10/1 (2012), 135–152
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching Mathematics and Computer Science

Analogous programming with a template class library

Tibor Gregorics

Abstract. In this paper, a template class library and its usage can be read. The classes of the library contain the C++ code of the algorithm of programming theorems. This library supports the implementation of the programs that are planned by analogous programming but the primary aim of its usage is to introduce the object-oriented programming style to show how a reusable code can be written with inheritances, overriding virtual methods, composition of objectcs and template parameters.

Key words and phrases: analogous programming, programming theorem, enumerator, object-oriented programming.

ZDM Subject Classification: P50.

1. Introduction

In a typical flow of teaching programming, simple algorithms must first be planned and encoded (procedural programming), which is mostly based on algorithm patterns employing analogous programming [2] [6]; then, the concept of data type is introduced, which leads to object-oriented programming. To solve a problem requiring only one class is not difficult, but to understand the solution of more complex problems where several classes (or template classes) are needed, even if there is inheritance between the classes, and if virtual methods are redefined, is very hard. Unfortunately, only two or three groups of problems are

This paper is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

known where object-oriented techniques based on few classes can be practiced. On top of all, these problems are totally different to the ones that are solved earlier when simple algorithms are taught.

But why cannot the same problems be used to illustrate the object-oriented techniques as they can be solved with analogous programming? If the template class library of programming theorems (see Figure 1) is made (there are only a few programming theorems), many problems can then be solved by reusing the code of this library. So, students can have practice in object-oriented programming facing problems which are not unknown and which can also be solved with procedural programming.

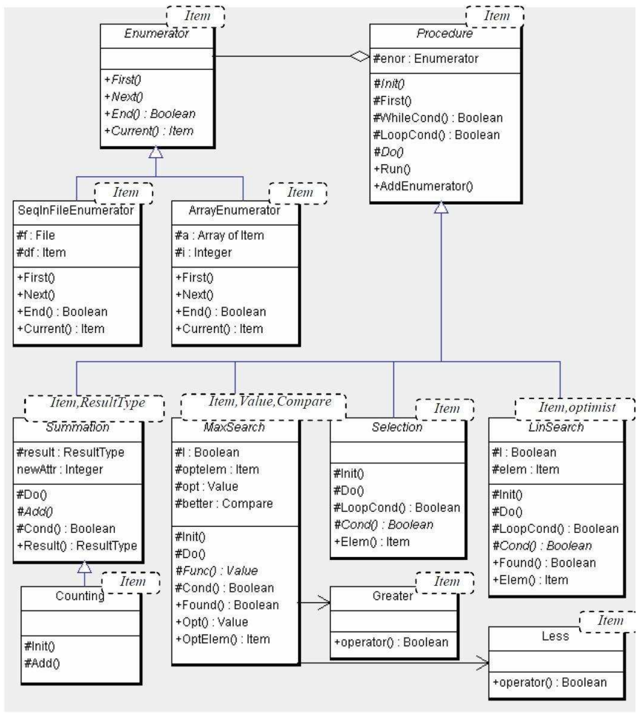

Figure 1. Template class library of programming theorems

In the object-oriented programming, a program derived from a programming theorem is executed by an activity object (more precisely, its special method). The class of this object is made by the inheritance from the corresponding template class that contains the programming theorem in the most general form, one characteristic of which is that it is over an enumerator [5]. The code inherited form the library must be tuned to the concrete problem. The special properties of a programming theorem can be given by the redefinition of the virtual methods (like the logical condition of the counting), by filling in the template parameters (e.g. the type of the enumerated elements), or by dynamically connecting the enumerator to the activity during the execution.

2. Elements of the class template library

The elements of the library are divided into two parts. One contains the classes of the programming theorems; the other consists of the classes of the famed enumerators. Both groups are derived from ancestor classes, respectively [1] [4] [7].

2.1. Class of general enumerators

The general properties of an enumerator [5] are fixed by the abstract class *Enumerator*. Every object that is an instance of this class has the enumeration operators: *First(), Next(), Current(), End()*. These operators (methods) are not defined at this level (they are abstract) because the type of the traversed elements is not known, hence it is denoted by the template parameter *Item*. This class is an interface, and the classes of all enumerators are going to implementate this. The C++ code of this class [8]:

```
template <typename Item>
class Enumerator {
public:
   virtual void First()    = 0;
   virtual void Next()     = 0;
   virtual bool End()        const = 0;
   virtual Item Current()    const = 0;
   virtual ~Enumerator(){}
};
```

2.2. Famed enumerator classes

Most often, the elements of an array or a sequential input file are enumerated in a programming theorem. The sequential input file is regularly based on a text file.

The class of the enumerator of a sequential input file implements the class *Enumerator*, as well. The file f itself is an important data member of this class. The operators *First()* and *Next()* read the next element of the file into a data member called *df*, the value of which can be asked by the operator *Current()*; the operator *End()* gives 'true', if the last reading has already been unsuccessful.

```
template <typename Item>
class SeqInFileEnumerator :  public Enumerator<Item> {
protected:
   std::ifstream   f;
   Item            df;
public:
   enum Exceptions { OPEN_ERROR };
   SeqInFileEnumerator(const std::string& str){
      f.open(str.c_str());
      if(f.fail()) throw OPEN_ERROR;
      if(typeid(Item)==typeid(char)) f.unsetf(std::ios::skipws);
   }
   void First()             { Next();}
   void Next()              { f >> df;}
   bool End()       const   { return f.fail();}
   Item Current()   const   { return df; }
};
```

Let us look at a C++ implementation where the elements of a text file should be read (enumerated) as a sequential input file. The constructor gets the name of the text file, it opens an input dataflow (*ifstream*) to the file, and it throws an exception if the file does not exist. The template parameter denotes the type of the elements to be read in. We suppose that the elements of the text file are separated by white spaces, except that the single characters of the text file are wanted to be read one-by-one. In this case, the constructor switches off the automatic setting which skips the white spaces during the reading process. The *operator>>* reads the next element of the text file, which is used in the operators *First()* and *Next()*.

The enumerator of a one-dimension array is defined by a template class derived from an *Enumerator* where the type of the elements of this array is still denoted by the template parameter; however, this class must define the enumeration operators. Hence, the class has got two data members: the array, and the index traversing the elements of the array. This index is set to the first element by the operator *First()*, is increased by the operator *Next()*, is examined by the operator *End()* whether it has reached the end of the array, and points at the current element that is given back by the operator *Current()*.

2.3. Class of general programming theorem

The template class *Procedure* is the ancestor of all programming theorems. The famed programming theorems use the same processing cycle: they process the elements that are produced by an enumerator.

```
Init();
for (enor.First(); !enor.End(); enor.Next()) Do(enor.Current());
```

This general process can be placed into the method *Run()* of *Procedure*. The method *Run()* is sealed (must not be redefined) but it calls two methods (*Init(), Do()*) which are not implemented at this level; they will have to be defined right after derivation according to the current problem.

The process of the method *Run()* can be transformed into a more general form.

```
template <typename Item>
void Procedure<Item>::Run(){
   if (enum==NULL) throw ExpectedEnumerator;
   Init();
   for( First(); LoopCond(); enor->Next()){
      Do(enor->Current());
   }
}
```

Firstly, the method *enor−>First()* is substituted by the method *First()*, the default definition of which is equal to *enor−>First()* but this can be changed if necessary. For example, if a process is based on the enumerator which was earlier used but which has broken off and now its enumeration is wanted to be continued. In this case, the enumeration by *enor−>First()* should not start again; therefore, the method *First()* must be redefined by the empty statement.

Secondly, the original loop condition !*enor−>End()* is expanded through the calling of the method *WhileCond()* which gives back 'true' as default but this method can be modified and, if we want, the enumeration can be stopped earlier. (Like, for example, the summation of the elements of a sequence is needed, but only the elements before the first negative one.) The return value of the method *WhileCond()* depends on the current element of the enumeration (this is its input parameter): this element satisfies the given condition as long as the process goes on.

Thirdly, this expanded loop condition is encapsulated into the method *Loop-Cond()* which is redefined later at the definition of two programming theorems: the linear search and selection.

The class *Procedure* provides a method (*AddEnumerator()*) that can connect an enumerator to the general process. This enumerator is referred to by the data member *enor* which is a pointer in the C++ code. In order that the process cannot start without an enumerator, the state of this data member is checked in the method *Run()*: if the enumerator does not exist (*enor==NULL*), then an exception (*MissingEnumerator*) is thrown. The type of the enumerated elements is not known at this point in the class *Procedure*; hence, it is substituted by a template parameter (*Item*).

```
template <typename Item>
class Procedure {
protected:
   Enumerator<Item> *enor;
   Procedure():enor(NULL){}
   virtual void Init()= 0;
   virtual void Do(const Item& current) = 0;
   virtual void First() {enor->First();}
   virtual bool WhileCond(const Item& current) const {return true;}
   virtual bool LoopCond() const
      { return !enor ->End() && WhileCond(enor->Current()); }
public:
   enum Exceptions {MissingEnumerator};
   void Run();
   void AddEnumerator(Enumerator<Item>* en) { enor = en;}
   virtual ~Procedure(){}
};
```

2.4. Classes of programming theorems

The programming theorem **summation** may solve several kinds of problems. It can calculate the sum or product of numbers, it can create the union of sets, it may be a conditional summation like a counting; furthermore, it can copy, pick out, and merge elements. This is reflected in the general template class *Summation*.

The template parameter *Item* denotes the type of the processed elements; the *ResultType* is the type of the result of summation. This output is referred by the data member *result*. In C++ code, this is a pointer which points to the memory site allocated or got by the constructor.The value of *result* can be asked with the method *Result()*.

The inherited method *Do()* is implemented by the conditional activity *if (Cond(e)) Add(e)*. In this activity, the variable e is the input parameter of *Do()*; that is, e is the currently enumerated element. The method *Do()* is not redefined in the classes derived from *Summation*. The method *Init()* (which initializes the member result) and the method *Add()* (which can modify the member *result*) cannot be defined in the *Summation*. They can only be defined in the classes which are derived from *Summation*. The method *Cond(), WhileCond()* and *First()* are also modified in these derived classes if needed.

```
template < typename Item, typename ResultType = Item >
class Summation :  public Procedure<Item> {
protected:
   ResultType *result;
   bool inref;
   Summation(){ inref = true; result = new ResultType; }
   Summation(ResultType *r){ inref = false; result = r; }
   void Do(const Item& e){ if(Cond(e)) Add(e);}
   virtual void Add(const Item& e) = 0;
   virtual bool Cond(const Item& e) const { return true;}
public:
   ResultType Result() { return *result; }
   ~Summation(){ if(inref) delete result;}
};
```

The programming theorem **counting**, which is a special summation, is used very frequently; hence, its general definition is made in the template class which is an inheritance of the summation. Here, the *ResultType* must be substituted by the integer. Thus, the reference **result* is an integer, too. This number is allocated by the first constructor of *Summation*, is set null in the method *Init()*, and is increased by the method *Add()*. In the descendants of *Counting*, the method *Cond()* will be allowed to be redefined as the current condition of counting.

```
template <typename Item >
class Counting :  public Summation<Item, int>{
public:
   Counting():Summation<Item,int>(){}
protected:
   void Init()              { *Summation<Item,int>::result = 0;}
   void Add(const Item& e)  { ++*Summation<Item,int>::result;}
};
```

The template class **selection** implements the method *Init(), LoopCond()* and *Do()* so that the method *Run()* selects the first element of an enumeration that satisfies a special condition. When we use this programming theorem, we suppose that this element exists. The condition can be given with the definition of the abstract method *Cond()* in the class which is derived from *Selection.* Therefore the method *LoopCond()* should be redefined as the negation of *Cond().* The method *Init()* and *Do()* belong to the empty statement.

```
template <typename Item >
class Selection :  public Procedure<Item>{
protected:
   void Init(){}
   void Do(const Item& e) {}
   bool LoopCond() const
      { return !Cond(Procedure<Item>::enor->Current()); }
   virtual bool Cond(const Item& e) const = 0;
};
```

The template class **linear search** is suitable to generate the pessimist (normal) or the optimist search. The pessimist linear search looks for the first element of an enumeration which satisfies a special condition. The optimist version checks whether the given condition holds on all elements of the enumeration. The origin of the name 'pessimist' and 'optimist' derives from the meaning of the value of the logical data member l before termination. The 'false' means that "we has not yet found any fitting element, it may not exist"; the 'true' means that "every element which have been checked is good, it is sure that all elements correspond to the given condition." A special template parameter (*optimist*) has been introduced to set the pessimist or optimist property of the search. Its default value is 'false' which defines a pessimist linear search. The method *Init()* and *LoopCond()* depend on the value of this template parameter. The method *Do()* changes the value of l to 'true' if the current element satisfies the condition. The data member elem contains the last-checked element of the enumeration. Search results can be asked with the methods *Found()* and *Elem()*.

The condition of the search can be given with the definition of the abstract method *Cond()* in the classes derived from *LinSearch*.

```
template < typename Item, bool optimist = false >
class LinSearch :  public Procedure<Item> {
protected:
   bool l;
   Item elem;

   void Init() {l = optimist; }
   void Do(const Item& e) {l = Cond(elem = e);}
   bool LoopCond() const
      { return (optimist?l:!l) && Procedure<Item>::LoopCond(); }
   virtual bool Cond(const Item& e) const = 0;
public:
   bool Found()   const { return l;}
   Item Elem()    const { return elem;}
};
```

The template class *MaxSearch* defines a **general maximum search** that encapsulates the ordinary maximum selection and the conditional maximum search. It has got three data members. The logical variable l signs the success of the search. The variable *optelem* saves the optimal element which is found until the current moment and the value of this element is in the variable *opt*. These three protected members are asked with the public methods of *Found(), OptItem()* and *Opt()*.

The virtual abstract method *Func()* maps the value from an element, and this value is compared to another. At this level, the type of the enumerated elements and the type of their value are unknown; they are substituted by template parameters. The parameter *Item* denotes the type of the element; *Value* is the type of the values compared to each other. The virtual method *Cond()* contains the constraint of the conditional maximum search. Its default value is 'true' (in this case, we have an ordinary maximum selecting) and it can be changed in the descendents of *MaxSearch* by the redefinition of *Cond()*.

The third template parameter is the type of the comparing (*Compare*). *Compare* can define the data member better which can select the best one out of two elements. If the member *better* can call the *operator(,)*, then the expression *better(left,right)* could compare the parameters *left* and *right*. After the template parameter *Compare* is substituted by the template class *Greater* (see below) (which interprets *operator(,)* as the relation ">"), then the expression *better(left,right)* is equal to the expression *left>right*. This is required so as to find the maximum.

```
template <typename Value> class Greater{
public:
   bool operator()(const Value& left, const Value& right)
      { return left >right; }
};
```

We can make the template class *Less*, which includes the relation "<" as an *operator(,)*. If *Compare* is substituted by *Less*, the minimum can be found.

```
template < typename Item, typename Value = Item,
          typename Compare = Greater<Value> >
class MaxSearch :  public Procedure<Item> {
protected:
   bool   l;
   Item   optelem;
   Value  opt;
   Compare better;

   void Init(){ l = false;}
   void Do(const Item& current) {
          Value val = Func(current);
          if ( !Cond(current) ) return;
          if (!l) { l = true; opt = val; optelem = current; }
          else if (better(val,opt)){ opt = val; optelem = current; }
   }
   virtual Value Func(const Item& e) const = 0;
   virtual bool Cond(const Item& e) const { return true;}
   public:
   bool   Found()     const { return l;}
   Value  Opt()       const { return opt;}
   Item   OptElem()   const { return optelem;}
};
```

As the class *MaxSearch* is derived from the class *Procedure*, its main task is to implement the abstract methods of *Procedure*. The method *Init()* sets the member variable *l* as 'false'. The member *Do()* contains the alternative construct which is well-known form the algorithm of the conditional maximum search. The concrete maximum searches can be derived from the *MaxSearch* where the method *Init()* and *Do()* should not be redefined but the method *Func()* must be and the method *Cond(), WhileCond()* and *First()* may be defined.

3. Solution of two problems

Now, two problems are going to be solved on the basis of our template class library. The abstract solution of both problems should be planned by using analogous programming. It will be enough to name the corresponding programming theorem, to define its specialties, and to give a fitting enumeration As we see, both problems require a unique enumerator, the class of which is derived from the class *Enumerator* defined by us. At the same time, the famed enumerator is needed since it can enumerate the elements of text files which are the input data of both problems. Additionally, in the second problem, the definition of the unique enumerator requires another programming theorem.

3.1. Merely common numbers

Example: *Two text files contain integers. The numbers are in increasing order and are separated by white spaces. Is it true that there is no number among them which does not appear in both?*

To solve this problem, the numbers of the text files should be enumerated, and every number should be denoted by 'true' if they are in both text files. If this sign is 'true' for all numbers, then the answer to the question of this problem is 'true'; otherwise, it is 'false'. It can be decided with an optimist linear search.

```
struct Number{
   int n;
   bool c;
};
```

In the class of the optimist linear search (*MyLinSearch*), the condition should be redefined. The enumerated elements are pairs of integer and logical value (*Number*) where the logical value shows whether the current integer is a common element of the files or not. The condition of the linear search (*Cond()*) needs this logical value.

```
class MyLinSearch:public LinSearch<Number,true>{
protected:
   bool Cond(const Number &e) const { return e.c;}
};
```

The solution needs the special enumerator (*NumbersEnumerator*) which can enumerate the integers of the two text files so that all integers are touched only once and that an integer is denoted by ‘true’ if it appears in both files.

```
class NumbersEnumerator :  public Enumerator<Number> {
protected:
   SeqInFileEnumerator<int> *x, *y;
   Number number;
   bool end;
public:
   NumbersEnumerator(const string &str1, const string &str2);
   ~NumbersEnumerator(){ delete x; delete y; }
   void First(){x->First(); y->First(); Next();}
   void Next();
   bool End() const { return end;}
   Number Current() const { return number;}
};
```

The implementation of the method *Next()* is derived from the merging algorithm.

```
void NumbersEnumerator::Next() {
   if(end = x ->End() && y->End()) return;
   if(y->End() || (!x->End() && x->Current()<y->Current())){
      number.n = x->Current(); number.c = false;
      x->Next();
   }else if(x->End() || (!y->End() && x->Current()>y->Current())){
      number.n = y->Current(); number.c = false;
      y->Next();
   }else if(!x->End() && !y->End() && x->Current()==y->Current()){
      number.n = x ->Current(); number.c = true;
      x->Next(); y->Next();
   }
}
```

The constructor indirectly opens the text files and creates their enumerators.

```
NumbersEnumerator::NumbersEnumerator(const string &str1,
                                     const string &str2){
   try{
      x = new SeqInFileEnumerator<int>(str1);
      y = new SeqInFileEnumerator<int>(str2);
   }catch(SeqInFileEnumerator<int>::Exceptions ex){
      if(ex==SeqInFileEnumerator<int>::OPEN_ERROR)
         cout << "Non-exisiting file" << endl; exit(1);
   }
}
```

The main program is very simple. It is enough to create the activity of the optimist linear search and the enumerator, to connect the enumerator to this activity, then to execute the activity, and to reveal the result.

```
   MyLinSearch lin;
   NumbersEnumerator it("input1.txt", "input2.txt");
   lin.AddEnumerator(&it);
   lin.Run();
   if(lin.Found())   cout << "All numbers are common";
   else              cout << "There is non-common number";
```

3.2. The best student

Example: *In a text file, students' marks are listed in rows. Every row contains a student's identity number and one mark separated by spaces. The rows are ordered according to identity numbers. Give the identity number of one of the best (average) student.*

This problem can be solved by maximum selection where the average of the students should be compared.

The type of *Student* is defined first which can describe either a pair of identity and mark, or a pair of identity and average. In order to read out a new row from the file, the *operator>>* is overloaded on *Student.*

```
struct Student {
   string id;
   double result;
};
ifstream& operator>>(ifstream &in, Student &e) {
   in >> e.id >> e.result;
   return in;
}
```

In the maximum selection (*MyMaxSearch*), it is enough to define the method *Func()* which is mapped from a pair of identity number and average to the average.

```
class MyMaxSearch :  public MaxSearch <Student, int> {
protected:
   int Func(const Student &e) const { return e.result;}
};
```

The pairs of identity number and mark can be read out from the file with an enumerator of *SeqInFileEnumerator<Student>*. In our maximum selection, it is not enough to enumerate these pairs. We need the unique enumerator (*StudentEnumerator*) that can enumerate all students along with their identity number and their averages. The production of averages requires the special summation which can count the marks of the same identity number and cumulate the sum of them. The average can be calculated from these two numbers.

The type of this new enumerator is derived from the class *Enumerator* and based on the enumerator of the text file (f); this enumerator can read the text file row by row. The enumerator f is created by the constructor which gets the name of the text file. The member *student* contains the current student's data; *end* signs if the enumeration of averages is finished.

```
class StudentEnumerator :  public Enumerator<Student> {
protected:
   SeqInFileEnumerator<Student>* f;
   Student student;
   bool end;

public:
   StudentEnumerator(const string &str);
   ~StudentEnumerator(){ delete f;}
   void First(){f->First(); Next();}
   void Next() ;
   bool End() const { return end;}
   Student Current() const { return student;}
};
```

The constructor indirectly opens the text file and creates its enumerators.

```
StudentEnumerator::StudentEnumerator(const string &str) {
   try{ f = new SeqInFileEnumerator<Student>(str); }
   catch(SeqInFileEnumerator<Student>::Exceptions ex){
      if(ex==SeqInFileEnumerator<Student>::OPEN_ERROR)
         { cout << "Non-exisiting file" << endl; exit(1);}
   }
}
```

The method *First()* is almost equal to the method *Next()*, but contains the reading of the first pair of identity-mark from the text file. The method *Next()* checks whether there exists a next row. If so, there also exists a next student, and his average has to be computed with another programming theorem. Moreover, not only one but two theorems are needed: one counts the marks of the same identity number; the other adds them. Both are the same loop summation: they can be transformed into one *(MySummation)*.

```
void StudentEnumerator::Next() {
   if(end = f->End()) return;
   student.id = f ->Current().id;
   MySummation sum(student.id);
   sum.AddEnumerator(f);
   sum.Run();
   student.result = sum.Result().sum/sum.Result().count;
}
```

These double activities initially require the current identity number, and results in two numbers which are saved in the structure of *Pair*.

```
struct Pair {
   double sum;
   int count;
};
```

An enumerator of our double summation *(MySummation)* would be required to traverse the marks belonging to the current identity number (it is saved in the member *id*). However, since we would like to avoid creating too many enumerators to every identity number, we have only one enumerator, the enumerator *f* (based on the text file) which enumerates all the marks. This enumerator has already begun when a summation starts to work, and has not finished at the end of the summation. This enumerator should not be restarted in the summation, so the method *First()* should be redefined with the empty statement. The double summation ends when the new identity number enumerated by *f* is different to the current one; hence, the method *WhileCond()* should accordingly be redefined.

```
class MySummation :  public Summation<Student,Pair> {
protected:
   string id;
   void First(){}
   void Init(){result->sum = 0; result->count = 0;}
   void Add(const Student &e)
      {result->sum+=enor->Current().result; ++result->count;}
   bool WhileCond(const Student &e) const { return e.id == id;}
public:
   MySummation(const string &str):Summation<Student,Pair>(),id(str){}
};
```

Finally, let us examine the main program. This program creates the activity objects of the maximum selection and of the enumerator, it connects them, then it executes the maximum selection, and it writes out the result.

```
  MyMaxSearch max;
  StudentEnumerator it("input.txt");
  max.AddEnumerator(&it);
  max.Run();
  if(max.Found())   cout << "The best student:  " << max.OptElem().id;
  else              cout << "There are no students!";
```

4. Conclusions

These two problems presented here may be enough to prove that numerous similar problems can be solved on the basis of our template class library.

The application of the library makes the usage of analogous programming more conscious. However, the algorithmic thinking cannot be used to create the solution here but the analysis of the problem should instead be in focus.

The implementation of the solutions gives a sample of the object-oriented programming. It reveals the role of inheritance, virtual methods and template parameters, and it exemplifies how the process of the code-reusability among editing, compiling and running can be dismembered.

The solutions based on the library are especially nice in the programmer's point of view. It is very interesting that there can be found only one loop in the solutions, specifically in the method *Run()* of the ancestor class of all programming theorems. Neither in the other classes of the library nor in its own code should a newer loop be written.

The template class library presented here is not made for industrial applications. It perfectly underlines the object-oriented techniques, but I do not think that its usage would be simple or expedient in practice. The implementation of a programming theorem (which is, in fact, one loop) is not hard; it can be done directly without deriving it from a template class library. The understanding and correct use of a complex template class library is more difficult. Therefore, this article shows the limit when it is worth using object-oriented tools to resolve a problem, and when it has no advantage in practice.

References

[1] G. Booch, J. Rumbaugh and I. Jakobson, *The Unified Modeling Language User Guide*, Second Edition, Addison-Wesley, 2005.

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

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

[4] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st Edition, ISBN 0201633612, Pearson Education Inc., Addison Wesley Professional, 1995.

[5] T. Gregorics, Programming theorems on enumerator, *Teaching Mathematics and Computer Science, Debrecen* **8**, no. 1 (2010), 89–108.

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

[7] J. Rumbaugh et al., *Object Oriented Modeling and Design*, Prentice Hall, 1991.

[8] 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 May, 2011)