Patrones de Diseño GOF

Este documento es una adaptación al lenguaje español de la DZone Refcard – Design Patterns elaborado por Jason McDonalds
http://refcardz.dzone.com/refcardz/design-patterns#refcard-download-social-buttons-display

A continuación se presentan los patrones de diseño GOF, tal como están listados en el libro *Design Patterns: Elements of Reusable Object-Oriented Software*. Cada patrón incluye un diagrama de clases, una explicación, información de uso y una descripción de un ejemplo del mundo real.

Los patrones GOF se catalogan en tres tipos:

C **Creacionales**: Usados para crear objetos que sean desacoplados de los sistemas que los implementan.

E **Estructurales**: Usados para construir estructuras complejas entre muchos objetos diferentes.

B **De comportamiento**: Usados para administrar relaciones y responsabilidades entre objetos.

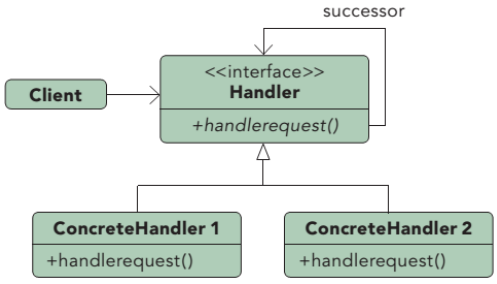
Los patrones pueden tener dos alcances: de objetos y de clases.

Alcance de objetos: Trata con relaciones de objetos que pueden ser cambiadas en tiempo de ejecución.

Alcance de clases: Trata con relaciones de clases que pueden ser cambiadas en tiempo de compilación.

C Abstract Factory	E Decorator	C Prototype
E Adapter	E Facade	E Proxy
E Bridge	C FactoryMethod	B Observer
C Builder	E Flyweight	C Singleton
B Chain of Responsability	B Interpreter	B State
	B Iterator	B Strategy
B Command	B Mediator	B Template Method
E Composite	B Memento	B Visitor

Chain of Responsability — objetos



Propósito: Le da a más de un objeto una oportunidad para responder una solicitud, al enlazar los objetos receptores juntos.

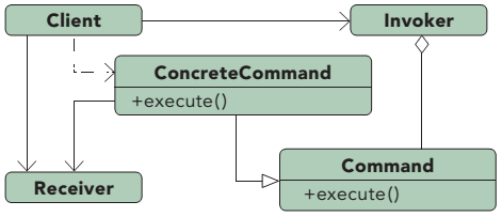
Usar cuando

- Múltiples objetos pueden manejar una solicitud y el que responde no tiene que ser un objeto específico.
- Un conjunto de objetos debe poder manejar una solicitud, determinado el manejador en tiempo de ejecución.
- Un resultado aceptable es una solicitud no tratada.

Ejemplo

El manejo de excepciones en algunos lenguajes implementa este patrón. Cuando una excepción es arrojada en un método, el motor de ejecución verifica si el método tiene un mecanismo para manejar la excepción o si se debe pasar a la pila de llamadas. Cuando sucede lo último, el proceso se repite hasta que se encuentre código para manejar la excepción o hasta que no haya más objetos padre para pasarles la solicitud.

Command — objetos



Propósito: Encapsular una solicitud, permitiendo que sea tratada como un objeto. Esto permite que la solicitud sea manejada en relaciones tradicionales basadas en objetos como encolamiento o *callback*.

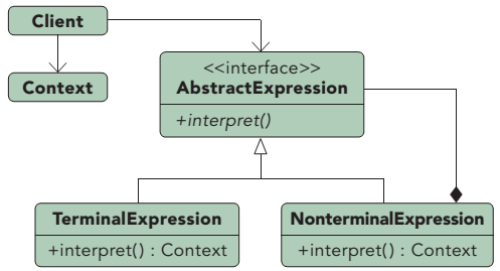
Usar cuando

- Se necesita funcionalidad de *callback*.
- Las solicitudes deben ser tratadas en tiempos variantes o en orden variante.
- Se requiere un historial de solicitudes.
- El invocador debe estar desacoplado del objeto que maneja la invocación.

Ejemplo

Las colas de trabajo son usadas ampliamente para facilitar el procesamiento asíncrono de algoritmos. Utilizando el patrón *command*, la funcionalidad que se debe ejecutar se le puede dar a una cola de trabajo para su procesamiento sin necesidad de que la cola tenga conocimiento de la implementación real que está invocando. El objeto *command* que es encolado implementa su algoritmo particular dentro de los confines de la interfaz que la cola está esperando.

Interpreter — clases



Propósito: Define una representación para una gramática al igual que un mecanismo para entenderla y actuar sobre ella.

Usar cuando

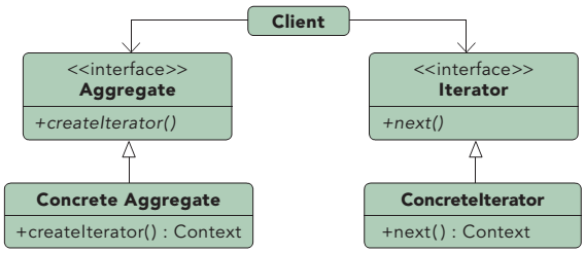
- Se debe interpretar una gramática que puede ser representada como un árbol de sintaxis.
- La gramática es simple.
- La eficiencia no es importante.
- Es deseable desacoplar la gramática de las expresiones subyacentes.

Ejemplo

Text based adventures (aventuras basadas en texto), ampliamente populares en los 80's, proveen un buen ejemplo de este patrón. Muchos tenían comandos simples, tal como

"*step down*" que permitía devolverse en el juego. Estos comandos podían estar anidados de tal forma que se alteraba su significado. Por ejemplo, "*go in*" podía tener un resultado diferente que "*go up*". Por medio de la creación de una jerarquía de comandos basada en el comando y el calificador (expresiones no-terminales y terminales), la aplicación podía fácilmente mapear muchas variaciones de comandos al árbol de acciones relacionado.

ITERATOR — objetos



Propósito: Permite acceder a los elementos de un objeto agregado sin permitir el acceso a sus representaciones subyacentes.

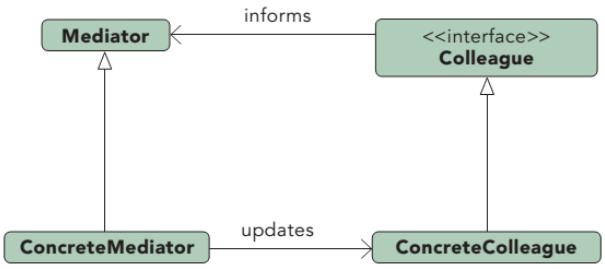
Usar cuando

- Se necesita acceso a los elementos sin acceder a la representación entera.
- Se necesitan recorridos múltiples o concurrentes de los elementos.
- Se necesita una interfaz uniforme para los recorridos.
- Existen diferencias sutiles entre los detalles de implementación de varios iteradores.

Ejemplo

La implementación del patrón *iterator* en Java permite a los usuarios recorrer varios tipos de datos sin preocuparse por la implementación subyacente de la colección. Dado que los clientes solo interactúan con la interfaz `Iterator`, a las colecciones se les permite definir el iterador apropiado para ellas mismas. Algunos pueden permitir acceso total al conjunto de datos subyacente, mientras que otros pueden restringir ciertas funcionalidades, como por ejemplo remover ítems.

MEDIATOR — objetos



Propósito: Permite bajo acoplamiento, encapsulando la forma en que diferentes conjuntos de objetos interactúan y se comunican entre sí. Permite que las acciones de cada conjunto de objetos varíen independientemente.

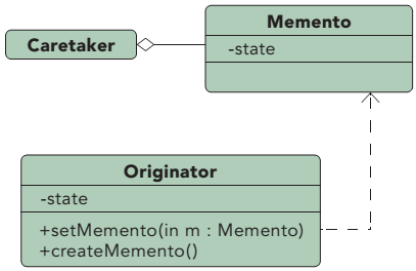
Usar cuando

- La comunicación entre conjuntos de objetos está bien definida y es compleja.
- Existen muchas relaciones y se requiere un punto común de control o comunicación.

Ejemplo

El software de listas de correo lleva registro de quienes se suscriben a la lista y provee un único punto de acceso a través del cual una persona se puede comunicar con la lista entera. Sin una implementación del patrón mediator, una persona que quiera enviar un mensaje al grupo tendría que llevar registro constantemente de quién se suscribe y quién no. Implementando el patrón *mediator*, el sistema está en la capacidad de recibir mensajes desde cualquier punto y luego determinar a qué destinatario direccionar el mensaje, sin necesidad de que el emisor del mensaje se tenga que preocupar de la lista de destinatarios.

MEMENTO — objetos



Propósito: Permite capturar y externalizar el estado interno de un objeto para que pueda ser restaurado posteriormente, todo esto sin violar el encapsulamiento.

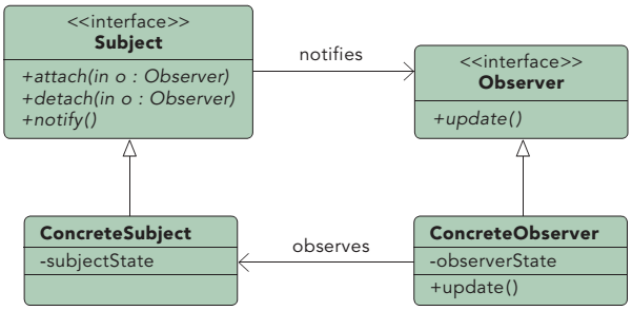
Usar cuando

- El estado interno de un objeto se debe guardar y restaurar posteriormente.
- El estado interno no se puede exponer por medio de interfaces sin exponer la implementación.
- Se debe preservar el encapsulamiento.

Ejemplo

La funcionalidad de deshacer se puede implementar adecuadamente usando el patrón memento. Serializando y deserializando el estado de un objeto antes de que el cambio ocurra, podemos preservar una foto de él que luego puede ser restaurada si el usuario decide deshacer la operación.

OBSERVER — objetos



Propósito: Permite que uno o más objetos sea notificados de los cambios de estado en otros objetos del sistema.

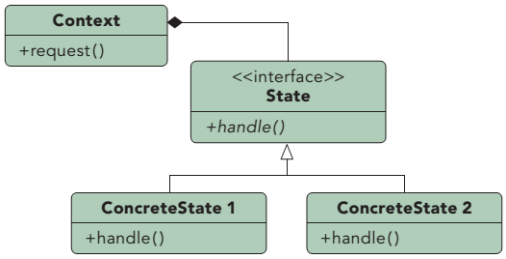
Usar cuando

- Los cambios de estado en uno o más objetos disparan comportamientos en otros objetos.
- Se requieren capacidades de difusión.
- Existe un entendimiento de que los objetos estarán ciegos a expensas de la notificación.

Ejemplo

Este patrón se puede encontrar en casi todos los ambientes GUI. Cuando los botones, el texto y otros campos son puestos en las aplicaciones, éstas típicamente registran un escuchador para esos controles. Cuando un usuario dispara un evento, como clic en un botón, el control recorre sus observadores registrados y envía una notificación a cada uno.

STATE — objetos



Propósito: Liga las circunstancias de un objeto con su comportamiento, permitiendo que el objeto se comporte de formas diferentes dependiendo de su estado interno.

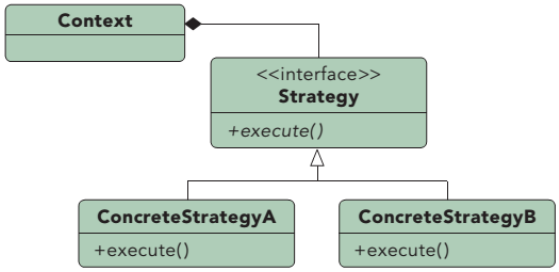
Usar cuando

- El comportamiento de un objeto debe ser influenciado por su estado.
- Condiciones complejas atan el comportamiento del objeto a su estado.
- Las transiciones entre estados deben ser explícitas.

Ejemplo

Un objeto `Correo` puede tener varios estados, los cuales cambiarán la forma en que el objeto maneja diferentes funciones. Si el estado es "no enviado", entonces la llamada al método `send()` enviará el mensaje, mientras que la llamada al método `recallMessage()` arrojará un error o no hará nada. Sin embargo, si el estado es "enviado", entonces la llamada al método `send()` arrojará un error o no hará nada, mientras que la llamada a `recallMessage()` enviará una notificación de respuesta a los destinatarios. Para evitar instrucciones condicionales en la mayoría de los métodos, existirán múltiples objetos de estado que manejarán la implementación respecto a su estado particular. Las llamadas dentro del objeto `Correo` serán delegadas al objeto de estado apropiado.

STRATEGY — objetos



Propósito: Define un conjunto de algoritmos encapsulados que pueden ser intercambiados para llevar a cabo comportamientos específicos.

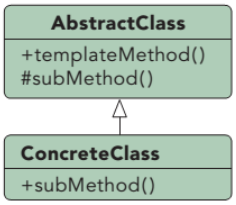
Usar cuando

- La única diferencia entre muchas clases relacionadas es su comportamiento.
- Se requieren múltiples versiones o variaciones de un algoritmo.
- Los algoritmos acceden o utilizan datos a los que el código de llamada no debe estar expuesto.
- Las instrucciones condicionales son complejas y difíciles de mantener.

Ejemplo

Cuando se importan datos a un sistema nuevo, se deben ejecutar diferentes validaciones. Configurando la importación para utilizar estrategias, la lógica condicional que determina que validación ejecutar puede ser removida y la importación puede ser desacoplada del código de validación. Esto nos permite llamar dinámicamente una o más estrategias durante la importación.

TEMPLATE METHOD — clases



Propósito: Identifica el marco de trabajo de un algoritmo, permitiendo implementar clases para definir el comportamiento.

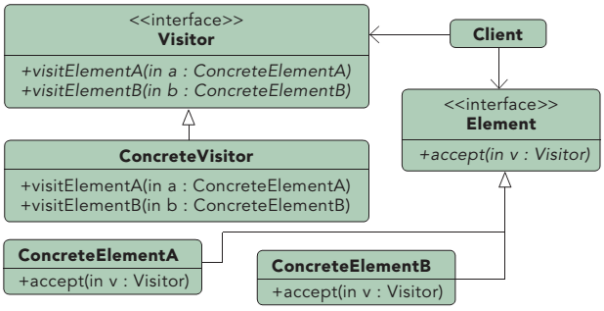
Usar cuando

- Se necesita una solo implementación abstracta de un algoritmo.
- El comportamiento común entre subclases se debe situar en una clase común.
- Las clases padre deben poder invocar uniformemente el comportamiento es sus subclases.
- La mayoría de las subclases debe implementar el comportamiento.

Ejemplo

Una clase padre, `InstantMessage`, tendrá, probablemente, todos los métodos requeridos para enviar un mensaje. Sin embargo, la serialización de los datos a enviar puede variar, dependiendo de la implementación. Un mensaje de video y un mensaje de texto plano requerirán diferentes algoritmos para serializar los datos correctamente. Las subclases de `InstantMessage` pueden proveer su propia implementación del método de serialización, permitiendo que la clase padre trabaje con ellos sin entender sus detalles de implementación.

VISITOR — objetos



Propósito: Permite que se apliquen una o más operaciones a un conjunto de objetos en tiempo de ejecución, desacoplando las operaciones de la estructura de objetos.

Usar cuando

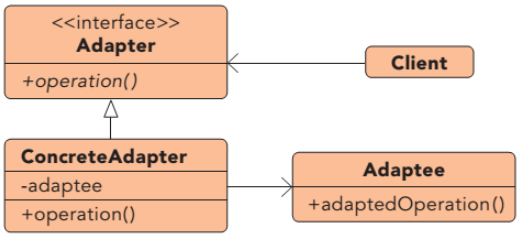
- Se deban llevar a cabo muchas operaciones no relacionadas en una estructura de objetos.
- La estructura de objetos no puede cambiar pero las operaciones llevadas a cabo sobre ella si pueden.
- Es aceptable exponer el estado interno o las operaciones de la estructura de objetos.
- Las operaciones se pueden llevar a cabo en múltiples estructuras de objetos que implementan el mismo conjunto de interfaces.

Ejemplo

Calcular impuestos en diferentes regiones sobre conjuntos de facturas requiere muchas variaciones diferentes de la lógica de cálculo. Implementar el patrón *visitor* permite que la lógica sea desacoplada de las facturas y los ítems. Esto permite que la jerarquía de ítems sea visitada por código de cálculo que

puede aplicar las tasas apropiadas para la región. Cambiar de región es tan simple como sustituir el *visitor*.

ADAPTER — clases y objetos



Propósito: Permite que las clases con interfaces dispares trabajen juntas por medio de la creación de un objeto común que les permite comunicarse e interactuar.

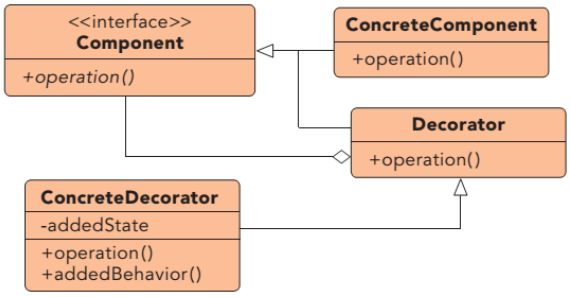
Usar cuando

- Una clase que va a ser usada no cumple los requisitos de la interfaz.

Ejemplo

Una aplicación de facturación requiere una interfaz con una aplicación de recursos humanos para intercambiar información de los empleados, sin embargo cada uno tiene su propia interfaz e implementación del objeto `Empleado`. Además, el número de identificación es almacenado en formatos diferentes en cada sistema. Creando un adaptador podemos crear una interfaz común entre las dos aplicaciones que les permita comunicarse usando sus objetos nativos y pueda transformar el formato del número de identificación en el proceso.

DECORATOR — objetos



Propósito: Permite la envoltura dinámica de objetos con el fin de modificar sus responsabilidades y comportamientos existentes.

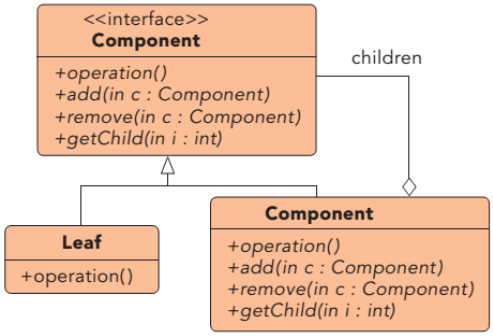
Usar cuando

- Las responsabilidades y comportamientos de un objeto deben ser modificables dinámicamente.
- Las implementaciones concretas deben ser desacopladas de las responsabilidades y comportamientos.
- Crear subclases para lograr la modificación es impráctico o imposible.
- Funcionalidad específica no deba residir en la parte alta de una jerarquía de objetos.
- Sea aceptable tener un montón de pequeños objetos rodeando una implementación concreta.

Ejemplo

Muchos negocios configuran sus sistemas de correo para aprovechar los decoradores. Cuando los mensajes son enviados de alguien en la compañía a una dirección externa, el servidor de correo decora el mensaje original con información de derechos de autor y confidencialidad; pero si el mensaje es interno, la información no es adjuntada. Esta decoración permite que el mensaje permanezca intacto hasta que una decisión en tiempo de ejecución se toma para envolver el mensaje con información adicional.

COMPOSITE — objetos



Propósito: Facilita la creación de jerarquías de objetos donde cada objeto puede ser tratado independientemente o como un conjunto de objeto anidados, a través de la misma interfaz.

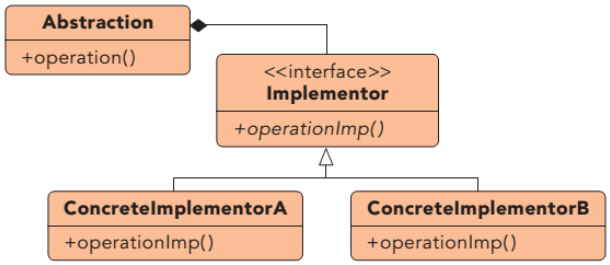
Usar cuando

- Se necesita representación jerárquica de objetos.
- Los objetos y las composiciones de objetos deben ser tratadas uniformemente.

Ejemplo

Algunas veces la información mostrada en un carrito de compras es el producto de un solo ítem, mientras que otras veces es una agregación de múltiples ítems. Implementando ítems como *composites* podemos tratar los agregados y los ítems de la misma manera, permitiéndonos iterar el árbol e invocar funcionalidad sobre cada ítem. Llamando el método `getCost()` en cualquier nodo, obtendríamos el costo de ese ítem más el costo de los ítems hijos, logrando que los ítems sean tratados uniformemente, ya sean ítems solos o agrupados.

BRIDGE — objetos



Propósito: Define una estructura abstracta de objetos independientemente de la implementación para limitar el acoplamiento.

Usar cuando

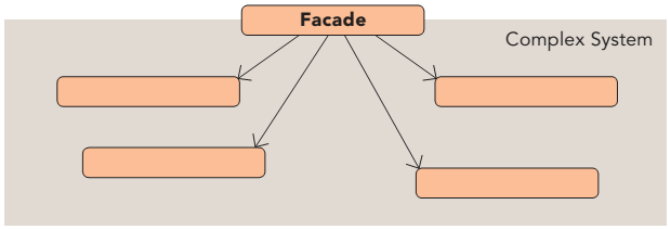
- Las abstracciones y las implementaciones no deban estar enlazadas en tiempo de compilación.
- Las abstracciones y las implementaciones se deban extender independientemente.
- Los cambios en la implementación de una abstracción no deben tener impacto en los clientes.
- Los detalles de implementación deben estar escondidos del cliente.

Ejemplo

La Máquina Virtual de Java (JVM) tiene su propio conjunto nativo de funciones que abstraen el uso de ventanas, sistemas de registro y ejecución de código byte, pero la implementación real de estas funciones se delega al sistema operativo sobre el que se ejecuta la JVM. Cuando una aplicación instruye la JVM para que pinte una ventana, ésta delega la operación a la

implementación concreta de la JVM que sabe cómo comunicarse con el sistema operativo para pintar la ventana.

FACADE objetos



Propósito: Provee una sola interfaz para un conjunto de interfaces dentro de un sistema.

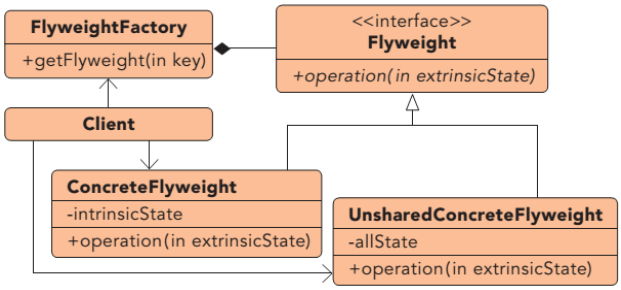
Usar cuando

- Se necesite una interfaz simple para proveer acceso a un sistema complejo.
- Existen muchas dependencias entre las implementaciones del sistema y los clientes.
- Los sistemas y subsistemas deben disponerse en capas.

Ejemplo

Exponiendo un conjunto de funcionalidades a través de un servicio web, el código del cliente solo se debe preocupar por la interfaz sencilla que está siendo expuesta y no por las relaciones complejas que pueden o no existir detrás de la capa del servicio web. Una sola llamada al servicio web para actualizar un sistema con nuevos datos puede involucrar comunicación con varias bases de datos y sistemas, sin embargo este detalle está oculto debido a la implementación del patrón *facade*.

FLYWEIGHT objetos



Propósito: Facilita la reutilización de muchos objetos de grano fino, haciendo la utilización de grandes números de objetos más eficiente.

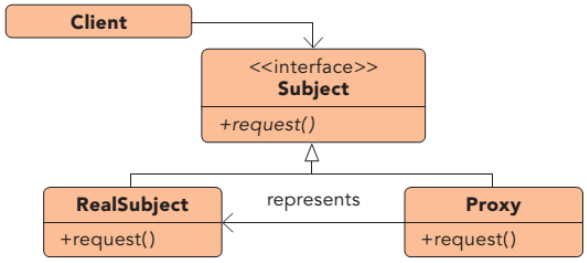
Usar cuando

- El costo de usar y almacenar muchos objetos similares es alto.
- La mayoría de estados de cada objeto se puede volver extrínseca.
- Unos pocos objetos compartidos pueden reemplazar muchos que no lo son.
- La identidad de cada objeto no importa.

Ejemplo

Los sistemas que permiten a los usuarios definir sus propios flujos de aplicación y distribuciones tienen, frecuentemente, una necesidad de llevar registro de grandes números de campos, páginas y otros ítems que son casi idénticos entre sí. Haciendo que estos ítems sean *flyweights*, todas las instancias de cada objeto pueden compartir el estado intrínseco, mientras mantienen el estado extrínseco separado. El estado intrínseco almacena las propiedades compartidas, tales como el aspecto de los campos de texto, la cantidad de datos que puede soportar y los eventos que expone. El estado extrínseco almacena las propiedades no compartidas, tales como a dónde pertenece el ítem, cómo reacciona a un clic del usuario y como maneja los eventos.

PROXY objetos



Propósito: Permite controlar el acceso a nivel de objetos, actuando como un entidad de paso o un objeto marcador de posición.

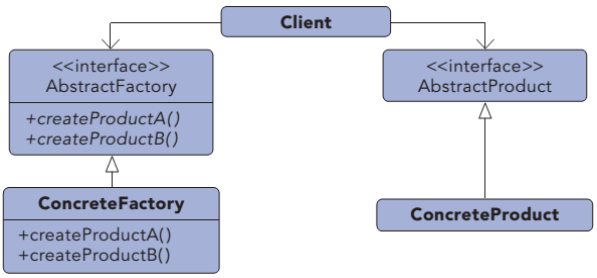
Usar cuando

- El objeto que está siendo representado es externo al sistema.
- Los objetos necesitan ser creados bajo demanda.
- Se requiere control de acceso para el objeto original.
- Se requiere adicionar funcionalidad cuando un objeto es accedido.

Ejemplo

Las aplicaciones de contabilidad proveen, frecuentemente, una forma para que los usuarios concilien sus estados bancarios con los datos de su libro de contabilidad bajo demanda, automatizando la mayoría del proceso. La comunicación con un tercero es una operación relativamente costosa que debe ser limitada. Usando un proxy para representar el objeto de comunicación, podemos limitar el número de veces o el intervalo en el que la comunicación es invocada. Adicionalmente, podemos envolver la compleja instanciación del objeto de comunicación dentro de la clase proxy, desacoplando el método de invocación de los detalles de implementación.

ABSTRACT FACTORY objetos



Propósito: Provee una interfaz que delega las llamadas de creación a una o más clases concretas con el fin de entregar objetos específicos.

Usar cuando

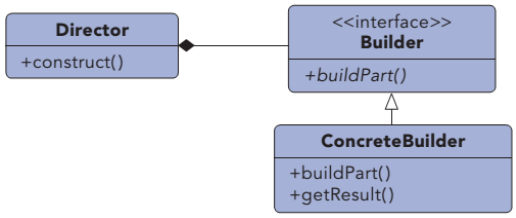
- La creación de los objetos debe ser independiente del sistema que los utiliza.
- Los sistemas deben ser capaces de usar múltiples familias de objetos.
- Familias de objetos deben ser usadas juntas.
- Se deben publicar librerías sin exponer los detalles de implementación.
- Las clases concretas deben ser desacopladas de los clientes.

Ejemplo

Los editores de correo permiten editar en múltiples formatos incluyendo texto plano, texto enriquecido y HTML. Dependiendo del formato usado, se necesitará crear diferentes

objetos. Si el mensaje es texto plano, entonces podría haber un objeto `Body` que representara solo el texto plano y un objeto `Attachment` que simplemente encriptara el adjunto en Base64. Si el mensaje es HTML, entonces el objeto `Body` representaría texto codificado en HTML y el objeto `Attachment` permitiría representaciones en línea y adjuntos estándar. Utilizando un *abstract factory* para la creación, podemos asegurar que se creen los conjuntos de objetos apropiados basados en el estilo de correo que se va a enviar.

BUILDER — objetos



Propósito: Permite la creación dinámica de objetos con base en algoritmos fácilmente intercambiables.

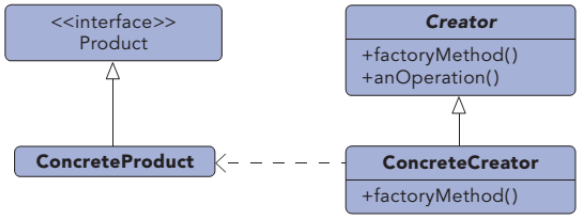
Usar cuando

- Los algoritmos de creación de objetos deben ser desacoplados del sistema.
- Se requieren múltiples representaciones de algoritmos de creación.
- Se necesita adicionar funcionalidad de creación nueva sin cambiar el código esencial.
- Se requiere control en tiempo de ejecución sobre el proceso de creación.

Ejemplo

Una aplicación de transferencia de archivos puede usar muchos protocolos diferentes para enviar archivos y el objeto de transferencia que será creado dependerá directamente del protocolo escogido. Usando el patrón *builder* podemos determinar el *builder* correcto para instanciar el objeto correcto. Si la configuración es FTP, entonces el *builder* FTP se usará al crear los objetos.

FACTORY METHOD — objetos



Propósito: Expone un método para crear objetos, permitiendo que las subclases controlen el proceso de creación.

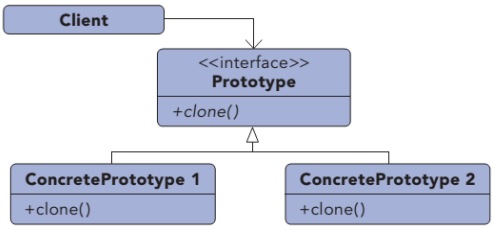
Usar cuando

- Una clase no sabrá qué clases se le solicitará crear.
- Las subclases deben especificar que objetos deben ser creados.
- Las clases padres deseen delegar la creación a las subclases.

Ejemplo

Muchas aplicaciones tienen algún tipo de estructura de usuarios y grupos para seguridad. Cuando la aplicación necesita crear un usuario, típicamente delega la creación a múltiples implementaciones de usuario. El objeto usuario padre maneja la mayoría de operación para cada usuario, pero las subclases definen el *factory method* que maneja las distinciones en la creación de cada tipo de usuario. Un sistema puede tener dos objetos, `AdminUser` y `StandardUser`, cada uno de los cuales extiende el objeto `User`. El objeto `AdminUser` puede llevar a cabo algunas tareas extra para conceder el acceso, mientras que el objeto `StandardUser` puede hacer lo mismo para limitar el acceso.

PROTOTYPE — objetos



Propósito: Crea objetos a través de la clonación, con base en una plantilla de un objeto.

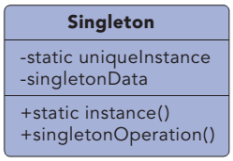
Usar cuando

- La composición, creación y representación de los objetos debe ser desacoplada de un sistema.
- Las clases que se van a crear se especifican en tiempo de ejecución.
- Existe un número limitado de combinaciones de estado en un objeto.
- Se requieren objetos o estructuras de objetos que son idénticas o se parecen mucho a otros objetos o estructuras de objetos.
- La creación inicial de cada objeto es una operación costosa.

Ejemplo

Los motores de procesamiento de precios frecuentemente requieren buscar valores de configuración diferentes, haciendo que la inicialización del motor sea un proceso relativamente costoso. Cuando se requieren múltiples instancias del motor, por ejemplo para importar datos de forma concurrente, el costo de inicializar muchos motores es alto. Utilizando el patrón *prototype* podemos asegurar que solo una copia del motor tenga que ser inicializada, luego simplemente clonamos el motor para crear un duplicado del objeto inicializado. El beneficio adicional de esto es que los clones se pueden optimizar para que solo incluyan información relevante para su situación.

SINGLETON — objetos



Propósito: Asegura que solo se permita una instancia de una clase dentro de un sistema.

Usar cuando

- Se requiere exactamente una instancia de una clase.
- Se requiere acceso controlado a un solo objeto.

Ejemplo

Muchos lenguajes proveen algún tipo de objeto `System` que permite la interacción con el sistema operativo nativo. Dado que la aplicación se está ejecutando en un solo sistema operativo, solo existe la necesidad de tener una sola instancia de ese objeto `System`. El patrón *singleton* es implementado por el motor de ejecución del lenguaje para asegurar que solo una copia del objeto `System` sea creada y que solo los procesos apropiados lo puedan acceder.