Politécnico Colombiano Jaime Isaza Cadavid

Facultad de Ingeniería

Patron Iterador

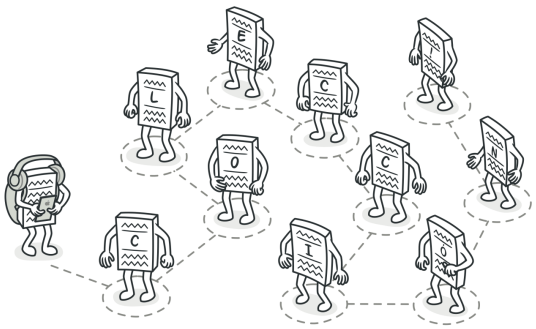
Diseño de software

Autor: Claudia
Apellidos: Gil Sánchez

9 de febrero de 2022

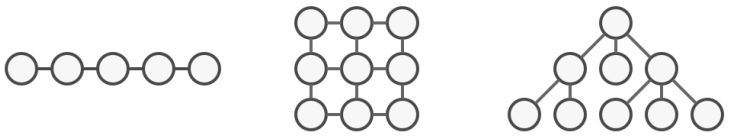
Introducción

El patron Iterador Proporciona una forma de acceder a elementos individuales en un objeto contenedor sin exponer los detalles internos del objeto.

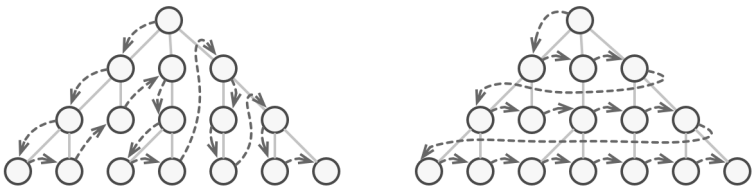


Iterador

s un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente. coexiste con colecciones. En términos generales, siempre que implementemos una colección, necesitamos proporcionar iteradores para esta colección al mismo tiempo, al igual que Colección, Lista, Conjunto, Mapa, etc. en Java. Estas colecciones tienen sus propios Iterador

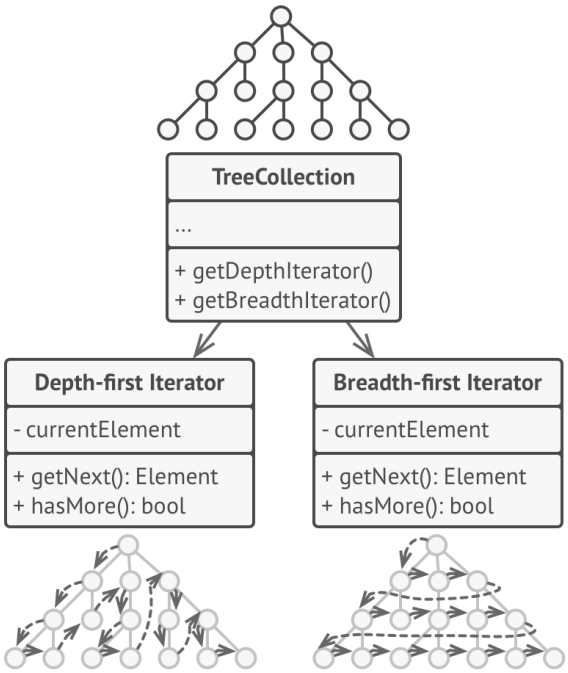


Varios tipos de colecciones.



La misma colección puede recorrerse de varias formas diferentes.

1. Los componentes del patrón son los siguientes:
    - Rol de iterador abstracto: este rol de resumen define la interfaz requerida para recorrer elementos
    - Implementación del iterador: implemente los métodos definidos en la interfaz del iterador, complete la iteración de la colección y mantenga la posición del cursor durante la iteración
    - Contenedor abstracto: generalmente es una interfaz que proporciona un método iterador (), como la interfaz de recopilación, la interfaz de lista, la interfaz de conjunto, etc. en Java, que implementan la interfaz Iterable
    - Contenedor concreto: es la clase de implementación concreta del contenedor abstracto, como la lista ordenada de la interfaz Lista implementa ArrayList, la lista vinculada de la interfaz Lista implementa LinkList, la lista hash de la interfaz Establecer implementa HashSet, etc.



Los iteradores implementan varios algoritmos de recorrido. Varios objetos iteradores pueden recorrer la misma colección al mismo tiempo.

Consecuencias:

Ventajas:Método transversal simplificado; ya sea una matriz o una lista, los usuarios solo necesitan iterar a través de iteradores; Podemos proporcionar una variedad de métodos de recorrido, por ejemplo, para listas ordenadas, podemos proporcionar un recorrido de orden positivo, un recorrido de orden inverso en dos iteradores; La encapsulación es buena, los usuarios solo necesitan que los iteradores atraviesen, sin tener que prestar atención a la implementación interna de su iteración.
Desventajas: para recorridos más simples (como matrices o listas ordenadas), iterar a través de iteradores es tedioso.

Aplicabilidad

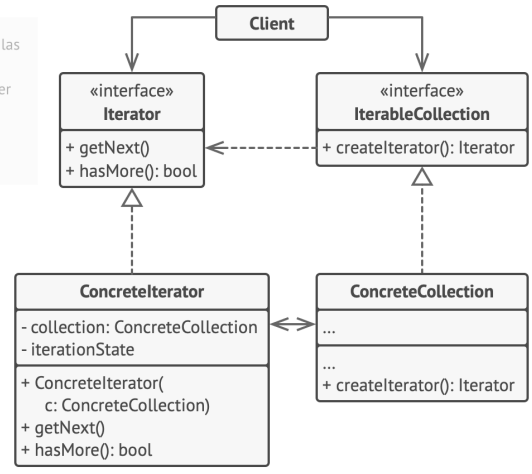
Es conveniente utilizar el patrón Iterador en las siguientes situaciones: Para acceder al contenido de un agregado sin exponer su representación interna Para permitir varios recorridos sobre un agregado Para proporcionar una interfaz uniforme para recorrer distintos tipos de agregados (esto es, permitir iteración polimórfica), Consecuencias Este patrón tiene tres consecuencias importantes: Soporta variaciones en el recorrido de una colección, puesto que estructuras complejas pueden requerir recorridos en muchas formas.

Si el algoritmo reside en el propio Iterador es más fácil usar el mismo sobre estructuras diferentes o diferentes algoritmos sobre la misma estructura, puede violarse el encapsulamiento, por necesidad de acceder a variables privadas de la estructura.

Resumen del patron

La iteración puede ser controlada de forma interna (controlada por el iterador), o de forma externa (controlada por el cliente). El algoritmo puede ser definido dentro de la colección, con lo cual el iterador es utilizado sólo como cursor de la posición actual o en la clase Iterator • Operaciones adicionales en el iterador - Se pueden adicionar operaciones a las ya básicas (First, Next, IsDone y CurrentItem) •Iteradores para tipos compuestos (Composite) •Iteradores nulos 2.5. Para permitir iterar polimórficamente sobre las diferentes implementaciones de colas, el iterador de la cola se complementa en términos de la interfaz e la clase abstracta Queue. Definen un método do: como un iterator interno el cual toma un bloque (es decir, un cierre) como parámetro.

Estructura



1 La interfaz **Iteradora** declara las operaciones necesarias para recorrer una colección: extraer el siguiente elemento, recuperar la posición actual, reiniciar la iteración, etc.

2 Los **Iteradores Concretos** implementan algoritmos específicos para recorrer una colección. El objeto iterador debe controlar el progreso del recorrido por su cuenta. Esto permite a varios iteradores recorrer la misma colección con independencia entre sí.

3 La interfaz **Colección** declara uno o varios métodos para obtener iteradores compatibles con la colección. Observa que el tipo de retorno de los métodos debe declararse como la interfaz iteradora de forma que las colecciones concretas puedan devolver varios tipos de iteradores.

4 Las **Colecciones Concretas** devuelven nuevas instancias de una clase iteradora concreta particular cada vez que el cliente solicita una. Puede que te estés preguntando: ¿dónde está el resto del código de la colección? No te preocupes, debe estar en la misma clase. Lo que pasa es que estos detalles no son fundamentales para el patrón en sí, por eso los omitimos.

5 El **Cliente** debe funcionar con colecciones e iteradores a través de sus interfaces. De este modo, el cliente no se acopla a clases concretas, permitiéndote utilizar varias colecciones e iteradores con el mismo código cliente.

Normalmente, los clientes no crean iteradores por su cuenta, en lugar de eso, los obtienen de las colecciones. Sin embargo, en algunos casos, el cliente puede crear uno directamente, como cuando define su propio iterador especial.

Ejemplo del patrón de diseño

iterators/ProfileIterator.java: Define la interfaz del perfil

```
package refactoring_guru.iterator.example.iterators;

import refactoring_guru.iterator.example.profile.Profile;

public interface ProfileIterator {
    boolean hasNext();

    Profile getNext();

    void reset();
}
```

iterators/FacebookIterator.java: Implementa la iteración por perfiles de Facebook

```
package refactoring_guru.iterator.example.iterators;

import refactoring_guru.iterator.example.profile.Profile;
import refactoring_guru.iterator.example.social_networks.Facebook;

import java.util.ArrayList;
import java.util.List;

public class FacebookIterator implements ProfileIterator {
    private Facebook facebook;
    private String type;
    private String email;
    private int currentPosition = 0;
    private List<String> emails = new ArrayList<>();
    private List<Profile> profiles = new ArrayList<>();

    public FacebookIterator(Facebook facebook, String type, String email) {
        this.facebook = facebook;
        this.type = type;
        this.email = email;
    }

    private void lazyLoad() {
        if (emails.size() == 0) {
            List<String> profiles = facebook.requestProfileFriendsFromFacebook(this.emai
            for (String profile : profiles) {
                this.emails.add(profile);
                this.profiles.add(null);
            }
        }
    }
```

```
    @Override
    public boolean hasNext() {
        lazyLoad();
        return currentPosition < emails.size();
    }

    @Override
    public Profile getNext() {
        if (!hasNext()) {
            return null;
        }

        String friendEmail = emails.get(currentPosition);
        Profile friendProfile = profiles.get(currentPosition);
        if (friendProfile == null) {
            friendProfile = facebook.requestProfileFromFacebook(friendEmail);
            profiles.set(currentPosition, friendProfile);
        }
        currentPosition++;
        return friendProfile;
    }

    @Override
    public void reset() {
        currentPosition = 0;
    }
}
```

iterators/LinkedInIterator.java: Implementa la iteración por perfiles de LinkedIn

```
package refactoring_guru.iterator.example.iterators;

import refactoring_guru.iterator.example.profile.Profile;
import refactoring_guru.iterator.example.social_networks.LinkedIn;

import java.util.ArrayList;
import java.util.List;

public class LinkedInIterator implements ProfileIterator {
    private LinkedIn linkedIn;
    private String type;
    private String email;
    private int currentPosition = 0;
    private List<String> emails = new ArrayList<>();
    private List<Profile> contacts = new ArrayList<>();

    public LinkedInIterator(LinkedIn linkedIn, String type, String email) {
        this.linkedIn = linkedIn;
        this.type = type;
        this.email = email;
    }

    private void lazyLoad() {
        if (emails.size() == 0) {
            List<String> profiles = linkedIn.requestRelatedContactsFromLinkedInAPI(this.
            for (String profile : profiles) {
                this.emails.add(profile);
                this.contacts.add(null);
            }
        }
    }
    @Override
    public boolean hasNext() {
        lazyLoad();
        return currentPosition < emails.size();
    }

    @Override
    public Profile getNext() {
        if (!hasNext()) {
            return null;
        }

        String friendEmail = emails.get(currentPosition);
        Profile friendContact = contacts.get(currentPosition);
        if (friendContact == null) {
            friendContact = linkedIn.requestContactInfoFromLinkedInAPI(friendEmail);
            contacts.set(currentPosition, friendContact);
        }
        currentPosition++;
        return friendContact;
    }

    @Override
    public void reset() {
        currentPosition = 0;
    }
}
```

Referencias

Santillan, D. (n.d.). Patron de diseño iterator. Slideshare.net. Retrieved February 5, 2022, from https://es.slideshare.net/dani13/patron-de-diseo-iterator-31722120.

programador clic. (n.d.). Programmerclick.Com. Retrieved February 5, 2022, from https://programmerclick.com/article/3482228142/.

Patrón iterador de serie de patrones de diseño: iteración sobre elementos en objetos agregados - Code World. (n.d.). Codetd.Com. Retrieved February 5, 2022, from https://www.codetd.com/es/article/10947578.

((Patrón iterador de serie de patrones de diseño: iteración sobre elementos en objetos agregados - Code World, n.d.).