Datové struktury a iterátory

Václav Snášel

Katedra MI, Tomkova 40, 779 00 Olomouc, Česká Republika

Abstrakt
V tomto příspěvku bych chtěl popsat použití iterátorů, při návrhu a realizaci abstraktních datových struktur a dále ukázat iterátor jako návrhový vzor.

1. ÚVOD

Při práci s abstraktními datovými strukturami potřebujeme velmi často pracovat s jednotlivými prvky. Je zřejmé, že by bylo vhodné, aby přístup k prvkům nezávisel na vybrané abstraktní datové struktuře. Dále se ukáže, že iterátor je vhodné používat jako návrhový vzor.

Průchod datovou strukturou můžeme realizovat mnoha způsoby. Pro jednoduchost si uvedeme příklad na abstraktní datové struktuře seznam viz R.Pecinovský, M.Virius. *Objektové programování*. Přístup k jednotlivým prvkům seznamu bychom mohli realizovat pomocí ukazatele, který můžeme nazvat pracovně kurzor. Za pomoci tohoto kurzoru bychom procházeli seznamem, přičemž přechod na další prvek bychom realizovali příkazem

Kurzor = Kurzor->Další

Uvedený postup má, ale několik nevýhod. Předpokládá totiž, že na všech místech programu, kde budeme chtít takovýto kurzor použít musíme zveřejnit vnitřní konstrukci atomu a zpřístupnit ji tak, aby bylo možno získat adresu jeho následníka či předchůdce. Tento postup není nejlepši protože zveřejnění adresy atomu umožňuje nekorektní zásah do datové struktury.

Další nevýhodou této konstrukce je, že je založena na znalosti realizace dané abstraktní datové struktury. To znamená, že při změně realizace této abstraktni datové struktury budeme nuceni procházet program a měnit místa, kde procházíme seznam pomocí Kurzoru. Tuto nevýhodu bychom mohli odstranit tím, že bychom pro přechod na další položku seznamu napsali funkci „Další" potom bychom mohli přechod na následující prvek realizovat takto:

Další(Kurzor)

Ani toto řešení není zcela vyhovující, protože pro abstraktní datovou strukturu umožňuje pouze jeden způsob průchodu. Například pro strom můžeme použít průchod do hloubky nebo do šířky, pro dvousměrný seznam průchod od začátku do konce nebo od konce do začátku. Nemá smysl uvažovat o tom, že bychom funkci Další doplnily o nějaký parametr, který by určoval typ průchodu.

2. ITERÁTORY

Ukazuje se, že by bylo vhodné realizovat průchod seznamem pomocí objektu. Jedna z možností jak přistoupit k realizaci takového objektu je popsána v [6]. Definice iterátoru z knihy [6] se nejeví jako nejlepší. A to z toho důvodu, že iterátor pro dvousměrný seznam zná strukturu procházené abstraktní datové struktury. To znamená, že kdybychom v programu nahradili z nějakých důvodů dvousměrný seznam jednosměrným seznamem museli bychom zachovat oba směry v realizaci iterátoru a to i v případě, že bychom potřebovali pouze směr jeden. Dále tento iterátor obsahuje metodu pro vkládání a rušení prvku. Mohl by nastat problém v okamžiku, kdy bychom chtěli pracovat se strukturou, kde by operace rušení prvku nebyla k dispozici.

```
  public:
    cDLIter();
       cDLIter(cDlist& L);  void Reset(cDlist& L);
       pData operator () ();
       pData operator ++ ();
       pData operator ++ (int);
       pData operator -- ();
       pData operator == (cDLIter i);
       pData operator != (cDLIter i);
       void PrevVloz(pData);
       void  Zrus  (int Vlatn=ClZl);
       friend ostream& operator << (ostream &, const cDLIter& )  private:
     cDList *Seznam;
     cDAtom *Kurzor;
```

Těmto problémům se lze snadno vyhnout dodržování následující zásady minimálnosti:

„Zjisti jaké jsou minimální vlastnosti objektu tak, aby mělo smysl objekt realizovat a pak se teprve pusť do realizace"

Tento princip bychom neměli chápat jako snahu o redukci počtu metod objektu, ale jako snahu o redukci vlastností objektu.

Daleko vhodnější se jeví možnost definovat třídu, která nám bude realizovat iterátor na obecné úrovni. Definice této třídy a příklady pro použití této třídy jsem převzal z knihy W.Ford, W.Topp. *Data Structures with C++*. Uveďme si definici třídy iterátor.

```
template <class T>class Iterator{   protected:            int iterationComplete;

  public:      Iterator(void);

    virtual void Next(void) = 0;
    virtual void Reset(void) = 0;

    virtual T& Data(void) = 0;

   virtual int EndOfList(void) const;
```

```
};
// Konstruktor nastaví iterationComplete do 0 (False)
template <class T>
Iterator<T>::Iterator(void): iterationComplete(0)
{}

template <class T>
int Iterator<T>::EndOfList(void) const
{
  return iterationComplete;
}
```

Tato třída je zcela nezávislá na nějaké abstraktní datové struktuře. Dále si ukážeme příklad realizace iterátoru pro seznam. Nejprve uvedeme deklaraci seznamu.

```
template <class T>class List{   protected:        // počet prvků v seznamu
      int size;
   public:
      List(void);

      virtual int ListSize(void) const;
      virtual int ListEmpty(void) const;
      virtual int Find (T& item) = 0;

      virtual void Insert(const T& item) = 0;
      virtual void Delete(const T& item) = 0;
      virtual void ClearList(void) = 0;
};

template <class T>
List<T>::List(void): size(0)
{}

template <class T>
int List<T>::ListSize(void) const
{
   return size;
}

template <class T>int List<T>::ListEmpty(void) const{   return size == 0;}
```

Nyní můžeme napsat potomka třídy Iterator.

```
template <class T>class SeqListIterator: public Iterator <T>{   private:
        SeqList<T>* ListPtr;        Node<T> *currPtr;     public:
        SeqListIterator(SeqList<T>& lst);
        virtual void Next(void) = 0;
        virtual void Reset(void) = 0;
        virtual T& Data(void) = 0;
```

```
        void SetList(SeqList<T>& lst);

    };
template      <class       T>SeqListIterator<T>::SeqListIterator(SeqList<T>&       lst)
Iterator<T>(), ListPtr(&lst)
{
        Reset();
 };


template <class T>void SeqListIterator<T>::Reset(){
iterationComplete = ListPtr->llist.ListEmpty();
if (ListPtr->lline.front == NULL)              return;
currPtr = ListPtr->lline.front;};template <class T>void
SeqListIterator<T>::SetList(SeqList<T>& lst){
ListPtr = &lst;          Reset(); };template <class T>T&
SeqListIterator<T>::Data(void){
if (ListPtr->llist.ListEmpty() || currPtr == NULL)              exit(1);
return(currPtr->data);};template <class T>void
SeqListIterator<T>::Next(void){          if (currPtr == NULL)
         return;        currPtr = currPtr->Next;
         if (currPtr == NULL)                 iterationComplete = 1;};
```

Nyní si ukážeme příklad na použití této třídy.

```
SeqList<int> L;SeqListIterator<int> iter(L);cout  <<  iter.data();
// tisk aktuálního prvku ze seznamuiter.next;
for(iter.Reset(); !iter.EndOfList(); iter.Next());cout  <<  iter.data() << "";
```

Další příklad ukáže použití iterátoru ve třídě realizující graf. tento iterátor umožňuje procházet vrcholy grafu.

```
template <class T>void Warshall(Graph<T>& G){
VertexIterator<T> vi(G), vj(G);      int i, j, k;      int
WSM[MaxGraphSize][MaxGraphSize];  // Warshallova matice
        int n = G.NumberOfVertices();

        // inicializace matice
        for(vi.Reset(),i=0;!vi.EndOfList();vi.Next(),i++)
                for(vj.Reset(),j= 0;!vj.EndOfList();vj.Next(), j++)
                        if (i == j)
                                WSM[i][i] = 1;
                        else
                                WSM[i][j] = G.GetWeight(vi.Data(),vj.Data());

        // tranzitivni uzávěr pomocí matice WSM
        for (i = 0; i < n; i++)          for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
        WSM[i][j] |= WSM[i][k] & WSM[k][j];   }
```

3. NÁVRHOVÉ VZORY

Mezi základní výhody iterátoru patří znovupoužitelnost. Znovupoužitelnost je významný prostředek ke zkvalitnění softwarových produktů a k podstatnému zvýšení produktivity jejich tvorby. Může být využita na různých úrovních. Objektově orientované jazyky podporují znovupoužití na úrovni zdrojového kódu. Znovupoužití na úrovni návrhu je v současné době předmětem intenzivního zkoumání. Návrh softwarového produktu byl dlouho považován za složitou záležitost. Bylo tomu tak zejména proto, že návrháři se v převážné většině případů pohybovali v neprobádaném území a nemohli tedy vycházet ze standardních a ověřených řešení, protože ty prostě neexistovaly. Jinak řečeno, doposud je návrh limitován zkušenostmi. Chceme-li zmírnit tento problém, potřebujeme prostředky umožňující zaznamenávání existujících zkušeností. Zkušený návrhář většinou nezačíná svůj návrh od nuly. Raději opakovaně používá řešení, která již fungují. Jestliže najde dobré řešení používá ho ve své práci stále znovu a znovu. Tyto zkušenosti z něj dělají experta. Postupně nachází opakující se vzory tříd a způsob komunikace mezi objekty, je schopen je zobecnit a implementovat v různých objektově orientovaných systémech. Tím se jeho řešení stávají flexibilní, elegantní a ve vysoké míře znovupoužitelné. Opakované používání úspěšných řešení značně zvyšuje produktivitu práce při návrhu nových systémů. Vliv na jejich kvalitu je taktéž nezanedbatelný.

Co znamená pojem vzor?Podobně jako u řady dalších inženýrských termínů neexistuje žádná přesná definice tohoto pojmu. Mohli bychom snad napsat "popis opakovatelných řešení při návrhu struktury a chování softwarového produktu" nebo stručněji "řešení problému zařazené do daného kontextu". Kontextem se míní popis problému, který vzor řeší, zdůvodnění adekvátnosti řešení, doporučený postup při jeho implementaci, vztah k ostatním vzorům, atd. Vzory tedy odpovídají nejen na otázku jak, ale také proč, a tím usnadňují pochopení a zvládnutí systému. K úplnému vysvětlení a pochopení podstaty pojmu, ale těchto několik slov zdaleka nestačí. Vzor je více než opakovatelné řešení. Vzor je způsob popisu řešení. Vzor popisuje kontext, ve kterém je toto řešení použitelné, i to, jak konkrétní kontext ovlivní parametry řešení a jeho důsledky. Ve vzorech jsou vlastně zaznamenávány užitné vlastnosti navrhovaných řešení. Někdy může být vhodné popsat i špatné řešení, pokud chceme upozornit na často se opakující chybu. Vzory pro návrh obsahují čtyři základní vlastnosti:

- Název vzoru

slouží k pojmenování daného řešení a jeho zařazení do katalogu vzorů. Pojmenování umožňuje návrhářům o vzorech hovořit a odkazovat se na ně v dokumentaci. Volba správného názvu má vliv na pochopení a vhodné používání daného vzoru.

- Problém

popisuje situace, ve kterých je možné daný vzor použít spolu s podmínkami, které musí být před jeho použitím splněny.

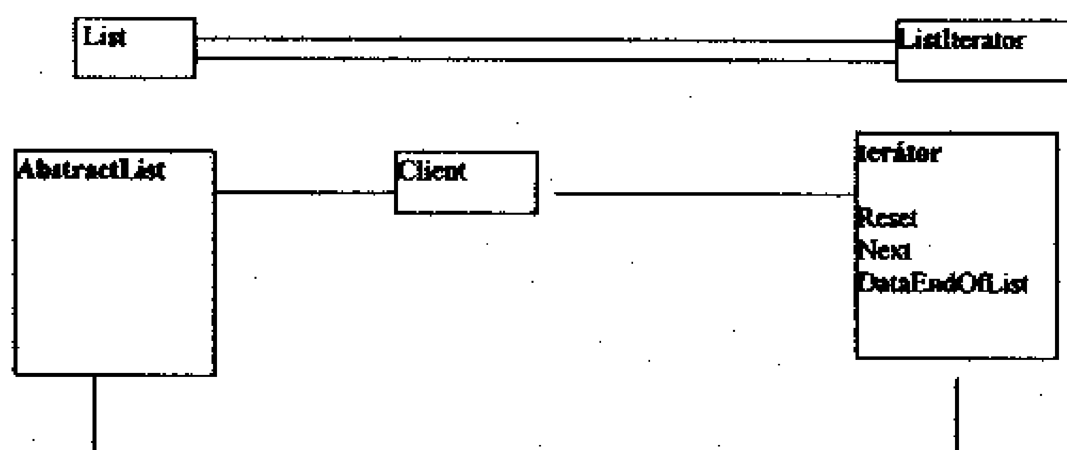
- Řešení

popisuje elementy, které tvoří řešení daného problému, jejich vzájemný vztah,

rozdělení zodpovědností a spolupráci. Nezabývá se však konkrétní implementací v konkrétním prostředí, protože vzor má sloužit jako šablona použitelná v různých situacích. Důsledky použitípopisuje výhody a nevýhody použitého řešení. Patří sem například paměťové a časové nároky nebo vliv na flexibilitu, rozšiřitelnost a přenositelnost.

S pojmem vzor jsou příbuzné (ne však totožné) pojmy paradigma, idiom, architektura, rámec (framework). Lze říci, že paradigma je velmi abstraktní vzor, který je použit v rozsahu celého systému. Idiom je řešení problémů malého rozsahu závislé na programovacím jazyku. Architektura je spojena s celkovou strukturou aplikace a může být vyjádřena pomocí více vzorů. V této souvislosti se s pojmem vzor často spojuje pojem mikroarchitektura. Rámec je tvořen kolekcí konkrétních tříd, které jsou zaměřeny na řešení problémů spojených s danou problémovou oblastí.
Vzory pro návrh většinou zahrnují obecný popis několika spolupracujících objektů a tříd, který lze upravit pro řešení obecného problému v konkrétním kontextu. Z pohledu rozsahu tedy mezi vzory nezařazujeme ani struktury, které lze vyjádřit samostatnou třídou, jako jsou např. seznamy, hašovací tabulky atd., ani komplexní, problémově orientované návrhy celých aplikací nebo subsystémů.

Iterátory můžeme chápat jako návrhové vzory viz E.Gamma, R.Helm, R.Johnson, J.Vlissides. *Design Patterns*. Tato kniha je vlastně katalogem návrhových vzorů. Jedním ze vzorů v tomto katalogu je i iterátor.



Iterátor zde chápeme jako obecné rozhraní pro procházení sdružených objektů. Konkrétní způsob procházení závisí na zvolené realizaci iterátoru. To na druhé straně umožňuje mít více iterátorů pro jednu realizaci abstraktní datové struktury.

3. ZÁVĚR

Používání iterátorů, jako vzorů usnadní řešení mnoha problémů. Samotná realizace iterátoru může přinášet některé problémy. Mezi tyto problémy patří modifikace abstraktní datové struktury v těle cyklu řízeného iterátorem. Výše uvedené implementace iterátoru na tyto modifikace nereagují.

Literatura:

1. G.Booch. Object-Oriented Analysis and Design with applications.

enjamin/Cummings, Redwood City, CA 1994. Second Edition
2. J.Coplien, D.Schmidt. Pattern Languages of Program Design.Addison-Wesley, 1995
3. W.Ford, W.Topp. *Data Structures with C++*. Prentice Hall 1996
4. E.Gamma, R.Helm, R.Johnson, J.Vlissides. *Design Patterns*. Addison Wesley 1995
5. G.Iannello. Programming Abstract Data types, Iterators and Generic Module in C. *Software – Practice & Experience. Vol. 20 (3), 1990*
6. R.Pecinovský, M.Virius. *Objektové programování 1,2*. Grada 1996
7. V. Sklenář, V.Snášel. Vzory pro návrh. Tvorba Software, 1995 Ostrava

–