**Patrones de Diseño**

Para poder entender que es un patrón de diseño debemos tomar en cuenta que estos son diseños habituales a problemas de software. Si hablamos en términos de construcción de una casa, debe de haber una sala, x cantidad de cuartos, x cantidad de baños, comedor, etc. En base a términos siempre es lo mismo, ahora consideremos que cada persona quiere que en cada cuarto tenga un diseño específico del otro, uno personalizado, quizá en el baño que tenga esto, y el otro sea diferente, de esta manera podemos interpretar en cómo funciona un patrón de diseño. Siempre hay pasos a seguir, siempre hay que llegar a la misma solución. Ojo con esto, no estamos hablando de que se pueden copiar estos patrones y reproducir en lo que deseemos, sino que cada patrón viene específicamente para un software, que ese patrón se deberá seguir a lo largo del desarrollo de este.

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón es más similar a un plano, ya que puedes observar cómo son su resultado y sus funciones, pero el orden exacto de la implementación depende de ti.

**¿En qué consiste el patrón?**

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Aquí tienes las secciones que suelen estar presentes en la descripción de un patrón:

* El propósito del patrón explica brevemente el problema y la solución.
* La motivación explica en más detalle el problema y la solución que brinda el patrón.
* La estructura de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
* El ejemplo de código en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Algunos catálogos de patrones enumeran otros detalles útiles, como la aplicabilidad del patrón, los pasos de implementación y las relaciones con otros patrones.

**Ventajas de usar patrones**

La realidad es que podrías trabajar durante años como programador sin conocer un solo patrón. Mucha gente lo hace. Incluso en ese caso, podrías estar implementando patrones sin saberlo. Así que, ¿por qué dedicar tiempo a aprenderlos?

* Los patrones de diseño son un juego de herramientas de **soluciones comprobadas** a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.
* Los patrones de diseño definen un lenguaje común que puedes utilizar con tus compañeros de equipo para comunicaros de forma más eficiente. Podrías decir: “Oh, utiliza un singleton para eso”, y todos entenderían la idea de tu sugerencia. No habría necesidad de explicar qué es un singleton si conocen el patrón y su nombre.

**Clasificaciones de los patrones**

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña. Me gusta la analogía de la construcción de carreteras: puedes hacer más segura una intersección instalando semáforos o construyendo un intercambiador completo de varios niveles con pasajes subterráneos para peatones.

Los patrones más básicos y de más bajo nivel suelen llamarse *idioms*. Normalmente se aplican a un único lenguaje de programación.

Los patrones más universales y de más alto nivel son los *patrones de arquitectura*. Los desarrolladores pueden implementar estos patrones prácticamente en cualquier lenguaje. Al contrario que otros patrones, pueden utilizarse para diseñar la arquitectura de una aplicación completa.

Además, todos los patrones pueden clasificarse por su *propósito*. Este libro cubre tres grupos generales de patrones:

* Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
  + **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.*
  + **Abstract Factory**: *es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.*
  + **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.*
  + **Prototype:** *es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.*
  + **Singleton:** *Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.*

**Prototype:**

**Problema:**

Digamos que tienes un objeto y quieres crear una copia exacta de él. ¿Cómo lo harías? En primer lugar, debes crear un nuevo objeto de la misma clase. Después debes recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto.

¡Bien! Pero hay una trampa. No todos los objetos se pueden copiar de este modo, porque algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.

Hay otro problema con el enfoque directo. Dado que debes conocer la clase del objeto para crear un duplicado, el código se vuelve dependiente de esa clase. Si esta dependencia adicional no te da miedo, todavía hay otra trampa. En ocasiones tan solo conocemos la interfaz que sigue el objeto, pero no su clase concreta, cuando, por ejemplo, un parámetro de un método acepta cualquier objeto que siga cierta interfaz.

Solución

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método clonar.

La implementación del método clonar es muy parecida en todas las clases. El método crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo. Se puede incluso copiar campos privados, porque la mayoría de los lenguajes de programación permite a los objetos acceder a campos privados de otros objetos que pertenecen a la misma clase.

Un objeto que soporta la clonación se denomina *prototipo*. Cuando tus objetos tienen decenas de campos y miles de configuraciones posibles, la clonación puede servir como alternativa a la creación de subclases.

Funciona así: se crea un grupo de objetos configurados de maneras diferentes. Cuando necesites un objeto como el que has configurado, clonas un prototipo en lugar de construir un nuevo objeto desde cero.

**Analogía en el mundo real**

En la vida real, los prototipos se utilizan para realizar pruebas de todo tipo antes de comenzar con la producción en masa de un producto. Sin embargo, en este caso, los prototipos no forman parte de una producción real, sino que juegan un papel pasivo.

Ya que los prototipos industriales en realidad no se copian a sí mismos, una analogía más precisa del patrón es el proceso de la división mitótica de una célula (biología, ¿recuerdas?). Tras la división mitótica, se forma un par de células idénticas. La célula original actúa como prototipo y asume un papel activo en la creación de la copia.

**Diagrama

Descripción generada automáticamente**

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

**Factory**

**Problema**

Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra dentro de la clase Camión.

Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de transporte marítimo para que incorpores la logística por mar a la aplicación.

Estupendo, ¿verdad? Pero, ¿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 decides añadir otro tipo de transporte a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

Al final acabarás 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. No te preocupes: 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*. Diagrama

Descripción generada automáticamente

A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

No obstante, hay una pequeña limitación: las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

Diagrama

Descripción generada automáticamente

Por ejemplo, tanto la clase Camión como la clase Barco deben implementar la interfaz Transporte, que declara un método llamado entrega. Cada clase implementa este método de forma diferente: los camiones entregan su carga por tierra, mientras que los barcos lo hacen por mar. El método fábrica dentro de la clase LogísticaTerrestre devuelve objetos de tipo camión, mientras que el método fábrica de la clase LogísticaMarítima devuelve barcos.

El código que utiliza el método fábrica (a menudo denominado código *cliente*) 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.

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

La clase base de diálogo utiliza distintos elementos UI para representar su ventana. En varios sistemas operativos, estos elementos pueden tener aspectos diferentes, pero su comportamiento debe ser consistente. Un botón en Windows sigue siendo un botón en Linux.

Cuando entra en juego el patrón Factory Method no hace falta reescribir la lógica del diálogo para cada sistema operativo. Si declaramos un patrón Factory Method que produce botones dentro de la clase base de diálogo, más tarde podremos crear una subclase de diálogo que devuelva botones al estilo de Windows desde el Factory Method. Entonces la subclase hereda la mayor parte del código del diálogo de la clase base, pero, gracias al Factory Method, puede representar botones al estilo de Windows en pantalla.

Para que este patrón funcione, la clase base de diálogo debe funcionar con botones abstractos, es decir, una clase base o una interfaz que sigan todos los botones concretos. De este modo, el código sigue siendo funcional, independientemente del tipo de botones con el que trabaje.

* Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
  + **Adapter:**  *es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles. (también llamado Adaptador, Envoltorio,Wrapper)*
  + **Bridge*:*** *es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.*
  + **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.*
  + **Decorador:**  *es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.*
  + **Facade:** *es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.*
  + **Flyweight:**  *es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.*
  + **Proxy:** *es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.*

**Adapter**

**Problema**

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**

Puedes crear un adaptador. Se trata de 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.

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.

Diagrama

Descripción generada automáticamente

Regresemos a nuestra aplicación del mercado de valores. Para resolver el dilema de los formatos incompatibles, puedes 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.

**Analogía en el mundo real**

Cuando viajas de Europa a Estados Unidos por primera vez, puede ser que te lleves una sorpresa cuanto intentes cargar tu computadora portátil. Los tipos de enchufe son diferentes en cada país, por lo que un enchufe español no sirve en Estados Unidos. El problema puede solucionarse utilizando un adaptador que incluya el enchufe americano y el europeo.

Texto

Descripción generada automáticamente con confianza media

Diagrama

Descripción generada automáticamente

**Diagrama

Descripción generada automáticamente**

**Flyweight**

**Problema**

Para divertirte un poco después de largas horas de trabajo, decides crear un sencillo videojuego en el que los jugadores se tienen que mover por un mapa disparándose entre sí. Decides implementar un sistema de partículas realistas que lo distinga de otros juegos. Grandes cantidades de balas, misiles y metralla de las explosiones volarán por todo el mapa, ofreciendo una apasionante experiencia al jugador.

Al terminarlo, subes el último cambio, compilas el juego y se lo envias a un amigo para una partida de prueba. Aunque el juego funcionaba sin problemas en tu máquina, tu amigo no logró jugar durante mucho tiempo. En su computadora el juego se paraba a los pocos minutos de empezar. Tras dedicar varias horas a revisar los registros de depuración, descubres que el juego se paraba debido a una cantidad insuficiente de RAM. Resulta que el equipo de tu amigo es mucho menos potente que tu computadora, y esa es la razón por la que el problema surgió tan rápido en su máquina.

El problema estaba relacionado con tu sistema de partículas. Cada partícula, como una bala, un misil o un trozo de metralla, estaba representada por un objeto separado que contenía gran cantidad de datos. En cierto momento, cuando la masacre alcanzaba su punto culminante en la pantalla del jugador, las partículas recién creadas ya no cabían en el resto de RAM, provocando que el programa fallara.

Diagrama

Descripción generada automáticamente

Solución

Observando más atentamente la clase Partícula, puede ser que te hayas dado cuenta de que los campos de color y *sprite* consumen mucha más memoria que otros campos. Lo que es peor, esos dos campos almacenan información casi idéntica de todas las partículas. Por ejemplo, todas las balas tienen el mismo color y sprite.

Diagrama

Descripción generada automáticamente

Otras partes del estado de una partícula, como las coordenadas, vector de movimiento y velocidad, son únicas en cada partícula. Después de todo, los valores de estos campos cambian a lo largo del tiempo. Estos datos representan el contexto siempre cambiante en el que existe la partícula, mientras que el color y el sprite se mantienen constantes.

Esta información constante de un objeto suele denominarse su *estado intrínseco*. Existe dentro del objeto y otros objetos únicamente pueden leerla, no cambiarla. El resto del estado del objeto, a menudo alterado “desde el exterior” por otros objetos, se denomina el *estado extrínseco*.

El patrón Flyweight sugiere que dejemos de almacenar el estado extrínseco dentro del objeto. En lugar de eso, debes pasar este estado a métodos específicos que dependen de él. Tan solo el estado intrínseco se mantiene dentro del objeto, permitiendo que lo reutilices en distintos contextos. Como resultado, necesitarás menos de estos objetos, ya que sólo se diferencian en el estado intrínseco, que cuenta con muchas menos variaciones que el extrínseco.

Diagrama

Descripción generada automáticamente

Regresemos a nuestro juego. Dando por hecho que hemos extraído el estado extrínseco de la clase de nuestra partícula, únicamente tres objetos diferentes serán suficientes para representar todas las partículas del juego: una bala, un misil y un trozo de metralla. Como probablemente habrás adivinado, un objeto que sólo almacena el estado intrínseco se denomina *Flyweight* (peso mosca).

**Almacenamiento del estado extrínseco**

¿A dónde se mueve el estado extrínseco? Alguna clase tendrá que almacenarlo, ¿verdad? En la mayoría de los casos, se mueve al objeto contenedor, que reúne objetos antes de que apliquemos el patrón.

En nuestro caso, se trata del objeto principal Juego, que almacena todas las partículas en su campo partículas. Para mover el estado extrínseco a esta clase, debes crear varios campos matriz para almacenar coordenadas, vectores y velocidades de cada partícula individual. Pero eso no es todo. Necesitas otra matriz para almacenar referencias a un objeto *flyweight* específico que represente una partícula. Estas matrices deben estar sincronizadas para que puedas acceder a toda la información de una partícula utilizando el mismo índice.

Diagrama

Descripción generada automáticamente

**Flyweight y la inmutabilidad**

Debido a que el mismo objeto flyweight puede utilizarse en distintos contextos, debes asegurarte de que su estado no se pueda modificar. Un objeto flyweight debe inicializar su estado una sola vez a través de parámetros del constructor. No debe exponer ningún método *set* (modificador) o campo público a otros objetos.

**Fábrica flyweight**

Para un acceso más cómodo a varios objetos flyweight, puedes crear un método fábrica que gestione un grupo de objetos flyweight existentes. El método acepta el estado intrínseco del flyweight deseado por un cliente, busca un objeto flyweight existente que coincida con este estado y lo devuelve si lo encuentra. Si no, crea un nuevo objeto flyweight y lo añade al grupo.

Existen muchas opciones para colocar este método. El lugar más obvio es un contenedor flyweight. Alternativamente, podrías crea un nueva clase fábrica y hacer estático el método fábrica para colocarlo dentro de una clase flyweight real.

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

El patrón extrae el estado intrínseco repetido de una clase principal Árbol y la mueve dentro de la clase flyweight TipodeÁrbol.

Ahora, en lugar de almacenar la misma información en varios objetos, se mantiene en unos pocos objetos flyweight vinculados a los objetos de Árbol adecuados que actúan como contexto. El código cliente crea nuevos objetos árbol utilizando la fábrica flyweight, que encapsula la complejidad de buscar el objeto adecuado y reutilizarlo si es necesario.

* Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.
  + **Chain of responsibility:**  *es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.*
  + **Command:** *es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.*
  + **Iterator:** *es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).*
  + **Mediator:** *es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.*
  + **Memento:** *es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.*
  + **Observer:** *es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.*
  + **State:** *es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.*
  + **Strategy:** *es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.*
  + **Template method:** *es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.*
  + **Visitor:** *es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.*

**Strategy**

**Problema**

Un día decidiste crear una aplicación de navegación para viajeros ocasionales. La aplicación giraba alrededor de un bonito mapa que ayudaba a los usuarios a orientarse rápidamente en cualquier ciudad.

Una de las funciones más solicitadas para la aplicación era la planificación automática de rutas. Un usuario debía poder introducir una dirección y ver la ruta más rápida a ese destino mostrado en el mapa.

La primera versión de la aplicación sólo podía generar las rutas sobre carreteras. Las personas que viajaban en coche estaban locas de alegría. Pero, aparentemente, no a todo el mundo le gusta conducir durante sus vacaciones. De modo que, en la siguiente actualización, añadiste una opción para crear rutas a pie. Después, añadiste otra opción para permitir a las personas utilizar el transporte público en sus rutas.

Sin embargo, esto era sólo el principio. Más tarde planeaste añadir la generación de rutas para ciclistas, y más tarde, otra opción para trazar rutas por todas las atracciones turísticas de una ciudad.

Aunque desde una perspectiva comercial la aplicación era un éxito, la parte técnica provocaba muchos dolores de cabeza. Cada vez que añadías un nuevo algoritmo de enrutamiento, la clase principal del navegador doblaba su tamaño. En cierto momento, la bestia se volvió demasiado difícil de mantener.

Cualquier cambio en alguno de los algoritmos, ya fuera un sencillo arreglo de un error o un ligero ajuste de la representación de la calle, afectaba a toda la clase, aumentando las probabilidades de crear un error en un código ya funcional.

Además, el trabajo en equipo se volvió ineficiente. Tus compañeros, contratados tras el exitoso lanzamiento, se quejaban de que dedicaban demasiado tiempo a resolver conflictos de integración. Implementar una nueva función te exige cambiar la misma clase enorme, entrando en conflicto con el código producido por otras personas.

**Solución**

El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

La clase original, llamada *contexto*, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.

Diagrama

Descripción generada automáticamente

En nuestra aplicación de navegación, cada algoritmo de enrutamiento puede extraerse y ponerse en su propia clase con un único método crearRuta. El método acepta un origen y un destino y devuelve una colección de puntos de control de la ruta.

Incluso contando con los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente. A la clase navegadora principal no le importa qué algoritmo se selecciona ya que su labor principal es representar un grupo de puntos de control en el mapa. La clase tiene un método para cambiar la estrategia activa de enrutamiento, de modo que sus clientes, como los botones en la interfaz de usuario, pueden sustituir el comportamiento seleccionado de enrutamiento por otro.

**Analogía en el mundo real**

Imagina que tienes que llegar al aeropuerto. Puedes tomar el autobús, pedir un taxi o ir en bicicleta. Éstas son tus estrategias de transporte. Puedes elegir una de las estrategias, dependiendo de factores como el presupuesto o los límites de tiempo.

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

**Iterator**

**Problema**

Las colecciones son de los tipos de datos más utilizados en programación. Sin embargo, una colección tan solo es un contenedor para un grupo de objetos.

La mayoría de las colecciones almacena sus elementos en simples listas, pero algunas de ellas se basan en pilas, árboles, grafos y otras estructuras complejas de datos.

Independientemente de cómo se estructure una colección, debe aportar una forma de acceder a sus elementos de modo que otro código pueda utilizar dichos elementos. Debe haber una forma de recorrer cada elemento de la colección sin acceder a los mismos elementos una y otra vez.

Esto puede parecer un trabajo sencillo si tienes una colección basada en una lista. En este caso sólo tienes que recorrer en bucle todos sus elementos. Pero, ¿cómo recorres secuencialmente elementos de una estructura compleja de datos, como un árbol? Por ejemplo, un día puede bastarte con un recorrido de profundidad de un árbol, pero, al día siguiente, quizá necesites un recorrido en anchura. Y, la semana siguiente, puedes necesitar otra cosa, como un acceso aleatorio a los elementos del árbol.

Añadir más y más algoritmos de recorrido a la colección nubla gradualmente su responsabilidad principal, que es el almacenamiento eficiente de la información. Además, puede que algunos algoritmos estén personalizados para una aplicación específica, por lo que incluirlos en una clase genérica de colección puede resultar extraño.

Por otro lado, el código cliente que debe funcionar con varias colecciones puede no saber cómo éstas almacenan sus elementos. No obstante, ya que todas las colecciones proporcionan formas diferentes de acceder a sus elementos, no tienes otra opción más que acoplar tu código a las clases de la colección específica.

**Solución**

La idea central del patrón Iterator es extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado *iterador*.

Además de implementar el propio algoritmo, un objeto iterador encapsula todos los detalles del recorrido, como la posición actual y cuántos elementos quedan hasta el final. Debido a esto, varios iteradores pueden recorrer la misma colección al mismo tiempo, independientemente los unos de los otros.

Normalmente, los iteradores aportan un método principal para extraer elementos de la colección. El cliente puede continuar ejecutando este método hasta que no devuelva nada, lo que significa que el iterador ha recorrido todos los elementos.

Todos los iteradores deben implementar la misma interfaz. Esto hace que el código cliente sea compatible con cualquier tipo de colección o cualquier algoritmo de recorrido, siempre y cuando exista un iterador adecuado. Si necesitas una forma particular de recorrer una colección, creas una nueva clase iteradora sin tener que cambiar la colección o el cliente.

**Diagrama

Descripción generada automáticamente**

**Analogía en el mundo real**

Planeas visitar Roma por unos días y ver todas sus atracciones y puntos de interés. Pero, una vez allí, podrías perder mucho tiempo dando vueltas, incapaz de encontrar siquiera el Coliseo.

En lugar de eso, podrías comprar una aplicación de guía virtual para tu smartphone y utilizarla para moverte. Es buena y barata y puedes quedarte en sitios interesantes todo el tiempo que quieras.

Una tercera alternativa sería dedicar parte del presupuesto del viaje a contratar un guía local que conozca la ciudad como la palma de su mano. El guía podría adaptar la visita a tus gustos, mostrarte las atracciones y contarte un montón de emocionantes historias. Eso sería más divertido pero, lamentablemente, también más caro.

Todas estas opciones —las direcciones aleatorias en tu cabeza, el navegador del smartphone o el guía humano—, actúan como iteradores sobre la amplia colección de visitas y atracciones de Roma.

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente Diagrama

Descripción generada automáticamente

El iterador ‘amigos’ puede utilizarse para recorrer los amigos de un perfil dado. El iterador ‘colegas’ hace lo mismo, excepto que omite amigos que no trabajen en la misma empresa que la persona objetivo. Ambos iteradores implementan una interfaz común que permite a los clientes extraer perfiles sin profundizar en los detalles de la implementación, como la autenticación y el envío de solicitudes REST.

El código cliente no está acoplado a clases concretas porque sólo trabaja con colecciones e iteradores a través de interfaces. Si decides conectar tu aplicación a una nueva red social, sólo necesitas proporcionar nuevas clases de colección e iteradoras, sin cambiar el código existente.

**Conclusión**

Un patrón nos ayudará a como desenvolver correctamente nuestro código, generando así una estructura, arquitectura más sostenible, más limpia, más segura y menos irrompible. Tal como se describió en este documento existen varios tipos de patrones de diseño.

De acuerdo con la magnitud del sistema que se esté implementando es necesario analizar que patrón de diseño ayudará a tener una mejor organización en código, y para los mantenimientos futuros, eso ayuda a que nuestro sistema sea robusto, y no presente fallas a la larga. Podemos optar en no aplicar algunos de estos patrones, pero nos estaríamos metiendo a territorio hostil a la hora de programar, y en un futuro sea muy difícil de mantener.