Iterator

Padrões Comportamentais

O padrão de projeto *Iterator* fornece uma maneira de acessar, sequencialmente, os elementos de um objeto agregado sem expor a sua representação subjacente.

Motivação (Por que utilizar?)

Um grupo de objetos pode ser chamado de coleção ou agregados. Tais objetos podem estar armazenados em diferentes estruturas de dados, como listas, matrizes, pilhas, árvores entre outras.

É interessante que forneçam uma maneira de acessar seus elementos sem expor sua estrutura de dados interna. O padrão de projeto *Iterator* torna isso possível. A idéia principal deste padrão é assumir a responsabilidade pela iteração de ponta a ponta sobre os elementos de objetos agregados. Isso é feito por meio de uma classe *Iterator*. Tal classe implementa uma interface que dita quais métodos são necessários para acessar os elementos de diferentes objetos agregados.

Lista ($lista)

Objeto 1	Objeto 2	Objeto 3	Objeto 4	Objeto 5	Objeto 6	Objeto 7	Objeto 8	Objeto 9
0	1	2	3	4	5	6	7	8

Matriz ($matriz)

	0	1	2
0	Objeto 1	Objeto 2	Objeto 3
1	Objeto 4	Objeto 5	Objeto 6
2	Objeto 7	Objeto 8	Objeto 9

Exemplo de agregados

Acima estão ilustrados dois tipos diferentes de objetos agregados, uma lista e uma Matriz. A iteração nestas duas estruturas de dados se dá de maneiras distintas, visto que a lista é unidimensional e a matriz é bidimensional.

Para iterar sobre os elementos da ***$lista*** podemos utilizar um código parecido com o abaixo:

```php
$tamanho = 9; //Tamanho da lista

//Para cada elemento da lista
for ($i = 0; $i < $tamanho; $i++) {
    echo $lista[$i] . ' '; //Imprima o item;
}
```

Já para iterar sobre a ***$matriz*** é necessário algo como:

```php
$qtdLinhas = 3; //Quantidade de linhas da matriz.
$qtdColunas = 3; //Quantidade de colunas da matriz.

//Para cada linha da matriz.
for ($linha = 0; $linha < $qtdLinhas; $linha++) {
    //Para cada coluna da matriz.
    for ($coluna = 0; $coluna < $qtdColunas; $coluna++) {
        echo $matriz[$linha][$coluna] . ' '; //Imprima o item;
    }
}
```

Os códigos acima são diferentes. Tudo bem os utilizar, pois trata-se de estruturas de dados simples (uma lista e uma matriz), mas repare que para acessar os dados de forma iterativa é necessário conhecer sua a estrutura interna, ou seja, a forma que os dados estão organizados. Isso deixa a **$lista** e **$matriz** totalmente expostas aos clientes que as utilizam.

Como poderíamos, com apenas um código, iterar sobre a lista e a matriz, e além disso ainda ocultar suas estruturas internas para o cliente?

O padrão *Iterator* tem como objetivo unificar e encapsular a iteração em agregados. Isso é possível por meio de uma interface comum implementada por classes do tipo *Iterator*, que por sua vez, encapsulam a forma de iteração de cada tipo de objeto agregado, expondo métodos padronizados independente da estrutura de dados do objeto agregado sobre qual atuam.

A interface *iterator* exige que uma classe implemente dois métodos:

```php
interface Iterator
{
    //Retorna um boolean que indica se existe um próximo elemento.
    public function hasNext(): bool;

    //Retorna o próximo elemento.
    public function next();
}
```

Obs.: Foi utilizado o nome `Iterator` *devido ao fato do PHP já possuir uma interface Iterator nativa.*

Vamos criar uma classe que gerencie uma lista utilizando o padrão *Iterator*.

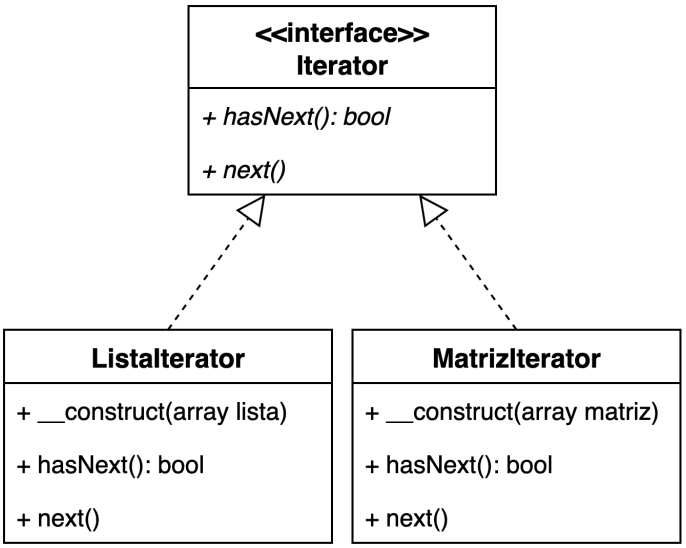


Diagrama de classes dos Iterators

```php
class ListaIterator implements Iterator
{
    private array $lista;
    private int $indice = 0;
    private int $tamanho;

    public function __construct(array $lista)
    {
        $this->lista = $lista;
        //Conta os elementos da lista.
        $this->tamanho = count($lista);
    }

    //Retorna um boolean que indica se existe um próximo elemento.
    public function hasNext(): bool
    {
        if ($this->indice >= $this->tamanho) {
            return false;
        }
        return true;
    }

    //Retorna o próximo elemento.
    public function next()
    {
        $item = $this->lista[$this->indice];
        $this->indice = $this->indice + 1;
        return $item;
    }
}
```

Agora o cliente não pode e nem precisa conhecer a estrutura interna de **ListaIterator** para poder iterar sobre seus elementos. O cliente apenas precisa utilizar os métodos **hasNext()** e **next()** disponibilizados pela classe.

Vamos ver agora como implementar um *Iterator* para a matriz.

```php
class MatrizIterator implements Iterator
{
    private array $matriz;
    private int $indiceLinha = 0;
    private int $indiceColuna = 0;
    private int $qtdLinhas;
    private int $qtdColunas;

    public function __construct(array $matriz)
    {
        $this->matriz = $matriz;
        //conta os arrays dentro da matriz.
        $this->qtdLinhas = count($matriz);
        //Conta a quantidade de elementos dentro do primeiro array da matriz.
        $this->qtdColunas = count($matriz[0]);
    }

    public function incrementarIndice()
    {
        //Se chegou ao limite de colunas.
        if ($this->indiceColuna >= $this->qtdColunas) {
            //Incrementa a linha.
            $this->indiceLinha = $this->indiceLinha + 1;
            //Volta a coluna para zero.
            $this->indiceColuna = 0;
        } else {
            //Apenas incrementa a coluna.
            $this->indiceColuna = $this->indiceColuna + 1;
        }
    }

    //Retorna um boolean que indica se existe um próximo elemento.
    public function hasNext(): bool
    {
        if ($this->indiceLinha > ($this->qtdLinhas - 1)) {
            return false;
        }
        return true;
    }

    //Retorna o próximo elemento.
    public function next()
    {
        $item = $this->matriz[$this->indiceLinha][$this->indiceColuna];
        $this->incrementarIndice();
        return $item;
    }
}
```

Repare que toda a lógica de iteração sobre os itens ficou sob a responsabilidade de cada classe. O cliente pode utilizar um único código para iterar sobre os elementos tanto da **$lista** quanto da **$matriz.**

Vamos criar uma pequena classe que itera sobre os elementos de um *Iterator*.

```
class ImpressoraDeAgregados
{
    public static function iterar(Iterator $iterator)
    {
        //Enquanto existir um próximo elemento.
        while ($iterator->hasNext()) {
            //imprima o elemento concatenado a um espaço em branco.
            echo ($iterator->next()) . " ";
        }
    }
}
```

Hora de testar.

```
//Vamos criar a lista
$lista = [1, 2, 3, 4, 5, 6, 7, 8, 9];

//e a matriz
$matriz = [0 => [1, 2, 3], 1 => [4, 5, 6], 2 => [7, 8, 9]];

//criar o Iterator da lista
$listaIterator = new Lista($lista);

//criar o iterator da matriz
$matrizIterator = new Matriz($matriz);

//iterar sobre a lista
echo "Elementos da Lista: <br>";
ImpressoraDeAgregados::iterar($listaIterator);

//iterar sobre a matriz
echo "<br><br>Elementos da Matriz: <br>";
ImpressoraDeAgregados::iterar($matrizIterator);
```

Saída:

```
Elementos da Lista:
1 2 3 4 5 6 7 8 9

Elementos da Matriz:
1 2 3 4 5 6 7 8 9
```

Já temos os *Iterators* implementados para a lista e para a matriz, mas como pode ser observado no teste acima o cliente ainda está criando listas e matrizes nativas. Embora o cliente não precise conhecer suas estruturas internas para fazer a iteração, ele ainda precisa conhecê-las para fazer a inserção de elementos.

O padrão Iterator diz que os agregados também devem implementar uma interface comum, isso permite que o cliente possa esperar um supertipo ao invés de um objeto (agregado) concreto.

Vamos criar a interface para os agregados:

```
interface Agregado
{
    //Retorna o iterator do agregado.
    public function createIterator(): Iterator;
}
```

Vamos criar os agregados propriamente ditos. Poderíamos implementar nos agregados métodos para diversas finalidades, tais comos:

- Adicionar um item;
- Recuperar um item em uma posição específica;
- Remover um item;
- Atualizar um item;
- Recuperar tamanho do agregado;

Porém nosso objetivo é entender o padrão *Iterator*, por isso vamos manter o código simples e implementar somente o necessário, serão eles os métodos para adicionar e recuperar itens e também métodos para recuperar o tamanho do agregado.

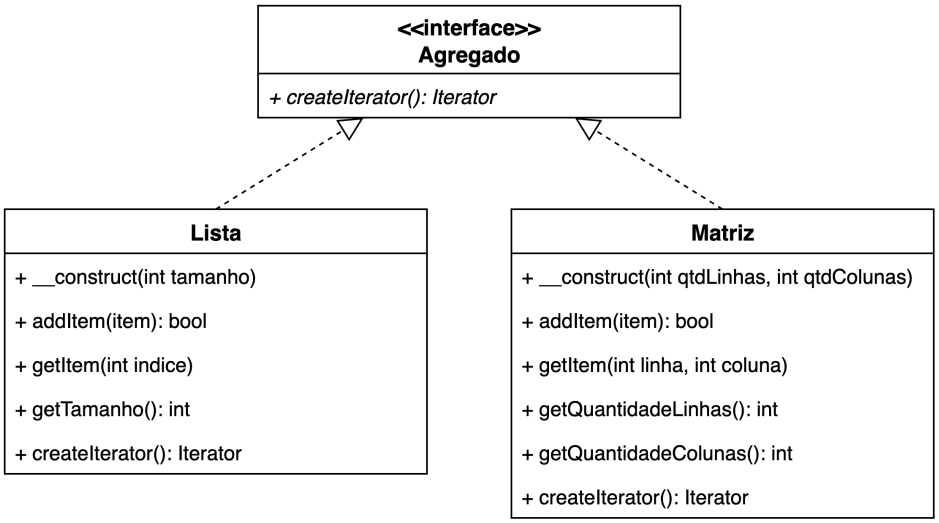


Diagrama de classe dos agregados

```php
class Lista implements Agregado
{
    private array $lista = [];
    private int $tamanho;

    //Recebe o tamanha máximo da lista no construtor
    public function __construct(int $tamanhoMaximo)
    {
        $this->tamanho = $tamanhoMaximo;
    }

    //Se ainda couber, aciona um item no fim da lista
    public function addItem($item): bool
    {
        if (count($this->lista) <= ($this->tamanho - 1)) {
            $this->lista[] = $item;
            return true;
        }
        return false;
    }

    //Retorna o item do índice passado por parâmetro
    public function getItem(int $indice)
    {
        return $this->lista[$indice];
    }

    //Retorna o Tamanha da lista
    public function getTamanho(): int
    {
        return $this->tamanho;
    }

    //Retorna o Iterator da Lista
    public function createIterator(): Iterator
    {
        return new ListaIterator($this);
    }
}
```

```php
class Matriz implements Agregado
{
    private array $matriz = [];
    private int $quantidadeLinhas;
    private int $quantidadeColunas;
    private int $linhaAtual = 0;
    private int $colunaAtual = 0;

    //Recebe no construtor a quantidade máxima de linhas e colunas
    public function __construct(int $quantidadeLinhas, int $quantidadeColunas)
    {
        $this->quantidadeLinhas = $quantidadeLinhas;
        $this->quantidadeColunas = $quantidadeColunas;
    }

    //Recebe um item de qualquer tipo e o insere na matriz
    public function addItem($item): bool
    {
        //Se chegou na última linha e última coluna
        if (
            $this->linhaAtual == ($this->quantidadeLinhas - 1)
            &&$this->colunaAtual == $this->quantidadeColunas
        ) {
            return false;
        }

        //Se chegou na última coluna
        if ($this->colunaAtual == ($this->quantidadeColunas)) {
            $this->linhaAtual = $this->linhaAtual + 1; //Incrementa a linha
            $this->colunaAtual = 0; //Zera coluna
        }

        //Insere o Item
        $this->matriz[$this->linhaAtual][$this->colunaAtual] = $item;
        $this->colunaAtual = $this->colunaAtual + 1; //Incrementa Coluna
        return true;
    }

    //Retorna o item que estiver na linha e coluna passadas por parâmetro
    public function getItem(int $linha, int $coluna)
    {
        return $this->matriz[$linha][$coluna];
    }

    //Retorna a quantidade de linhas da Matriz
    public function getQuantidadeLinhas(): int
    {
        return $this->quantidadeLinhas;
    }

    //Retorna a quantidade de colunas da matriz
    public function getQuantidadeColunas(): int
    {
        return $this->quantidadeColunas;
    }

    //Retorna o Itertor da Matriz
    public function createIterator(): Iterator
    {
        return new MatrizIterator($this);
    }
}
```

Precisamos agora adaptar os *iterators* **ListaIterator** e **MatrizIterator** para atuar sobre os agregados que acabamos de criar.

```
class ListaIterator implements Iterator
{
    private Lista $lista; //Agora espera uma instância de Lista
    private int $indice = 0;
    private int $tamanho;

    //Agora recebe uma instância de Lista por parâmetro
    public function __construct(Lista $lista)
    {
        $this->lista = $lista;
        //utiliza o método getTamanho() ao invés de count()
        $this->tamanho = $lista->getTamanho();
    }

    //Retorna um boolean que indica se existe um próximo elemento.
    public function hasNext(): bool
    {
        if ($this->indice >= $this->tamanho) {
            return false;
        }
        return true;
    }

    //Retorna o próximo elemento.
    public function next()
    {
        //utiliza o método getItem() ao invés de recuperar o item direto do array.
        $item = $this->lista->getItem($this->indice);
        $this->indice = $this->indice + 1;
        return $item;
    }
}
```

```php
class MatrizIterator implements Iterator
{
    private Matriz $matriz; //Agora espera uma instância de Matriz
    private int $indiceLinha = 0;
    private int $indiceColuna = 0;
    private int $quantidadeLinhas;
    private int $quantidadeColunas;

    //Agora recebe uma instância de Matriz por parâmetro
    public function __construct(Matriz $matriz)
    {
        $this->matriz = $matriz;
        //utiliza o método getQuantidadeLinhas() ao invés de count()
        $this->quantidadeLinhas = $matriz->getQuantidadeLinhas();
        //utiliza o método getQuantidadeColunas() ao invés de count()
        $this->quantidadeColunas = $matriz->getQuantidadeColunas();
    }

    public function incrementarIndice()
    {
        //Se chegou ao limite de colunas.
        if ($this->indiceColuna == ($this->quantidadeColunas - 1)) {
            //Incrementa a linha.
            $this->indiceLinha = $this->indiceLinha + 1;
            //Volta a coluna para zero.
            $this->indiceColuna = 0;
        } else {
            //Apenas incrementa a coluna.
            $this->indiceColuna = $this->indiceColuna + 1;
        }
    }

    //Retorna um boolean que indica se existe um próximo elemento.
    public function hasNext(): bool
    {
        if ($this->indiceLinha > ($this->quantidadeLinhas - 1)) {
            return false;
        }
        return true;
    }

    //Retorna o próximo elemento.
    public function next()
    {
        //utiliza o método getItem() ao invés de recuperar o item direto do array.
        $item = $this->matriz->getItem($this->indiceLinha, $this->indiceColuna);
        $this->incrementarIndice();
        return $item;
    }
}
```

Os agregados implementam a mesma interface, portanto podemos criar uma nova impressora de agregados que depende apenas desta interface.

```
class ImpressoraDeAgregados
{
    public Agregado $agregado;

    //Define um agregado ao cliente em tempo de execução.
    public function setAgregado(Agregado $agregado)
    {
        $this->agregado = $agregado;
    }

    public function iterar()
    {
        //Cria o iterador do agregado.
        $iterator = $this->agregado->createIterator();
        //Enquanto existir um próximo elemento.
        while ($iterator->hasNext()) {
            //imprima o elemento concatenado a um espaço em branco.
            echo $iterator->next() . " ";
        }
    }
}
```

Vamos testar a nova impressora de agregados.

```
//Criamos uma lista com tamanho máximo de 9 elementos
$lista = new Lista(9);

//Adicionamos 10 itens a ela, porém não aceitará mais que 9.
$lista->addItem(1);
$lista->addItem(2);
$lista->addItem(3);
$lista->addItem(4);
$lista->addItem(5);
$lista->addItem(6);
$lista->addItem(7);
$lista->addItem(8);
$lista->addItem(9);
$lista->addItem(10);

//Criamos de uma matriz que possui 3 linhas e 3 colunas.
$matriz = new Matriz(3, 3);
//Adicionamos 10 itens a ela, porém não aceitará mais que 9.
$matriz->addItem(1);
$matriz->addItem(2);
$matriz->addItem(3);
$matriz->addItem(4);
$matriz->addItem(5);
$matriz->addItem(6);
$matriz->addItem(7);
$matriz->addItem(8);
$matriz->addItem(9);
$matriz->addItem(10);

//Criamos uma impressora de Agregados
$cliente = new ImpressoraDeAgregados();
```

```php
//Passamos a lista para ele e fazemos a iteração
$cliente->setAgregado($lista);
echo "Elementos da Lista: <br>";
$cliente->iterar();

//Agora passamos a matriz para ele e fazemos a iteração
$cliente->setAgregado($matriz);
echo "<br><br>Elementos da Matriz: <br>";
$cliente->iterar();
```

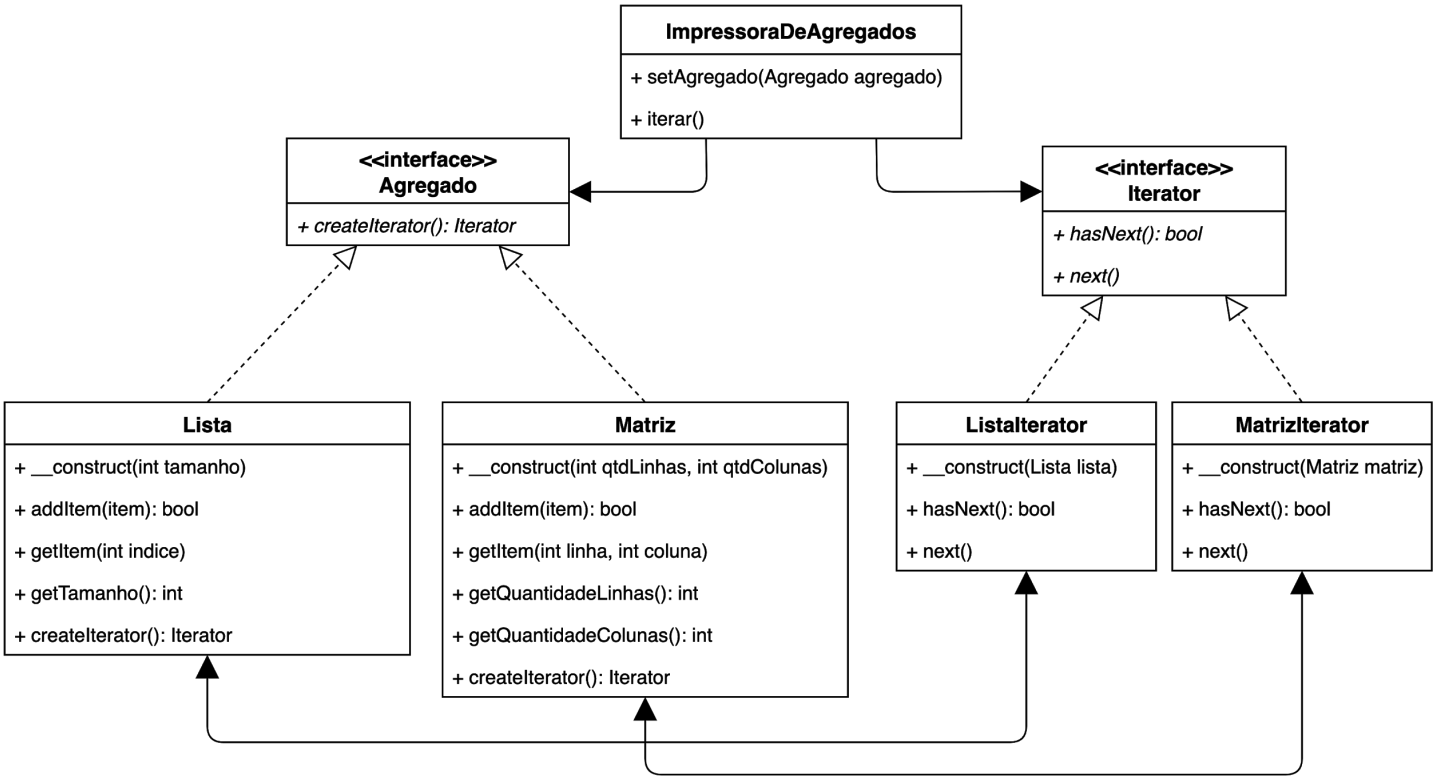
Saída:

```
Elementos da Lista:
1 2 3 4 5 6 7 8 9

Elementos da Matriz:
1 2 3 4 5 6 7 8 9
```

Toda a responsabilidade de como iterar sobre os elementos são das classes que implementam o **Iterator** que são **ListaIterator** e **MatrizIterator**, essa responsabilidade deixa de ser do cliente. Da mesma forma a criação do *Iterator* e a adição de elementos passam a ser responsabilidade das classe **Lista** e **Matriz** que implementam a interface **Agregado**.

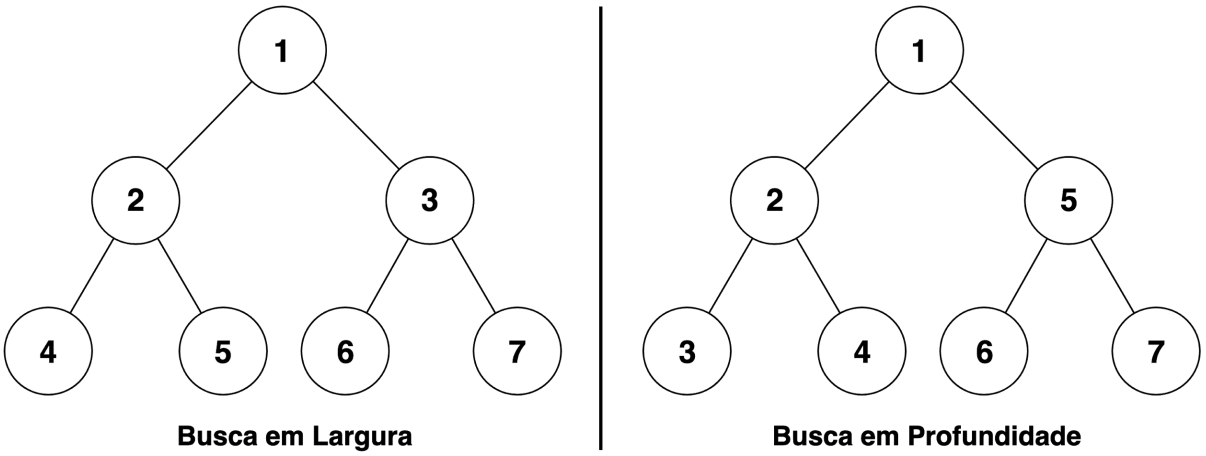
Com o *Iterator* é possível implementar novos tipos de agregados, de modo que ao passá-los para o cliente, nada precise ser modificado nele.



Cada agregado precisa retornar um *Iterador* Concreto e cada *Iterator* concreto gerência um agregado.

Diagrama de classe dos agregados

No nosso exemplo foram utilizadas estruturas de dados e iterações simples, porém o iterator também pode ser utilizado para iterar sobre estruturas de dados mais complexas, como um grafo por exemplo, fazendo uma busca em largura ou busca em profundidade.



Exemplo de dois tipos de interações sobre grafos

Aplicabilidade (Quando utilizar?)

- Quando é necessário acessar o conteúdo de um objeto agregado sem expor sua representação interna.
- Quando é preciso suportar vários tipos de iteração em objetos agregados.
- Quando é necessário fornecer uma interface uniforme para iterar sobre diferentes estruturas agregadas (ou seja, para suportar iteração polimórfica).

Componentes

- **Cliente:** Precisa iterar sobre os objetos agregados.
- **Agregado:** Define uma interface para criação de um objeto *Iterator*. O cliente espera essa interface ao invés de agregados concretos.
- **AgregadoConcreto:** Possui uma coleção de objetos e implementa o método que retorna um objeto do tipo *Iterator* referente a sua coleção.
- **Iterator:** Fornece a interface que todos os iteradores concretos devem implementar, bem como um conjunto de métodos para acessar os elementos de um agregado.

- **IteratorConcreto:** Implementa a interface a interface de Iterator e mantém o controle da posição corrente no percurso do agregado.

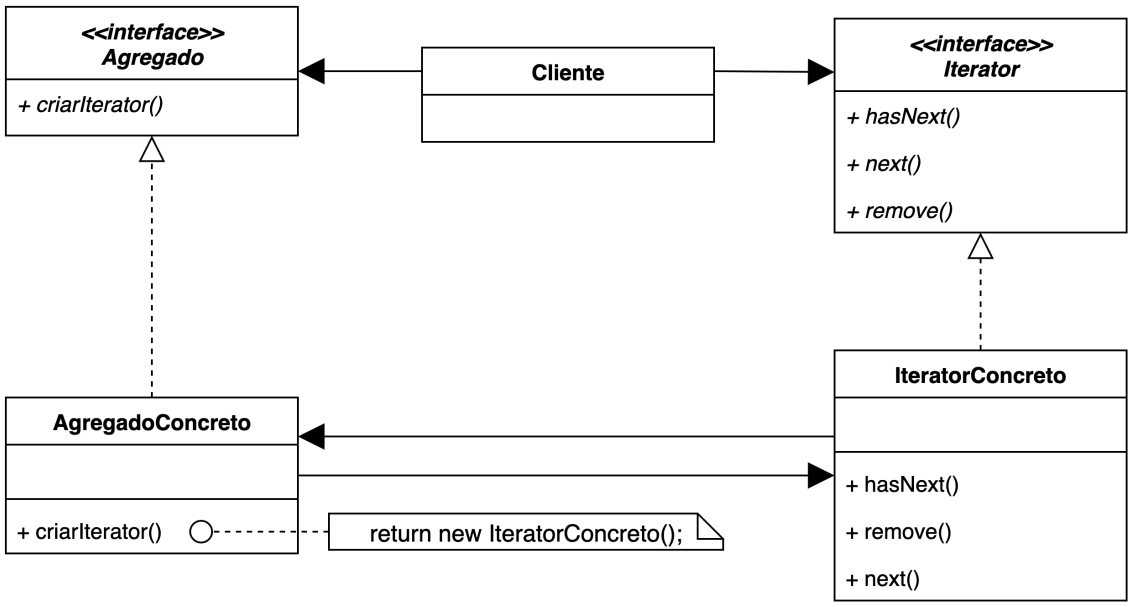


Diagrama de Classes

Consequências

- Suporte a variações no percurso de um agregado: Agregados complexos podem ser percorridos de várias maneiras, encapsulando e simplificando sua iteração para objetos externos (Clientes).

- Simplificação da interface dos agregados: Já que os *Iterators* são os responsáveis pelas iterações, os agregados não precisam implementar métodos para tal finalidade.

- Padronização da forma como é feita a iteração em objetos agregados de diferentes tipos (iteração polimórfica).

- Um agregado pode sofrer mais de uma iteração ao mesmo tempo: Um *iterator* controla seu próprio estado de iteração. Portanto, podem existir mais de uma iteração sendo executada sobre o mesmo objeto agregado ao mesmo tempo.

- O código passa a seguir o *"princípio de responsabilidade única"*. Toda a responsabilidade de como iterar sobre os elementos é da classe que implementa o *Iterator.* Essa responsabilidade deixa de ser do cliente e do agregado.

- O código também passa a seguir o *"Princípio Aberto/Fechado"*. É possível implementar novos tipos de agregados/coleções seguindo o padrão *Iterator,* ao passá-los para o cliente existente nada precisará ser modificado nele.