Iterator

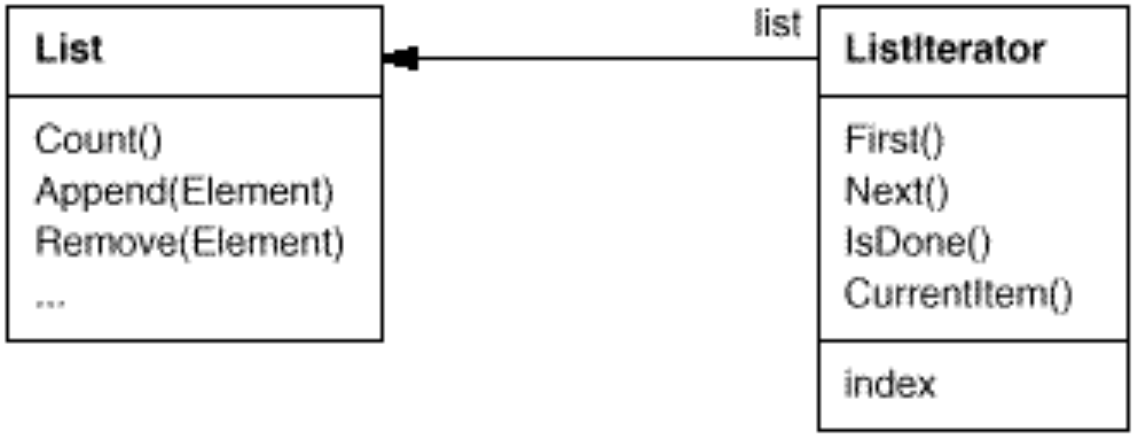
Iterator – Úvod

- **Známý také jako**
  - Cursor
- **Kategorie**
  - Behavioral patterns
- **Smysl**
  - Umožnit sekvenční přístup k prvkům v nějaké kolekci bez nutnosti znát její vnitřní strukturu

Iterator – Účel

- **Účel**
  - umožňuje procházení (iteraci) prvků v nějaké kolekci
  - dovoluje provádět různé druhy iterací
  - umožňuje provádět více iterací na jedné kolekci současně
  - jednotné rozhraní pro procházení různých kolekcí
  - odstínění implementace a vnitřní struktury kolekce
  - zjednodušení rozhraní kolekce

- **Příklad použití**



Iterator – Vlastnosti

- **Požadované vlastnosti**
  - získání prvního prvku z kolekce
  - získání dalšího prvku z kolekce
  - zjištění, zda byly navštíveny všechny prvky
- **Volitelné vlastnosti**
  - přesun na daný prvek
  - změna směru průchodu
  - zadání filtru
  - zadání komparátoru pro ovlivnění pořadí vracených objektů
- **Změny v kolekci**
  - přidání prvku, vyjmutí prvku

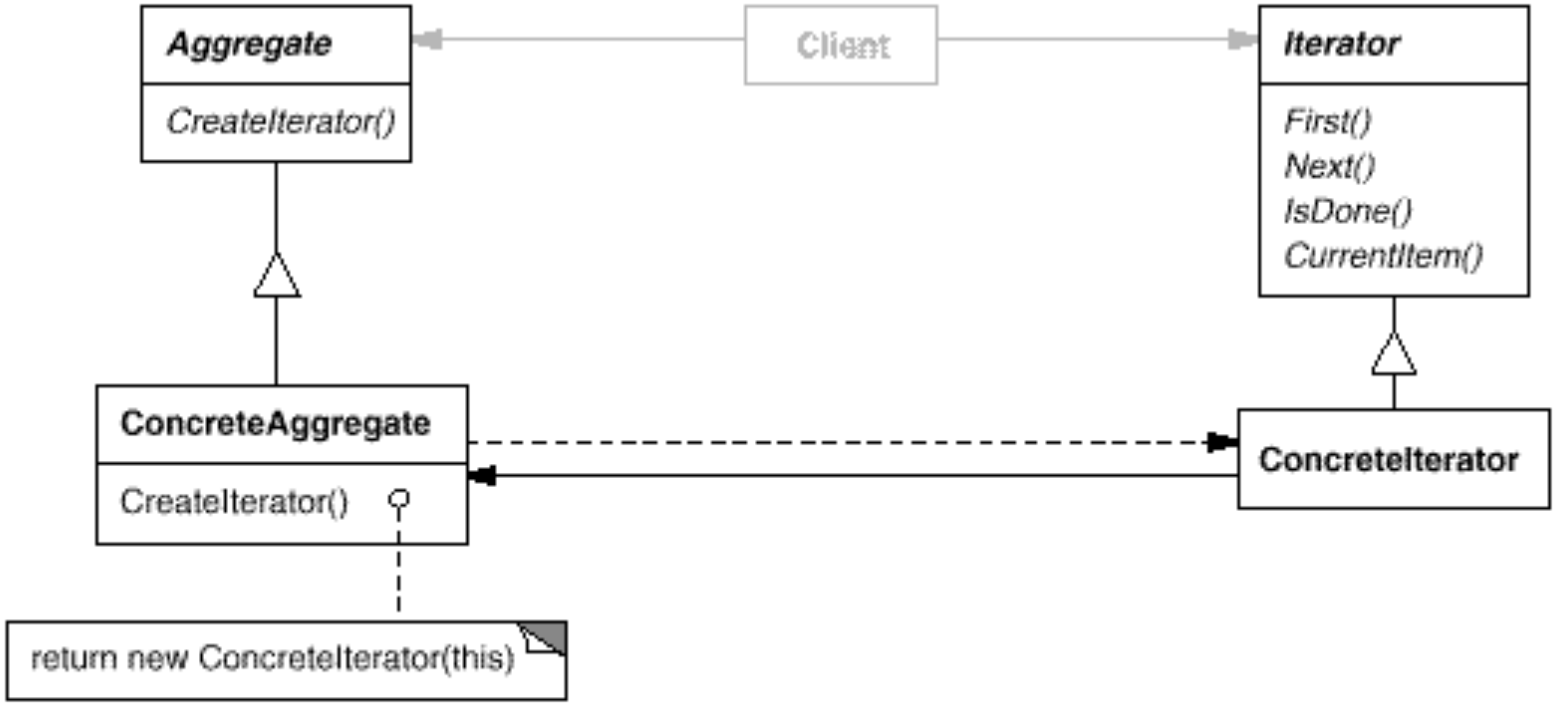
Iterator – Další možnosti

- **Řízení průběhu iterace**
  - Externí iterátor
    - průchod kolekcí řídí klient s využitím iterátoru
    - složitější, ale mocnější řešení (porovnání dvou kolekcí)
  - Interní iterátor
    - průchod kolekcí řídí iterátor sám
    - klient jen specifikuje, co provádět s prvky kolekce
    - potřeba dostatečných výrazových prostředků v jazyce

- **Implementace algoritmu pro průchod kolekce**
  - Iterátorem
    - flexibilnější
    - narušuje zapouzdření
  - Samotnou kolekcí
    - iterátor („cursor“) jen udržuje aktuální pozici

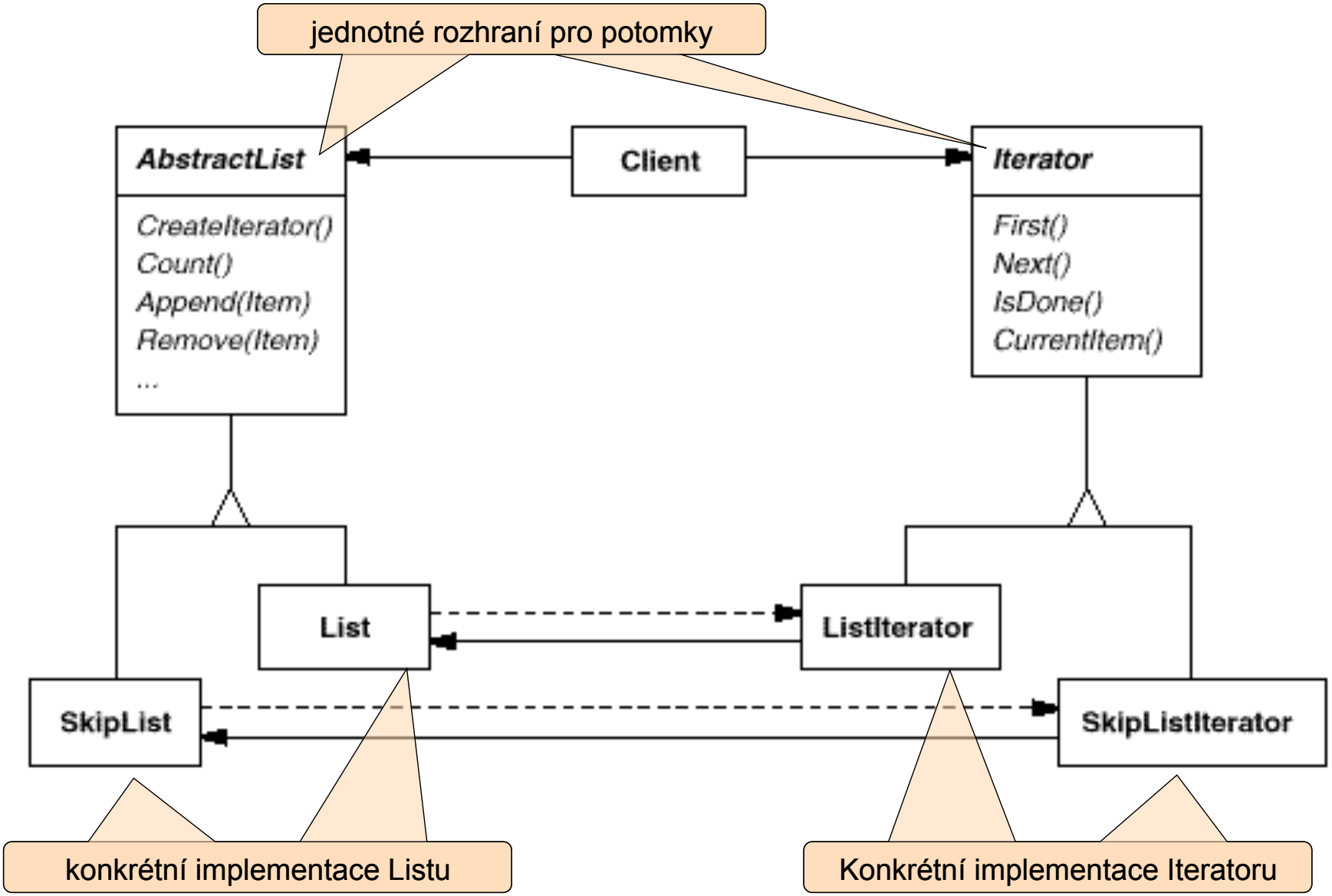
- **Robustní iterátory**
  - korektní chování i při změnách v kolekci během iterace

Iterator – Obecná struktura



- **Účastníci**
  - Aggregate
    - definuje rozhraní pro vytvoření konkrétního iteratoru
  - ConcreteAggregate
    - umí vytvořit instanci konkrétního iteratoru
  - Iterator
    - definuje rozhraní pro sekvenční přístup k prvkům kolekce
  - ConcreteIterator
    - implementuje rozhraní Iterator
    - pamatuje si aktuální pozici

Iterator – Polymorfismus



Iterator – Zapouzdření

- **Iterátor může potřebovat přístup k funkcím mimo rámec veřejného rozhraní kolekce**
  - zpřístupnění těchto metod - narušuje zapouzdření
  - deklarovat *friend class Iterator* - ztěžuje odvozování dalších typů iterátoru
  - společná nadtřída pro všechny Iterátory, ve které jsou jako *protected* zpřístupněny metody pro přístup k neveřejným funkcím kolekce

```
class Iterator {

protected:

  List* _list;

  bool LstIsLast(Item* it) {
    return _list->IsLast(it);
  }

  Item* LstGetNext(Item* it) {
    _list->GetNext(it);
  }

  Iterator (List* lst) :_list(lst) {}

public:
  virtual

};
```

```
class ForwardIterator : public Iterator {

protected:
  Item* _current;

public:

  bool IsDone() {
    return LstIsLast(_current);
  }

  void Next() {
    _current = LstGetNext(_current);
  }

  Item* CurrentItem() {
    return _current;
  }

};
```

Iterator – Implementace

- **Dědičnost a polymorfismus iterátorů v C++**
  - u objektů alokovaných na zásobníku není možné využít polymorfismus
  - vyšší náročnost dynamické alokace
  - klient je zodpovědný za rušení iterátorů, které již nejsou potřeba
    - náročné pro klienta, problém s výjimkami, nutno hlídat všechny možné cesty opuštění bloku kódu s iterátorem…
  - řešení: smart pointers - návrhový vzor *Proxy*

- **Prázdný iterátor *(Null Iterator)***
  - iterátor, který neukazuje na žádný prvek
  - vhodné pro řešení okrajových podmínek
    - *IsDone()* vrací vždy *true*
    - je výsledkem iterace přes prázdnou množinu

Iterator – Příklad implementace

```
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 : Iterator<Item> {

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 použití

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

```
AbstractList<Employee*>* employees;

// ...

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

PrintEmployees(* iterator );

delete iterator;
```

Iterator - 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:

  Iterator<Item>* _i;

  /* disallow copy and assignment to
     avoid multiple deletions of _i: */

  IteratorPtr(const IteratorPtr&);

  IteratorPtr& operator= (const IteratorPtr&);

};
```

```
AbstractList<Employee*>* employees;

// ...

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

PrintEmployees(* iterator );

delete iterator;
```

```
AbstractList<Employee*>* employees;

// ...

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

PrintEmployees(* iterator );
```

Iterator – související návrhové vzory

- **Známé použití**
  - kolekce v objektově orientovaných jazycích (C++, Java, C#, …)

- **Související návrhové vzory**
  - Composite
    - časté použití iterátorů pro procházení všech prvků rekurzivní struktury jako Composite
  - Factory Method
    - řešení tvorby instance iterátoru u polymorfních iterátorů
  - Adapter
    - k přizpůsobení rozhraní různých iterátorů
  - Proxy
    - automatická správa instancí iterátorů