JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 5, No. 1, January-February 2006

Constructing persistent object-oriented models with standard C++

Alexander Kozynchenko, Department of Information Technology and Media, Mid Sweden University, Sundsvall, Sweden

Abstract

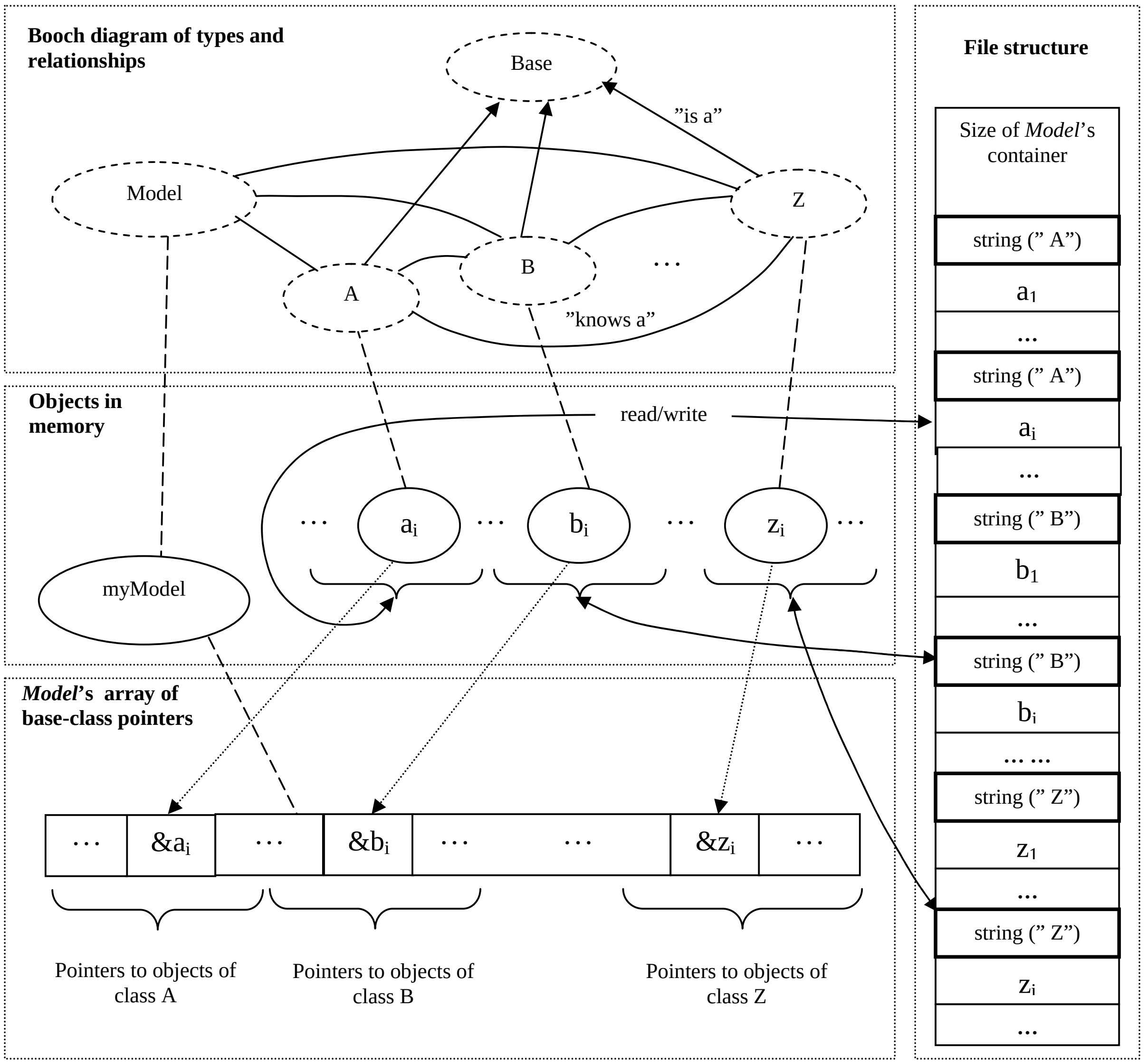
In this paper, it is suggested an approach and a design pattern for developing object-oriented models that need to be persistent, including the databases of moderate size, with using only the standard C++ and its file storage facilities, and without using specific C++ dialects or any support of external libraries providing the persistence. Objects of the model may be of a great variety of types, belonging to a complex class hierarchy, and are considered to be of rather general structure, containing both pointers to any other model's objects and dynamically allocated arrays of various types. The main idea consists in that all types involved are considered as classes derived from the unique base class with the minimal common interface. Classes' objects are allocated dynamically, and the pointers are kept in the model's base-class pointers container, which provides sorting, searching, and changing the objects kept. The objects' serialization, reading, and management is implemented with using the virtual functions, list of type names, and object factory technique.

1 INTRODUCTION

In the paper, it is treated a problem of developing the object-oriented models with objects dynamically allocated on the free store and having been linked to each other with pointers. Object-oriented databases and knowledge-based systems are considered to be an important kind of such models. Once created, the model may be running some time, change, and then be waiting for the next run. Naturally, such a model should be kept permanently in the secondary storage ready to load and use instead of creating it and its relationships from the very beginning each time. When programming the persistent object-oriented models in C++, it would be useful to find out the ways of writing the programs on the standard C++ language [Stroustrup1997] rather than using extensions to the C++ like the ObjectStore and POET systems [Piattini2000] or support of class libraries like MFC.

2 OUTLINE OF THE STRUCTURE OF THE PROPOSED OBJECT-ORIENTED MODEL AND C++ DESIGN PATTERN

The possible solution of the problem in point is as follows. The object-oriented model may be represented as a class, say, `Model`, which main purpose is to be a repository for addresses of objects of the user-defined types. All these types involved in constructing the model are considered to be as classes derived from the common base class, say, `Base`, like shown at the Booch diagram in Fig. 1.



Booch diagram of types and relationships
Base
"is a"
Model
Z
B
A
. . .
"knows a"
Objects in memory
read/write
. . . a_i . . . b_i . . . z_i . . .
myModel
Model's array of base-class pointers
. . . | $\&a_i$ | . . . | $\&b_i$ | . . . | . . . | $\&z_i$ | . . .
Pointers to objects of class A
Pointers to objects of class B
Pointers to objects of class Z
File structure
Size of *Model*'s container
string (" A")
a_1
...
string (" A")
a_i
...
string (" B")
b_1
...
string (" B")
b_i
... ...
string (" Z")
z_1
...
string (" Z")
z_i
...

Figure 1. Generalized scheme of the C++ persistent object-oriented model

The derived classes may, in turn, be the members of other inheritance hierarchy. The dynamically allocated objects of these derived classes are identified by the base-class pointers, which are placed into a container, for instance, the STL `deque` being a member of Model. An example of the declaration of class Model that would provide minimal functionality is as follows:

```
typedef deque<Base*> Repository;
typedef Repository::iterator It;
typedef pair<It, It> Pair;
typedef Base* (*FuncPCr)();
typedef bool (*FuncPLess)(Base*, Base*);
struct FuncPointers
{
  FuncPCr pCreate;
  FuncPLess pLess;
  FuncPointers(FuncPCr pC, FuncPLess pL)
                               :pCreate(pC),pLess(pL){}
};
typedef map<string, FuncPointers> TypeList;

class Model
{
private:
  Repository v;
  static TypeList typeList;
  filebuf fb;
  filebuf* pWrite;
  void OpenToWrite();
  Base* CreateObject(const string& typeName);
public:
  Model();
  ~Model();
  Repository& GetV();
  void SetTypeList(const string& typeName,
                                     const FuncPointers&
fps);
  static TypeList& GetTypeList();
  void SetV(Base* p);
  void PrintModel() const;
  void WriteToFile();
  void ReadFromFile();
  struct LessTypeNames
  {
    bool operator()(Base* lhs, Base* rhs) const;
  };
  void SortByTypes();
  Pair EqualRange(It first, It last,
                  const string& typeName) const;
  struct LessInType
  {
    bool operator()(Base* lhs, Base* rhs) const;
  };
  void SortInType(const string& typeName);
};
```

The second Model's data member, typeList, representing a list of types involved in the model is used in the object factory (see, e.g., [Alexandrescu2001]) when reading data from a secondary storage and creating objects, as well as in the factory-like algorithm when sorting objects of a given type. It is implemented as an associative container, the STL map, with keys of the string type given as type names (or type IDs) defined in the base class. Associated values are objects of type

FuncPointers that keep pointers, FuncPCr and FuncPLess, to global functions providing the object creation and a less-than sorting criterion for a given class (e.g., functions CreateA and LessA described in the next section). The member typeList is filled up in the Model's default constructor.

There are two more Model's auxiliary data members, fb and pWrite, declared as an object of class basic_filebuf and a pointer to this type, which are defined in the Model's default constructor as well. This is made for the sake of data safety: the failed attempt to open a file at the stage of creating a Model's object will not cause the risk of losing any Model's data. So, the Model's default constructor is defined as follows:

```
Model::Model()
{
  SetTypeList(A::GetTypeName(), FuncPointers(CreateA,
LessA);
  SetTypeList(B::GetTypeName(), FuncPointers(CreateB,
LessB);
  //...
  SetTypeList(Z::GetTypeName(), FuncPointers(CreateZ,
LessZ);
  OpenToWrite();
}
```

where A, B, …, Z – types involved in the model. Private helper function OpenToWrite is intended for keeping the file open for writing while the Model's object exists. It should contain some exception handling:

```
void Model::OpenToWrite()
{
  pWrite = fb.open("myfile.dat",ios::out|ios::binary);
  if(!pWrite)
   {
   perror("perror says open for writing failed" );
   getchar();
   this->~Model();
   abort();
  }
}
```

Member function SetV adds a new base-class pointer to the array.Definitions of other important member functions and nested function-like structures will be given in the sections 4 and 5.

Thus, each object involved in the model is identified by both the pointer and the corresponding index of the array's element where this pointer is kept. After finishing a work with the Model, all objects must be stored in the hard disk in such a way that all relationships accomplished by the pointers would be restored correctly when retrieving the data later on. In order to meet this requirement, we have to keep the "one-to-one" relationship between the object's data member – a pointer to some dynamically allocated object, and the index of the database's array cell where this pointer is kept.

For solving problems related to model persistence and data processing, the proper scheme of a type name management is developed. It includes type names (actually type identifiers) of type `string` provided by the `Model`'s objects and kept in `typeList`. Duplicate type names clash is excluded in associative arrays such as the STL `map`. As stated above, `typeList` is involved in the factory algorithms.

3 DEFINITIONS OF CLASSES INVOLVED IN THE MODEL. OBJECTS' SERIALIZATION AND READING

Here it's time to break describing the class `Model` and to talk about the structures of objects held in the model. The class `Base` has minimal functionality necessary for making objects persistent and for preserving objects' relationships. It may be declared as follows:

```
class Base;
typedef deque<Base*> Links;
typedef deque<int> Indexes;

class Base
{
private:
  string typeName;
  Links links;
  Indexes indexes;
public:
  Base(const string& typeName): typeName(typeName){}
  virtual ~Base(){}
  string& GetTypeName();
  void PutLink(Base* p);
  Links& GetLinks();
  void PutIndex(int i);
  Indexes& GetIndexes();
  void WriteTypeName(ostream& out);
  void ReadTypeName(istream& in);
  virtual void Write(ostream& out);
  virtual void Read(istream& in);
};
```

The class `Base` contains the data member `typeName` that is initialized in `Base`'s parameterized constructor, as well as two dynamic arrays - `links` and `indexes`. The array `links` keeps base-class pointers to other objects to be linked with, providing the association relationships. Just before writing data to the hard disk and terminating the program the `links` may be emptied because the pointers become invalid. Indexes of the `Model`'s array `v` corresponding to the `Base`'s pointers are kept in the array `indexes`. That is, if `links[j]` is equal to `v[i]` then `indexes[j]` is equal to `i`. In contrast with the `links`, the array `indexes` is used only when writing and reading data, so it can be empty at almost all run time. For the sake of brevity, we don't give definitions of the access functions - `PutLink`, `GetLinks` etc., they are implemented in a usual way.

The class `Base` has two non-virtual member functions for writing/reading the type name:

```
void Base::WriteTypeName(ostream& out)
{
  out << typeName.size();
  if(typeName.size() > 0)
    for(int i = 0; i < typeName.size(); ++i)
     out << typeName[i];
}

void Base::ReadTypeName(istream& in)
{
  int sizeName;
  in >> sizeName;
  string tempName(sizeName,' ');
  if(sizeName > 0)
    for(int i = 0; i < sizeName; ++i)
      in >> tempName[i];
  typeName.swap(tempName);
}
```

Two virtual member functions `Write` and `Read` are intended respectively for the correct serialization and for subsequent creation of transient objects, and are to be overridden in derived classes. Their `Base`'s definitions are as follows:

```
void Base::Write(ostream& out)
{
  //serialization of the links:
  if(links.size() != indexes.size())
    throw SomeExceptionHandler("Sizes are not equal");
  char delimiter = '\0';
  out << indexes.size() << delimiter;
  if(links.size() > 0)
    for(int i = 0; i < links.size(); ++i)
      out << indexes[i] << delimiter;
}

void Base::Read(istream& in)
{
  //reading links:
  int sizeIndexes;
  char delimiter;
  in >> sizeIndexes >> delimiter ;
  Indexes tempIndexes(sizeIndexes);
  if(sizeIndexes > 0)
    for(int i = 0; i < sizeIndexes; ++i)
      in >> tempIndexes[i] >> delimiter;
  indexes.swap(tempIndexes);
}
```

Reliable equivalents of the streaming operators `<<` and `>>` are `write` and `read` member functions of the classes `ostream` and `istream` applying without delimiters.

As an example of the application class involved in the model, we take the class `A` of rather general view, namely, having a dynamic array of **`char`** (C-style string) and the STL dynamic array - `vector` of **`double`**.

```
class A: public Base
{
private:
  char* s;
  int sizeS;
```

```
    vector<double> a;
    static string typeName;
  protected:
    A(const A& rhs);
  public:
    A(const char* j = "");
    ~A();
    static string& GetTypeName();
    void PutA(double x);
    void PutS(char x);
    char* GetS() const;
    vector<double>& GetA();
    void Print() const;
    void Write(ostream& out);
    void Read(istream& in);
    friend ostream& operator<<(ostream& lhs, const A& rhs);
  };

  //global functions being related to the class A:
  Base* CreateA();
  bool LessA(Base* lhs, Base* rhs);
  //function comparing C-style strings lexicographically:
  bool Less(const char* lhs, const char* rhs);
```

Omitting ordinary routines of accessing, deep copy, etc. and focusing an attention to the correct serialization, we give the definitions of the overridden virtual functions `A::Write` and `A::Read`:

```
  void A::Write(ostream& out)
  {
    Base::Write(out);
    //serialization of the vector of double a:
    char delimiter = '\0';
    out << a.size() << delimiter;
    if(a.size() > 0)
      for(int i = 0; i < a.size(); ++i)
        out << a[i] << delimiter;
    //serialization of the dynamic array (C-style string) s:
    out << sizeS;
    if(sizeS > 0)
      for(int i = 0; i < sizeS; ++i)
        out << s[i];
  }

  void A::Read(istream& in)
  {
    Base::Read(in);
    //reading the vector a:
    int sizeA;
    char delimiter;
    in >> sizeA >> delimiter;
    a.resize(sizeA);
    vector<double> temp(sizeA);
    if(sizeA > 0)
      for(int i = 0; i < sizeA; ++i)
        in >> temp[i] >> delimiter;
    a.swap(temp);
    //reading the C-style string s:
    in >> sizeS;
    if(sizeS > 0)
    {
      delete [] s;
      s = new char[sizeS];
      for(int i = 0; i < sizeS; ++i)
        in >> s[i];
    }
  }
```

We should also note a global helper function `CreateA` responsible for creating an object of class `A` in the object factory and called when reading data in the model:

```
  Base* CreateA() { return new A; }
```

Another helper function that gives a sorting criterion specific for the class `A` is a global predicate `LessA`:

```
  bool LessA(Base* lhs, Base* rhs)
  {
    return = Less(reinterpret_cast<A*>(lhs)->GetS(),
                           reinterpret_cast<A*>(rhs)->GetS());
  }
```

It fulfils a static downcast of the arguments and calls a global predicate `Less` providing the lexicographical comparison of C-style strings.

4 MODEL'S DATA SERIALIZATION AND RETRIEVING

Here we can return to discussing the `Model`'s member functions, centering on data writing/ reading routines and other related functions. First of all, it is a function for adding new pairs of `typeName` and functin pointers to the associative array `typeList`:

```
  void Model::SetTypeList(const string& typeName,
                                              const FuncPointers&
  fps)
  {
    typeList.insert(pair<string, FuncPointers>(typeName,
  fps));
  }
```

Further, let us consider a member function responsible for serialization – the function `Model::WriteToFile`. At the moment just before writing the model to the secondary storage, the array `links` of each object being kept in the `Model`'s array holds base-class pointers to others objects according to some relationship scheme. Elements of the array `indexes` intended for keeping indexes of the `Model`'s array `v` have not been defined so far because at run time the objects can be identified by pointers. Obviously, these pointers will become invalid after computer shutting down, so other objects' identifiers should be introduced: it could be the indexes of the `Model`'s array.The initial fragment of the function `Model::WriteToFile` sets the indexes `k` of the `Model`'s array elements `v[k]` as the elements of the `indexes` arrays:

```
for(int i = 0; i < v.size(); ++i)
{
  v[i].GetIndexes().resize(v[i].GetLinks().size());
  for(int j = 0; j < v[i]->GetLinks().size(); ++j)
    for(int k = 0; k < v.size(); ++k)
      if(v[i]->GetLinks()[j] == v[k])
        v[i]->GetIndexes()[j] = k;
}
```

The next fragment for writing the objects to a binary file follows:

```
ostream out(pWrite);
char delimiter = '\0';
out << v.size() << delimiter;
for(i = 0; i < v.size(); ++i)
{
  v[i]->WriteTypeName(out);
  v[i]->Write(out);
}
```

It writes down the size of the `Model`'s array first. Then a name of the type is being written just before the corresponding object as shown in the Fig.1, the segment "*File structure*". In principle, such a serialization procedure may be just an intermediate saving, and we can continue to work with the model. The `filebuf` object `fb` is not closed and a pointer `pWrite` is not equal to zero.

When we start retrieving the data from a file by calling a function `Model::ReadFromFile`, first we close the file object `fb` if it is opened for writing and then fulfill the following operations:

- Open the file for reading with some kind of exception handling.
- Model's array size is being read and the array `v` is resized.
- Further, the object factory is organized: within the loop, the object's type name is being read and an object of the same type is created by default and dynamically allocated due to calling a private helper member function `CreateObject`.
- The polymorphic call of the object's virtual function `Read` fills the object with the actual content.
- `Model`'s array is being filled with base-class pointers to the objects in the order of reading.
- Relationships between objects are restored with using the `Model`'s array indexes kept in the `indexes` arrays.
- File object `fb` is closed and re-opened for possible subsequent writing.

The code example for `ReadFromFile` is as follows:

```
void Model::ReadFromFile()
{
  if(pWrite != NULL)
  {
    fb.close();
    pWrite = NULL;
  }
  filebuf* pRead = fb.open("Model.dat",
ios::in|ios::binary);
```

```
    if(!pRead)
    {
      perror("perror says open for reading failed");
      this->~Model();
      abort();
    }
    istream in(pRead);
    int size;
    char delimiter;
    in >> size >> delimiter;
    v.resize(size);
    Base* pB;
    for(int i = 0; i < size; ++i)
    {
      int sizeName;
      in >> sizeName;
      string typeName(sizeName, ' ');
      if(sizeName > 0)
        for(int i = 0; i < sizeName; ++i)
          in >> typeName[i];
      pB = CreateObject(typeName);
      pB->Read(in);
      v[i] = pB;
    }
  //restore objects' relationships:
    for(i = 0; i < v.size(); ++i)
    {
      v[i]->GetLinks().resize(v[i]->GetIndexes().size());
      for(int j = 0; j < v[i]->GetIndexes().size(); ++j)
        v[i]->GetLinks()[j] = v[v[i]->GetIndexes()[j]];
    }
    fb.close();
    OpenToWrite();  //keep the file open for furhter writing
  }
```

A helper function `CreateObject` operating with the list of types finds a proper function pointer to a global function, like `CreateA`, that, in turn, directly creates an object of a type specified in the argument:

```
  Base* Model::CreateObject(const string& typeName)
  {
    TypeList::const_iterator it = typeList.find(typeName);
    if(it == typeList.end())
    {
      perror("perror says unknown type found");
      this->~Model();
      abort();
    }
    return (it->second.pCreate)();
  }
```

5 MODEL'S DATA MANAGEMENT: SORTING OBJECTS

Objects being kept in the `Model`'s container should be sorted somehow in order to ensure effective data management. Sorting may be implemented in two stages:

first, sorting objects according to their type names held in the `Base`'s data member `typeName;`

second, sorting objects within each type using some keys specified for a given type.

The first step can be done by lexicographical sorting of the type names with a predicate provided by a nested function-like class `LessTypeNames`. The corresponding function object is used in the `Model`'s sorting member function `SortByTypes`. The code example is shown below:

```
bool Model::LessTypeNames::operator()(Base* lhs,
                                      Base* rhs) const
{
  if(lhs->GetTypeName().compare(rhs.GetTypeName()) <= 0)
    return true;
  else
    return false;
};

void Model::SortByTypes()
{
  stable_sort(v.begin(), v.end(), LessTypeNames());
}
```

In the second stage, it is required to delimit the objects'subsequences of the same type by using a member function `EqualRange`:

```
Pair Model::EqualRange(It first, It last,
                       const string& typeName) const
{
  Pair eRange;
  bool flag = false;
  while(first != last)
  {
    if(typeName.compare((*first)->GetTypeName()) == 0
                                        && flag == false)
    {
      eRange.first = first;
      flag = true;
    }
    if(typeName.compare((*first)->GetTypeName()) != 0
                                        && flag == true)
    {
      eRange.second = first;
      flag = false;
    }
    ++first;
  }
```

```
    if(flag == true)
      eRange.second = first;
    return eRange;
  }
```

Having known the delimiters obtained from `EqualRange`, we can sort objects of the same type in the member function `SortInType`:

```
  void Model::SortInType(const string& typeName)
  {
    Pair eRange = EqualRange(v.begin(), v.end(), typeName);
    sort(eRange.first, eRange.second, LessInType());
  }
```

The sorting predicate is provided by a nested function-like class `LessInType`:

```
  bool Model::LessInType::operator()(Base* lhs, Base* rhs)
  const
  {
    TypeList::const_iterator it =
                     GetTypeList().find(lhs->GetTypeName());
    return (it->second.pLess)(lhs, rhs);
  }
```

The function-call operator uses the list of types and function pointers for choosing a proper global predicate (e.g., `LessA`) for sorting objects of a specified type, that is, actually, based on the object factory technique.

6 SUMMARY

The described object-oriented model and the design pattern can represent a basis for developing persistent research models, object-oriented databases, and knowledge-based systems with using standard C++. The model can keep objects of various types, including ones involved in inheritance hierarchies. Objects may form a complex network of association relationships provided by the base-class pointers. This network remains persistent as well. Classes involved into the model must satisfy some requirements such as: be derived from a common base class having facilities for data serialization and management, contain a string data member as a type name identifier and pass it to the base class, define overridden virtual functions for data writing to and reading from a secondary storage, be accompanied with relevant global functions for object creation and for using as a sorting predicate. The corresponding function pointers together with the list of type names are used in the model's object factories when retrieving data and sorting. Using `std::deque` instead of `std::vector` in the base class arrays is a reliable solution. The design pattern provides only restricted data management facilities specific for the model in point, namely, two-stage internal sorting "by types"-"within a type". Minimal exception handling necessary for the correct program termination is included. Model allows repeated writing and reading in run time. The size of the model is naturally restricted by the virtual memory limits. However, it seems to be likely to use this approach as a basis for developing models of much greater sizes.

7 ACKNOWLEDGEMENTS

The author wishes to thank Mr. Mogens Hansen for very valuable comments and suggestions that helped indeed the author in improving the original idea.

REFERENCES

[Stroustrup1997] Bjarne Stroustrup: *The C++ Programming Language*, 3rd edition, Addison-Wesley, 1997

[Piattini2000] *Advanced Database Technology and Design*/ Mario G. Piattini, Oscar Díaz, editors, Artech House, 2000

[Alexandrescu2001] Andrei Alexandrescu: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001

About the author

Alexander Kozynchenko is a lecturer on computer science at the Mid Sweden University. He had obtained the honours diploma of electrical and mechanical engineer in 1974, the degree of candidate of science on aircraft control systems in 1982, and the rank of senior researcher in 1987. E-Mail: alexander.kozynchenko@miun.se