**Propósito**

**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.).

**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.

**Aplicabilidad**

 Utiliza el patrón Iterator cuando tu colección tenga una estructura de datos compleja a nivel interno, pero quieras ocultar su complejidad a los clientes (ya sea por conveniencia o por razones de seguridad).

 El iterador encapsula los detalles del trabajo con una estructura de datos compleja, proporcionando al cliente varios métodos simples para acceder a los elementos de la colección. Esta solución, además de ser muy conveniente para el cliente, también protege la colección frente a acciones descuidadas o maliciosas que el cliente podría realizar si trabajara con la colección directamente.

Utiliza el patrón para reducir la duplicación en el código de recorrido a lo largo de tu aplicación.

 El código de los algoritmos de iteración no triviales tiende a ser muy voluminoso. Cuando se coloca dentro de la lógica de negocio de una aplicación, puede nublar la responsabilidad del código original y hacerlo más difícil de mantener. Mover el código de recorrido a iteradores designados puede ayudarte a hacer el código de la aplicación más breve y limpio.

 Utiliza el patrón Iterator cuando quieras que tu código pueda recorrer distintas estructuras de datos, o cuando los tipos de estas estructuras no se conozcan de antemano.

 El patrón proporciona un par de interfaces genéricas para colecciones e iteradores. Teniendo en cuenta que ahora tu código utiliza estas interfaces, seguirá funcionando si le pasas varios tipos de colecciones e iteradores que implementen esas interfaces.

**Pros**

* *Principio de responsabilidad única*. Puedes limpiar el código cliente y las colecciones extrayendo algoritmos de recorrido voluminosos y colocándolos en clases independientes.
* *Principio de abierto/cerrado*. Puedes implementar nuevos tipos de colecciones e iteradores y pasarlos al código existente sin descomponer nada.
* Puedes recorrer la misma colección en paralelo porque cada objeto iterador contiene su propio estado de iteración.
* Por la misma razón, puedes retrasar una iteración y continuar cuando sea necesario.

## Contras

* Aplicar el patrón puede resultar excesivo si tu aplicación funciona únicamente con colecciones sencillas.
* Utilizar un iterador puede ser menos eficiente que recorrer directamente los elementos de algunas colecciones especializadas.