INF011 – Padrões de Projeto

19 – *Iterator*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

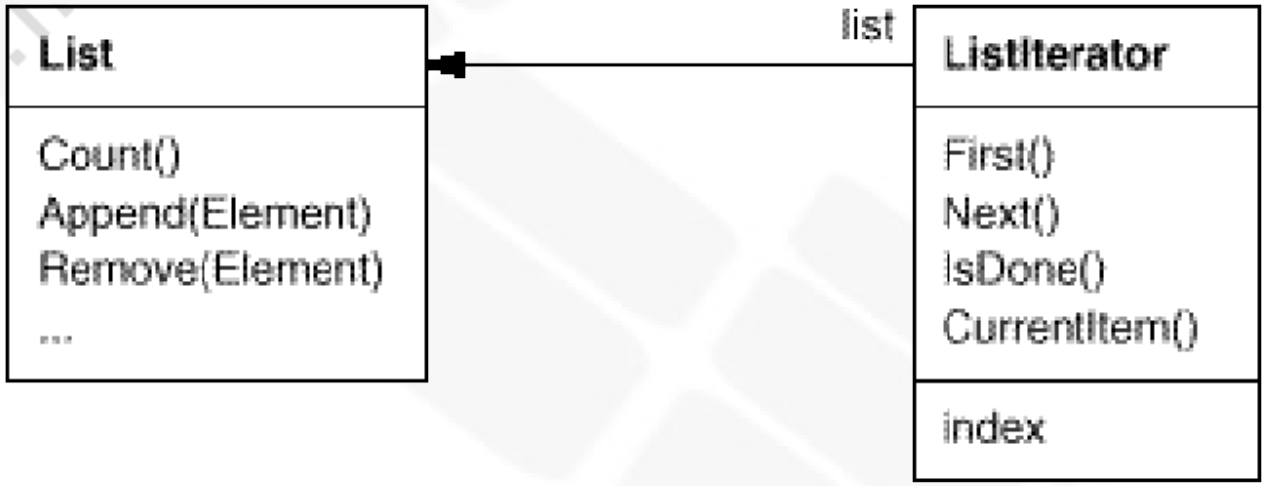
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Iterator

- Propósito:
  - Disponibilizar uma forma de acesso sequencial aos elementos de um agregado sem expor a sua representação subjacente
- Também conhecido como: *Cursor*
- Motivação:
  - Deve-se poder percorrer uma lista encadeada, por exemplo, sem conhecer sua estrutura interna
  - Podem existir diferentes formas de varredura e não deseja-se que todo este código esteja na classe da lista
  - Podem existir diferentes varreduras simultâneas

Iterator

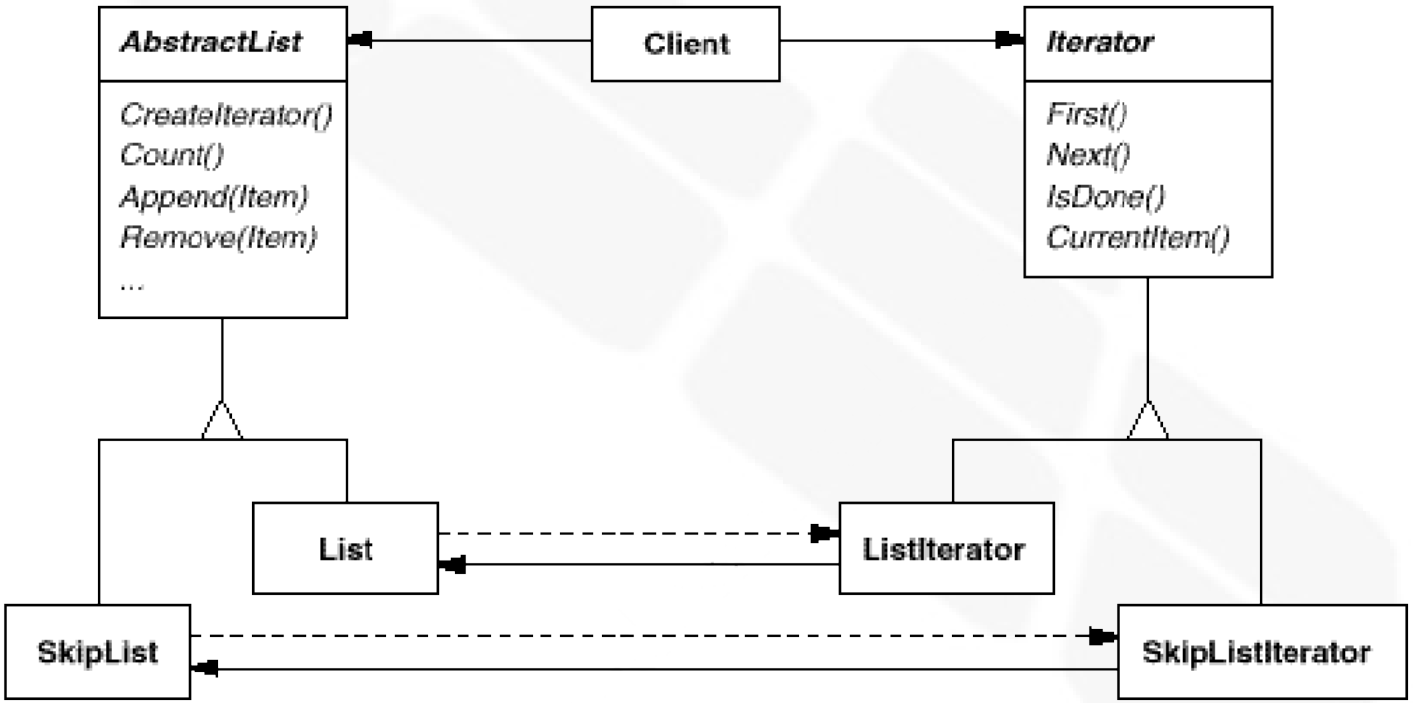
- Motivação:



  - O *Iterator* define a interface de acesso aos elementos da lista e armazena o elemento atual
  - Pode-se definir *iterators* para diferentes políticas de varredura sem enumerá-los na classe *List*. Ex: *FilteringListIterator*
  - A dependência explícita com o tipo do agregado (*List*) pode ser removida com o uso de *iterators* polimórficos

Iterator

- Motivação:
  - *Iterators* polimórficos



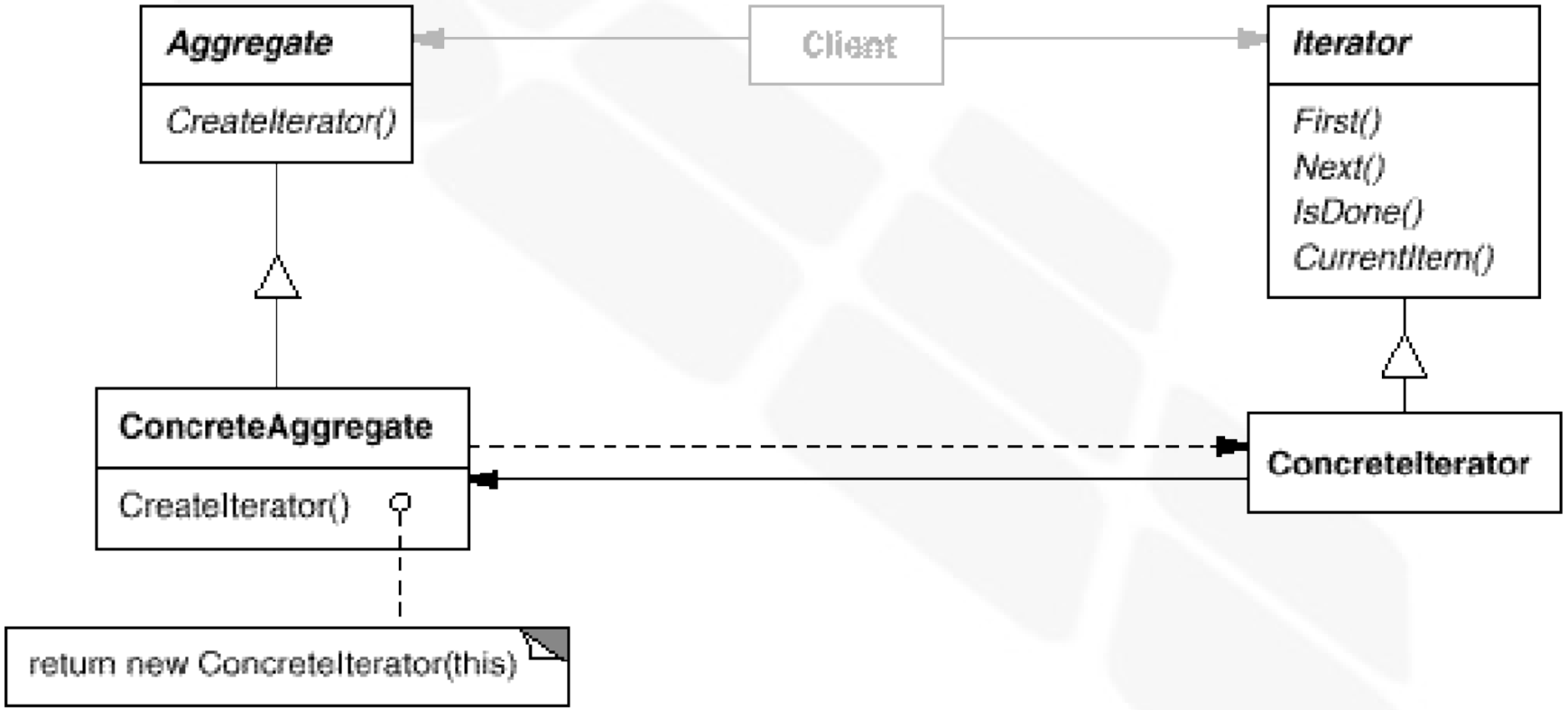
  - Utiliza-se um *Factory Method* no agregado para que o cliente não instancie um *iterator* específico

Iterator

- Aplicabilidade:
  - Para acessar o conteúdo de um agregado sem expor a sua representação interna
  - Para suportar múltiplas varreduras em objetos agregados
  - Para disponibilizar uma interface uniforme para varredura de objetos agregados (iteração polimórfica)

Iterator

- Estrutura:



Iterator

- Participantes:
  - *Iterator*:
    - Define a interface para acessar e percorrer elementos
  - *ConcreteIterator*:
    - Implementa a interface do *Iterator*
    - Mantém a posição atual da varredura do agregado
  - *Aggregate*:
    - Define a interface de criação do *Iterator*
  - *ConcreteAggregate*:
    - Implementa a interface de criação do *Iterator*, retornando uma instância de *ConcreteIterator* apropriada

Iterator

- Colaborações:
  - Um *ConcreteIterator* mantém uma referência para o elemento atual do agregado e é capaz de calcular o próximo elemento da varredura

Iterator

- Consequências:
  - Suporta variações na varredura do agregado:
    - Diferentes formas de varredura de árvores: em ordem, pré-ordem
    - Para mudar a política de varredura simplesmente utiliza-se um outro tipo de *Iterator*
    - Sub-classes do *Iterator* podem ser facilmente criadas com o objetivo de suportar novas formas de varredura
  - Simplificam a interface do agregado:
    - A interface de varredura do *Iterator* elimina a necessidade de uma interface similar na classe do agregado
  - Suportam diferentes varreduras simultâneas:
    - Cada *Iterator* possui seu próprio estado

Iterator

- Implementação:
  - Quem controla a iteração ?
    - *Iterator* externo: o cliente controla a varredura (mais flexíveis)
    - *Iterator* interno: o cliente solicita que o *iterator* execute uma determinada operação e esta é por ele aplicada a todos os elementos do agregado (menos flexíveis, porém mais fáceis de utilizar)
    - Implementar um *operator=()* entre agregados é difícil com um *iterator* interno

Iterator

- Implementação:
  - Quem define o algoritmo de varredura ?
    - O próprio *iterator*:
      - Pode-se utilizar diferentes algoritmos de iteração no mesmo agregado
      - Pode-se reutilizar o mesmo algoritmo em outros agregados
      - Por outro lado, o *iterator* pode precisar acessar dados privados do agregado, violando o encapsulamento (*friend*)
    - O agregado, recebendo o *iterator* como parâmetro:
      - Neste caso o *iterator* é chamado de *cursor* e é responsável somente pela referência ao item atual da varredura

Iterator

- Implementação:
  - Quão robusto é o *iterator* ?
    - Se elementos são adicionados ou removidos durante a varredura, pode-se deixar de percorrer elementos ou acessá-los duas vezes
    - Uma solução fácil, porém custosa, é realizar a varredura em uma cópia do agregado
    - Um *robust iterator* garante que inserções e remoções não irão interferir na varredura, mesmo sem realizar cópia do agregado:
      - Pode-se, por exemplo, registrar os *iterators* nos agregados. Eles seriam notificados, e re-organizados, sempre que acontecesse uma modificação no agregado

Iterator

- Implementação:
  - Operações adicionais no *Iterator*:
    - As operações básicas são: *first()*, *next()*, *isDone()* e *currentItem()*
    - Pode-se adicionar entretanto *previous()* e *skipTo()*
  - Usando *Iterators* polimórficos em C++:
    - *Iterators* polimórficos adicionam um custo, pois são alocados dinamicamente por um *Factory Method*
    - O cliente é responsável pela sua desalocação, o que pode ser difícil se existem múltiplos pontos de saída e tratamento de *exceptions*
    - Entretanto, pode-se utilizar um *Proxy* (*smart pointer*)

Iterator

- Implementação:
  - Uso de *Iterators* em *composites*:
    - *Iterators* externos que realizam varreduras em *Composites* podem ser difíceis de implementar (requer pilha ou recursão)
    - Pode ser mais fácil definir um *iterator* interno ou, se a estrutura interna do *composite* permitir, utilizar uma abordagem baseada em *cursor*
  - *Iterators null*:
    - *Iterator* degenerado, útil para tratar condições de contorno
    - Um *iterator null* é aquele cuja varredura está sempre concluída (*isDone()* sempre igual a *true*)
    - Permitem que varreduras em árvores sejam implementadas de maneira uniforme (folhas retornam *iterators null* para percorrer os filhos não-existentes)

Iterator

- Código exemplo:
  - Interfaces *List* e *Iterator*:

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

Iterator

- Código exemplo:
  - Interfaces *List* e *Iterator*:

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

Iterator

- Código exemplo:
  - Sub-classes de *Iterator*:

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;

private:
    const List<Item>* _list;
    long _current;
};
```

Iterator

- Código exemplo:
  - Sub-classes de *Iterator*:

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

```
template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

Iterator

- Código exemplo:
  - Sub-classes de *Iterator*:

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```

Iterator

- Código exemplo:
  - Utilizando o *Iterator*:

```
void PrintEmployees (Iterator<Employee*>& i) {
    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Print();
    }
}
```

```
List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

Iterator

- Código exemplo:
  - *Iterator* polimórfico:

```
template <class Item>
class AbstractList {
public:
    virtual Iterator<Item>* CreateIterator() const = 0;
    // ...
};
```

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

Iterator

- Código exemplo:
  - *Iterator* polimórfico:

```
// we know only that we have an AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

Iterator

- Código exemplo:
  - *Smart pointer* para o *Iterator*:

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }
private:
    // disallow copy and assignment to avoid
    // multiple deletions of _i:

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);
private:
    Iterator<Item>* _i;
};
```

```
AbstractList<Employee*>* employees;
// ...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```

Iterator

- Usos conhecidos:
  - Comuns em sistemas orientados a objetos
  - *Booch*
  - ET++
  - *Object Windows 2.0*

Iterator

- Padrões relacionados:
  - *Iterators* são frequentemente aplicados em estruturas recursivas tais como os *Composites*
  - *Iterators* polimórficos dependem de *Factory Methods* para a criação das sub-classes apropriadas de *Iterator*
  - O *Iterator* frequentemente utiliza o *Memento* (internamente) para armazenar o estado da iteração

INF011 – Padrões de Projeto

19 – *Iterator*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas