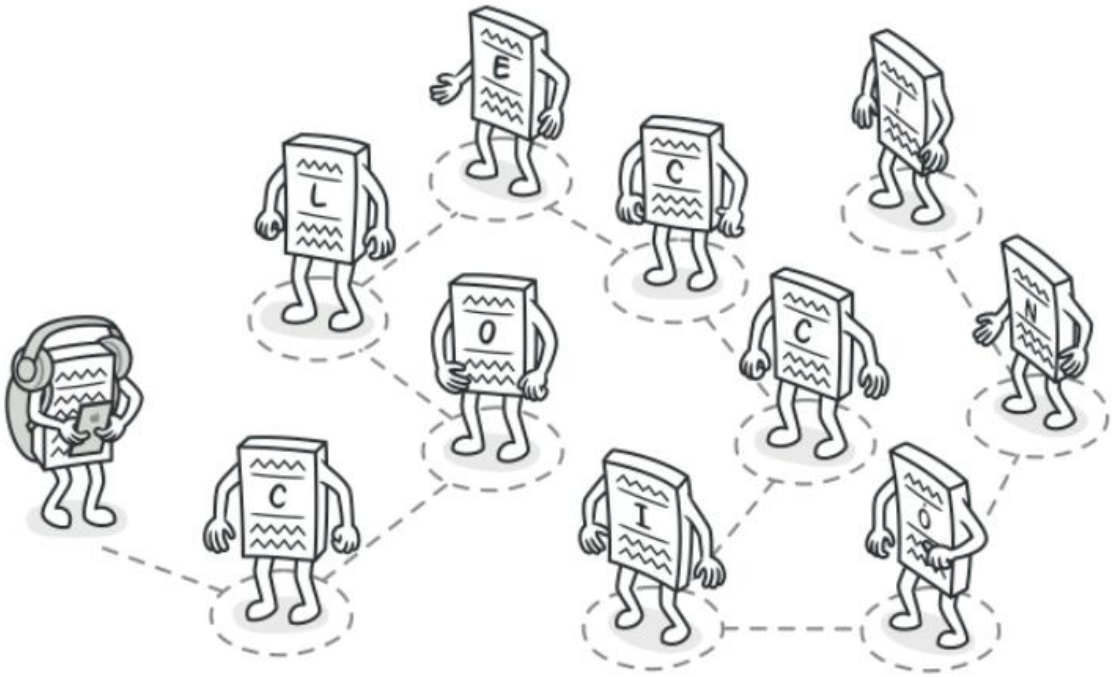
# Patrón de Diseño Iterator



**Objetivo del patrón:** El patrón Iterator proporciona un mecanismo uniforme para recorrer los elementos de una colección sin exponer su implementación interna.

## Contexto y Problema

- **Problema común en el diseño de software:**
    - Se necesita iterar sobre una colección de elementos sin exponer los detalles de su implementación (por ejemplo, listas, árboles o pilas).
    - En caso de necesitar varios tipos de iteración (ej., recorrer en orden inverso), sería engorroso implementar múltiples métodos dentro de cada colección.
- **Ejemplo:** En una aplicación de e-commerce, hay colecciones de productos que necesitan ser recorridas para mostrarse en la página de resultados, sin importar si están almacenadas en listas, pilas, o árboles. Sin el patrón Iterator, cada tipo de colección tendría su propia lógica para el recorrido, haciendo el código más complejo y menos flexible.

## Definición del Patrón Iterator

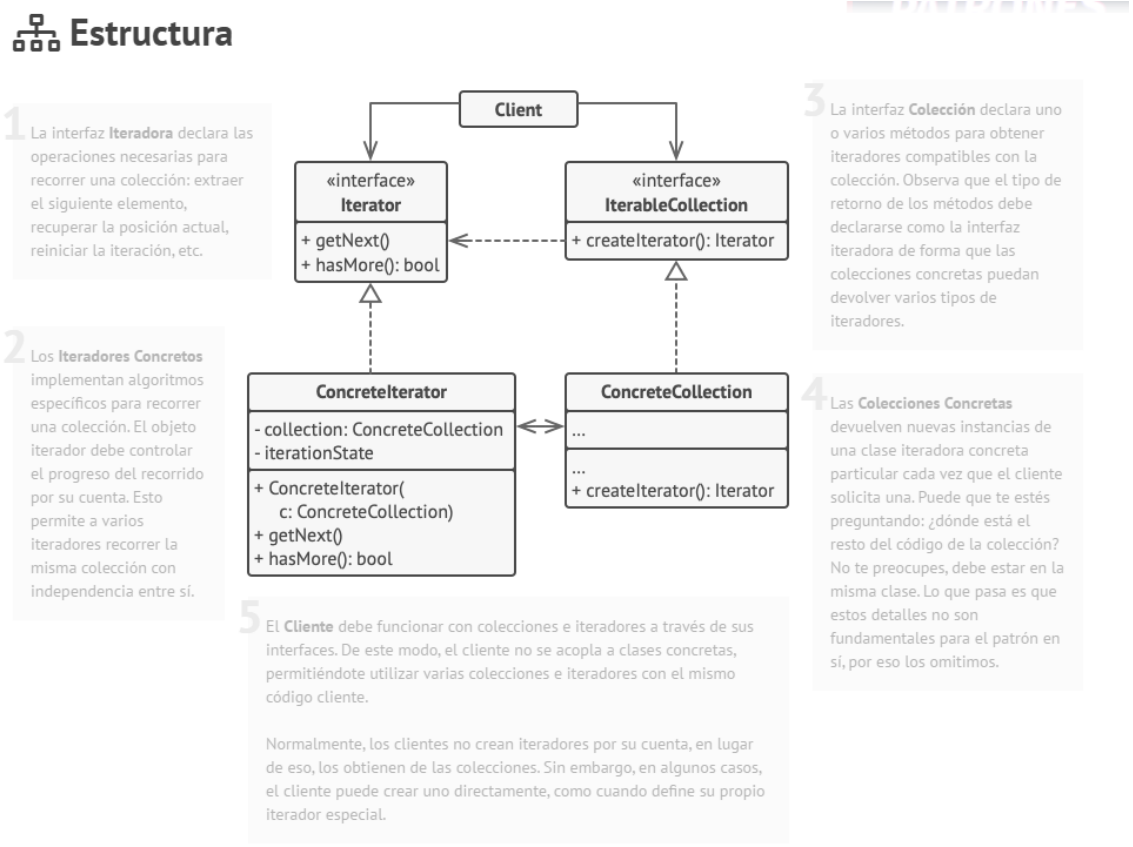
**Iterator:**
El patrón Iterator es un patrón de comportamiento que permite recorrer los elementos de una colección de forma secuencial sin exponer la estructura interna de la colección.

Este patrón proporciona una interfaz común para iterar sobre distintos tipos de colecciones de manera uniforme.

## Estructura del Patrón Iterator

1. **Iterator (Iterador):**
    - Interfaz o clase abstracta que define los métodos para recorrer los elementos de una colección, como next() y has_next().
2. **ConcreteIterator (Iterador Concreto):**
    - Implementación específica del Iterator para una colección particular. Mantiene el estado actual del recorrido.
3. **Aggregate (Colección o Agregado):**
    - Interfaz o clase abstracta que declara el método para crear un iterador.
4. **ConcreteAggregate (Colección Concreta):**
    - Implementación específica de una colección que devuelve un ConcreteIterator adecuado para esa colección.

Esta estructura permite desacoplar la lógica de recorrido de la implementación de la colección, facilitando el cambio o la adición de diferentes tipos de iteradores.



## Ejemplo Práctico

**Contexto:**
Imaginemos una biblioteca de libros en línea que almacena listas de libros. Queremos iterar sobre estas listas de libros para mostrarlos sin importar si están organizados como listas o conjuntos.

```python
# Iterator
class IteradorLibro:
    def next(self):
        pass

    def has_next(self):
        pass
# ConcreteIterator
class IteradorListaLibros(IteradorLibro):
    def __init__(self, lista_libros):
        self.lista_libros = lista_libros
        self.indice = 0

    def next(self):
        libro = self.lista_libros[self.indice]
        self.indice += 1
        return libro

    def has_next(self):
        return self.indice < len(self.lista_libros)

# Aggregate y ConcreteAggregate
class ColeccionLibros:
    def crear_iterador(self):
        pass

class ListaLibros(ColeccionLibros):
    def __init__(self, libros):
        self.libros = libros

    def crear_iterador(self):
        return IteradorListaLibros(self.libros)

#Cliente
libros = ListaLibros(["Libro Y", "Libro Z", "Libro M", "Libro X"])
iterador = libros.crear_iterador()

while iterador.has_next():
    print(iterador.next())

# Salida: "Libro Y", "Libro Z", "Libro M","Libro X"

```

## Ventajas y Desventajas

**Ventajas:**

- **Desacoplamiento:** Separa la lógica de iteración de la implementación interna de la colección.

- **Flexibilidad:** Es posible agregar diferentes tipos de iteradores para una misma colección (ej., iterador inverso).
- **Uniformidad:** Permite un mecanismo de acceso uniforme para diferentes colecciones.

### Desventajas:

- **Sobrecarga:** Puede requerir una clase iteradora adicional para cada colección concreta, añadiendo complejidad al diseño.
- **Rendimiento:** La creación de múltiples iteradores en tiempo de ejecución puede aumentar el consumo de memoria si se utiliza en grandes colecciones.

## ¿Cuándo Usar el Patrón Iterator?

### Situaciones recomendadas:

- **Colecciones complejas:** Cuando se necesita recorrer colecciones complejas y heterogéneas sin exponer sus detalles internos.
- **Varias formas de recorrido:** Si se requieren diferentes maneras de recorrer una colección, el patrón permite tener iteradores específicos para cada propósito.
- **Interfaces comunes:** Cuando es importante ofrecer una interfaz uniforme de acceso a elementos para distintas colecciones.

## Conclusión y Preguntas

### Conclusión:

El patrón Iterator facilita el acceso a los elementos de una colección sin exponer su estructura interna, haciendo que el código sea más flexible y fácil de mantener. Es especialmente útil para implementar múltiples formas de recorrer colecciones y para manejar diferentes tipos de estructuras de datos de manera uniforme.