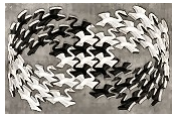

Iterator

Lukáš Marek, Ondřej Sýkora



Iterator

■ Známý jako

- Cursor

■ Účel

- kategorie: Behavioral patterns
- umožňuje procházení(iteraci) objektů jako je List
- lze provádět různé druhy iterací
- dovoluje více iterací najednou

■ Motivace

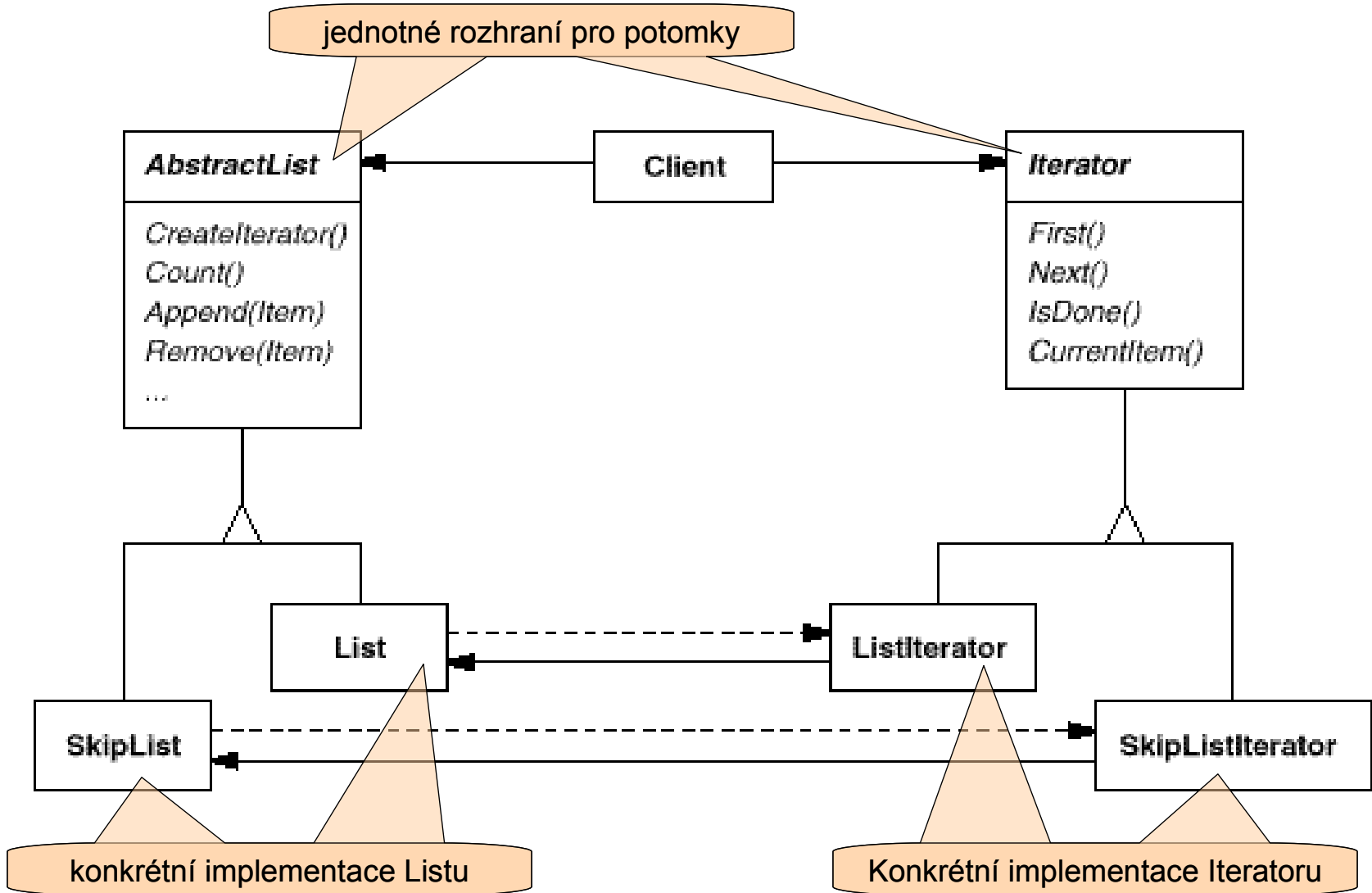
- List jako kolekce prvků
- iterator přebírá zodpovědnost za iterování
- iterování podle kritérii
- polymorfní iterator(List a SkipList)
- problém s vytvořením polymorfního iteratoru

■ Použitelnost

- přístup k prvkům agregovaného objektu bez odhalení vnitřní struktury
- více souběžných procházení jedním objektem
- jednotné rozhraní pro procházení různých objektů



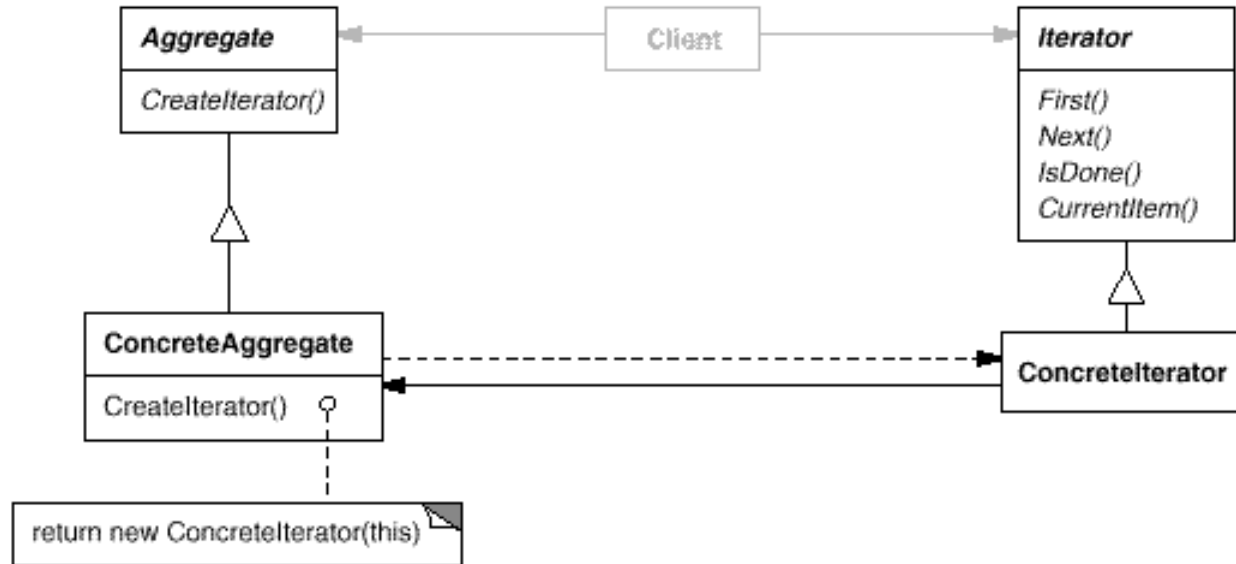
Iterator - motivace





Iterator - struktura

■ Struktura



■ Účastníci

□ Iterator

- definuje rozhraní pro přístup a iterování objektem

□ ConcreteIterator

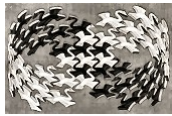
- implementuje rozhraní iteratoru
- pamatuje si aktuální pozici v iterovaném objektu

□ Aggregate

- definuje interface pro vytvoření iteratoru

□ ConcreteAggregate

- vrací instanci konkrétního iteratoru



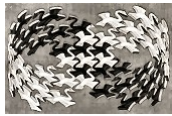
Iterator - důsledky

■ Důsledky(shrnutí)

- podporuje různé druhy(algoritmy) iterací složitějších struktur
 - iterace stromu preorder, inorder, postorder
 - změnou iteratoru je možné jednoduše změnit iteraci

- zjednodušuje rozhraní aggregate
 - aggregate nemusí mít podobné rozhraní jako iterator

- více(různých) iterací na jednom objektu ve stejný moment
 - díky tomu že sám iterator si pamatuje stav iterace



Iterator - implementace

■ Implementace

□ Kdo řídí průběh iterace?

■ Externí iterace

- klient sám určuje, kdy se přesune na další prvek
- větší flexibilita, za cenu zvýšení složitosti na straně klienta

```
for(Iterator i = list.CreateIterator(); !i.IsDone(); i.Next())  
    std::out << i.CurrentItem();
```

■ Interní iterace

- klient dodá jen specifikaci akce, o její aplikaci na všechny prvky se stará agregátor
- méně flexibilní - vždy nejvýše jeden aktivní iterátor
- obtížná implementace v jazycích bez dostatečných výrazových prostředků (anonymní funkce, lambda výrazy)

```
map (\x -> x * x) [1,2,3,4]
```

```
interface IteratorAction {  
    void action(Object o);  
}  
...  
list.forEach(new IteratorAction() {  
    public void action(Object o) {  
        System.out.println(o.toString());  
    }  
});
```

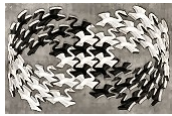


Iterator - implementace

- Zapouzdření agregátoru
 - Iterátor může požadovat přístup k funkcím mimo rámec veřejného rozhraní agregátoru
 - zpřístupnění těchto metod narušuje zapouzdření agregátoru
 - pro agregátor lze deklarovat *friend class* Iterator
 - ztěžuje odvozování dalších typů iterátoru
 - řešení: společná nadtřída pro všechny Iterátory, ve které jsou jako *protected* zpřístupněny funkce pro přístup k neveřejným funkcím agregátoru.

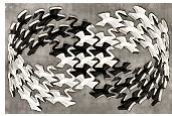
```
class IteratorBase {
protected:
    List* _list;
    bool LstIsLast(Item* it) {
        return _list->IsLast(it);
    }
    Item* LstGetNext(Item* it) {
        _list->GetNext(it);
    }
    IteratorBase(List* lst)
        : _list(lst) {}
public:
    virtual
};
```

```
class ForwardIterator
    : public IteratorBase {
protected:
    Item* _current;
public:
    bool IsDone() {
        return LstIsLast(_current);
    }
    void Next() {
        _current = LstGetNext(_current);
    }
    Item* GetCurrent() {
        return _current;
    }
};
```



Iterator - implementace

- ❑ Dědičnost a polymorfismus iterátorů v C++
 - u objektů alokovaných na zásobníku není možné využít polymorfismus
 - ❑ vyšší náročnost dynamické alokace
 - ❑ na klientovi je, aby iterátory po použití správným způsobem mazal
 - nebezpečné vzhledem k výjimkám
 - nutné ohlídat pro všechny možnosti, jak může být opuštěn blok kódu, kde je iterátor definován
 - řešení: smart pointers - návrhový vzor *Proxy*
- ❑ Iterace přes složené objekty (*Composite*)
 - obtížná implementace externího iterátoru
- ❑ Prázdný iterátor (*Null Iterator*)
 - iterátor, který neukazuje na žádný prvek
 - vhodné pro řešení okrajových podmínek
 - ❑ `IsDone()` vždy vrací *true*
 - ❑ následuje po posledním prvku
 - ❑ výsledek iterace přes prázdnou množinu



Iterator - implementace

- ❑ Kde je implementován algoritmus průchodu?
 - Průchod implementován v agregátoru (*Kursor*)
 - ❑ v iterátoru je uložena jen aktuální pozice, algoritmus pro přechod na další prvek je implementován přímo v agregátoru
 - Průchod implementován v iterátoru
 - ❑ větší flexibilita - pomocí dědičnosti je možné definovat nové iterátory
 - ❑ narušení zapouzdření agregátoru
- ❑ Robustnost iterátoru
 - *Robustní iterátor* „přežije“ změny v agregátoru
 - ❑ přidání prvku, odstranění prvku, změnu ve struktuře uložení dat v agregátoru
 - ❑ co když je odstraněn prvek, na který iterátor ukazoval
- ❑ Rozšíření dostupných operací
 - přesun na předchozí prvek
 - navigace ve stromové struktuře



Iterator - příklad

```
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();
};
```

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

```
template <class Item>
class List : public AbstractList<Item> {
public:
    // ...
    long Count() const;
    Item& Get(long index) const;
    // ...
    Iterator<Item>* CreateIterator() const {
        return new ListIterator<Item>(this);
    }
};
```

```
template <class Item>
class ListIterator {
protected:
    const List<Item>* _list;
    long _current;

public:
    virtual void First() {
        _current = 0;
    }

    virtual void Next() {
        _current++;
    }

    virtual bool IsDone() const {
        return _list->Count() <= _current;
    }

    Item GetCurrentItem() const {
        if(IsDone()) {
            throw IteratorOutOfBounds;
        }
        return _list->Get(_current);
    }

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



Iterator - příklad

□ Dealokace dynamických iterátorů

```
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:

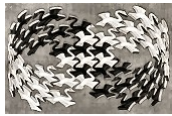
    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=
        (const IteratorPtr&);

private:
    Iterator<Item>* _i;
};
```

```
template <class Item>
class Iterator {
public:
    Iterator(IteratorImpl<Item>* aImpl)
        : _impl(aImpl) { _impl->AddRef(); }
    ~Iterator() { _impl->Release(); }

    void First() { _impl->First(); }
    void Next() { _impl->Next(); }
    bool IsDone() { _impl->IsDone(); }
    Item GetCurrentItem() {
        return _impl->GetCurrentItem();
    }

private:
    // ...
    IteratorImpl<Item>* _impl;
};
```



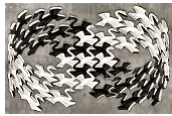
Iterator - související NV

■ Znamé použití

- kolekce v objektově orientovaných jazycích
 - C++, Java, C#,...
- The Boost.Iterator Library

■ Související NV

- Composite
 - časté použití iterátorů pro procházení všech prvků Composite
- Factory Method
 - vytváření konkrétních iterátorů pomocí interface abstraktního agregátoru
- Proxy
 - automatická správa instancí iterátoru na zásobníku



Otázky

- 1) Proč je možné provádět více iterací zároveň na jednom objektu?
- 2) Proč aggregate využívá Factory Method?
- 3) V čem je interní iterátor méně flexibilní?