Patrones de Diseño

de Software

Programación Orientada a Objetos

Ing. Juan Andrés García

Patrones de Diseño de Software

- Los patrones de diseño (design patterns) son soluciones habituales a problemas comunes en el diseño de software. El patrón es una descripción del problema y la esencia de su solución, de modo que la solución puede reutilizarse en diferentes tipos de desarrollos.

Interfaz: se refieren a un medio común para que los componentes de software no relacionados se comuniquen entre sí.

Clasificación de los Patrones de Diseño

- Los patrones de diseño de software son soluciones generales a problemas comunes que surgen al diseñar software. Se dividen en tres categorías principales: Creacionales, Estructurales y de Comportamiento.

Patrones Creacionales

- Los patrones creacionales se centran en la creación de objetos de manera flexible y eficiente.
- Estos patrones proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.

Tipos de patrones creacionales

- Factory Method
- Abstract Factory
- Builder
- Singleton
- Prototype

Patrón de Creacional: Factory Method

- Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Características:

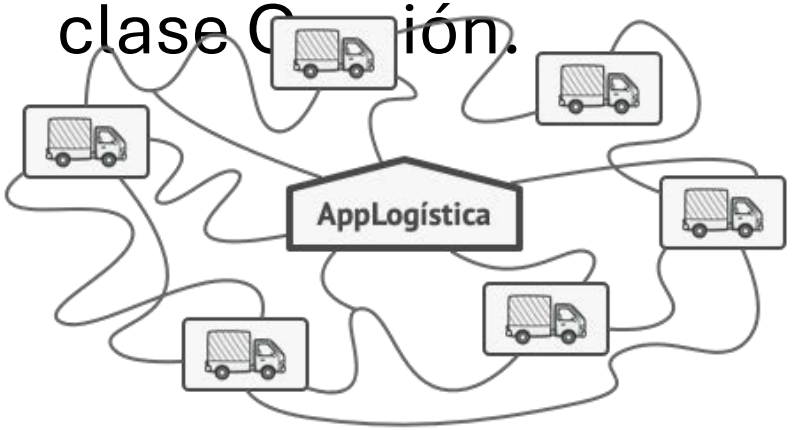
- Define una interfaz para crear objetos, pero permite a las subclases alterar el tipo de objetos que se crearán.
- Encapsula la lógica de instanciación en subclases, proporcionando flexibilidad en la creación de objetos.
- Permite la extensión del código sin modificar el código existente, siguiendo el principio de apertura y cierre.

Cómo implementarlo:

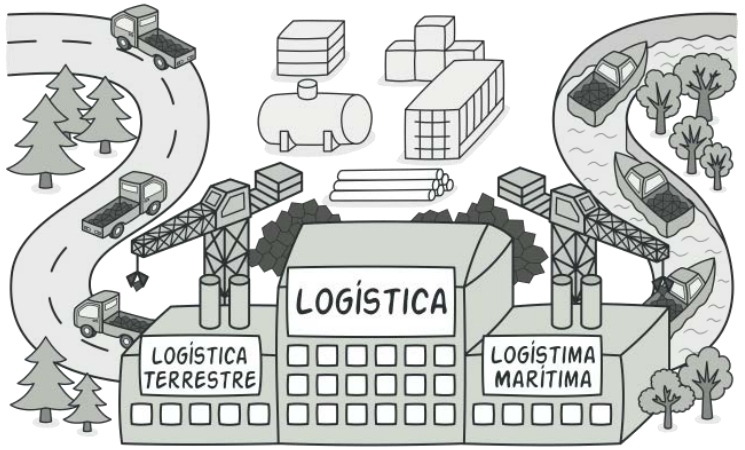
- Crear una interfaz o clase base que declare un método para crear objetos. Las subclases implementarán este método para crear objetos específicos.

Ejemplo Factory Method

- Se desea una aplicación de gestión logística. La primera versión de la aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra enfocada de la clase C[illegible]ión.

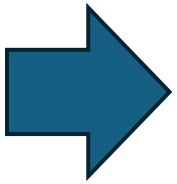


- Al cabo de un tiempo, la aplicación se vuelve bastante popular. Cada día se recibe decenas de peticiones de empresas de transporte marítimo para incorporar la logística por mar a la aplicación.



¿Qué pasa con el código?

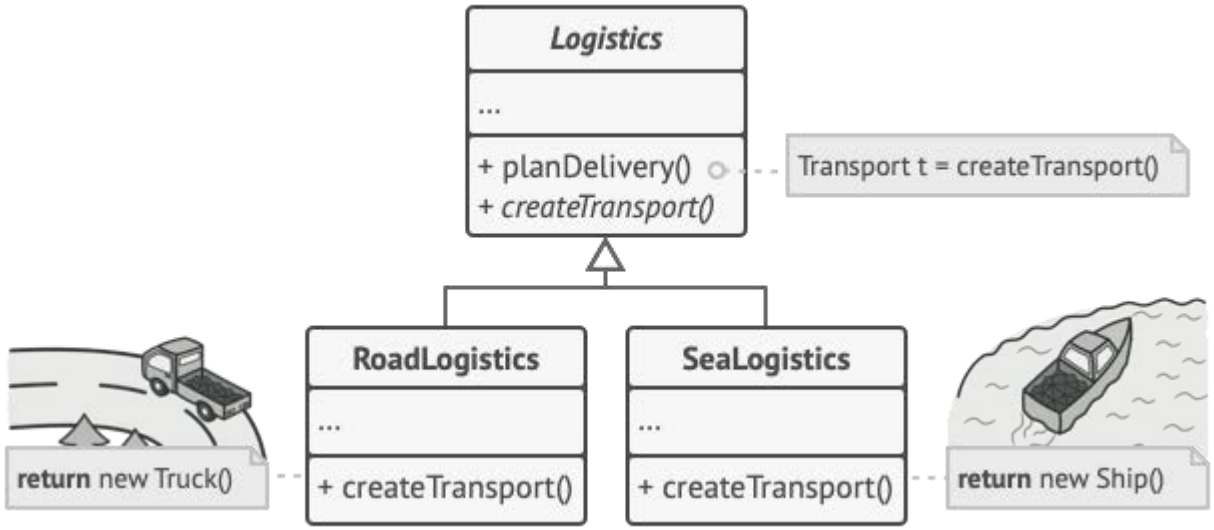
- En este momento, la mayor parte de tu código está acoplado a la clase Camión. Para añadir barcos a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde se decide añadir otro tipo de transporte a la aplicación, probablemente se tendría que volver a hacer todos estos cambios.



consecuencia

Al final se acabaría con un código bastante sucio, plagado de condicionales que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos de transporte.

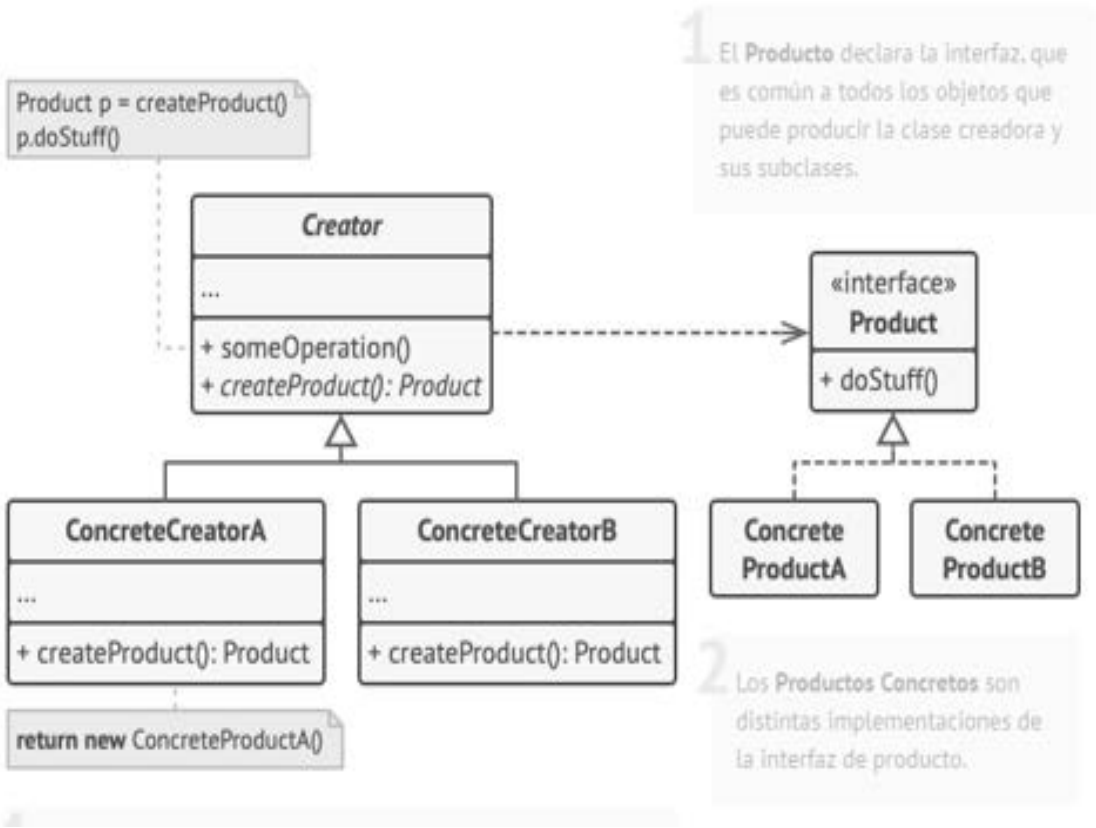
Solución



- El patrón Factory Method sugiere que, en lugar de llamar al operador new para construir objetos directamente, se invoque a un método fábrica especial.
- Aclaración: los objetos se siguen creando a través del operador new, pero se invocan desde el método fábrica. Los objetos devueltos por el método fábrica a menudo se denominan productos.
- El código que utiliza el método fábrica no encuentra diferencias entre los productos devueltos por varias subclases, y trata a todos los productos como la clase abstracta Transporte. El cliente sabe que todos los objetos de transporte deben tener el método entrega, pero no necesita saber cómo funciona exactamente.

Estructura

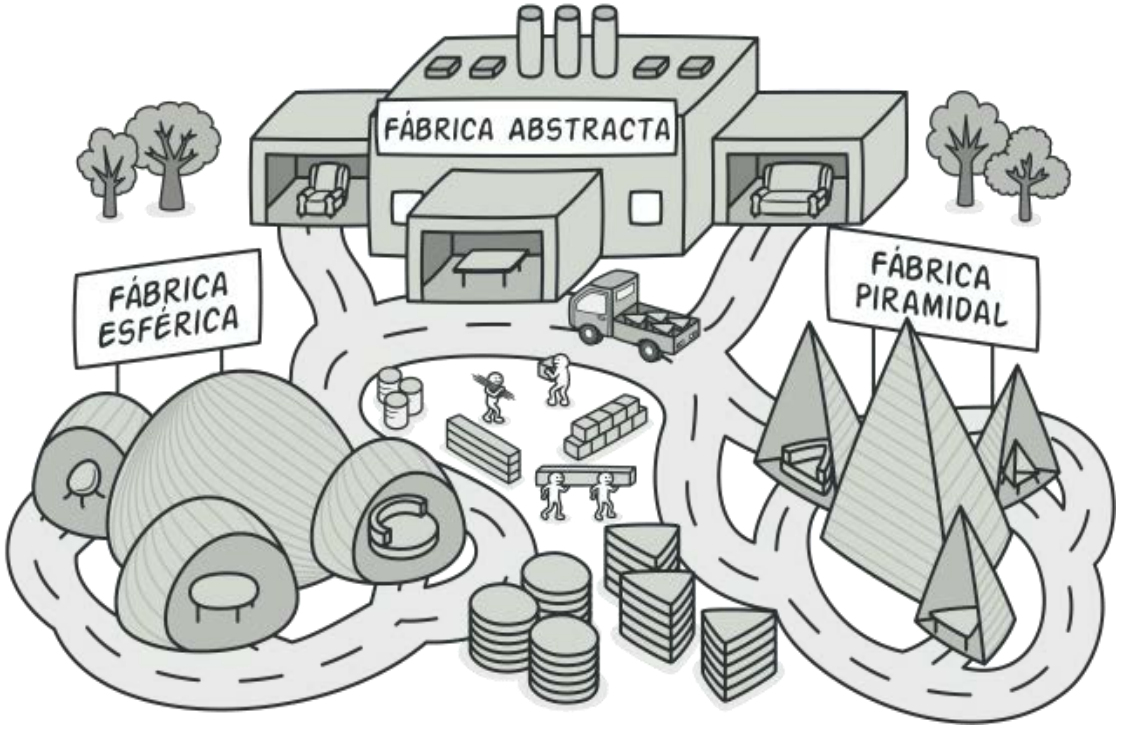
- La clase Creadora declara el método fábrica que devuelve nuevos objetos de producto. Es importante que el tipo de retorno de este método coincida con la interfaz de producto.
- Se puede declarar el patrón Factory Method como abstracto para forzar a todas las subclases a implementar sus propias versiones del método. Como alternativa, el método fábrica base puede devolver algún tipo de producto por defecto.
- La clase de creación de producto no es la principal responsabilidad de la clase creadora. Normalmente, esta clase cuenta con alguna lógica de negocios central relacionada con los productos. El patrón Factory Method ayuda a desacoplar esta lógica de las clases concretas de producto.



Patrón Creacional: Abstract Factory

- Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

- Características:
  - Proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.
  - Permite la creación de objetos interrelacionados sin acoplar el código al conjunto de clases concretas.
  - Garantiza que los productos creados por una fábrica sean compatibles entre sí.



Patrón Abstract Factory: Ejemplo

- Se necesita crear un simulador de tienda de muebles. El código está compuesto por clases que representan lo siguiente:
  - Una familia de productos relacionados, digamos: Silla + Sofá + Mesilla.
  - Algunas variantes de esta familia. Por ejemplo, los productos Silla + Sofá + Mesilla están disponibles en estas variantes: Moderna, Victoriana, ArtDecó.

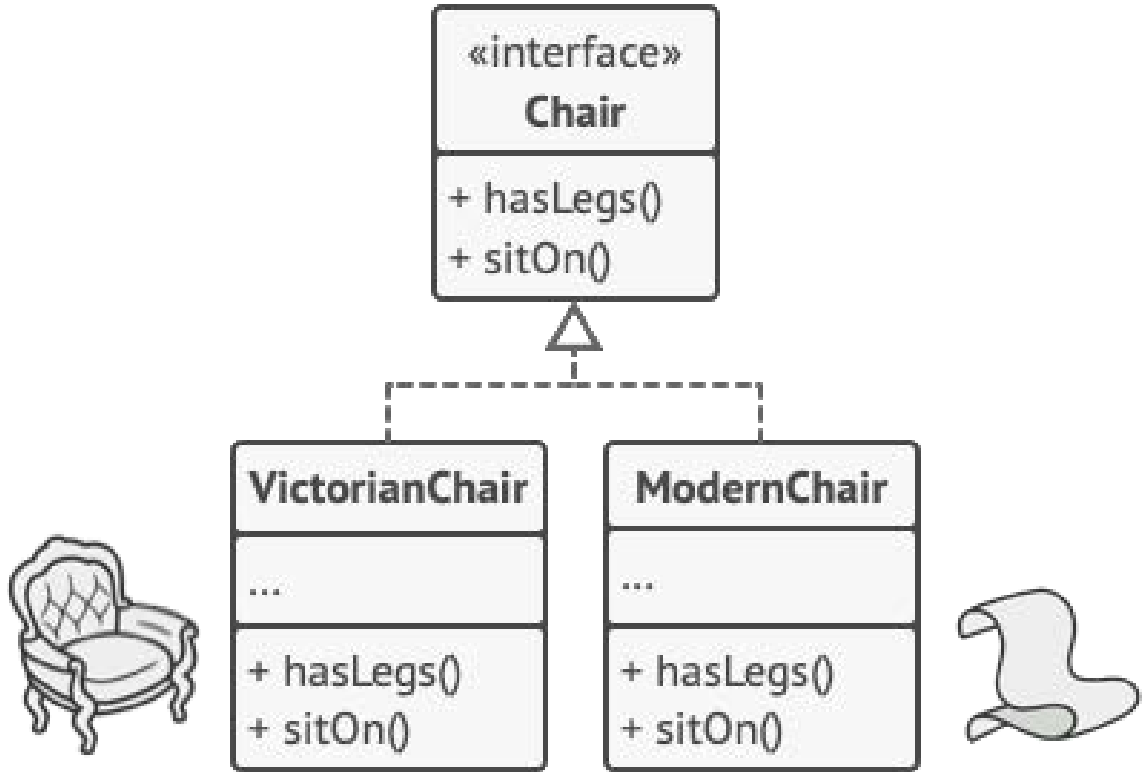
Familias de productos y sus variantes.

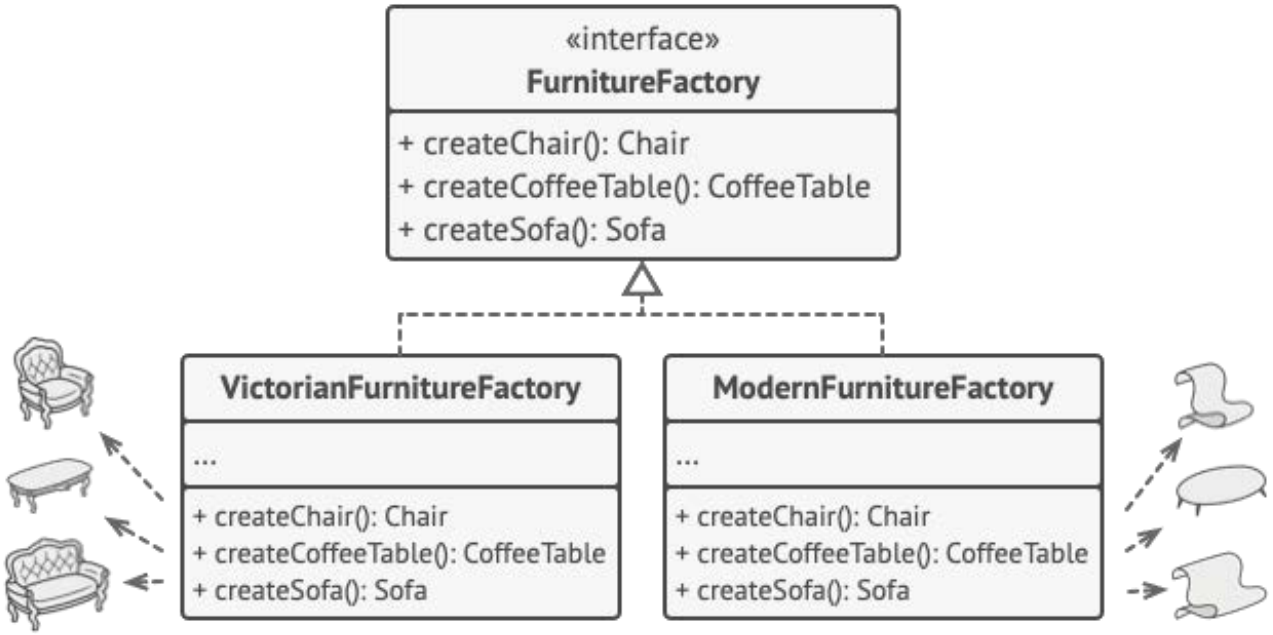
Patrón Abstract Factory: Problema

- Se necesita implementar una forma de crear objetos individuales de mobiliario para que combinen con otros objetos de la misma familia.
- Además, no queremos cambiar el código existente al añadir al programa nuevos productos o familias de productos. Los comerciantes de muebles actualizan sus catálogos muy a menudo, y debemos evitar tener que cambiar el código principal cada vez que esto ocurra.

Abstract Factory: Solución Paso 1

- Lo primero que sugiere el patrón Abstract Factory es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces. Por ejemplo, todas las variantes de silla pueden implementar la interfaz Silla, así como todas las variantes de mesilla pueden implementar la interfaz Mesilla, y así sucesivamente.





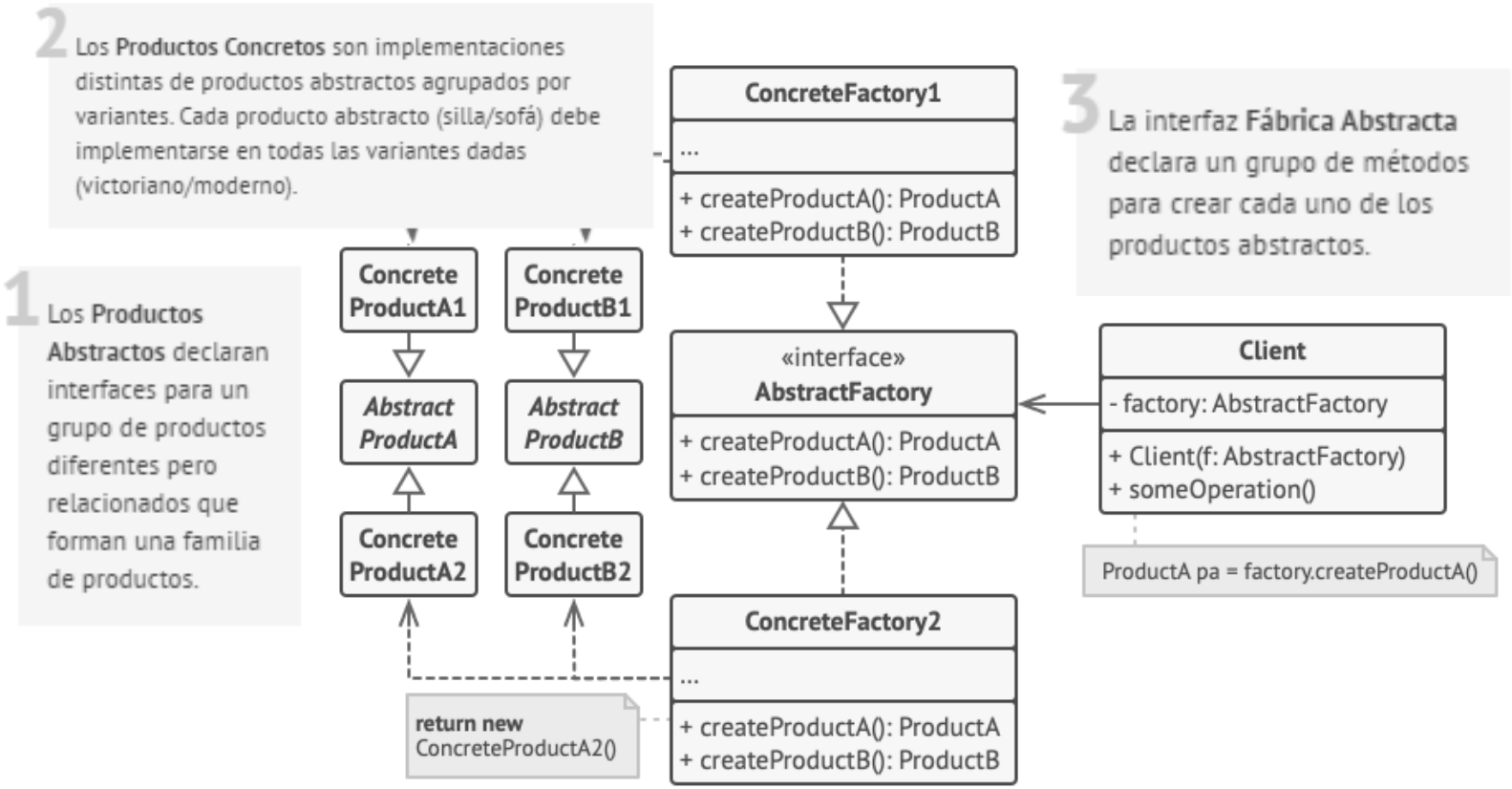
Abstract Factory: Solución Paso 2

- El paso 2 consiste en declarar la Fábrica abstracta: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos (por ejemplo, crearSilla, crearSofá y crearMesilla). Estos métodos deben devolver productos abstractos representados por las interfaces que extrajimos previamente: Silla, Sofá, Mesilla, etc.

Abstract Factory: Solución Paso 3

- Ahora bien, ¿qué hay de las variantes de los productos? Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz FábricaAbstracta. Una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la FábricadeMueblesModernos sólo puede crear objetos de SillaModerna, SofáModerno y MesillaModerna.
- El código cliente tiene que funcionar con fábricas y productos a través de sus respectivas interfaces abstractas. Esto nos permite cambiar el tipo de fábrica que pasamos al código cliente, así como la variante del producto que recibe el código cliente, sin descomponer el propio código cliente.

Estructura



4 Las **Fábricas Concretas** implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.

5 Aunque las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación deben devolver los productos *abstractos* correspondientes. De este modo, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto que obtiene de una fábrica. El **Cliente** puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.

Aplicabilidad

- Este Patrón de diseño se debe implementar cuando:
  - Un sistema debe ser independiente de cómo se crean, componen y representan sus productos.
  - Un sistema debe ser configurado con una familia de productos de entre varias.
  - Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.
  - Quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones.

¿Cómo implementarlo?

- Mapea una matriz de distintos tipos de productos frente a variantes de dichos productos.
- Declara interfaces abstractas de producto para todos los tipos de productos. Después haz que todas las clases concretas de productos implementen esas interfaces.
- Declara la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.
- Implementa un grupo de clases concretas de fábrica, una por cada variante de producto.
- Crea un código de inicialización de la fábrica en algún punto de la aplicación. Deberá instanciar una de las clases concretas de la fábrica, dependiendo de la configuración de la aplicación o del entorno actual. Pasa este objeto de fábrica a todas las clases que construyen productos.
- Explora el código y encuentra todas las llamadas directas a constructores de producto. Sustitúyelas por llamadas al método de creación adecuado dentro del objeto de fábrica.

Patrón Creacional: Builder

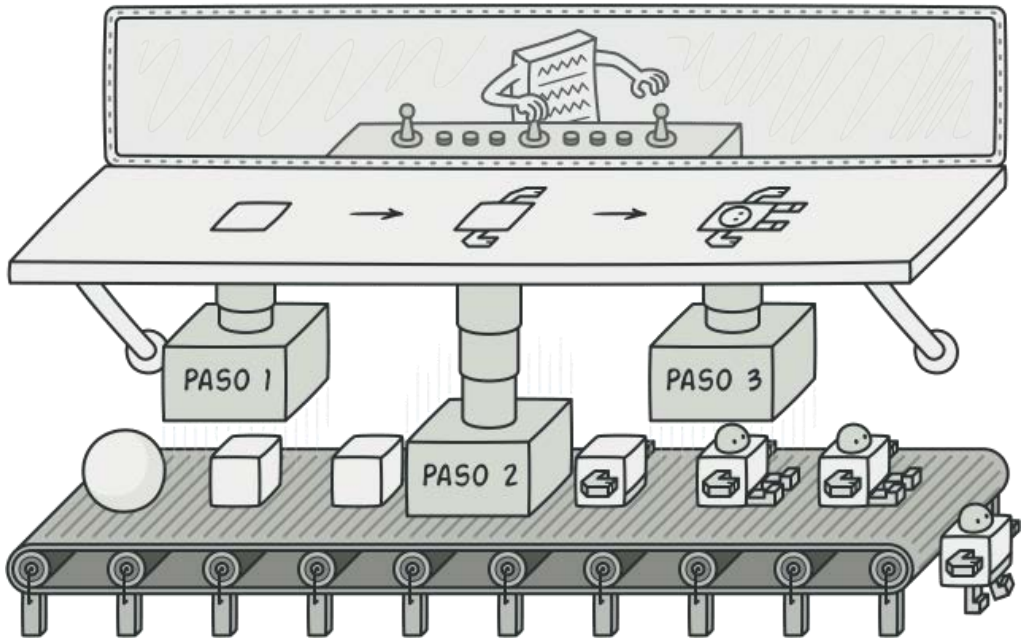
- Es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.
- El Builder separa la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear diferentes representaciones.

Características

- Permite la construcción de objetos complejos paso a paso.
- Separa la construcción de la representación, lo que permite crear diferentes representaciones del mismo objeto de construcción.
- Facilita la construcción de objetos complejos, ocultando los detalles de su creación.

Implementación

- Cómo implementarlo:
  - Definir una interfaz que especifique los pasos de construcción del objeto. Luego, implementar esta interfaz en varias clases concretas para construir diferentes representaciones del objeto.
  - Ejemplo:
    - Un constructor de pizzas que permite construir pizzas con diferentes ingredientes y tamaños. Los clientes pueden usar el mismo proceso de construcción para crear pizzas de pepperoni, vegetarianas o de cualquier otro tipo.

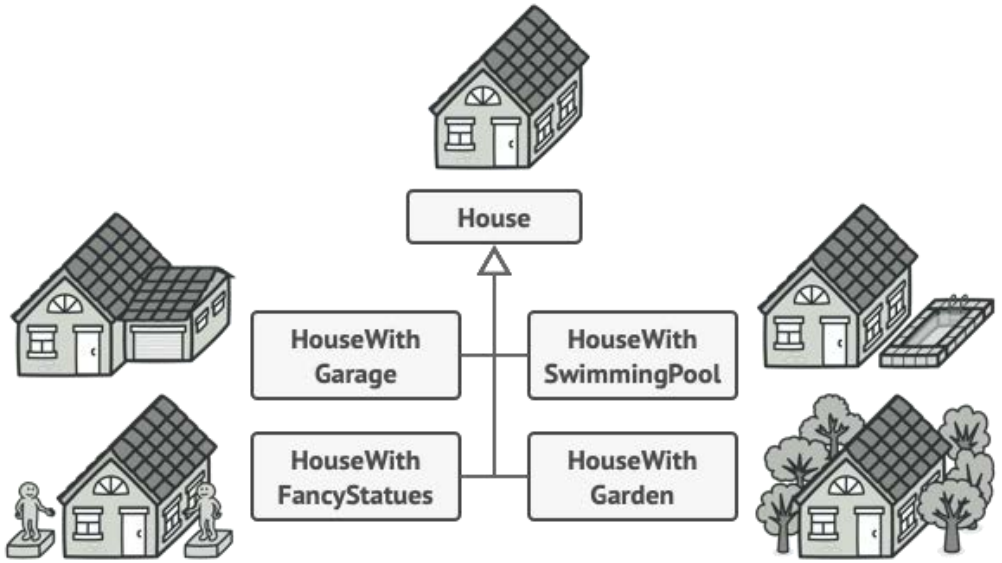


Problema

- Imagina un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados. Normalmente, este código de inicialización está sepultado dentro de un monstruoso constructor con una gran cantidad de parámetros. O, peor aún: disperso por todo el código cliente.

Ejemplo

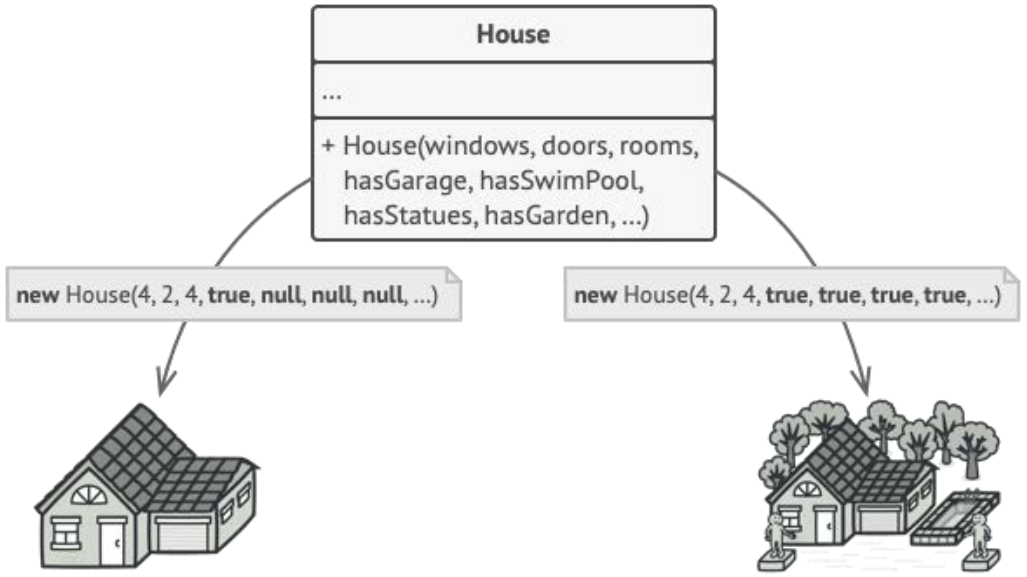
- Se necesita crear un objeto Casa. Para construir una casa sencilla, debemos construir cuatro paredes y un piso, así como instalar una puerta, colocar un par de ventanas y ponerle un tejado. Pero ¿qué pasa si se quiere una casa más grande y luminosa, con un jardín y otros extras (como sistema de calefacción, instalación de fontanería y cableado eléctrico)?



Análisis

- La solución más sencilla es extender la clase base Casa y crear un grupo de subclases que cubran todas las combinaciones posibles de los parámetros. Pero, en cualquier caso, acabarás con una cantidad considerable de subclases. Cualquier parámetro nuevo, como el estilo del porche, exigirá que incrementes esta jerarquía aún más.

- Existe otra posibilidad que no implica generar subclases. Puedes crear un enorme constructor dentro de la clase base Casa con todos los parámetros posibles para controlar el objeto casa. Aunque es cierto que esta solución elimina la necesidad de las subclases, genera otro problema.



Solución

- El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores.
- El patrón organiza la construcción de objetos en una serie de pasos (construirParedes, construirPuerta, etc.). Para crear un objeto, se ejecuta una serie de estos pasos en un objeto constructor. Lo importante es que no se necesita invocar todos los pasos. Se puede invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.

  - Puede ser que algunos pasos de la construcción necesiten una implementación diferente cuando se tenga que construir distintas representaciones del producto. Por ejemplo, las paredes de una cabaña pueden ser de madera, pero las paredes de un castillo tienen que ser de piedra.

  - En este caso, se puede crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente. Entonces se puede utilizar estos constructores en el proceso de construcción (por ejemplo, una serie ordenada de llamadas a los pasos de construcción) para producir distintos tipos de objetos.

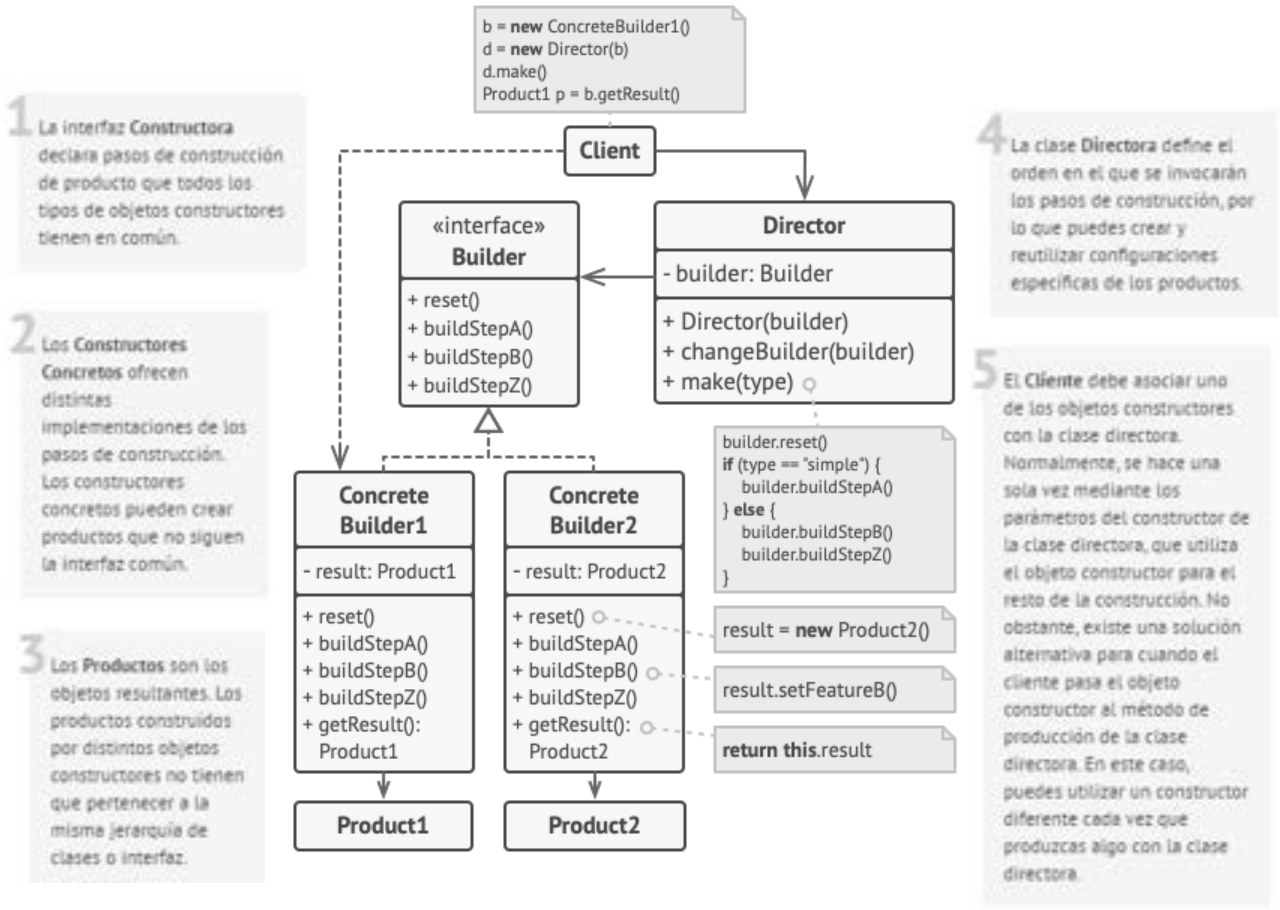
Clase directora

- Se puede ir más lejos y extraer una serie de llamadas a los pasos del constructor que utilizas para construir un producto y ponerlas en una clase independiente llamada directora. La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos.

No es estrictamente necesario tener una clase directora en el programa, ya que se pueden invocar los pasos de construcción en un orden específico directamente desde el código cliente. No obstante, la clase directora puede ser un buen lugar donde colocar distintas rutinas de construcción para poder reutilizarlas a lo largo del programa.

Además, la clase directora esconde por completo los detalles de la construcción del producto al código cliente. El cliente sólo necesita asociar un objeto constructor con una clase directora, utilizarla para iniciar la construcción, y obtener el resultado del objeto constructor.

Estructura


b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
Client
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
Concrete
Builder1
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
Concrete
Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
Product1
Product2
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
result = new Product2()
result.setFeatureB()
return this.result
1 La interfaz Constructora declara pasos de construcción de producto que todos los tipos de objetos constructores tienen en común.
2 Los Constructores Concretos ofrecen distintas implementaciones de los pasos de construcción. Los constructores concretos pueden crear productos que no siguen la interfaz común.
3 Los Productos son los objetos resultantes. Los productos construidos por distintos objetos constructores no tienen que pertenecer a la misma jerarquía de clases o interfaz.
4 La clase Directora define el orden en el que se invocarán los pasos de construcción, por lo que puedes crear y reutilizar configuraciones específicas de los productos.
5 El Cliente debe asociar uno de los objetos constructores con la clase directora. Normalmente, se hace una sola vez mediante los parámetros del constructor de la clase directora, que utiliza el objeto constructor para el resto de la construcción. No obstante, existe una solución alternativa para cuando el cliente pasa el objeto constructor al método de producción de la clase directora. En este caso, puedes utilizar un constructor diferente cada vez que produzcas algo con la clase directora.

Patrón de diseño: Singleton

- Singleton es un patrón creacional que garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a esa instancia.
- Características:
  - Garantiza que una clase tenga solo una instancia en toda la aplicación.
  - Proporciona un punto de acceso global a esa instancia única.
  - Controla el acceso a la instancia única, evitando la creación de múltiples instancias.

Problema

- El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:
  - Garantizar que una clase tenga una única instancia. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.
  - Funciona así: imagina que se ha creado un objeto y al cabo de un tiempo decides crear otro nuevo. En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.
  - Se debe tener en cuenta que este comportamiento es imposible de implementar con un constructor normal, ya que una llamada al constructor siempre debe devolver un nuevo objeto por diseño.

Proporcionar un punto de acceso global a dicha instancia.
¿Y qué sucede con esas variables globales que se utilizó para almacenar objetos esenciales? Aunque son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

Cómo implementarlo

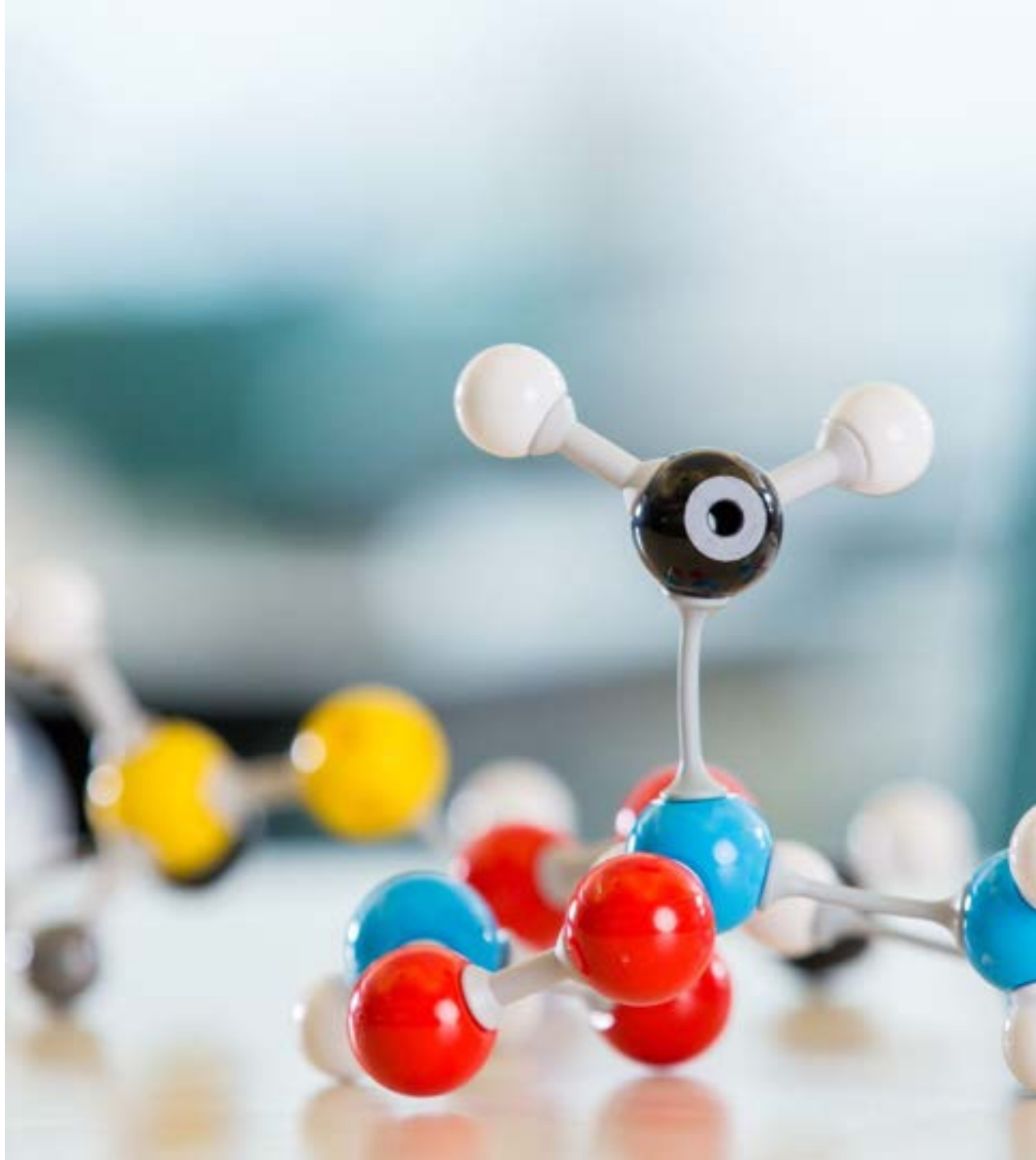
- Se debe definir un método estático en la clase que devuelva la instancia única de la clase. Dentro de este método, comprobar si ya se ha creado una instancia y devolverla si existe, o crear una nueva instancia si no.
- Ejemplo:
  - Una clase de registro Singleton que proporciona métodos para registrar eventos en una aplicación. Solo se necesita una instancia de esta clase en toda la aplicación para mantener un registro centralizado de eventos.

APLICABILIDAD

- Se debe haber exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
- La única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.
- El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.

Patrones estructurales

- Los patrones estructurales se ocupan de la composición de clases y objetos.
- Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.
- Los patrones estructurales de clases hacen uso de la herencia para componer interfaces o implementaciones. A modo de ejemplo sencillo, pensemos en cómo la herencia múltiple mezcla dos o más clases en una sola.



Patrones estructurales: Adapter

El Adapter es un patrón estructural que permite que interfaces incompatibles trabajen juntas. Convierte la interfaz de una clase en otra interfaz que un cliente espera.

Características:

- Permite que objetos con interfaces incompatibles trabajen juntos.
- Convierte la interfaz de una clase en otra interfaz que un cliente espera.
- Permite la reutilización de clases existentes sin modificar su código fuente.

Adapter: Ejemplo

- Imagina que estás creando una aplicación de monitoreo del mercado de valores. La aplicación descarga la información de bolsa desde varias fuentes en formato XML para presentarla al usuario con bonitos gráficos y diagramas.
- En cierto momento, decides mejorar la aplicación integrando una inteligente biblioteca de análisis de una tercera persona. Pero hay una trampa: la biblioteca de análisis solo funciona con datos en formato JSON.

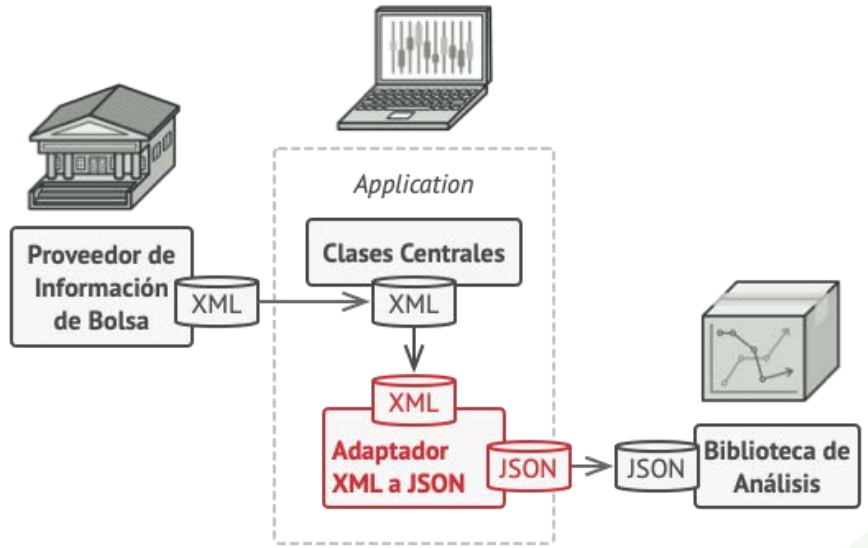
Podrías cambiar la biblioteca para que funcione con XML. Sin embargo, esto podría descomponer parte del código existente que depende de la biblioteca. Y, lo que es peor, podrías no tener siquiera acceso al código fuente de la biblioteca, lo que hace imposible esta solución.

Solución

Solución

Se puede crear un adaptador que funcione como un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión que tiene lugar tras bambalinas. El objeto envuelto ni siquiera es consciente de la existencia del adaptador. Por ejemplo, puedes envolver un objeto que opera con metros y kilómetros con un adaptador que convierte todos los datos al sistema anglosajón, es decir, pies y millas.

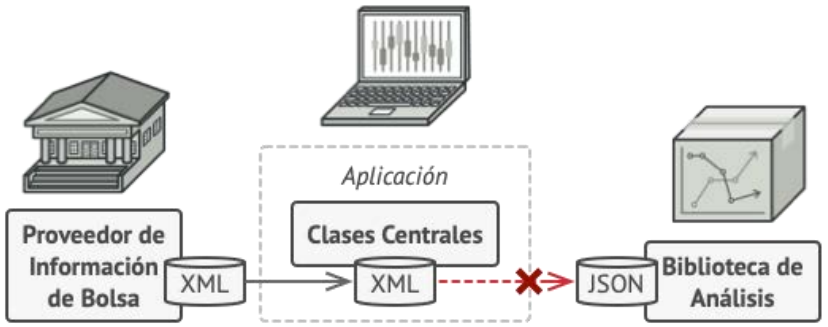


Solución

Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar. Funciona así:

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2. Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

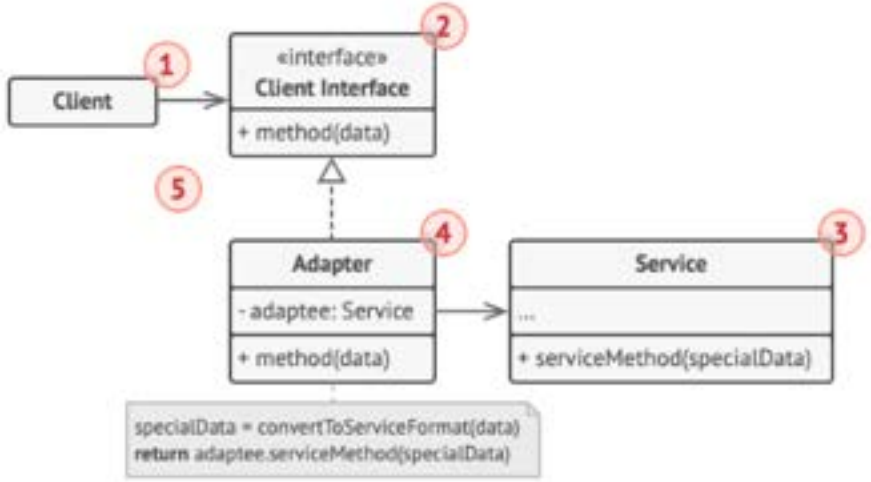
En ocasiones se puede incluso crear un adaptador de dos direcciones que pueda convertir las llamadas en ambos sentidos.



Solución

- Regresemos a la aplicación del mercado de valores. Para resolver el dilema de los formatos incompatibles, se puede crear adaptadores de XML a JSON para cada clase de la biblioteca de análisis con la que trabaje tu código directamente.
- Después ajustas tu código para que se comunique con la biblioteca únicamente a través de estos adaptadores. Cuando un adaptador recibe una llamada, traduce los datos XML entrantes a una estructura JSON y pasa la llamada a los métodos adecuados de un objeto de análisis envuelto.

Adapter: Estructura



- Un adaptador de clases usa la herencia múltiple para adaptar una interfaz a otra:

1. La clase **Cliente** contiene la lógica de negocio existente del programa.
2. La **Interfaz con el Cliente** describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente.
3. **Servicio** es alguna clase útil (normalmente de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.
4. La clase **Adaptadora** es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz de cliente y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.
5. La **Clase adaptadora** no necesita envolver objetos porque hereda comportamientos tanto de la clase cliente como de la clase de servicio. La adaptación tiene lugar dentro de los métodos sobrescritos. La clase adaptadora resultante puede utilizarse en lugar de una clase cliente existente.
6. El código cliente no se acopla a la clase adaptadora concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, puedes introducir nuevos tipos de adaptadores en el programa sin descomponer el código cliente existente. Esto puede resultar útil cuando la interfaz de la clase de servicio se cambia o sustituye, ya que puedes crear una nueva clase adaptadora sin cambiar el código cliente.

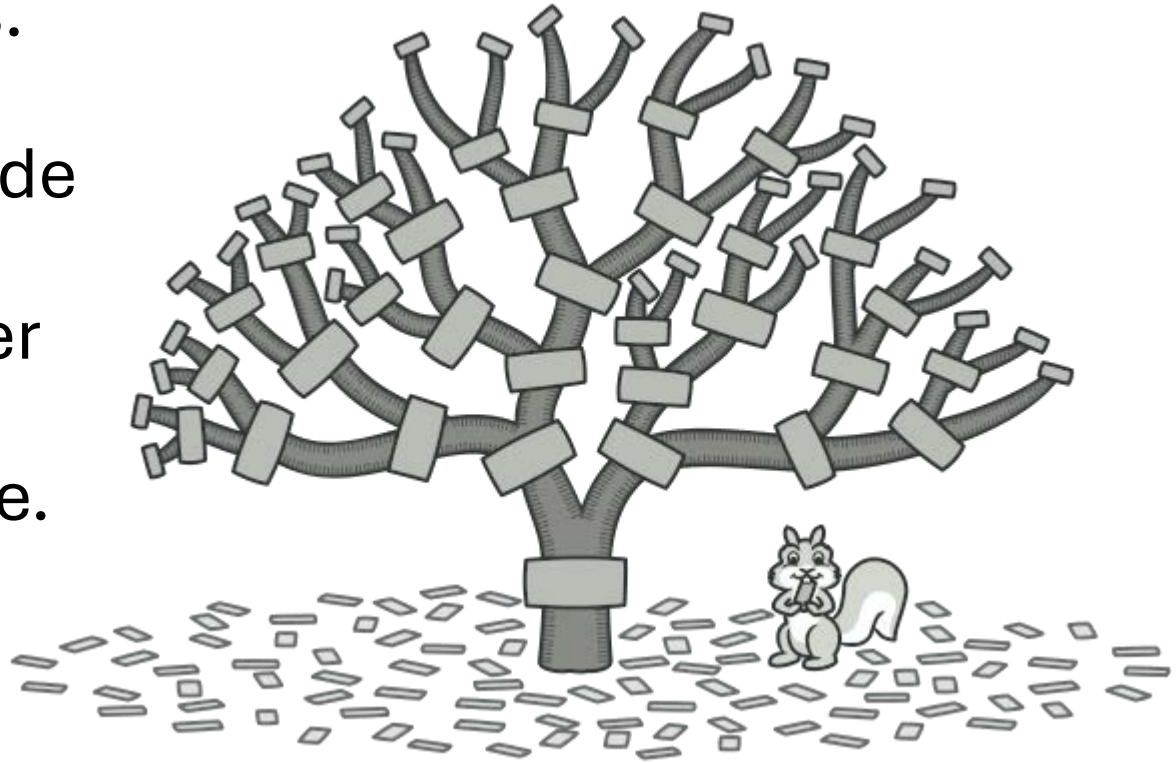
Patrón Estructural: Composite

- La implementación del patrón Composite implica la creación de una clase base común para todos los elementos (Componente), clases concretas para los elementos individuales (Hoja) y las composiciones (Compuesto), y la implementación de las operaciones comunes en la clase Compuesto.

- Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

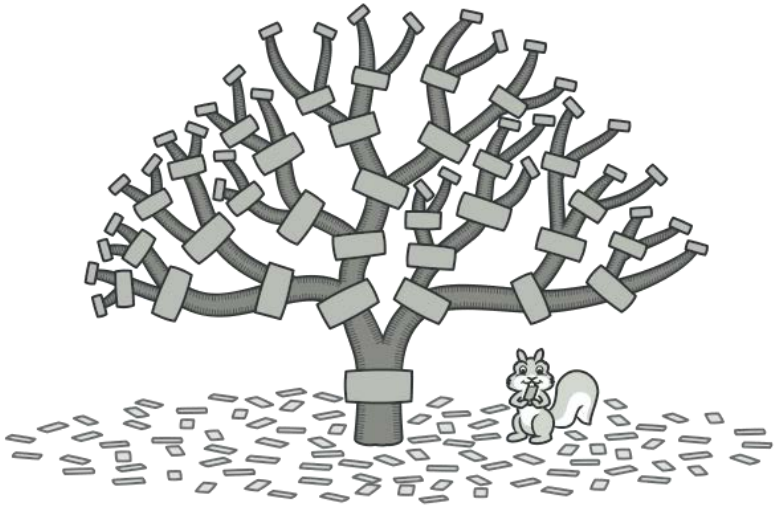
Problema

- Por ejemplo, imagina que tienes dos tipos de objetos: Productos y Cajas. Una Caja puede contener varios Productos así como cierto número de Cajas más pequeñas. Estas Cajas pequeñas también pueden contener algunos Productos o incluso Cajas más pequeñas, y así sucesivamente.



Problema

- Se decide crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos… y otras cajas. ¿Cómo se determinaría el precio total de ese pedido?



Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de Productos y Cajas a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

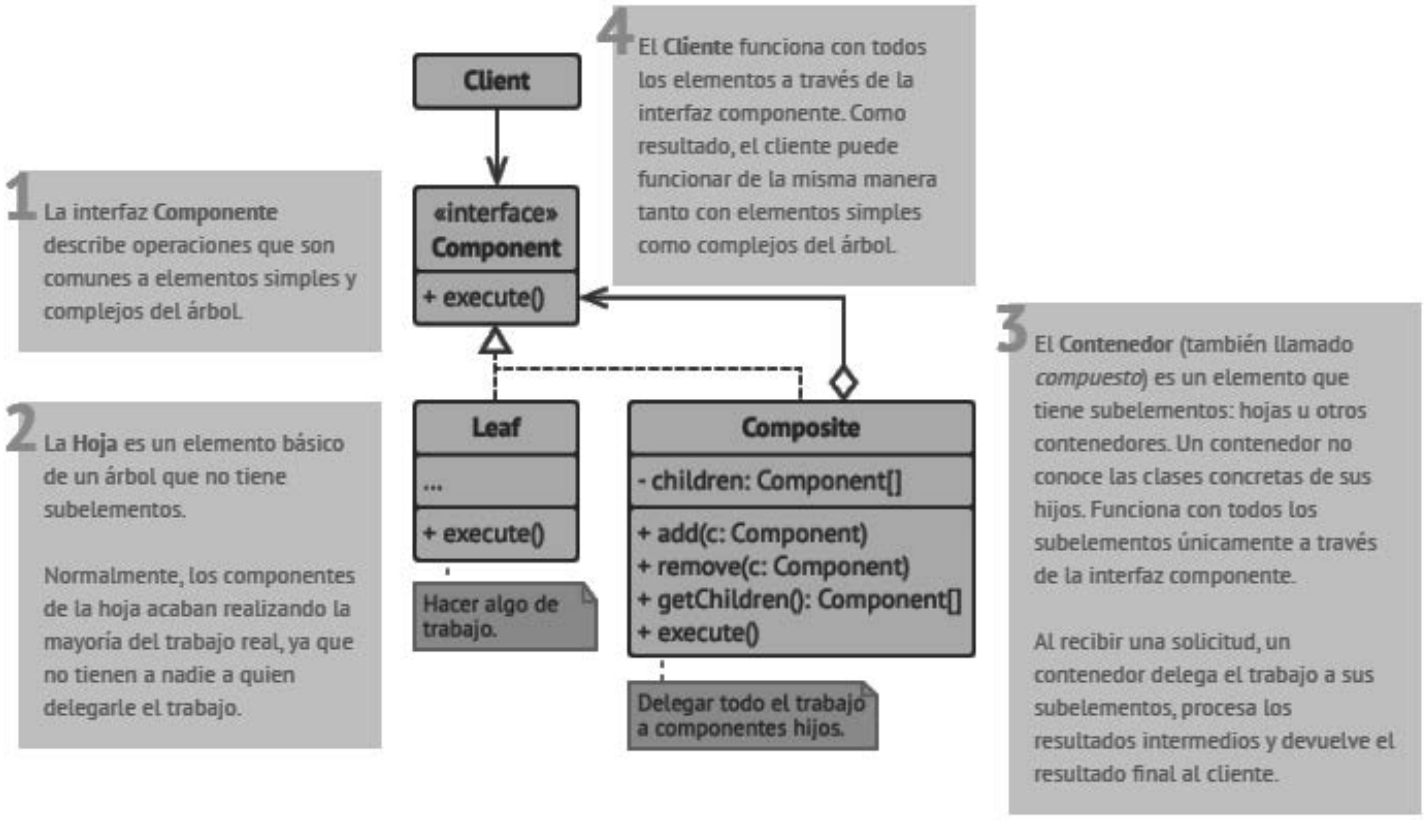
Solución

El patrón Composite sugiere que trabajes con Productos y Cajas a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método?

- Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

Solución



La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol.

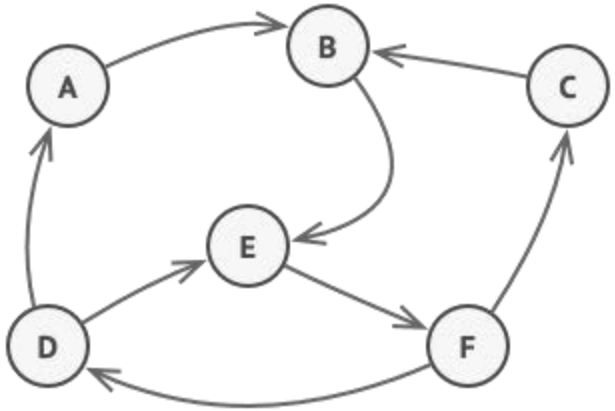
No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol

Patrones de comportamiento

- Los patrones de comportamiento tienen que ver con algoritmos y con la asignación de responsabilidades a objetos. Los patrones de comportamiento describen no sólo patrones de clases y objetos, sino también patrones de comunicación entre ellos.
- Estos patrones describen el flujo de control complejo que es difícil de seguir en tiempo de ejecución, lo que nos permite olvidarnos del flujo de control para concentrarnos simplemente en el modo en que se interconectan los objetos.

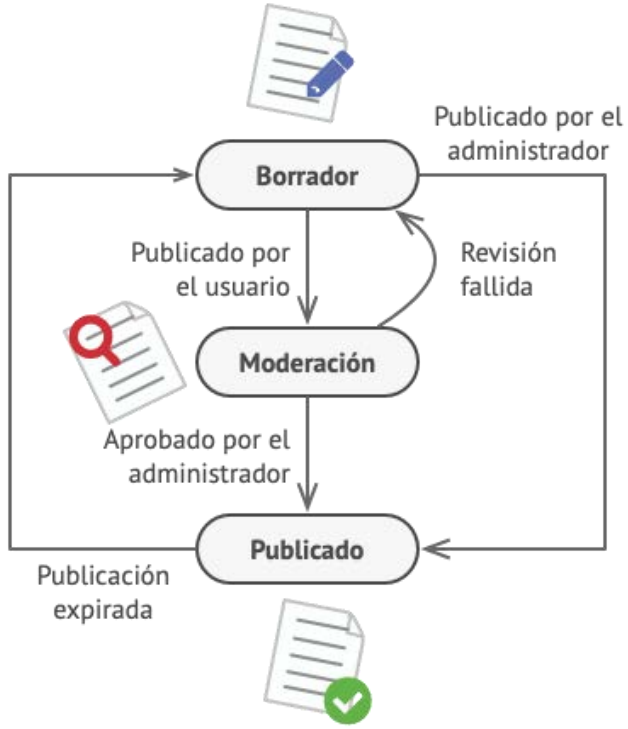
Patrón de Comportamiento: STATE

- Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.
- La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número finito de estados. Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados. Estas normas de cambio llamadas transiciones también son finitas y predeterminadas.



Problema

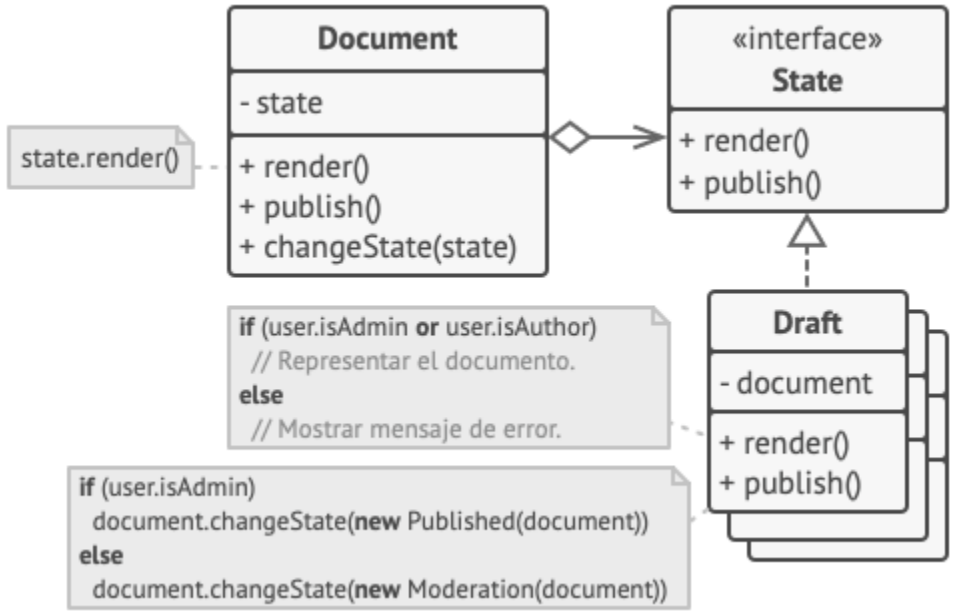
- Imagina que tienes una clase Documento. Un documento puede encontrarse en uno de estos tres estados: Borrador, Moderación y Publicado. El método publicar del documento funciona de forma ligeramente distinta en cada estado:
  - En Borrador, mueve el documento a moderación.
  - En Moderación, hace público el documento, pero sólo si el usuario actual es un administrador.
  - En Publicado, no hace nada en absoluto.



Las máquinas de estado se implementan normalmente con muchos operadores condicionales (if o switch) que seleccionan el comportamiento adecuado dependiendo del estado actual del objeto. Normalmente, este “estado” es tan solo un grupo de valores de los campos del objeto.

Solución

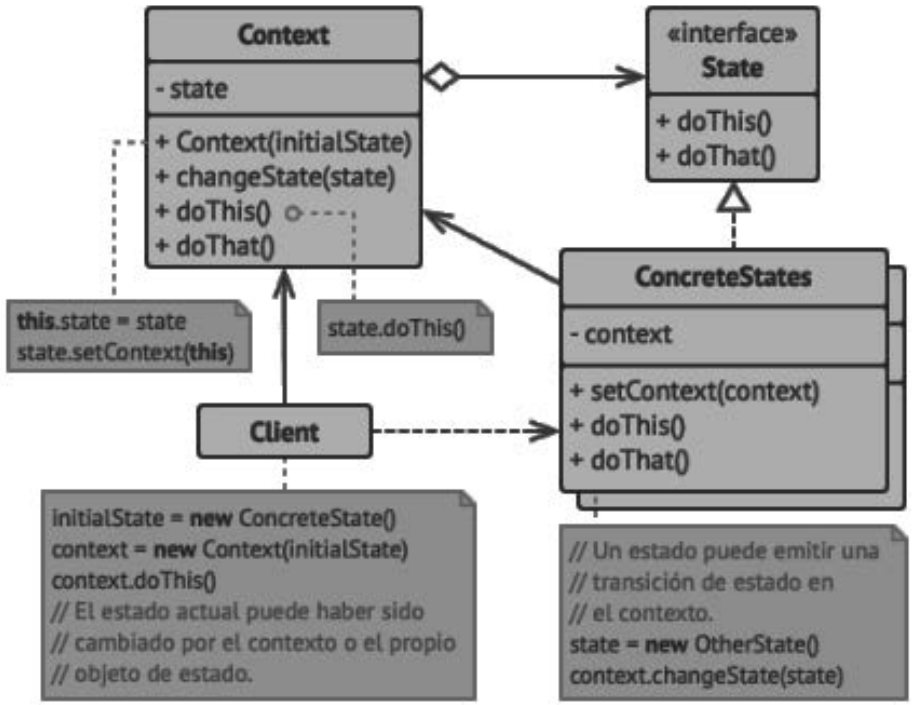
- El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.
- En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado contexto, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.



Estructura

1 La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

2 La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.



3 Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

4 Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.