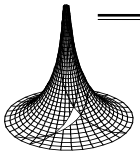


8/1 (2010), 89–108
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Programming theorems on enumerator

Tibor Gregorics

Abstract. This paper deals with the examination of the programming patterns best known by programmers: the programming theorems. It is a significant issue that in what way these patterns can be formulated in order to solve a relatively broad spectrum of problems using a small number of patterns. In this paper, the well known programming theorems are applied to the processing of enumerators. To this end, the robustness of patterns gained this way will be presented, and it will be also pointed out how the programs thus constructed can be implemented in the modern object-oriented programming environments: in language C++, Java and C#.

Key words and phrases: programming pattern, analogous programminng, programming theorem, enumerator, iterator.

ZDM Subject Classification: P50.

1. Introduction

The fundamental aim of software engineering is the creation of products of proper quality. To ensure this, various technologies, regulations and standards are applied. Among these, there have a major role the so called *programming patterns* which give a guaranteed-quality solution to each part-problem revealed during software production. Several groups of programming patterns are known: some help testing or generating test data, others support implementation or encoding [1], [3], and many more assist program design. These latter contain the so called design patterns but also the algorithm patterns to solve small sub problems: the so called programming theorems [7].

A *programming theorem* consists of a general problem and an algorithm which solves it. If a concrete problem is an instance of the general one, then the solution of the concrete problem will be given from the appropriate conversion of the algorithm of the programming theorem [6]. This technique guarantees only best the correctness of the generated solution if the problems are phrased pro forma and it can be shown that the concrete problem, which is to be solved, is a special case of the general problem of the programming theorem. After the differences are revealed between the concrete and general problem, they should only be substituted to the algorithm of the programming theorem [5]. This technique is called *analogous programming* [4], [7].

The main condition of the usability of this method is that the programming theorems are not that numerous, thus necessarily general; meanwhile, not so abstract, hence fitting to concrete problems. These theorems are well known in the programming methodology: summation, counting, maximum selection, linear searching and selection. The only difference is that; for some, the problems are related to arrays; for others, to functions mapping from an integer interval. In this paper, the programming theorems which are related to enumerators are introduced.

2. Enumerator

If a piece of data can be represented by a group of elementary values (e.g. sets, arrays, or sequences, etc.), then its processing means the processing of its elements. One of the most important properties of this type of data is that their elementary values can be got one by one. This kind of data-type is called the enumerable (iterated) type; the instance of it is called the collection or the container. The enumeration of a collection means the ability to point out the first element, to step to the next one, to refer to the current one, and to ask if there is the end of the enumeration.

Not only can the elements of a collection be enumerated, but also, for example, the proper divisors of an integer or the integers of an interval. It can be seen that enumeration may also relate to non-enumerable types.

The enumerating operators do not belong to the data, the elements of which are enumerated. It would be strange if an integer, the proper divisors of which are needed, has got operators such as "step to the first proper divisor", "step to the next proper divisor", "take the current proper divisor" and "ask if there is the end of the enumeration". Although, an integer interval has only got operators to

ask the limits of the interval; in order to enumerate its element, a special object, the index is needed. It is because the enumeration operators always bind to an individual object that refers to the data, the elements of which are wanted to be enumerated. This object is the enumerator (or iterator). It can step on the first element of the enumeration; then, on the next one, it can give back the current element and check if the enumeration has finished. Hence the enumerator always includes the enumerating operators: *First(), Next(), Current()* and *End()*.

2.1. Concept of enumerator

The enumerator is the object that can produce – one by one – the elements of an enumerable data such as an array, a set, a sequential file or an integer providing its proper divisors. The enumerator can be viewed as a finite sequence of elements.

Let the enumerator be notated by t. The enumeration begins with the calling of the operator *t.First()*[1]. This operator steps on the first element of the enumeration if an element exists at all. After this, the enumerator's state is "launched". Then the operator *t.Next()* moves to the other elements of the enumeration. The operator *t.Current()* gets the current element that the enumeration has stepped on. The condition of executing the operators *Next()* and *Current()* is that the enumeration has not ended yet. This can be checked with the operator *t.End()* which replies with a logical true if the enumeration is over. Obviously, the enumeration must not be infinite to ensure the occurrence of this situation.

The effects of these operators are not always defined. It is not generally known; for example, what the effect of the operators *t.Next()*, *t.Current()* and *t.End()* is if they are executed before the operator *First()*, or what the effect of the operators *t.Next()* and *t.Current()* is after the enumeration ended. It is similarly not defined what should happen if the operator *t.First()* is executed for a second time.

An object can be named as enumerator if its type implements the operators *First()*, *Next()*, *End()* and *Current()*. The *enor(E)* denotes the enumerator-type which can traverse the finite sequence of set E. In the specification of problems, where enumerators are used, the enumerator can be referred to as a finite sequence; t_i is the i^{th} element of such a sequence and $|t|-$ is the length of it.

In reality, the enumerator is not represented with a sequence. (Neither even if it is the enumerator of a sequence-structured piece of data.) Of course in the

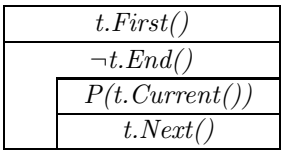
[1]In this paper, the object-oriented style is used.

representation of the enumerator, there is some reference to the data which is wanted to be enumerated. (This could be a single natural number if its divisors should be generated; an array, a sequential input file, a set, or else a multiplicity set if their elements are required; or a graph, the nodes of which are needed to be traversed with any strategy so as to come by the values contained there.) Besides this, the representation may also contain other kind of components which support the enumeration. In the course of the enumeration, there is always an actual elemental value which can be got at that exact moment. In case of the enumeration of the data of a one-dimension array; that is, a vector, an index-variable is sufficient; in case of a sequential input file, the element last read out should be restored, so does the fact whether the last read-out was successful; in case of the enumeration of a nature number's divisors; for instance, the last given divisor is required.

2.2. Enumeration

The processing of an enumerator means that an activity is executed on enumerated elements. This activity may be a summation, a counting, or a maximum selecting, etc.

The values given back by the enumerator are usually processed in some way. This procession can be various; let's nominate it as the general $P(e)$ which means an arbitrary procession of an e elemental value. No further explanation is needed why the enumeration-based procession is executed by the algorithm-scheme given below. It is noted that since the number of the enumerable elements is finite, this procession terminates certainly in a finite number of steps.



2.3. Famous enumerators

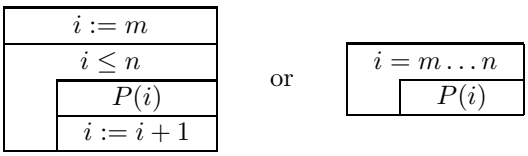
Some important enumerators will be hereinafter examined; the representation of which – except for one – is based on some famous iterated type. It will be discussed what the implementation of the operations *First()*, *Next()*, *End()* and *Current()*, which ensure the enumeration, is. It will be an important part of

the examinations to show how the algorithm-scheme presented above doing the general procession is modified in case of a concrete enumerator-type object.

Let's see first the **integer-interval enumerator** that can enumerate the integers from m to n one by one. Certainly, other (backward, two by two, etc.) interval enumerators can be constructed. The classical one is particular because the popular versions of programming theorems are used to state over interval (or the domain of an array). It proves that those programming theorems can be derived from the more general ones on enumerator.

The classical enumerator of an interval can be represented by the limits of the interval (m and n) and the index i. The variable i contains the current element of the enumeration, so it implements the operator *Current()*. The operator *First()* is the assignment $i := m$; *Next()* is the assignment $i := i + 1$. The condition $i > n$ substitutes the operator *End()*. (The operator *First()* can be executed iteratively; it starts again the enumeration. All operators can always be executed.)

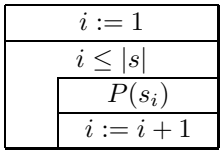
Now, the processing of an interval-enumerator can be compounded from the general algorithm-scheme and from the implementation of the operators of the interval-enumerator. Certainly, this algorithm can be also written in form of a counter loop.



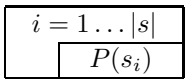
It is easy to verify that the operators ($i := n$, $i := i - 1$, $i < m$) implement the backward enumerator over the interval $m \ldots n$, and the operators ($i := m$, $i := i + 2$, $i > n$) implement nevertheless the enumerator two by two.

It is very simple to make a **sequence enumerator**. It can be represented by the sequence that is wanted to be enumerated, and by an index. The index i points the current element of the sequence s, so the expression s_i implements the operator *Current()*. In the classical traversal of the sequence, the operator *First()* is the assignment $i := 1$; *Next()* is the assignment $i := i + 1$. The condition $i > |s|$ substitutes the operator *End()*. (The operator *First()* can be executed iteratively; it starts again the enumeration. The operator *Current()* is only defined during the traversal while the others are always.)

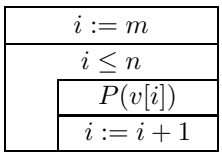
Using these remarks, here is the processing of the sequence.



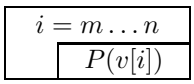
or



The **vector enumerator** is very similar to the sequence enumerator. The only difference between them is that the element of a vector can be indexed by the interval $[m \ldots n]$ and not by interval $[1 \ldots |s|]$. Its processing algorithm is in form of a counter loop:

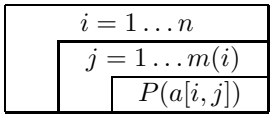


or



A matrix is a vector of vectors. The inside vectors are named as rows, elements of the i^{th} row are indexed from 1 to $m(i)$ in which $m(i)$ is the number of the elements of the i^{th} row. Frequently $m(i)$ is equal to the same m for all i. The rows are indexed from 1 to n in which n is the number of the rows. The row-by-row traversal of a matrix can be solved by only one index but the two-index version is more popular.

In the two-index version the **matrix enumerator** is represented by a matrix (a) and two index-variables (i and j). One index (i) is used to go through among the rows of the matrix, the other (j) to go through the elements of the actual row. During the go-through, $a[i, j]$ will be *Current()*. First, $a[1, 1]$ should be stepped on, so the operator *First()* is implemented by $i, j := 1, 1$. Grabbing the next element, if the index j has not yet reached the end of the actual row, it has to be increased by one. In contrast, the index i is increased by one to step on the next row; the index j is situated at the beginning of this row. To sum up, $IF(j \leq m(i) : j := j + 1; j > m(i) : i, j := i + 1, 1)$ implements the operator *Next()*. The term *End()* is substituted by $i > n$. The matrix-procession is different to the previous ones but the following two-loop algorithm is much more known in practice.



A sequential input file is a sequence that has got only one operator: it can read and remove its first element. This reading can be notated by $st, e, f : read$ where st gets the status of reading, e contains the element read from the file (undefined if the file is empty) and f is the sequential input file. If the file is not empty, the reading is successful and its status (st) is "norm", otherwise "*abnorm*". The **enumerator of a sequential input file** can be represented by the file (f), the current element (e) that has been read recently from the file, and the status (st) of the recent reading. The enumeration of the sequential input file reads over the elements of the file. The operator *First()* is the first $st, e, f : read$, and the additional readings substitute the operator *Next()*. The operator *Current()* gives back the value of e. The *End()* is false while the status st is norm. All operators are well-defined outside the enumeration.

$st, e, f : read$
$st = norm$
$P(e)$
$st, e, f : read$

The representation of **the enumerator of the set** h does not require special data; its operators can be implemented by the operators of the set. Obviously, we cannot speak about the order of elements in a set but the order of their enumeration. The value of the operator *Current()* can be produced by the deterministic element-selection: $mem(h)$. To read the first element, nothing is needed to be done, so the operator *First()* is the empty statement. The operator *Next()* deletes the element selected out from the set before: $h := h - \{mem(h)\}$. The implementation of this operator is simpler than the general $h := h - \{e\}$ because $mem(h)$ is surely in set h. The *End()* will be true if the set is empty. The effect of operators *Current()* and *Next()* are not defined if the set is empty.

$h \neq \emptyset$
$P(mem(e))$
$h := h - \{mem(e)\}$

3. Programming theorems on enumerator

Now we introduce the general programming theorems.

All of them have got an enumerator as input data. The description of programming theorems consist of three parts: the text of the problem to be solved,

its formal specification and the solving algorithm. The formal specification contains the state space, the precondition and the post condition. The state space, A, includes the data (variables) relevant to the problem with their types. The precondition, *Pre*, expresses that the input is given and contains the constraint (if any) the input must satisfy. The post condition, *Post*, determines the value of the output variables.

3.1. Summation

Problem: There is given an enumerator, the elements of which are in the set E, and a function $f : E \to H$, where H is an arbitrary set with some associative adding operation, $+ : H \times H \to H$, and 0 is a neutral element of this operation. Compute the sum of the values that the function f maps from the elements of t, i.e. $\sum_{i=1}^{|t|} f(t_i)$! (If t is empty, this sum is equal to zero.)

Specification:[2]

$$A = (t : enor(E), s : \mathbb{N})$$

$$Pre = (t = t')$$

$$Post = \left(s = \sum_{i=1}^{|t'|} f(t'_i)\right)$$

Algorithm:

$s := 0$
$t.First()$
$\neg t.End()$
$s := s + f(t.Current())$
$t.Next()$

3.2. Counting

Problem: There is given an enumerator, the elements of which are in the set E, and a condition $\beta : E \to \mathbb{L}$. How many enumerated elements satisfy the condition?

Specification:

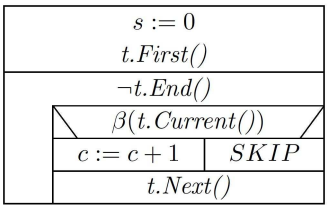
$$A = (t : enor(E), c : \mathbb{N})$$

[2]Here we use the general formal specification where A is the statespace, Pre is the precondition, $Post$ is the post condition of the problem.

$$Pre = (t = t')$$

$$Post = \left(c = \sum_{\substack{i=1 \\ \beta(t'_i)}}^{|t'|} 1 \right)$$

Algorithm:



3.3. Maximum selecting

Problem: There is given an enumerator, the elements of which are in the set E, and a function $f : E \to H$ where H is a well ordered set. We suppose that the enumerator is not empty. Which enumerated element is the greatest?

Specification:

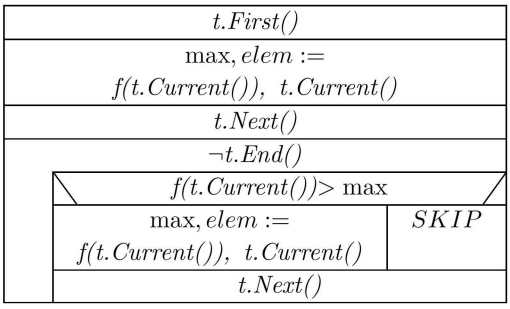
$$A = (t : enor(E), \max : H, elem : E)$$

$$Pre = (t = t' \wedge |t| > 0)$$

$$Post = \left(\left(\max = f(elem) = \max_{i=1}^{|t'|} f(t'_i) \right) \wedge \exists i \in [1 \ldots |t'|] : t'_i = elem \right)$$

where $[1 \ldots |t'|] = [1, |t'|] \cap \mathbb{N}$

Algorithm:



3.4. Conditional maximum searching

Problem: There is given an enumerator, the elements of which are in the set E, a function $f : E \to H$ where H is well ordered set, and a condition $\beta : E \to \mathbb{L}$. Which enumerated element is the greatest among them that satisfies the condition β?

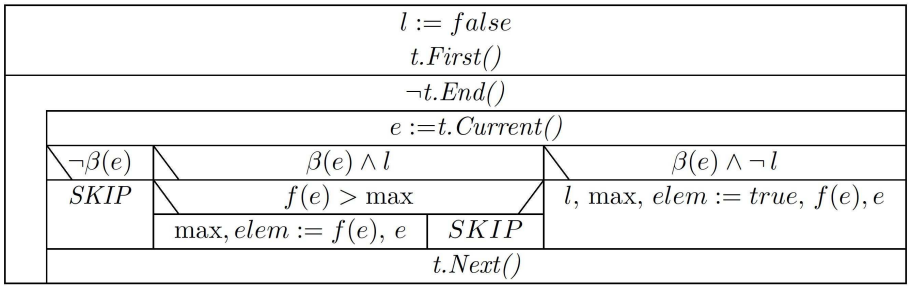
Specification:

$$A = (t : enor(E), l : \mathbb{L}, \max : H, elem : E)$$

$$Pre = (t = t')$$

$$Post = \Big((l = \exists i \in [1 \dots |t'|] : \beta(t'_i)) \wedge \Big(l \to \max = f(elem) = \max_{\substack{i=1 \\ \beta(t'_i)}}^{|t'|} f(t'_i)\Big) \wedge \exists i \in [1 \dots |t'|] : t'_i = elem)\Big)$$

Algorithm:



3.5. Selection

Problem: There is given an enumerator, the elements of which are in the set E, and the condition $\beta : E \to \mathbb{L}$. There exists at least one element that satisfies this condition. Let us give the first one!

Specification:

$$A = (t : enor(E), elem : E)$$

$$Pre = (t = t' \wedge \exists i \in [1 \dots |t|] : \beta(t_i))$$

$$Post = (\exists i \in [1 \dots |t'|] : t'_i = elem \wedge \beta(elem) \wedge \forall j \in [1 \dots i-1] : \neg\, \beta(t'_i))$$

Algorithm:

$t.First()$
$\neg\beta(t.Current())$
$t.Next()$
$elem := t.Current()$

3.6. Linear searching

Problem: There is given an enumerator, the elements of which are in the set E, and the condition $\beta : E \to \mathbb{L}$. Let us give the first element of the enumeration of t that satisfies the condition β.

Specification:

$$A = (t : enor(E), l : \mathbb{L},\ elem : E)$$

$$Pre = t(= t')$$

$$Post = ((l = \exists i \in [1 \ldots |t'|] : \beta(t'_i)) \wedge (l \to \exists i \in [1 \ldots |t'|] : t'_i = elem \wedge \beta(elem) \wedge \forall j \in [1 \ldots i-1] : \neg\beta(t'_i)))$$

Algorithm:

$l := false$ $t.First()$
$\neg l \wedge \neg t.End()$
$elem := t.Current()$
$l := \beta(elem)$
$t.Next()$

3.7. Computing of recursive function

Problem: There is given an enumerator, the elements of which are in the set E, the k-order m-basic (positive integer k, integer m) function $f : Z \to H$ where $f(i) = h(t_i, f(i-1), \ldots, f(i-k))$ for all $i \geq 1$, h is a function $E \times H^k \to H$, and $f(m-1) = e_{m-1}, \ldots,\ f(m-k) = e_{m-k}$, where $e_{m-1}, \ldots, e_{m-k} \in H$. Compute the value of the function f in the argument $|t|$!

Specification:

$$A = (t : enor(E), y : H)$$

$$Pre = (t = t')$$

$$Post = (y = f(|t'|))$$

Algorithm:

$y, y_1, \ldots, y_{k-1} := e_{m-1}, e_{m-2}, \ldots, e_{m-k}$ *t.First()*
$\neg$*t.End()*
$y, y_1, y_2, \ldots, y_{k-1} := h($*t.Current()*$, y, y_1, \ldots, y_{k-1}), y, y_1, \ldots, y_{k-2}$
t.Next()

4. Conclusions

An opinion of the usability of the programming theorems can be formed in two respects.

One is the robustness; that is, that the theorems are general enough to assist in solving many problems. The number of the theorems should not however be too big; at the same time, their phrasing should not be too abstract. The number and the level of abstraction of the programming theorems have already passed a lot of tests. Indeed a small set of similar programming theorems makes it possible to solve a huge set of small practical problems with analogous programming [6], [7]. Here it should only be justified whether their variants on enumerators are not too broad, which are no doubt more general than the ones on arrays or on functions mapping from interval; these are better known in the programming society.

Second is that how the programming environment can fit to our programming theorems, that is, how the abstract programs made with programming theorems on enumerator can be translated into a programming language.

Now, we are going to examine how our programming theorems can be applied in the program designing and how they can be implemented on some well-known programming languages.

4.1. Designing

When a problem shows the signs of summation, counting or maximum selecting, etc., an appropriate programming theorem can be applied. The question is that what kind of enumerator can traverse the elements that the programming theorem processes. If this enumerator is detected, it is enough to define its operators (*First()*, *Next()*, *End()* and *Current()*).

The solution of those problems in which the elements of a well-known collection should be processed is very simple. The general schema of the appropriate

programming theorem is needed, its function (or functions) should be defined, and the *First()*, *Next()*, *End()* and *Current()* should be substituted with the operators of the concrete enumerator. For example: "Which is the maximum element of a non-empty set?" (maximum selecting over a set), "How many words beginning with the letter 'a' are there in a text?" (counting over a sequential file), "Search a positive element in a matrix" (linear searching over matrix), or "Select the red cars from the file of different vehicles" (summation over sequential file).

The problems which need a special enumerator verify much more the robustness of the programming theorems on enumerator. Let's see the next example: "Compute the sum of the prim divisors of a natural number."

This problem can be solved as a conditional summation over the interval $2 \dots n$, in which the condition checks whether an element is a prim divisor of the number n or not.

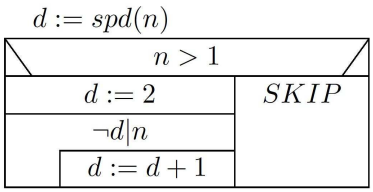
$$A = (n : \mathbb{N}, sum : \mathbb{N})$$
$$Pre = (n = n')$$
$$Post = \left(Ef \wedge sum = \sum_{\substack{i=2 \\ i|n' \wedge prim(i)}}^{n} i\right)$$

Another and better solution is that if an enumerator is imagined that can traverse all prim divisors of the number n. In this case, a summation on enumerator solves the problem in which $E = H = \mathbb{N}$ and $f(e) = e$ for all number e.

$$A = (t : enor(\mathbb{N}), sum : \mathbb{N})$$
$$Pre = (t = t')$$
$$Post = \left(Ef \wedge sum = \sum_{i=1}^{|t'|} t'_i\right)$$

$sum := 0$
$t.First()$
$\neg t.End()$
$sum := sum + t.Current()$
$t.Next()$

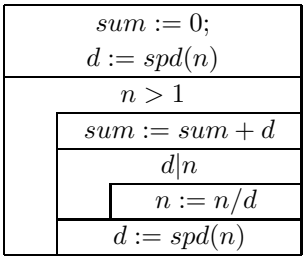
If the number n is at least 2, then its smallest prim divisor can be found by a selection programming theorem starting from 2 and searching for the first divisor. Note this activity by $d := spd(n)$. It is sure that d is a prim. This is the operator *t.First()*.



The next prim divisor can be produced (by the operator *t.Next()*) using this $d := fpd(n)$ but before this, the search for the number n must be reduced. It should be divided by the prim divisor found before as many times as it can be divided. The operator *t.Current()* gives back the value of d, and the operator *t.End()* will be true if $n = 1$. The enumerator is represented by the number n and d.

$$
\begin{array}{lcl}
t & \sim & n,\ d : \mathbb{N} \\
t.First() & \sim & d := spd(n) \\
t.Next() & \sim & LOOP(d|n;\ n := n/d);\ d := spd(n) \\
t.End() & \sim & n = 1 \\
t.Current() & \sim & d
\end{array}
$$

The final solution:



There often occur problems in which a programming theorem should be used in the way that the solving program stops if a certain condition comes true. For instance, if it is examined in an array whether there are at least three even numbers or not, then it is a counting that should not go through all the elements of the array if the result reaches three. For such a problem, the enumerator is needed in which only the operator *End()* differs from that of the array's famous enumerator.

Similarly, those problems can be easily handled in which there is needed a reverse traversal than usual. For example, an array's last element with that attribute is searched.

In general, those problems are difficult in which the groups of a collection's elemental values are to be processed. For example, the longest word of a text file is searched. In this case, the enumerator should not traverse the characters but the length of the words. The operators *First()* and *Next()* search for the beginning of the first and the next word, respectively; if such exists, then it also searches for its end and it computes its length; finally, the *Current()* gives back this length. The operator *End()* turns true if no more words are found.

The solution of the problems of merging also requires a unique enumerator. Let's see this example: "A sequential input file is given containing n pieces of integers; all of them are strictly monotone increasing ordered." Here an enumerator is needed which enumerates all the integers from the files so that it gives a number at only once and that it also notes that a number belongs to which file.

4.2. Implementation

The algorithms created with the assist of the programming theorems on enumerator can be with ease implemented in the object-oriented programming languages. The reason is that the type of an enumerator can be easily described by a class. For example, let's examine the realisation in C++ of the enumerator that can traverse a double queue represented by a two-direction chained list without the implementation of operators Push_front(), Pop_front(), Push_back(), Pop_back().

```
class Sequence
{
private:
    struct Node
    {
        int val;
        Node *next;
        Node *prev;
        Node(int c, Node *n, Node *p): val(c), next(n), prev(p){ };
    };
    Node *first;
    Node *last;
    int iteratorCount;
public:
    void Push_front(int e);
    int Pop_front();
    void Push_back(int e);
    int Pop_back();
friend class Iterator;
    class Iterator
    {
    public:
        Iterator(Sequence *s):seq(s),current(NULL)
                                    {++(seq → iteratorCount);}
        ∼ Iterator()                {- -(seq → iteratorCount);}
        int Current()const          {return current →val;}
        void First()                {current = seq → first;}
        bool End()   const          {return current = NULL;}
        void Next()                 {current = current → next;}
    private:
        Sequence *seq;
        Node    *current;
    };
    Iterator CreateIterator(){return Iterator(this);}
};
```

Many times we do not need a unique enumerator like before. For example the enumerators of the famous collections have been ready in the Standard Template Library (STL) of C++. If an algorithm, that can solved a problem, is derived from our programming theorem which processes famous collections the algorithm is implemented simply. Bjarne Stroustrup wrote in [9] that the iterators (means enumerators) are the screws that fasten together the collections and the algorithms.

In STL of C++ the standard collections have here got a begin() and an end() method, with the assist of which the traversal of the collection's elements is solved. In the program-passage shown below, the vector is a special collection-template, the elements of which (so be integers this time) are wanted to be enumerated in order to be subjected to the operation of a function void f(int). So the below code-type (based on [9]) is needed:

```
vector⟨int⟩ coll;
...
vector⟨int⟩::iterator it = coll.begin();
while(it!= coll.end())
{
      f(*it);
      ++it;
}
```

The same can be applied in Java language to the collection List⟨Integer⟩ and to a function void f(Integer) [2]. Here the enumerator (its type is Iterator⟨Integer⟩) can be created by the method iterator() and it has got the methods hasNext() and next(). These correspond to the operators *First()*, *Next()*, *Current()*, and *End()*.

```
List⟨Integer⟩ coll = new List⟨Integer⟩();
...
Iterator⟨Integer⟩ it = coll.iterator();
while(it.hasNext())
{
    f(it.next());
}
```

The C# [8] ensures language elements very similar to Java. The class implemented the interface IEnumerable contains the operator GetEnumerator() which can create an enumerator with the methods int Current(){get;}, bool MoveNext() and void Reset().

```
List⟨int⟩ coll = new List⟨int⟩();
...
IEnumerator⟨int⟩ it = coll.GetEnumerator();
it.Reset();
while(it.MoveNext())
{
    f(it.Current);
}
```

The C# has got two further language elements which make the application of the enumerators and the definition of the unique enumerators very simple. With the assist of the loop foreach, enumeration can be applied (either a unique enumerator or a famous one) without so much as the enumerator object should be indirectly created, or else its methods should be indirectly called. The program-part given above can be likewise written down on the condition that the function f does not change the collection. (This kind of loop foreach can be found in C++ and Java, too.)

```
List⟨int⟩ c = new List⟨int⟩();
...
foreach(int e in coll)
{
    f(e);
}
```

If a unique enumerator to an object is wanted to be created in C#, then the class of the object should implement the interface IEnumerable. This happens by the definition of the method GetEnumerator(). In the realisation of this method, the elements to be traversed can be created one by one and are given with the assist of the order yield return. For example, let's have a C# program-part of the solution of the problem "Define the summation of the prim divisors of an n positive integer."

```
class PrimDivisor : IEnumerable
{
   static void Main(string[] args)
   {
      int sum = 0;
      PrimDivisor ds = new PrimDivisor(int.Parse(Console.ReadLine()));
      foreach (int p in ds)
      {
           sum += p;
      }
      Console.WriteLine("The sum of prim divisors: {0} ", sum);
      Console.ReadKey();
   }
   private int n;
   private int d;
   public PrimDivisor(int i)
   {
      n = i > 0 ? i : 1;
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      while (n != 1)
      {
           spd();
           yield return d;
           n = n / d;
      }
   }
   int spd()
   {
      d = 2;
      while (n % d != 0) ++d;
   }
}
```

References

[1] A. Alexandrescu and H. Sutter, *C++ Codings Standards: 101 Rules, Guidelines, and Best Practices*, 1st Edition, ISBN 0321113586, Pearson Education In., Addison Wesley Professional, 2005.

[2] E. Angster, *Objektumorientált tervezés és programozás*, 4KÖR Bt, 2004 (in Hungarian).

[3] K. Beck, *Implementation Patterns*, 1st Edition, ISBN 0321413091, Pearson Education In., Addison Wesley Professional, 2008.

[4] Sz. Csepregi, A. Dezső, T. Gregorics and S. Sike, *Automatic Implementation of Service Required by Components*, PROVECS'2007 Workshop, ETH Technical Report, 2007, 567.

[5] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, 1973.

[6] Á. Fóthi, *Bevezetés a programozáshoz*, ELTE Eötvös Kiadó, 2005 (in Hungarian).

[7] T. Gregorics and S. Sike, Generic algorithm patterns, *Proceedings of Formal Methods in Computer Science Education FORMED 2008*, Satellite workshop of ETAPS 2008, Budapest, March 29, 2008, 141–150.

[8] J. Sharp, *Visual C# 2005 Step by Step*, Microsoft Press, 2005.

[9] B. Stroustrup, *The C++ Programming Language*, 2001.

TIBOR GREGORICS
ELTE, FACULTY OF INFORMATICS
DEPT. SOFTWARE TECHNOLOGY AND METHODOLOGY
1117 BUDAPEST PÁZMÁNY PÉTER SÉTÁNY 1/C

E-mail: gt@inf.elte.hu

(Received June, 2009)