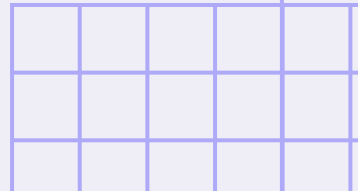


Iterator



Integrantes!



Jorge Loría

A63265



Michelle Fonseca

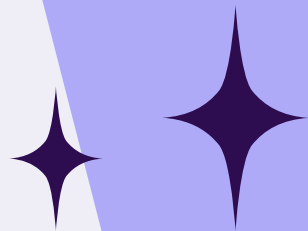
B93034



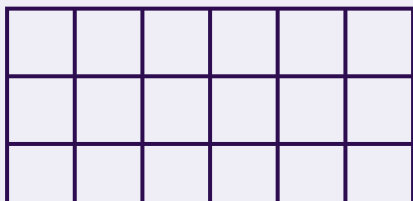
Julían Sedó

B97388

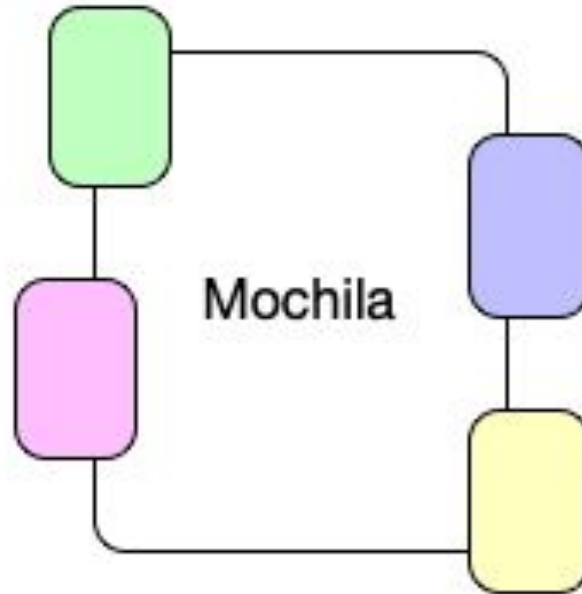




Introducción

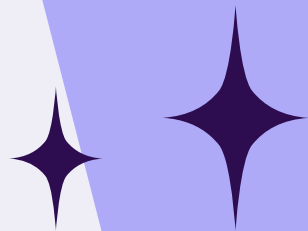


Metáfora de la vida real

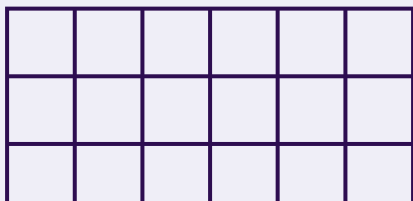


Fuente: <https://www.arquitecturajava.com/el-patron-iterador-y-su-flexibilidad/>





Problema



Problema

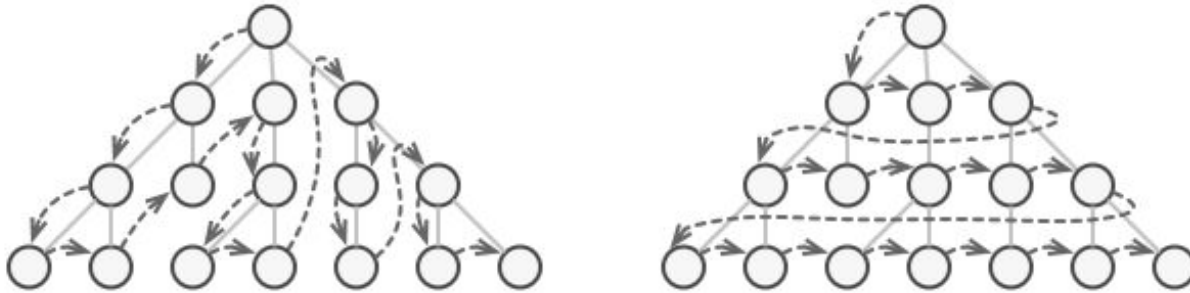
El patrón Iterator se usa en el contexto de las listas y conjuntos.

Tenemos una serie de objetos que internamente trabajan con conjuntos de elementos.

Necesitamos manipularlos abstrayéndose de cómo están implementados internamente.

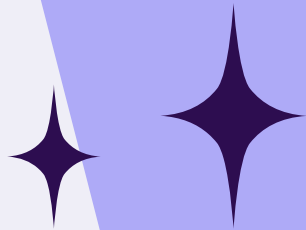


Problema

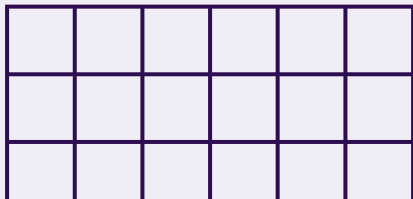


La misma colección puede recorrerse de varias formas diferentes.

Fuente: <https://refactoring.guru/es/design-patterns/iterator>



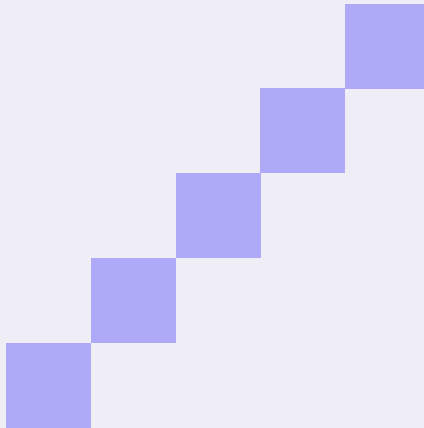
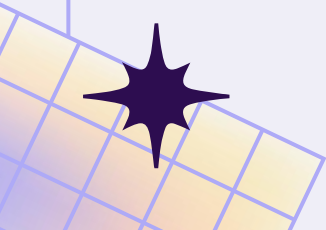
Solución



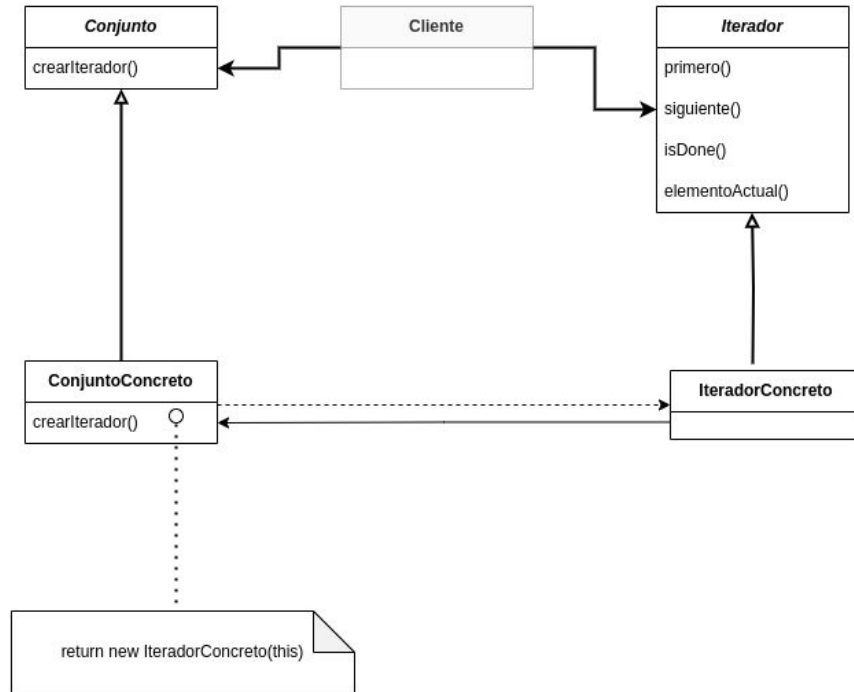


Solución

La solución consiste en crear una interfaz Iterator que estandarice los métodos para tratar la colección de elementos.



Solución



Solución

Esta interfaz definirá una serie de operaciones para manipular los elementos del conjunto,

- siguiente() para obtener el siguiente elemento
- haySiguiente() para comprobar que sigue habiendo elementos en el conjunto,
- actual() para obtener el elemento actual
- primero() para mover el cursor al primer elemento y a la obtener una referencia al mismo.

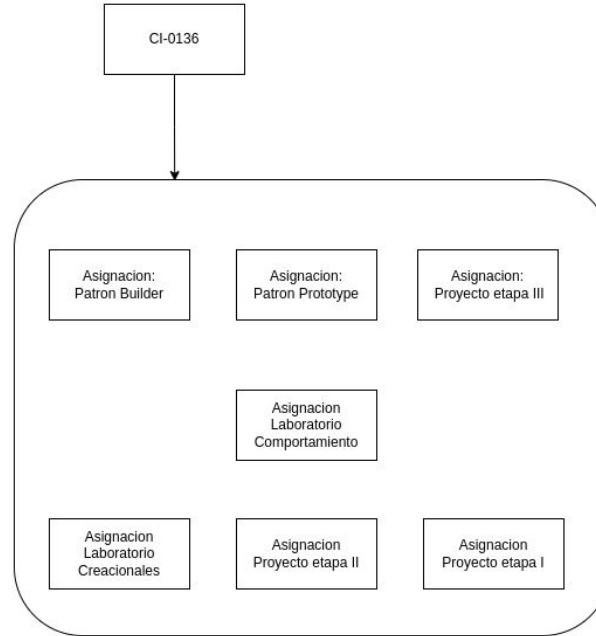
Solución

Participantes:

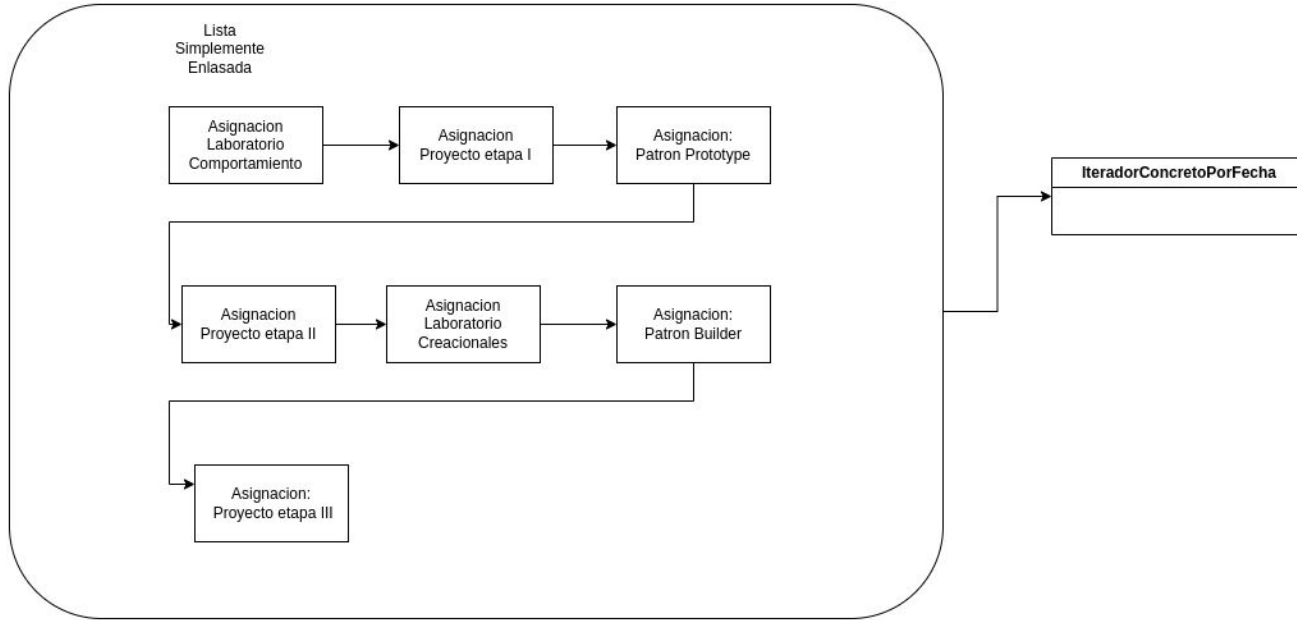
- Iterador: define interfaz para acceder y recorrer elementos
- Iterador Concreto:
 - Implementa la interfaz Iterator
 - Mantiene la posición actual en el recorrido del agregado
- Conjunto: define la interfaz para crear un objeto Iterator
- Conjunto Concreto: implementa una interfaz de creación del Iterator para devolver la instancia de Concreteliterator apropiada

Solución: Mediación Virtual

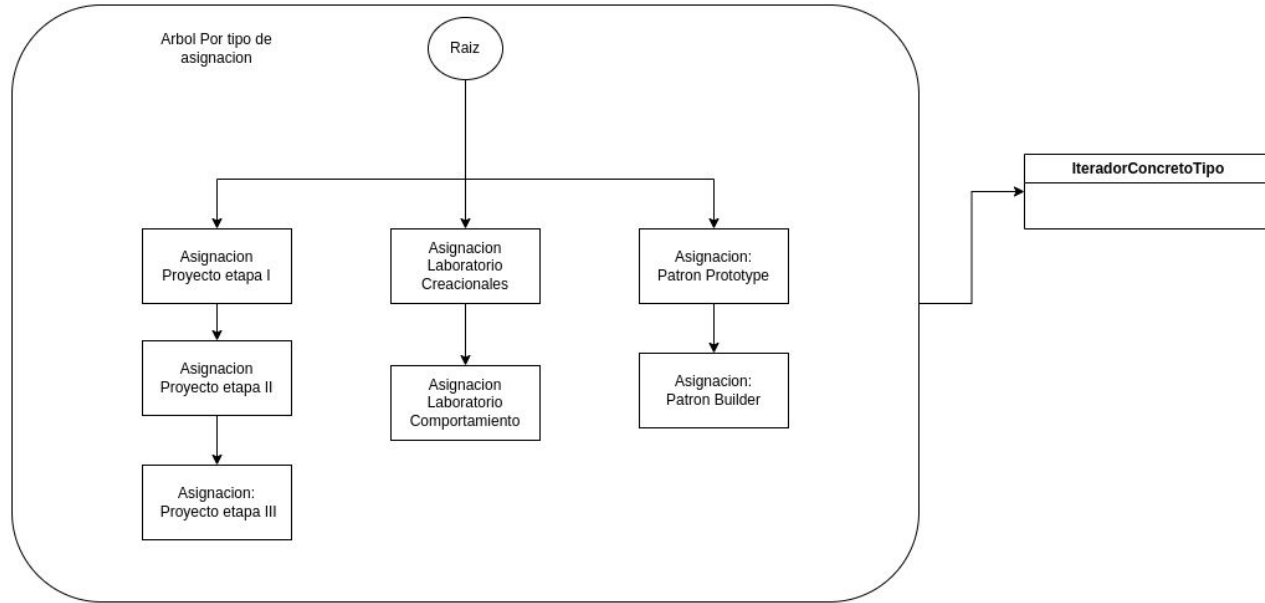
Queremos desplegar las asignaciones de un curso particular para que el usuario los vea.

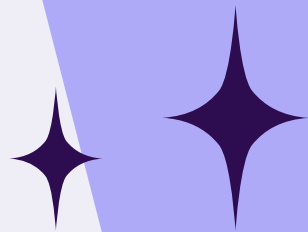


Solución: Mediación Virtual

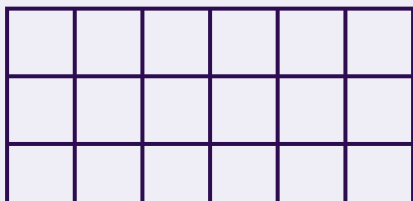


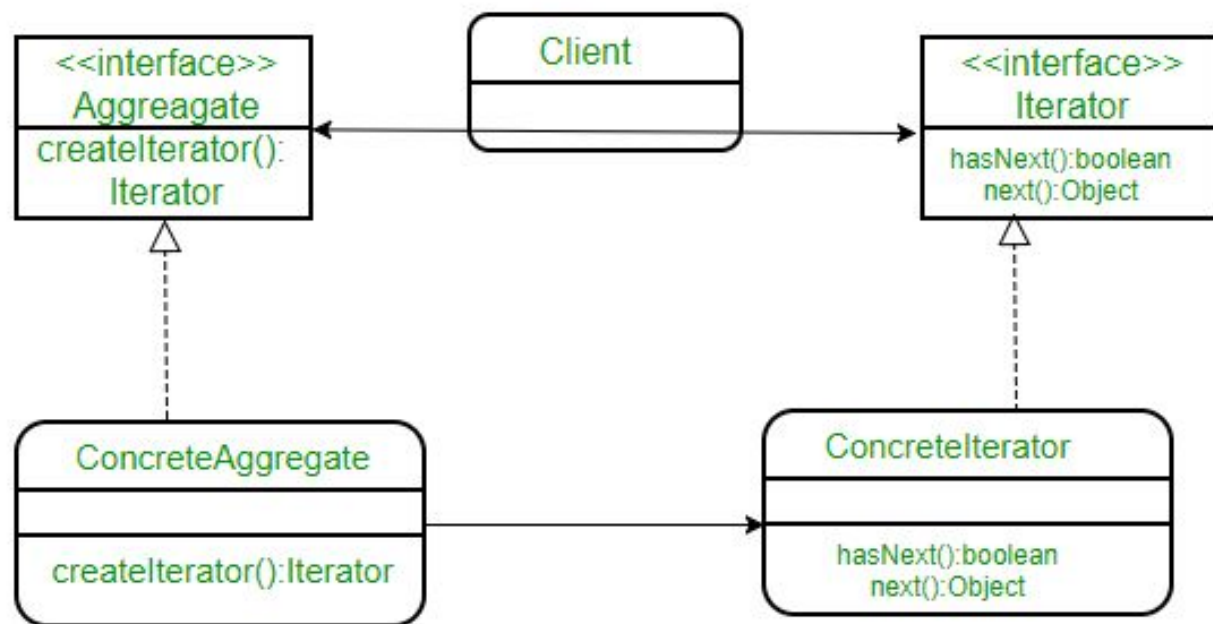
Solución: Mediación Virtual

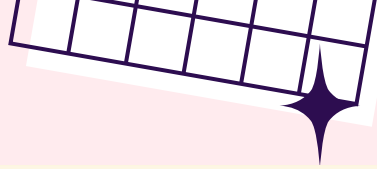




Ejemplo de código



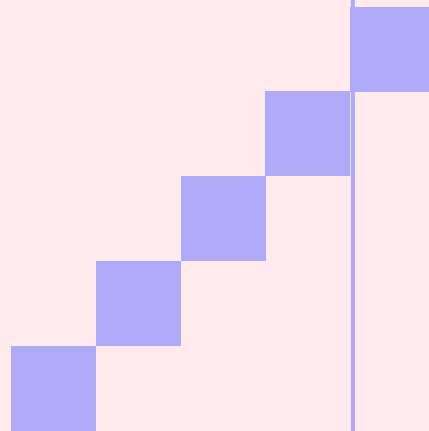
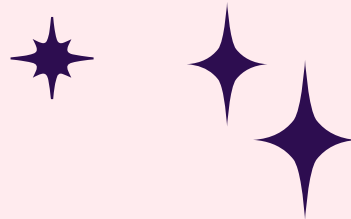




```
1 // Una clase simple de Notificación
2 class Notification
3 {
4     // Para almacenar un mensaje de notificación
5     String notification;
6
7     public Notification(String notification)
8     {
9         this.notification = notification;
10    }
11    public String getNotification()
12    {
13        return notification;
14    }
15 }
16
17 // Intefaz "Collection"
18 interface Collection
19 {
20     public Iterator createIterator();
21 }
```

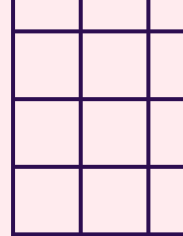


```
1 // Colección de notificaciones
2 class NotificationCollection implements Collection
3 {
4     static final int MAX_ITEMS = 6;
5     int numberOfItems = 0;
6     Notification[] notificationList;
7
8     public NotificationCollection()
9     {
10         notificationList = new Notification[MAX_ITEMS];
11
12         // Notificaciones para el ejemplo
13         addItem("Notification 1");
14         addItem("Notification 2");
15         addItem("Notification 3");
16     }
17
18     public void addItem(String str)
19     {
20         Notification notification = new Notification(str);
21         if (numberOfItems >= MAX_ITEMS)
22             System.err.println("Full");
23         else
24         {
25             notificationList[numberOfItems] = notification;
26             numberOfItems = numberOfItems + 1;
27         }
28     }
29
30     public Iterator createIterator()
31     {
32         return new NotificationIterator(notificationList);
33     }
34 }
```





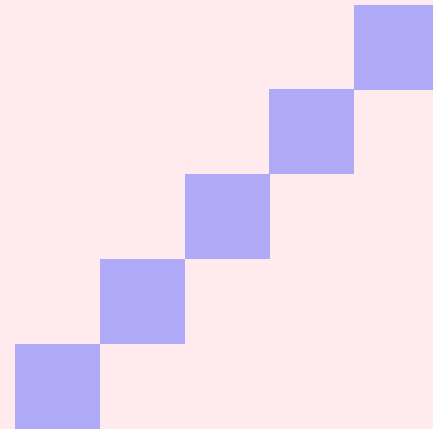
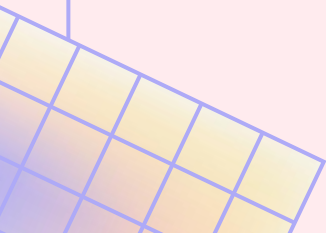
```
1  // También se podría usar Java.Util.Iterator
2  interface Iterator
3  {
4      // indica si hay más elementos sobre los que iterar
5      boolean hasNext();
6
7      // retorna el siguiente elemento
8      Object next();
9  }
```



```
1  // ConcreteIterator
2  class NotificationIterator implements Iterator
3  {
4      Notification[] notificationList;
5
6      // mantiene la posición actual del iterador sobre el array
7      int pos = 0;
8
9      // El constructor toma el array de notificationList sobre el que se va a iterar.
10     public NotificationIterator (Notification[] notificationList)
11     {
12         this.notificationList = notificationList;
13     }
14
15     public Object next()
16     {
17         // retorna el siguiente elemento en el array e incrementa pos
18         Notification notification = notificationList[pos];
19         pos += 1;
20         return notification;
21     }
22
23     public boolean hasNext()
24     {
25         if (pos >= notificationList.length ||
26             notificationList[pos] == null)
27             return false;
28         else
29             return true;
30     }
31 }
```



```
1 // Contiene la colección de notificaciones
2 // como un objeto de NotificationCollection
3 class NotificationBar
4 {
5     NotificationCollection notifications;
6
7     public NotificationBar(NotificationCollection notifications)
8     {
9         this.notifications = notifications;
10    }
11
12    public void printNotifications()
13    {
14        Iterator iterator = notifications.createIterator();
15        System.out.println("-----NOTIFICATION BAR-----");
16        while (iterator.hasNext())
17        {
18            Notification n = (Notification)iterator.next();
19            System.out.println(n.getNotification());
20        }
21    }
22 }
```





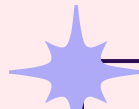
```
1  class Main
2  {
3      public static void main(String args[])
4      {
5          NotificationCollection nc = new NotificationCollection();
6          NotificationBar nb = new NotificationBar(nc);
7          nb.printNotifications();
8      }
9  }
```

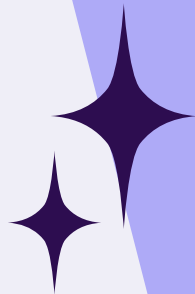
-----NOTIFICATION BAR-----

Notification 1

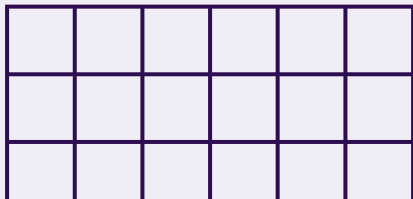
Notification 2

Notification 3





Implementación



Use el patrón cuando...

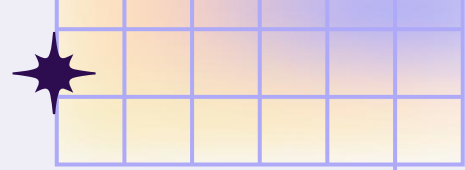
su colección tenga una estructura de datos compleja pero desee ocultar su complejidad a los clientes

El iterador encapsula los detalles de trabajar con una estructura de datos compleja, proporcionando al cliente varios métodos simples para acceder a los elementos de la colección. Si bien este enfoque es muy conveniente para el cliente, también protege la colección de acciones descuidadas o maliciosas que el cliente podría realizar si trabaja directamente con la colección.

Use el patrón para...

*reducir la duplicación
del código transversal
en su aplicación*

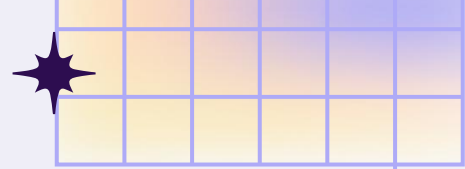
El código de los algoritmos de iteración no triviales tiende a ser muy grande. Mover el código transversal a los iteradores designados puede ayudar a hacer que el código de la aplicación sea más ágil y limpio.

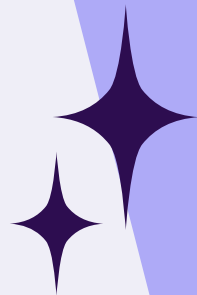


Use el patrón cuando...

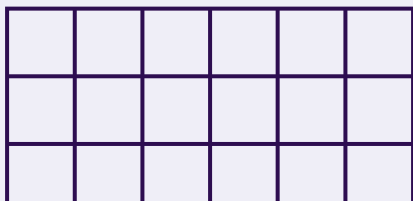
*desee que su código pueda
atravesar diferentes
estructuras de datos o
cuando los tipos de estas
estructuras se desconozcan
de antemano.*

El patrón proporciona un par de interfaces genéricas tanto para colecciones como para iteradores. Dado que su código ahora usa estas interfaces, seguirá funcionando si le pasan varios tipos de colecciones e iteradores que implementan estas interfaces.





Relación con otros patrones





01


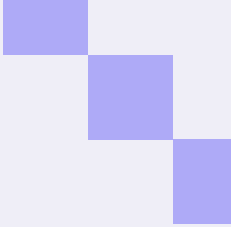
Puede usar Factory Method junto con Iterator para permitir que las subclases devuelvan diferentes tipos de iteradores que sean compatibles con las colecciones.

02

Puede usar Memento junto con Iterator para capturar el estado de iteración actual y revertirlo si es necesario.

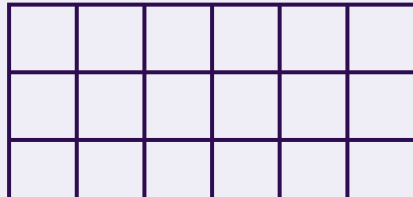
03

Puede usar Visitor junto con Iterator para atravesar una estructura de datos compleja y ejecutar alguna operación sobre sus elementos, incluso si todos tienen diferentes clases.





Consecuencias



Ventajas

- Principio de responsabilidad única (SRP): Puede limpiar el código del cliente y las colecciones extrayendo los algoritmos transversales voluminosos en clases separadas.
- Principio Abierto/Cerrado. Puede implementar nuevos tipos de colecciones e iteradores y pasarlos al código existente sin romper nada.
- Se puede iterar sobre la misma colección en paralelo, cada objeto iterador contiene su propio estado de iteración.



Desventajas

- Aplicar el patrón puede resultar excesivo si la aplicación sólo trabaja con colecciones sencillas.
- Utilizar un iterador puede ser menos eficiente que recorrer directamente los elementos de algunas colecciones especializadas.

Caules, C. Á. (2022). El patrón Iterador y su flexibilidad. Arquitectura Java.

<https://www.arquitecturajava.com/el-patron-iterador-y-su-flexibilidad/>

Design Patterns - Iterator Pattern. (n.d.).

https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

GeeksforGeeks. (2022). Iterator Pattern. GeeksforGeeks.

<https://www.geeksforgeeks.org/iterator-pattern/> Iterator. (n.d.-a).

<https://refactoring.guru/es/design-patterns/iterator> Iterator. (n.d.-b).

<https://reactiveprogramming.io/blog/es/patrones-de-diseno/iterator> SL, P.

E. C. (n.d.). Patrones de Diseño (XVII): Patrones de Comportamiento -

Iterator. Programación En Castellano.

https://programacion.net/articulo/patrones_de_diseno_xvii_patrones_de_comportamiento_iterator_1020