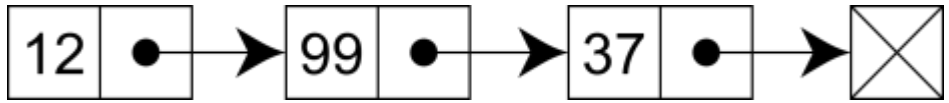
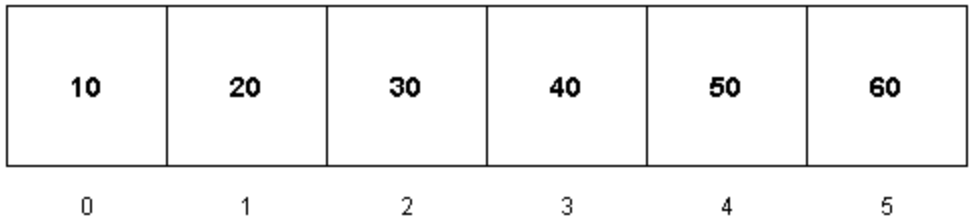
ITERATOR

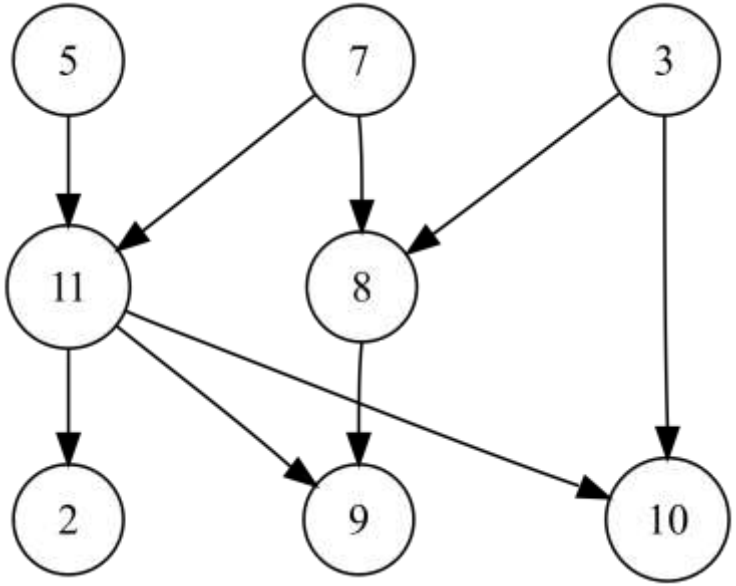
Le Duc Hung

Iterator – Motivace



- **Agregovaný objekt (Kolekce)**
  - Seznam
  - Pole
  - DAG





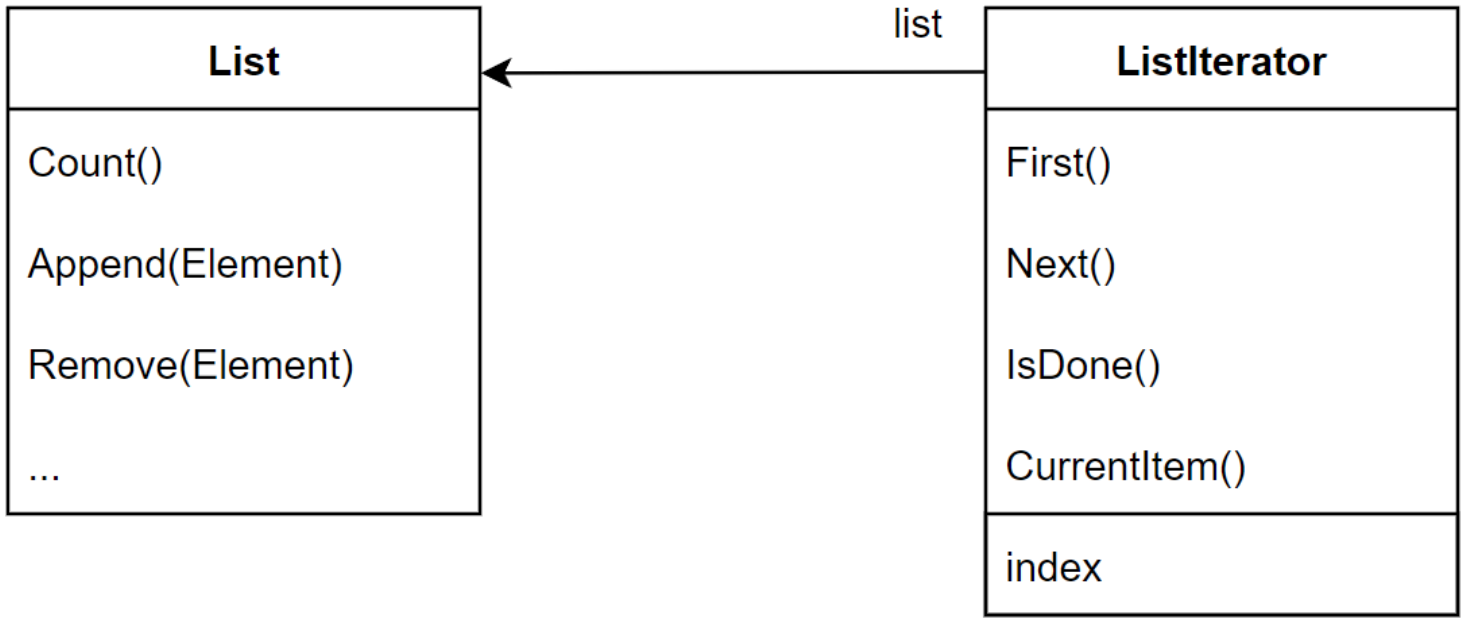
Iterator – Cíl

- **Co je iterátor?**
  - snadné a jednotné rozhraní pro sekvenční přístup k prvkům kolekce
  - bez nutnosti odhalovat její interní reprezentaci (zapouzdření)
  - rozhraní iterace odděleno od rozhraní kolekce
    - záměna kolekcí beze změny uživatelského kódu
  - více možností iterace nad stejnou kolekcí

Iterator – Cíl, použití a výhody

- **Knihovna kolekcí**
  - uživatel kolekce se nemusí zatěžovat složitým principem fungování kolekce
    - pokud mu stačí sekvenční přístup
  - rozhraní kolekce je jednodušší
- **Knihovna algoritmů**
  - funkce definované v knihovně nemusí znát konkrétní kolekci, nad kterou pracují
- **Další výhody**
  - odlišné způsoby iterace nad kolekcí, transformace iterátorů
  - více současných iterací nad jednou kolekcí (vzájemně neovlivnitelných)
  - líné vyhodnocování (zvýšená efektivita, nekonečné kolekce)

Iterator – Základní forma



- **Základní principy**
  - Iterator určuje způsob iterace nad kolekcí
  - **rozhraní kolekce zůstává jednoduché**
- **Rozhraní iterátoru**
  - další možná rozšíření:
    - Previous()
    - SkipTo()
    - Reset()

Iterator – Základní forma - použití

- **Příklad použití rozhraní iterátoru (GoF, C++)**

```
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;
};
```

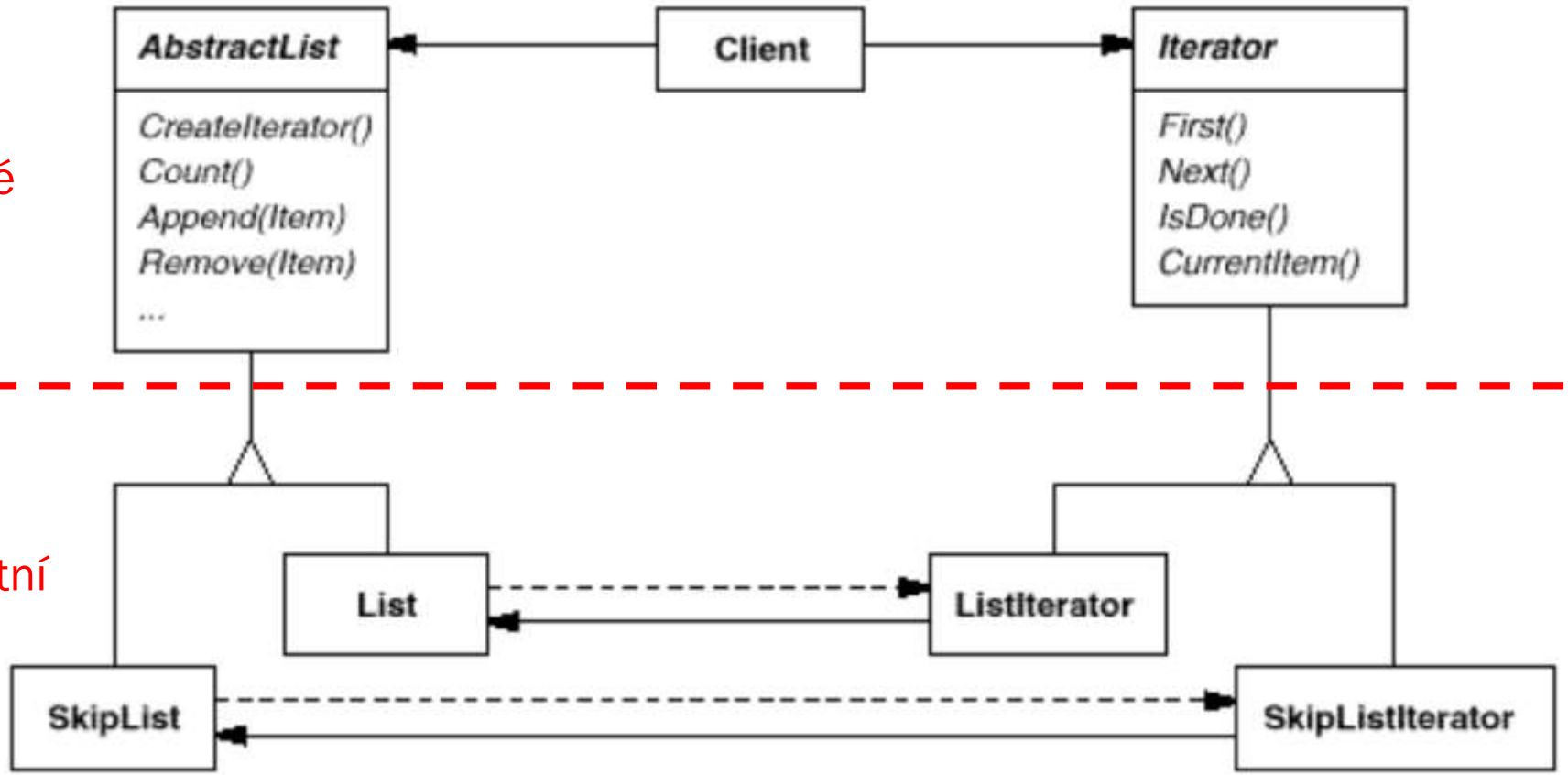
```
struct Employee {
  std::string _name;

  void Print() {
    std::cout << _name << std::endl;
  }
};

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

- **Funkce `PrintEmployees()` je stále svázána s kolekcí `List`**
  - polymorfismus (rozhraní)

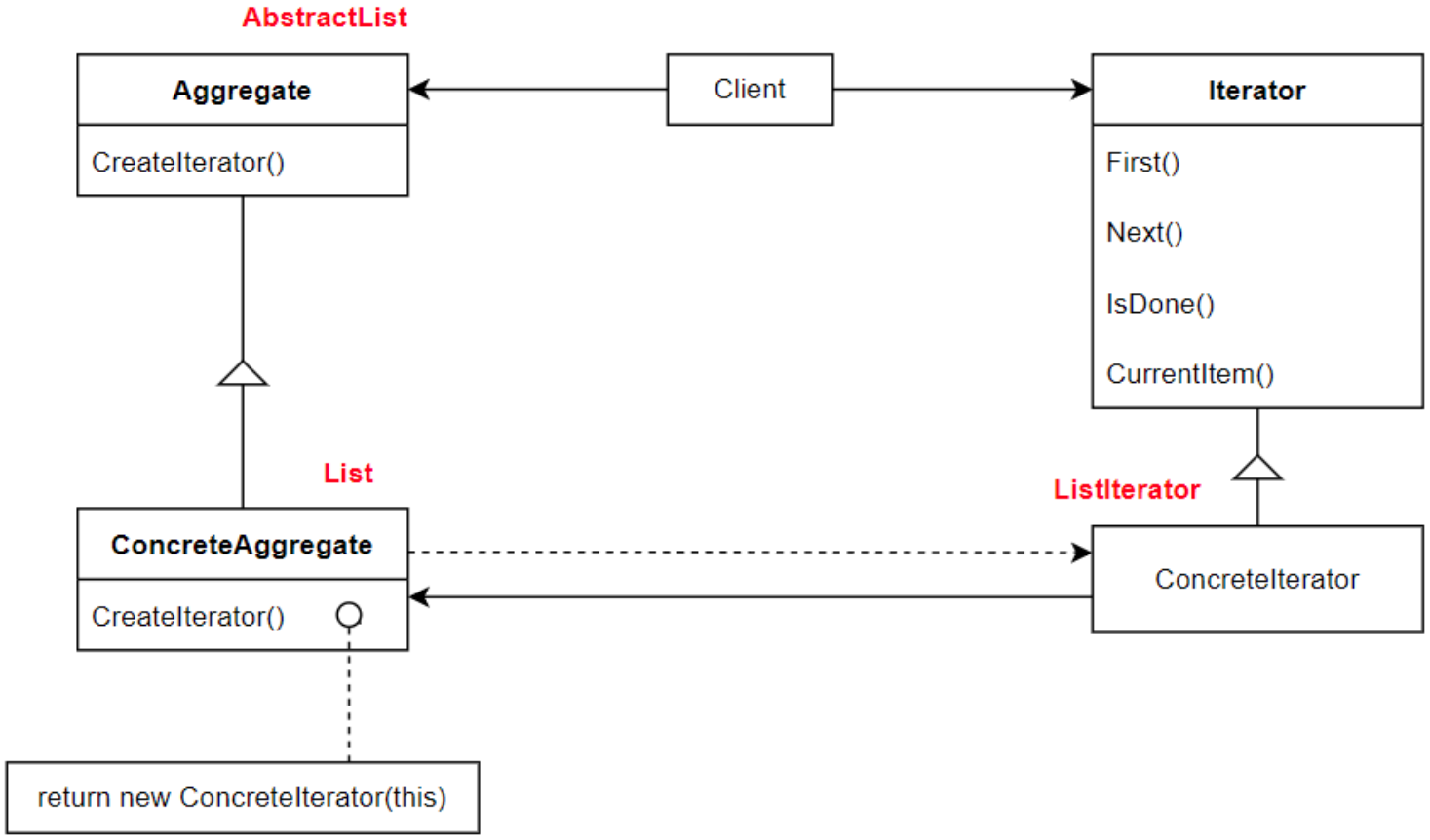
Iterator – Polymorfismus

Obecné

Konkrétní



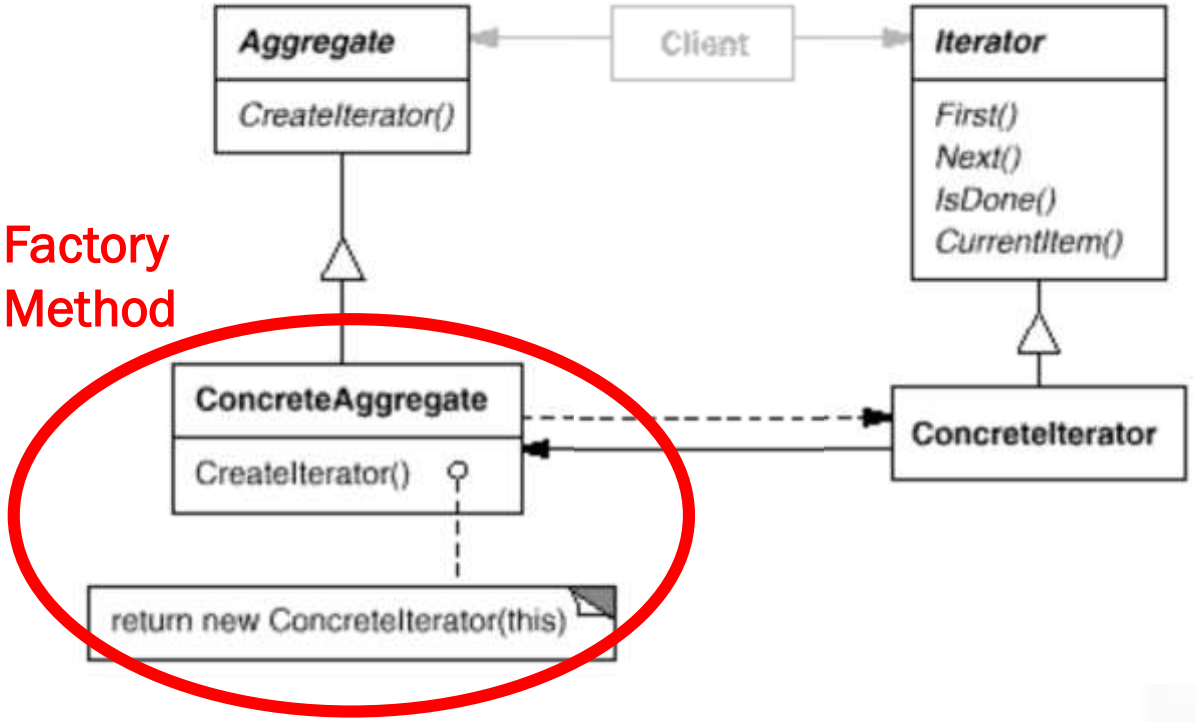
Iterator – Polymorfismus

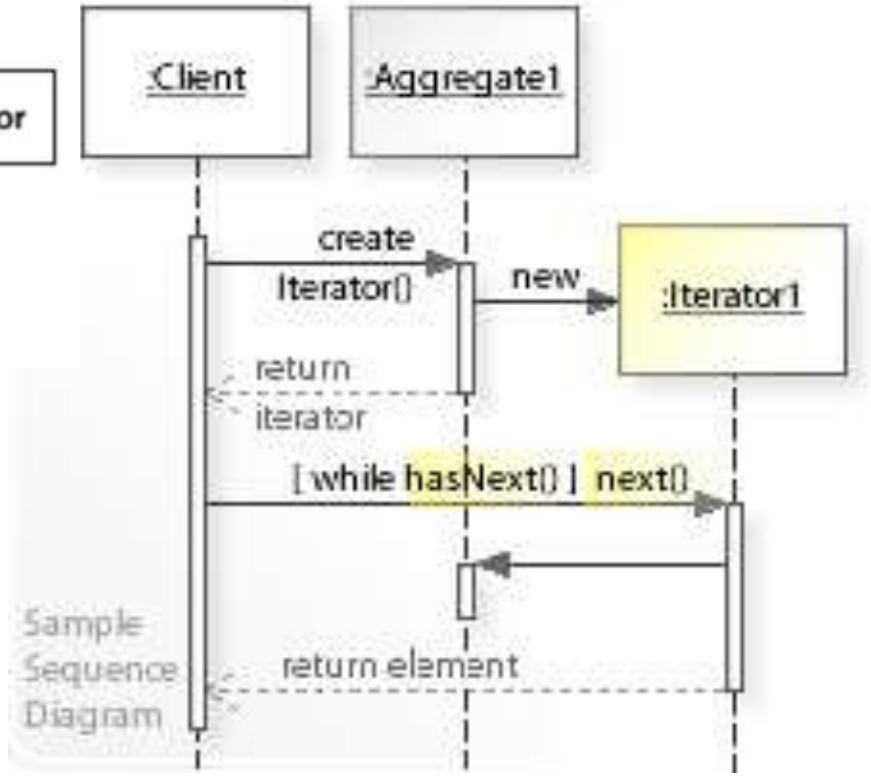


- **Abstraktní kolekce (Aggregate)**
  - rozhraní pro vytváření iterátorů
- **Konkrétní kolekce**
  - vytváří instance konkrétního iterátoru
- **Abstraktní iterátor (Iterator)**
  - rozhraní pro sekvenční iteraci nad kolekcí
- **Konkrétní iterátor**
  - implementace rozhraní iterátoru
  - udržuje informaci o právě vybraném prvku

Iterator – Polymorfismus

Factory Method





Iterator – Factory Method

- **Řeší problém, jak vytvářet instance konkrétních iterátorů**
  - abychom mohli psát kód, který je nezávislý na použité konkrétní kolekci
  - zodpovědnost za vytvoření má kolekce

- **Důsledky**
  - jde o „propojení“ obou hierarchií tříd
  - přináší dynamickou alokaci instancí iterátorů

C++11

```
auto it = employees.begin();
it->PrintEmployees();
```

JAVA

```
Iterator<Employee> it = employees.iterator();
PrintEmployees(it);
```

C#

```
IEnumerator<Employee> it = employees.GetEnumerator();
PrintEmployees(it);
```

Iterator – Rozdělení

- **Podle toho, kdo řídí průběh iterace**

Externí (pull, activní) iterátor	Interní (push, pasivní) iterátor
Řízení je na uživateli	Řízení iterace je na kolekci/iterátoru
Dotazování na následující prvek	Uživatel specifikuje pouze akce
Jednodušší na používání	Složitější na používání
Složitější implementace	Jednodušší implementace

Flexibilnější
 	
`iterator = words.GetEnumerator();`
`while(iterator.MoveNext()) {`
`  Console.WriteLine(iterator.Current);`
`}`
	Ne vždy se hodí - porovnávání kolekcí
	
	`words.ForEach(`
	`  x => Console.WriteLine(x)`
	`);`

Iterator – Rozdělení, zapouzdření

- **Podle toho, kdo implementuje algoritmus iterace nad kolekcí:**
  - **Kolekce**
    - tzv. **cursor**
    - iterátor pouze udržuje aktuální pozici v kolekci
  - **Iterátor**
    - flexibilnější
    - znovupoužitelný kód
    - „narušuje“ zapouzdření kolekce
      - potřebuje přístup k vnitřní struktuře kolekce
      - nad rámec veřejného rozhraní
    - řešení:
      - C#/Java: vnořená private třída
      - C++: friend class

Iterator – Implementace - Polymorfismus

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

```
void PrintEmployees(
  std::unique_ptr<Iterator<Employee*>>& it
)
{
  for (it->First(); !it->IsDone(); it->Next()) {
    it->CurrentItem()->Print();
  }
}
```

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

```
std::unique_ptr<AbstractList<Employee*>> employees;
auto iterator = employees->CreateIterator(); //Initialization of employees
PrintEmployees(iterator);
// No need to manually delete the iterator, it will be automatically destroyed
```

Iterator – Implementace - Delete

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

  Iterator<Item>* _i;
}
```

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

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

Iterator – Otázky

- **Jaké je chování při modifikacích kolekce během iterování?**
  - typicky nebezpečná operace
  - řešení:
    - iterace nad kopií kolekce
    - robustnější iterátor
      - např. kolekce vede evidenci iterátorů a tyto dle potřeby aktualizuje
  - v C# a Javě zakázáno -> výjimka, v C++ nedefinované chování
- **Nad čím vším můžeme iterovat?**
  - cokoliv nás napadne, fantazii se meze nekladou
  - virtuální nebo „nekonečný“ iterátor
    - proudová data načítána z externího zdroje
    - transformační iterátor
      - aplikace funkce na hodnotu
    - číselné posloupnosti (Fibonacciho aj.), generátor náhodných čísel
      - kolekce je generována iterátorem, prvky nejsou přítomny v paměti
  - filtr - výběr pouze určitých prvků dle predikátu
  - XML dokument, adresářová struktura, ...

Iterator – Interakce s Composite

- **Iterator nad NV Composite**
  - třeba si pamatovat celou cestu vnoření
  - snazší použít *interní iterátor*
    - volá sám sebe rekurzivně na potomky komponentu
      - cestu si „pamatuje“ **volací zásobník**
  - různé způsoby průchodu stromovou strukturou → různé iterátory
    - `PreorderCompositeIterator, BreadthFirstCompositeIterator, ...`

- **`NullIterator`**
  - neobsahuje žádný prvek
  - `IsDone()` vždy vrátí `true`
  - pomáhá řešit okrajové případy
    - např. list kompozitní struktury může vracet instanci třídy `NullIterator`

Iterator – Shrnutí

- **Klíčové momenty**
  - iterátor s jednotným rozhraním
  - instance vytvářeny pomocí Factory Method
  - externí vs. interní iterátory – podle volby umístění řízení iterace
  - normální vs. cursor – podle volby umístění algoritmu iterace
  - iterátory nebývají odolné vůči změnám kolekce v průběhu iterace
- **Kde se s iterátory setkáme**
  - kolekce v objektově orientovaných jazycích – C++, C#, Java, …
  - na pozadí konstrukcí „foreach“
    - posun iterátorů z knihoven do syntaxe jazyků
- **Související návrhové vzory**
  - Factory Method – vytváření iterátorů
  - Proxy – ochrana syrových ukazatelů, resource management
  - Composite – iterace nad rekurzivní strukturou
  - Memento – zachytávání stavu iterace