refactoring.guru

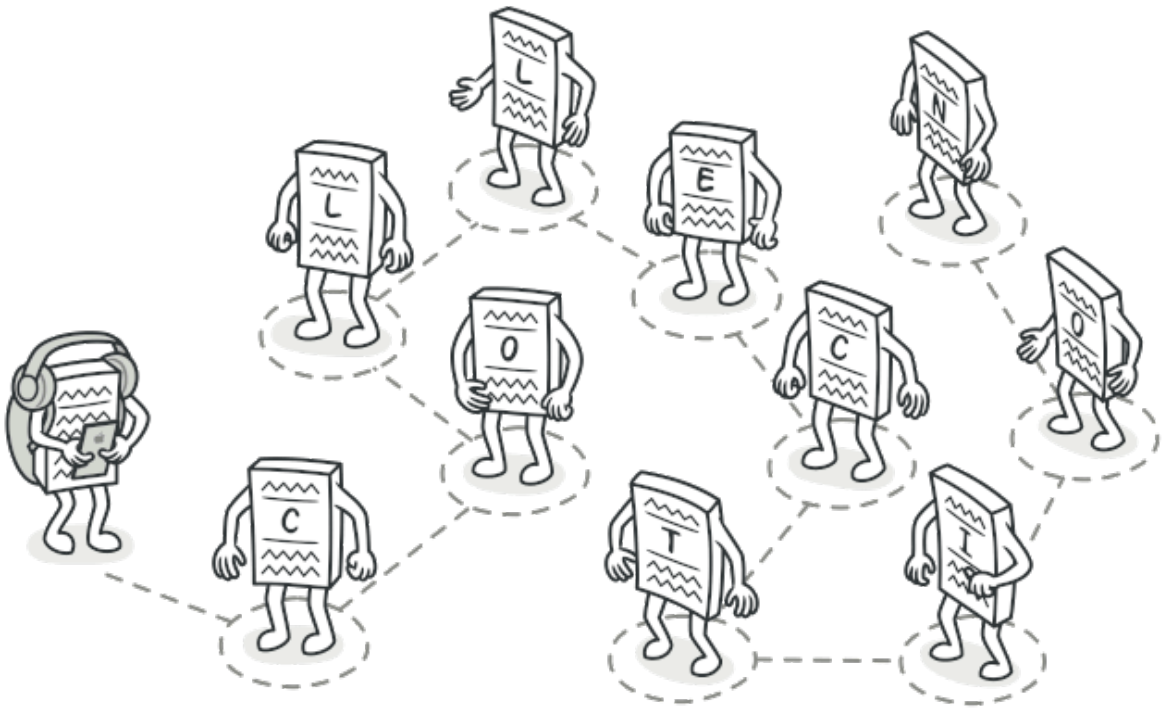
Iterator

14–19 minutes

Também conhecido como: Iterador

Propósito

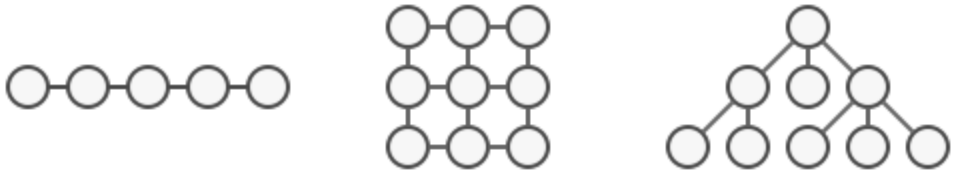
O **Iterator** é um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.)



Problema

Coleções são um dos tipos de dados mais usados em

programação. Não obstante, uma coleção é apenas um contêiner para um grupo de objetos.

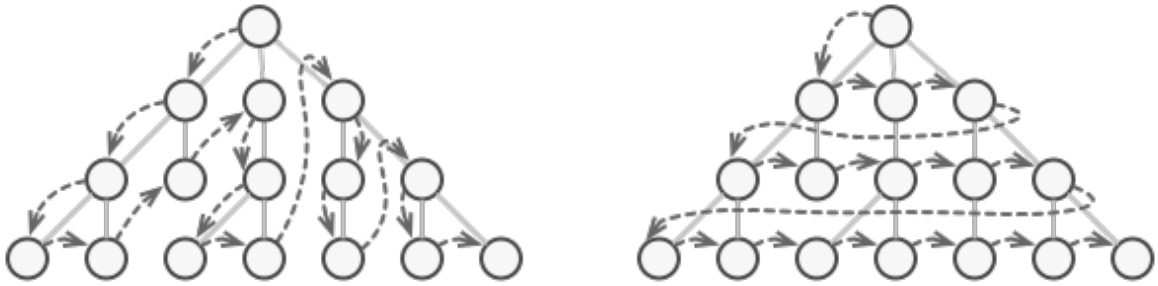


Vários tipos de coleções.

A maioria das coleções armazena seus elementos em listas simples. Contudo, alguns deles são baseados em pilhas, árvores, grafos, e outras estruturas complexas de dados.

Mas independente de quão complexa uma coleção é estruturada, ela deve fornecer uma maneira de acessar seus elementos para que outro código possa usá-los. Deve haver uma maneira de ir de elemento em elemento na coleção sem ter que acessar os mesmos elementos repetidamente.

Isso parece uma tarefa fácil se você tem uma coleção baseada numa lista. Você faz um loop em todos os elementos. Mas como você faz a travessia dos elementos de uma estrutura de dados complexas sequencialmente, tais como uma árvore. Por exemplo, um dia você pode apenas precisar percorrer em profundidade em uma árvore. No dia seguinte você pode precisar percorrer na amplitude. E na semana seguinte, você pode precisar algo diferente, como um acesso aleatório aos elementos da árvore.



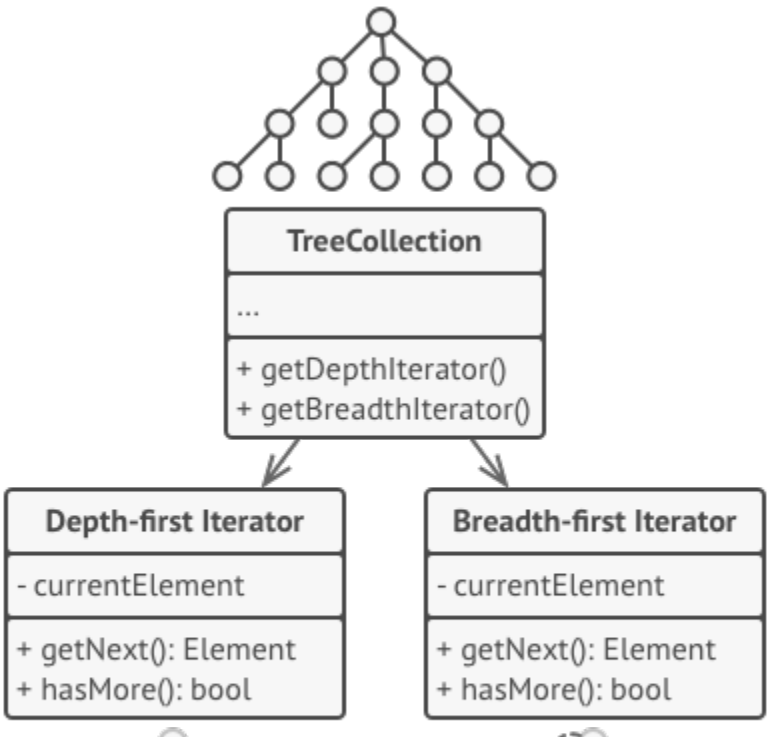
A mesma coleção pode ser atravessada de diferentes formas.

Adicionando mais e mais algoritmos de travessia para uma coleção gradualmente desfoca sua responsabilidade primária, que é um armazenamento de dados eficiente. Além disso, alguns algoritmos podem ser feitos para aplicações específicas, então incluí-los em uma coleção genérica pode ser estranho.

Por outro lado, o código cliente que deveria trabalhar com várias coleções pode não se importar com a maneira que elas armazenam seus elementos. Contudo, uma vez que todas as coleções fornecem diferentes maneiras de acessar seus elementos, você não tem outra opção além de acoplar seu código com as classes de coleções específicas.

Solução

A ideia principal do padrão Iterator é extrair o comportamento de travessia de uma coleção para um objeto separado chamado um *iterador*.



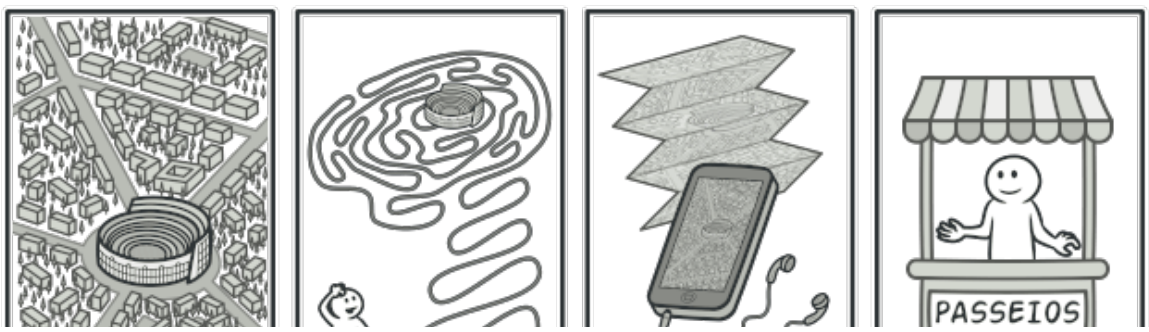
Iteradores implementam vários algoritmos de travessia. Alguns objetos iterador podem fazer a travessia da mesma coleção ao mesmo tempo.

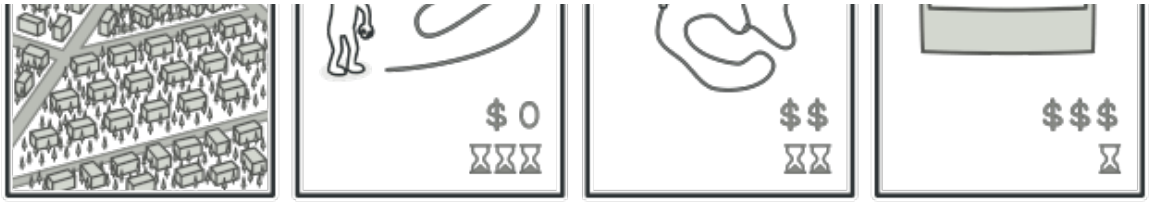
Além de implementar o algoritmo em si, um objeto iterador encapsula todos os detalhes da travessia, tais como a posição atual e quantos elementos faltam para chegar ao fim. Por causa disso, alguns iteradores podem averiguar a mesma coleção ao mesmo tempo, independentemente um do outro.

Geralmente, os iteradores fornecem um método primário para pegar elementos de uma coleção. O cliente pode manter esse método funcionando até que ele não retorne mais nada, o que significa que o iterador atravessou todos os elementos.

Todos os iteradores devem implementar a mesma interface. Isso faz que o código cliente seja compatível com qualquer tipo de coleção ou qualquer algoritmo de travessia desde que haja um iterador apropriado. Se você precisar de uma maneira especial para a travessia de uma coleção, você só precisa criar uma nova classe iterador, sem ter que mudar a coleção ou o cliente.

Analogia com o mundo real





Várias maneiras de se caminhar por Roma.

Você planeja visitar Roma por alguns dias e visitar todas suas principais atrações e pontos de interesse. Mas uma vez lá, você poderia gastar muito tempo andando em círculos, incapaz de achar até mesmo o Coliseu.

Por outro lado, você pode comprar um guia virtual na forma de app para seu smartphone e usá-lo como navegação. É inteligente e barato, e você pode ficar em lugares interessantes por quanto tempo quiser.

Uma terceira alternativa é gastar um pouco da verba da viagem e contratar um guia local que conhece a cidade como a palma de sua mão. O guia poderia ser capaz de criar um passeio que se adeque a seus gostos, mostrar todas as atrações, e contar um monte de histórias interessantes. Isso seria mais divertido, mas, infelizmente, mais caro também.

Todas essas opções—direções aleatórias criadas em sua cabeça, o navegador do smartphone, ou o guia humano—agem como iteradores sobre a vasta coleção de locais e atrações de Roma.

Estrutura

1. A interface **Iterador** declara as operações necessárias para percorrer uma coleção: buscar o próximo elemento, pegar a posição atual, recomeçar a iteração, etc.
2. **Iteradores Concretos** implementam algoritmos específicos para

percorrer uma coleção. O objeto iterador deve monitorar o progresso da travessia por conta própria. Isso permite que diversos iteradores percorram a mesma coleção independentemente de cada um.

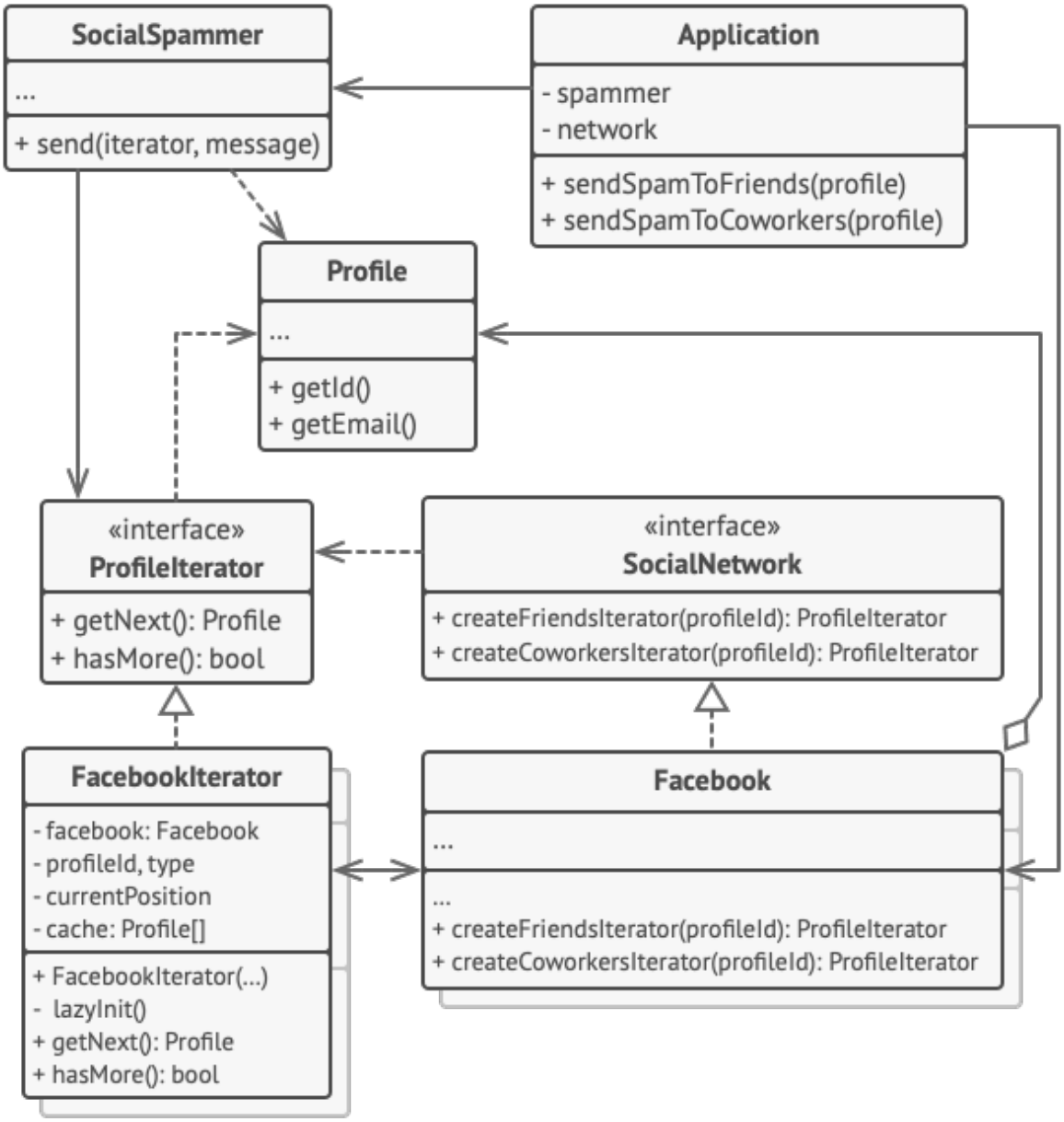
3. A interface **Coleção** declara um ou mais métodos para obter os iteradores compatíveis com a coleção. Observe que o tipo do retorno dos métodos deve ser declarado como a interface do iterador para que as coleções concretas possam retornar vários tipos de iteradores.
4. **Coleções Concretas** retornam novas instâncias de uma classe iterador concreta em particular cada vez que o cliente pede por uma. Você pode estar se perguntando, onde está o resto do código da coleção? Não se preocupe, ele deve ficar na mesma classe. É que esses detalhes não são cruciais para o padrão atual, então optamos por omiti-los.
5. O **Cliente** trabalha tanto com as coleções como os iteradores através de suas interfaces. Dessa forma o cliente não fica acoplado a suas classes concretas, permitindo que você use várias coleções e iteradores com o mesmo código cliente.

   Tipicamente, os clientes não criam iteradores por conta própria, mas ao invés disso os obtêm das coleções. Ainda assim, em certos casos, o cliente pode criar um diretamente; por exemplo, quando o cliente define seu próprio iterador especial.

Pseudocódigo

Neste exemplo, o padrão **Iterator** é usado para percorrer uma coleção especial que encapsula acesso ao grafo social do Facebook. A coleção fornece vários iteradores que podem

percorrer perfis de várias maneiras.



Exemplo de uma iteração sobre perfis sociais.

O iterador ‘amigos’ pode ser usado para verificar os amigos de um dado perfil. O iterador 'colegas’ faz a mesma coisa, exceto por omitir amigos que não trabalham na mesma companhia como uma pessoa alvo. Ambos iteradores implementam uma interface comum que permite os clientes recuperar os perfis sem mergulhar nos detalhes de implementação como autenticação e pedidos REST.

O código cliente não está acoplado à classes concretas porque funciona com coleções e iteradores somente através de interfaces. Se você decidir conectar sua aplicação com uma nova rede social, você simplesmente precisa fornecer as novas classes de iteração e coleção sem mudar o código existente.

```
// A interface da coleção deve declarar um método fábrica para
// produzir iteradores. Você pode declarar vários métodos se há
// diferentes tipos de iteração disponíveis em seu programa.
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator


// Cada coleção concreta é acoplada a um conjunto de classes
// iterador concretas que ela retorna. Mas o cliente não é, uma
// vez que a assinatura desses métodos retorna interfaces de
// iterador.
class Facebook implements SocialNetwork is
    // ...o grosso do código da coleção deve vir aqui...

    // Código de criação do iterador.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")


// A interface comum a todos os iteradores.
interface ProfileIterator is
```

```
    method getNext():Profile
    method hasMore():bool


// A classe iterador concreta.
class FacebookIterator implements ProfileIterator is
    // O iterador precisa de uma referência para a coleção que
    // ele percorre.
    private field facebook: Facebook
    private field profileId, type: string

    // Um objeto iterador percorre a coleção independentemente
    // de outros iteradores. Portanto ele tem que armazenar o
    // estado de iteração.
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
            cache = facebook.socialGraphRequest(profileId, type)

    // Cada classe iterador concreta tem sua própria
    // implementação da interface comum do iterador.
    method getNext() is
        if (hasMore())
```

```
            result = cache[currentPosition]
            currentPosition++
            return result

        method hasMore() is
            lazyInit()
            return currentPosition < cache.length


// Aqui temos outro truque útil: você pode passar um iterador
// para uma classe cliente ao invés de dar acesso a ela à uma
// coleção completa. Dessa forma, você não expõe a coleção ao
// cliente.
//
// E tem outro benefício: você pode mudar o modo que o cliente
// trabalha com a coleção no tempo de execução ao passar a ele
// um iterador diferente. Isso é possível porque o código
// cliente não é acoplado às classes iterador concretas.
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)


// A classe da aplicação configura coleções e iteradores e então
// os passa ao código cliente.
class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer
```

```
method config() is
    if working with Facebook
        this.network = new Facebook()
    if working with LinkedIn
        this.network = new LinkedIn()
    this.spammer = new SocialSpammer()

method sendSpamToFriends(profile) is
    iterator = network.createFriendsIterator(profile.getId())
    spammer.send(iterator, "Very important message")

method sendSpamToCoworkers(profile) is
    iterator = network.createCoworkersIterator(profile.getId())
    spammer.send(iterator, "Very important message")
```

Aplicabilidade

Utilize o padrão Iterator quando sua coleção tiver uma estrutura de dados complexa por debaixo dos panos, mas você quer esconder a complexidade dela de seus clientes (seja por motivos de conveniência ou segurança).

O iterador encapsula os detalhes de se trabalhar com uma estrutura de dados complexa, fornecendo ao cliente vários métodos simples para acessar os elementos da coleção. Embora essa abordagem seja muito conveniente para o cliente, ela também protege a coleção de ações descuidadas ou maliciosas que o cliente poderia fazer se estivesse trabalhando com as coleções diretamente.

Utilize o padrão para reduzir a duplicação de código de travessia em sua aplicação.

O código de algoritmos de iteração não triviais tendem a ser muito pesados. Quando colocados dentro da lógica de negócio da aplicação, ele pode desfocar a responsabilidade do codigo original e torná-lo um código de difícil manutenção. Mover o código de travessia para os iteradores designados pode ajudar você a tornar o código da aplicação mais enxuto e limpo.

Utilize o Iterator quando você quer que seu código seja capaz de percorrer diferentes estruturas de dados ou quando os tipos dessas estruturas são desconhecidos de antemão.

O padrão fornece um par de interfaces genérica tanto para coleções como para iteradores. Já que seu código agora usa essas interfaces, ele ainda vai funcionar se você passar diversos tipos de coleções e iteradores que implementam essas interfaces.

Como implementar

1. Declare a interface do iterador. Ao mínimo, ela deve ter um método para buscar o próximo elemento de uma coleção. Mas por motivos de conveniência você pode adicionar alguns outros métodos, tais como recuperar o elemento anterior, saber a posição atual, e checar o fim da iteração.
2. Declare a interface da coleção e descreva um método para buscar iteradores. O tipo de retorno deve ser igual à interface do iterador. Você pode declarar métodos parecidos se você planeja ter grupos distintos de iteradores.
3. Implemente classes iterador concretas para as coleções que você quer percorrer com iteradores. Um objeto iterador deve ser ligado

com uma única instância de coleção. Geralmente esse link é estabelecido através do construtor do iterador.

4. Implemente a interface da coleção na suas classes de coleção. A ideia principal é fornecer ao cliente com um atalho para criar iteradores, customizados para uma classe coleção em particular. O objeto da coleção deve passar a si mesmo para o construtor do iterador para estabelecer um link entre eles.
5. Vá até o código cliente e substitua todo o código de travessia de coleção com pelo uso de iteradores. O cliente busca um novo objeto iterador a cada vez que precisa iterar sobre os elementos de uma coleção.

Prós e contras

- *Princípio de responsabilidade única*. Você pode limpar o código cliente e as coleções ao extrair os pesados algoritmos de travessia para classes separadas.
- *Princípio aberto/fechado*. Você pode implementar novos tipos de coleções e iteradores e passá-los para o código existente sem quebrar coisa alguma.
- Você pode iterar sobre a mesma coleção em paralelo porque cada objeto iterador contém seu próprio estado de iteração.
- Pelas mesmas razões, você pode atrasar uma iteração e continuá-la quando necessário.
- Aplicar o padrão pode ser um preciosismo se sua aplicação só trabalha com coleções simples.
- Usar um iterador pode ser menos eficiente que percorrer elementos de algumas coleções especializadas diretamente.

Relações com outros padrões

- Você pode usar Iteradores para percorrer árvores Composite.
- Você pode usar o Factory Method junto com o Iterator para permitir que uma coleção de subclasses retornem diferentes tipos de iteradores que são compatíveis com as coleções.
- Você pode usar o Memento junto com o Iterator para capturar o estado de iteração atual e revertê-lo se necessário.
- Você pode usar o Visitor junto com o Iterator para percorrer uma estrutura de dados complexas e executar alguma operação sobre seus elementos, mesmo se eles todos tenham classes diferentes.