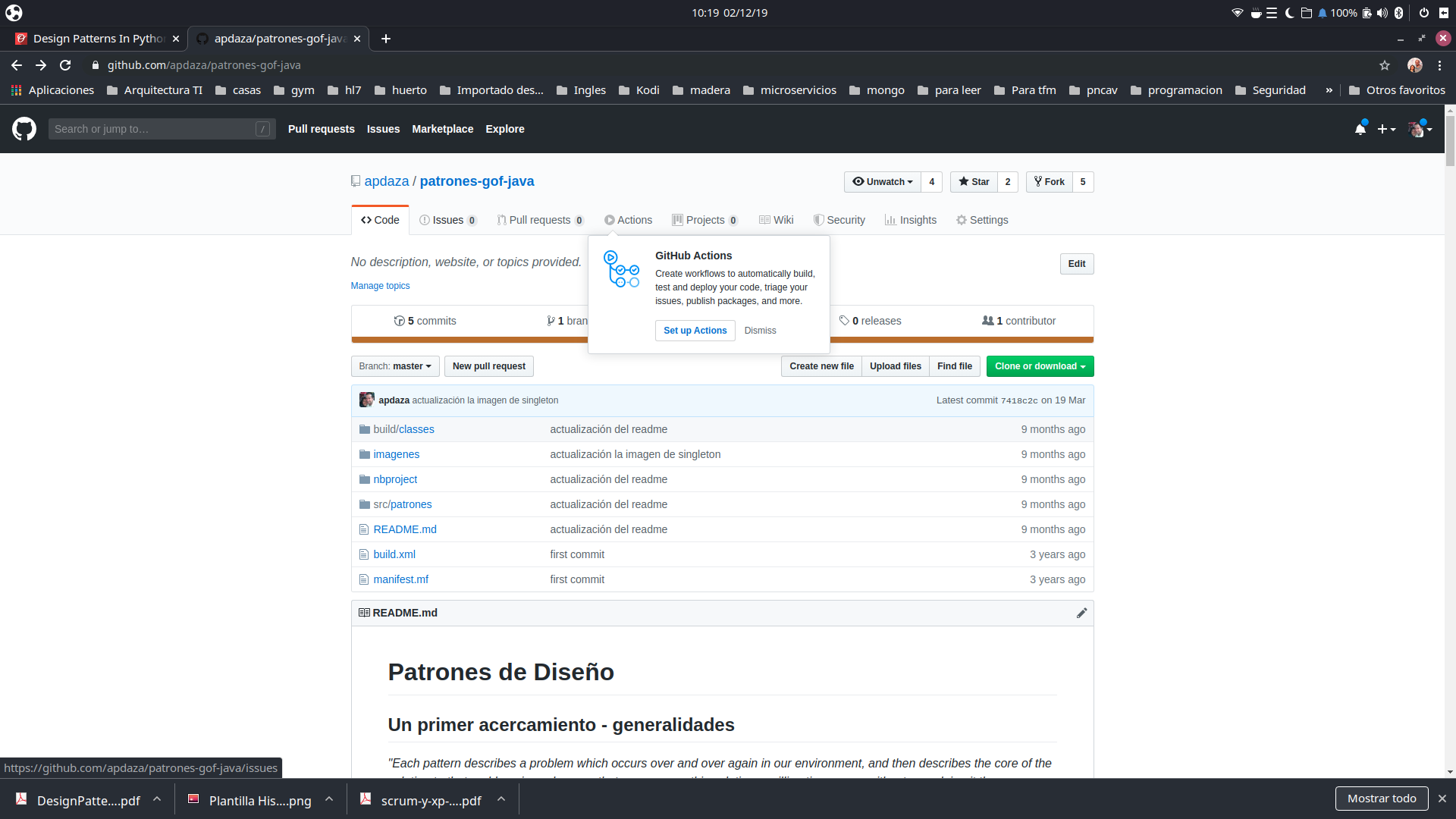
En esta lección magistral introduciremos algunos de los patrones de Gamma, que resuelven problemas de diseño habituales en los sistemas informáticos; especialmente aquellos cuyo diseño está orientado a objetos. En concreto, introduciremos los siguientes patrones: fábrica abstracta, adaptador, y mediador.



Accede a la lección magistral a través del aula virtual.

**Patrones de diseño software**

En este repositorio web encontrarás una descripción de los patrones de diseño. Se muestra un diagrama UML para cada uno de los casos, y lo más interesante es que se aportan ejemplos de implementación concretos en código Java.



Accede a la web a través del aula virtual o desde la siguiente dirección web:

[https://github.com/apdaza/patrones-gof-java](https://informaticapc.com/patrones-de-diseno/)

**The GoF Design Patterns Reference**

En esta web encontrarás mucha información sobre los patrones de diseño analizados por Gamma y sus colaboradores. Cada patrón está acompañado de ejemplos de aplicación, indicaciones de implementación y código de ejemplo. Además, la web ofrece un curso de certificación y la posibilidad de descargar el manual en PDF.



Accede a la web a través del aula virtual o desde la siguiente dirección web:

http://www.w3sdesign.com/

Pattern Languages of Programs (PLoP)

En la web oficial de esta familia de conferencias encontrarás información sobre eventos, libros, fechas de conferencias sectoriales o regionales específicas, y una recopilación de catálogos de patrones de diseño para dominios de aplicación específicos.

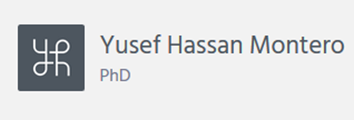


Accede a la web a través del aula virtual o desde la siguiente dirección web:

<https://hillside.net/>

Patrones de diseño de interacción

En este artículo de su blog personal, el profesor Yusef Hassan, de la Universidad Internacional de La Rioja, presenta una introducción a los beneficios que supone la utilización de patrones de diseño. En el artículo también se explica la diferencia entre patrones de diseño y guías de usabilidad, y se ofrece una lista de recursos con colecciones de patrones y bibliografía relacionada.

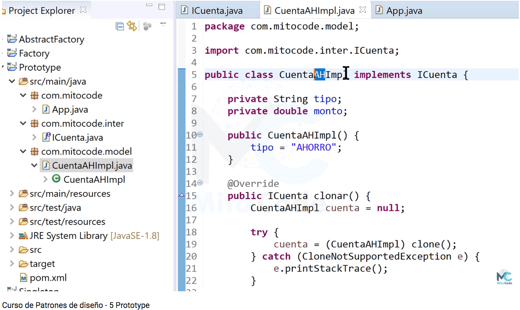


Accede al documento a través del aula virtual o desde la siguiente dirección web:

<http://yusef.es/blog/2012/11/patrones-de-diseno-de-interaccion/>

Curso de patrones de diseño

En esta *playlist* encontrarás una serie de vídeos en los que se explican algunos de los patrones que hemos estudiado en este tema. También se muestra cómo implementarlos en código Java utilizando el IDE Eclipse.



Accede a los vídeos a través del aula virtual o desde la siguiente dirección web:

<https://youtu.be/M3VT1v54cq4>

**Design Patterns in Java Tutorial**

Este completo tutorial ofrece una introducción a los patrones de diseño analizados en este tema, y otros muchos, así como código de ejemplo en lenguaje Java.



Accede al tutorial a través del aula virtual o desde la siguiente dirección web:

https://www.tutorialspoint.com/design\_pattern/index.htm

**Patrones de diseño en Python**

Verma, R. & Giridhar, C. (2011). *Design Patterns In Python: a learner’s approach to understand design patterns via Python programming language*. Autoedición.

En este libro (disponible gratuitamente bajo licencia *Creative Commons Attribution-NonCommercialNoDerivs* 3.0) encontrarás la descripción de 7 patrones de diseño y de código de ejemplo en lenguaje Python.



Accede al documento a través del aula virtual o desde la siguiente dirección web:

<https://openlibra.com/es/book/design-patterns-in-python>

Actividades

Trabajo: Diseño de una aplicación con patrones

En esta actividad vamos a aplicar conceptos de los temas 5 y 6 de la asignatura para realizar el **diseño de una aplicación cliente para la red social Twitter**.

Partiremos de la **especificación de requisitos desarrollada en   
la primera actividad de la asignatura**.

Objetivos de la actividad

Realizando esta actividad alcanzarás los siguientes objetivos:

* Aprender a **analizar una especificación de requisitos funcionales**, en la forma de historias de usuario, y **desarrollar un diseño acorde**.
* **Aplicar los conceptos sobre patrones de diseño** estudiados al diseño de una aplicación real.
* **Diseñar la interfaz gráfica** de la aplicación utilizando patrones de diseño.

Descripción de la actividad

La actividad consiste en desarrollar el diseño de una aplicación cliente de Twitter aplicando patrones de diseño estudiados en el tema. Partiremos de la especificación de requisitos que definimos durante la primera actividad de la asignatura. A partir de ellos deberás:

* **Diseñar una arquitectura general del sistema** que considere los componentes internos fundamentales y sus interacciones con otros sistemas externos.
* **Diseñar una arquitectura de clases** que modele los conceptos y funcionalidades de una manera acorde a la especificación.
* **Diseñar una interfaz gráfica** empleando patrones de diseño de interacción.

Entrega

Cada alumno entregará **de manera individual un documento en formato PDF** que deberá contener los siguientes apartados:

1. **Introducción**. Descripción de la aplicación que estamos diseñando y resumen de las principales funcionalidades (explica principalmente aquellas en las que te vas a centrar en las labores de diseño).

Extensión aproximada: media página.

1. **Diseño de la arquitectura del sistema**. Aquí deberás representar un diseño de alto nivel del sistema en su conjunto, considerando tanto la aplicación desarrollada como otros sistemas con los que debe interactuar (por ejemplo, la API de Twitter). Explica sus componentes principales.

Extensión aproximada: una página.

1. **Diseño de componentes y clases**. Aquí deberás realizar un diseño más detallado de la aplicación. No es necesario diseñar la aplicación entera, sino resolver 3 problemas específicos empleando 3 patrones adecuados. Deberías seguir una estructura similar a la siguiente:
   * **Problema**: descripción del problema que quieres resolver.
   * **Funcionalidad asociada**: descripción de la historia de uso de la que has partido.
   * **Solución**: descripción del patrón de diseño aplicado y diagrama de clases correspondiente.

Extensión aproximada: una página por cada patrón.

1. **Diseño de la interfaz de usuario**. Finalmente, debes incluir un diseño preliminar de lo que podría ser la interfaz de usuario. Para ello, debes aplicar 3 patrones de diseño de interfaz, indicando el nombre del patrón, el problema que resuelve, y cómo lo has utilizado en tu diseño.

Extensión aproximada: una página.

Criterios de evaluación

Al evaluar esta actividad se tendrán en cuenta los siguientes aspectos:

* **Correcta estructura y redacción del documento**, sin faltas ortográficas y con capacidad de síntesis.
* **Estética del documento,** cuidando los aspectos visuales de los paneles y gráficos representados.
* **Calidad del diseño arquitectónico general** planteado y **correcta explicación**.
* **Correcta utilización de los patrones de diseño** para resolver problemas específicos.
* **Adecuación del diseño** de interfaz planteado a las características de la aplicación, y correcta aplicación y utilización de los patrones de diseño de interfaz.

Test

1. ¿En qué tres categorías suelen agruparse los patrones de diseño?

A. Creacionales, estructurales y conductuales.

B. Creativos, estructurales y de comportamiento.

C. Adaptativos, compuestos y mediadores.

D. Todas las respuestas anteriores son correctas.

1. ¿Qué tipo de problemas ayuda a resolver el patrón observador?

A. El comportamiento de un objeto depende de su estado interno.

B. Es necesario garantizar que en cada instante solo existe una instancia de un objeto.

C. Los cambios de estado de un objeto deben ser notificados a un conjunto de objetos interesados

D. Necesitamos adaptar la interfaz de un componente, para que un cliente existente pueda utilizarla.

1. ¿Qué tipo de problemas ayuda a resolver el patrón singleton?

A. El comportamiento de un objeto depende de su estado interno.

B. Es necesario garantizar que en cada instante sólo existe una instancia de un objeto.

C. Los cambios de estado de un objeto deben ser notificados a un conjunto de objetos interesados

D. Necesitamos adaptar la interfaz de un componente, para que un cliente existente pueda utilizarla.

1. ¿Qué tipo de problemas ayuda a resolver el patrón estado?

A. El comportamiento de un objeto depende de su estado interno.

B. Es necesario garantizar que en cada instante sólo existe una instancia de un objeto.

C. Los cambios de estado de un objeto deben ser notificados a un conjunto de objetos interesados

D. Necesitamos adaptar la interfaz de un componente, para que un cliente existente pueda utilizarla.

1. ¿Qué tipo de problemas ayuda a resolver el patrón fachada?

A. El comportamiento de un objeto depende de su estado interno.

B. Es necesario garantizar que en cada instante sólo existe una instancia de un objeto.

C. Los cambios de estado de un objeto deben ser notificados a un conjunto de objetos interesados

D. Reducir el acoplamiento y facilitar el uso de un conjunto de interfaces de objetos de un subsistema.

1. Supongamos que estamos desarrollando un navegador web. Sabemos que muchas páginas web contienen documentos de gran extensión, con gran número de imágenes insertadas, lo que puede provocar que la carga del documento sea lenta. ¿Qué tipo de patrón de diseño podríamos aplicar para mejorar la experiencia del usuario y agilizar la carga de documentos?

A. El patrón composite.

B. El patrón *builder*.

C. El patrón proxy remoto.

D. El patrón proxy virtual.

1. ¿Con qué otros patrones de diseño guarda relación el patrón fachada?

A. Con el patrón adaptador. La fachada ofrece un punto de acceso común que adapta la manipulación de un conjunto de objetos en un subsistema, mientras que el adaptador lo hace para componentes específicos.

B. Con la cadena de responsabilidad, porque cada uno de los objetos que pasan el evento a su sucesor puede ser visto como una fachada.

C. Con el patrón observador, porque el observador se comporta como una fachada frente a los objetos que son notificados de cambios de estado.

D. Ninguna de las respuestas es correcta.

1. ¿Qué tipo de problemas ayudan a resolver los patrones de diseño de interacción?

A. Problemas de funcionalidad.

B. Problemas de usabilidad.

C. Problemas de configuración.

D. Problemas de acoplamiento.

1. ¿Cuáles de las siguientes características son deseables en un patrón de diseño?

A. Que haya mostrado su efectividad en más de una ocasión.

B. Que esté publicado en un libro.

C. Que ofrezca una descripción genérica que permita adaptarla a diversos contextos.

D. Las respuestas A y C son correctas.

1. Supongamos que estamos desarrollando un juego de coches de carreras. El usuario puede configurar los coches a voluntad, seleccionando entre diferentes tipos de neumáticos, motores, carrocerías, etc. De esta manera, es necesario modelar objetos complejos que representen instancias concretas de cada coche en tiempo de ejecución. ¿Qué patrón de diseño sería más adecuado para resolver este problema?

A. La factoría abstracta.

B. El prototipo.

C. El constructor o *builder*.

D. El singleton.