Case Study in ER Diagrams in the Context of Development of Generic Visualization Methods

Boris P. Leontyev*

Abstract—**Purpose of this paper is development of Generic Visualization methods (GVM) to enhance a Periodic Table of Generic Visualization Methods (PTGVM) by examining a particular case of ER diagrams. Work is based on the concept of the three-layer architecture for an information visualization system, including: (1) Periodic Table of GVM (PTGVM); (2) middleware, translating conceptual visualizations into classes, comprising processing logic in the spirit of Generic Programming, and translating methods of classes into instructions of a Graphic Programming Language of Computer-Aided Publishing System; and (3) graphic package, realizing visualization physically. Middleware can be developed exploiting such methods as (a) inheritance; (b) polymorphism; (c) parametrized templates (Boost C++ Graph Library). This paper relates to the case (a). Hierarchy of classes, representing nodes, was designed and implemented, programs to draw ER diagrams were developed, exemplary ER diagrams were generated. Quantitative signature for this particular representative visualization was defined to position it precisely inside a PTGVM. Fragments of implementation code are given.**

Keywords: generic programming, information visualization methods, periodic table

1 Introduction

Scientific, engineering, educational materials existing in different electronic and hard-copy publishable forms (books, articles, reports, etc.) and formats (PDF, HTML, etc.) may happen to contain hundreds of illustrations, or figures, or visualizations as an outcome of some information visualization process. Language of visualization is quite common, unalienable in these as well in many other fields.

Advantages of visualizations can be exploited to full extent if pertinent sophisticated technologies and tools are used. Required technologies can be specified as industrial-strength, integrative, effective, efficient and generic.

It can be expected, that these technologies should benefit, if based on computer science methods (object-oriented programming (OOP), software engineering, etc.).

There are two prerequisites, two prospective tools for further research such as (1) Periodic Table of Generic Visualization Methods (PTGVM) [1], [2], and (2) Boost C++ Graph Library (BGL) [3].

1.1 Periodic Table of Visualization Methods

At first, there is a publication [1], which introduced a concept of Periodic Table of Visualization Methods (PTVM).

PTVM provides following: (1) descriptive overview of the visualization domain; (2) repository, and a problem-solving tool, relating specific visualizations to visualization methods; (3) reduced complexity of visualization because of structuring domain of visualization; (4) research tool, recognizing similarities and differences among different types of visualization, and revealing logic of their development.

1.2 Periodic Table of Generic Visualization Methods

[2] examined a problem of developing Generic Visualization Methods (GVM) to enhance PTVM of [1] . Term "Generic" here implies use of programming technologies of highest degree of abstraction as a tool to develop information visualizations. [2] found [1] to be a suitable prototypical generalization approach for further research.

One of the ideas of [2] was to propose programming techniques to render a PTVM not mostly intuitive, but empowered with precise numerical evaluations.

It proposed to call a PTVM a PTGVM.

This proposal was based on a hypothesis, that there should be some organizing principle of building-up a Periodic Table, lying in the nature of things, and generic programming techniques can be considered as a candidate to reveal this organizing principle similar to the principle of organizing electrons in atoms.

If such a suggestion could be proved to be true, if discovered, use of such an organizing principle would have a great impact on practice of implementing visualizations.

*16/Jun/2008, Department of Computer Science, National University of Rwanda, BP 347, Huye, Province du Sud, Rwanda, Afrique Centrale, Tel: (250)03240730 Email: b.leontiev@tltsu.ru

[2] introduced a three-layer architecture of the information visualization system and enhanced signature with two extra dimensions to characterize visualizations quantitatively.

Three layer architecture includes: (1) PTGVM, characterizing information visualization methods from conceptual most abstract point of observation as a front-end; (2) middleware, comprising a hierarchy of classes, their methods, and logic of their processing. In other words middleware: (a) translates conceptual visualization methods of layer (1) into classes in terms of OOP, (b) comprises processing logic in the spirit of generic programming, and (c) translates methods of classes into instructions of a Graphic Programming Language (GPL) of Computer-Aided Publishing System (CAPS); (3) CAPS with graphical packages as a back-end.

Quantitative signatures for representative visualizations characterize complexity of application area (stands for groups), and complexity of content area for periods.

Use of signatures makes it possible to position visualizations precisely inside a PTGVM.

1.3 Boost C++ Graph Library

BGL [3] is a powerful problem-solution software tool, creating solutions to entire families of problems, it implements graph algorithms be broadly applicable. Idea of this paper is to prepare grounds to exploit potential of BGL for a search for solution of the problem of developing generic visualization methods.

2 Statement of the Problem

Tools and techniques, developed in [2] have a potential of putting each of the GVM into proper box in a PTGVM indisputably, make the table more precise, correct and enhance it as well as GVM themselves.

To make it fully operational and functional, more concrete, less declarative it is necessary to create a data base of representative visualizations (examining particular case studies) and run all of them through the proposed technology of developing visualizations:

1. properly attributing them;
2. debugging and improving constituents of technology: (a) PTGVM; (b) library of ADT, classes with their data members and member functions, etc.

Organizing principle of a PTGVM is use of ADT, classes to discriminate groups, so, they have to be discovered for concrete case study of this paper.

ER diagrams can be attributed to a graph visualization type, and BGL should prospectively be utilized to generate this particular case of visualizations.

If to keep in mind, that the final accomplishment for ER diagrams case study is development of a genuine generic visualization method in the framework of a PTGVM, following stages of development can be discriminated in the growing degree of complexity:

1. a stage of using imperative programming;
2. a stage of using generic programming:
   (a) OOP with inheritance;
   (b) OOP with polymorphism;
   (c) OOP with parametrized templates (BGL), i.e. a final genuine generic implementation.

Each of the stages, becoming more generic, could be considered to be a specific problem, solved separately in a sequence. After preceding problem has been solved, solution of the next is started.

This paper focuses on OOP with inheritance stage (2(a)) of using generic programming techniques.

3 Solution of the Problem

3.1 General Description of Approach

Source [4] (as a reference example) lets to distinguish following objects, which should be used in drawing ER diagrams:

1. Entities. They comprise a multiline string label, single- or double-line (in a case of a weak entity) rectangular frame;
2. Attributes of entities. They comprise a multiline string label, which can be underlined, if an attribute is a key attribute, and an elliptic frame;
3. Relations. They comprise again a multiline string label, and a single or double line (if a weak entity is involved) rhombic frame;
4. Links with cardinality as a one-character label. They can be realized as single- or double- connection lines. They connect entities and attributes, entities and relations, relations and attributes.

Multiline string label (both for entities and relations) has such characteristic dimensions as width and height, and they have to be computed.

Distance from the frame (rectangular or rhombic) to the label block can be set to `\fboxsep`, as it is done usually in LaTeX.

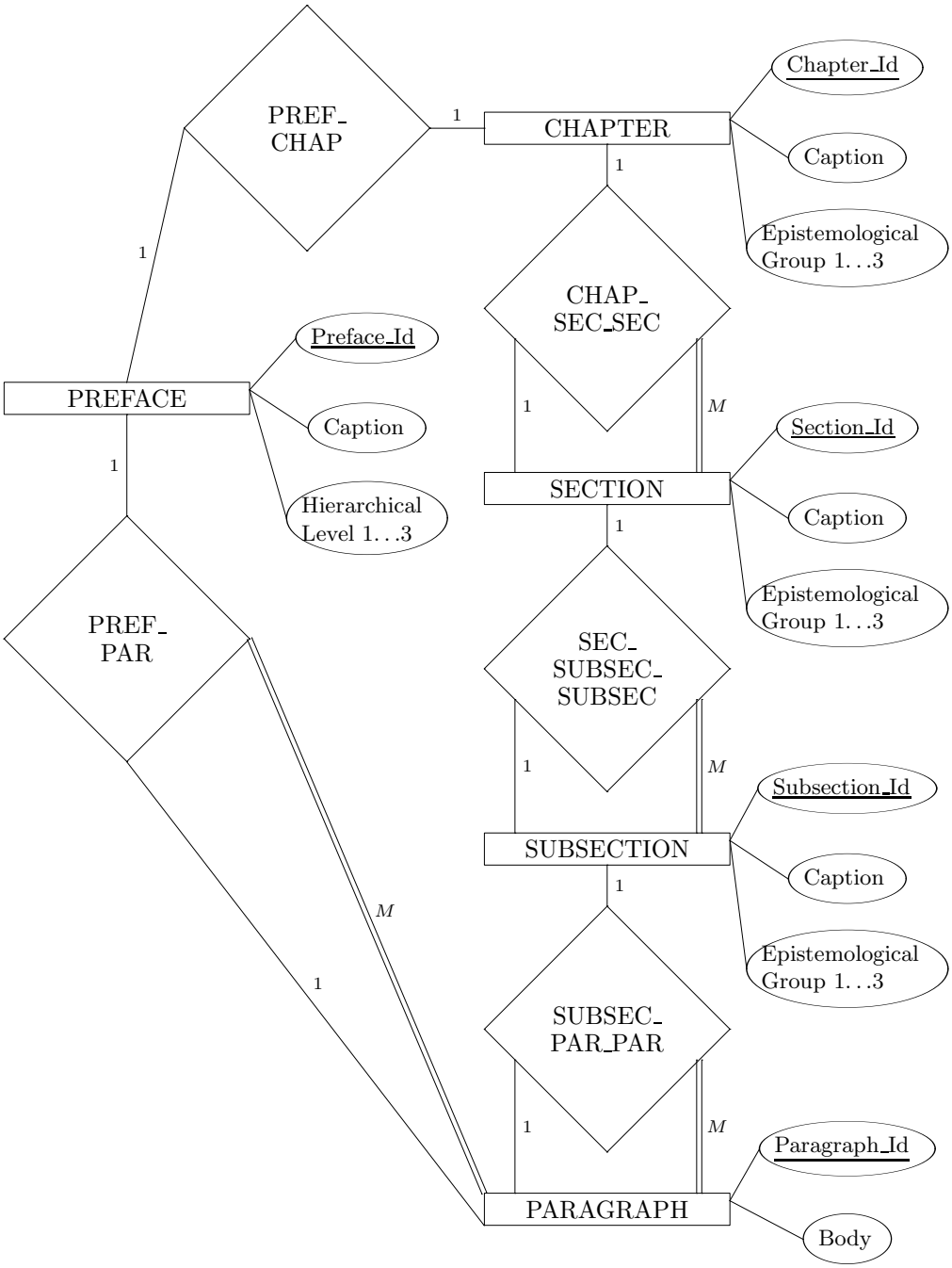

Figure 1: ER diagram for prototypical and target forms covering range of entities PARAGRAPH, CHAPTER, PREFACE

To render ER diagrams more aesthetically appealing horizontal characteristic dimensions of entities and relations should be harmonized in some way, for example, they should be the same. Labels of the objects should be optimized in such a way, that their forms should match closely encompassing frames.

In principal, in order to draw an ER diagram, it is required to master following elementary actions:

1. to build-up elementary objects, and visualize them;
2. put objects into a plane, mutually orienting them;
3. interconnect objects with lines, using some characteristic points of the objects.

It can be expected, that three classes have to be designed, and implemented, namely `Entity`, `Relation`, `Arrow`.

Methods or member functions of these classes should generate XY-pic-code [5], corresponding to the objects of the visualization.

Potential advantages of exploiting OOP become quite evident. Changes into design of objects are done once in the class implementation code, and after that all the instances of this class (which can be quite numerous) use the same new features, properties (data and methods) immediately, saving visualization development time.

Such a practice involves less errors, it becomes possible to focus on more conceptual, than low-level design, there is less typing in the process.

As for the inheritance, class `Relation` can be derived from class `Entity`, and it should override method `Draw()`, because in the case of `Entity` a frame should be rectangular, contrary to rhombic frame in a case of `Relation`.

Main organizing program should create (instantiate) objects, invoke their methods, generate XY-pic-code, and write it into a `tex`-file for further processing by LaTeX and other utilities.

In the following subsection principal details of technical implementation of the approach with corresponding fragments of code are briefly given.

3.2 Details of Technical Implementation with Fragments of Code

Declaration of the class `Entity` follows.

```
class Entity {
//
public:
  Entity( int, string, string, string, int,
       string, string, string, std::vector<
      struc_attrib > vector_attributes, int,
       int, int, int );
  void Define_Dimensions_for_Label() const;
  ~Entity();
  string Get_Ident_Partial() ;
  void Redefine_SizeX_Using_StandardWidth()
       const;
  void Define_Offsets() const ;
  void Draw() ;
  string Get_Predecessor() ;
  int Get_i_Object_Rel_Pos_u() ;
  // etc. for l (left), d (down), r (right)
  string Get_Label () ;
  int Get_Number_of_the_Object() ;
  void SizeXY_One_Quarters() ;

private:
  int i_Number_of_the_Object ;
  string str_Predecessor ;
  string str_Ident_Partial ;
  string str_Label ;
  int i_Number_of_Lines ;
  string str_Label_First_Line ;
  string str_Label_Last_Line ;
  string str_Max_Width_Label_Fragment ;
  //
  ofstream ofs_File_to_Write_to ;
  string str_File_Name_to_Write_to ;
  ofstream ofs_File_to_Draw_Object ;
  //
  std::vector< struc_attrib > vector_
      Attributes ;
  //
  int i_Object_Rel_Pos_u, i_Object_Rel_Pos_
      l, i_Object_Rel_Pos_d, i_Object_Rel_
      Pos_r ;

}; // end class Entity
```

Following structure defines attributes of `Entity`.

```
struct struc_attrib {
  string str_Label, str_Control_Width ;
  bool bool_Key ;
  int i_Attrib_Rel_Pos_u, i_Attrib_Rel_Pos_
      l, i_Attrib_Rel_Pos_d, i_Attrib_Rel_
      Pos_r ;
} ;
```

Declaration of the class `Relation` follows.

```
class Relation2 : public Entity {
public:
  Relation2 ( int, string, string, string,
      int, string, string, string, std::
      vector< struc_attrib >, int, int, int,
       int ) ;
  void print(ofstream) const ;
  void Define_Dimensions_for_Label() ;
  void Define_Rhombic_Dimension() ;
  void Redefine_StandardWidth_Using_Rhombic
      _Size() ;
  void Redefine_Rhombic_Size_Using_
      StandardWidth() ;
  void Define_Offsets() ;
```

```
  void Define_Size_Rhombic_One_Quarter() ;
  void Draw() ;
  ~Relation2 ();

private:
  ofstream ofs_File_to_Draw_Object ;
};
```

An example of creating an object `PARAGRAPH` of class `Entity` follows.

```
// 001 PARAGRAPH: 1-line entity
//     "PARAGRAPH"
// Attribute #1
//
struc_attrib_Utile.str_Label = "Paragraph
    \\_Id" ;
struc_attrib_Utile.str_Control_Width = "
    Paragraph\\_Id" ;
struc_attrib_Utile.bool_Key = true ;
struc_attrib_Utile.i_Attrib_Rel_Pos_u = 2;
// etc. for l (left), d (down), r (right)
//
vector_Attributes.push_back( struc_attrib_
    Utile ) ;
//
// Attribute #2
//
struc_attrib_Utile.str_Label = "Body" ;
struc_attrib_Utile.str_Control_Width = "
    Body" ;
struc_attrib_Utile.bool_Key = false ;
struc_attrib_Utile.i_Attrib_Rel_Pos_u = 0;
// etc. for l (left), d (down), r (right)
//
vector_Attributes.push_back( struc_attrib_
    Utile ) ;
str_predecessor = "" ;
Entity ent_Paragraph ( 1, str_predecessor,
     "Paragraph", "PARAGRAPH", 1, "
    PARAGRAPH", "PARAGRAPH", "PARAGRAPH",
    vector_Attributes,
i_object_rel_pos_u, i_object_rel_pos_l, i_
   object_rel_pos_d, i_object_rel_pos_r ) ;
ent_Paragraph.Define_Dimensions_for_Label(
     );
```

An example of creating an object `SEC_SUBSEC_SUBSEC` of class `Relation`

```
  //
  // 004 SEC_SUBSEC_SUBSEC: 3-line relation
       -> rhombus
  //
  //     SEC_
  //     SUBSEC_
  //     SUBSEC
  //
 i_object_rel_pos_u = 6, i_object_rel_pos_l
     = 0, i_object_rel_pos_d = 0, i_object_
    rel_pos_r = 0 ;
 vector_Attributes.erase( vector_Attributes
    .begin(), vector_Attributes.end()) ;
 str_predecessor = "Subsection" ;
 // str_predecessor = "" ;
 Relation2 rel_Sec_Subsec_Subsec ( 4, str_
    predecessor, "SecSubsecSubsec", "SEC
    \\_\\\\SUBSEC\\_\\\\SUBSEC", 3, "SEC
    \\_", "SUBSEC", "SUBSEC\\_", vector_
    Attributes, i_object_rel_pos_u, i_
    object_rel_pos_l, i_object_rel_pos_d, i
    _object_rel_pos_r ) ;

 rel_Sec_Subsec_Subsec.Define_Dimensions_
    for_Label() ;
 rel_Sec_Subsec_Subsec.Define_Rhombic_
    Dimension() ;
 rel_Sec_Subsec_Subsec.Redefine_
    StandardWidth_Using_Rhombic_Size() ;
```

Implementation of the method `Entity::Define_Dimensions_for_Label()` follows.

```
void Entity::Define_Dimensions_for_Label()
    const
{
  cout << "\\newlength{\\" << str_Ident_
      Partial << "SizeX}" << endl ;
  cout << "\\newlength{\\" << str_Ident_
      Partial << "SizeY}" << endl ;
//
cout << "\\setlength{\\" << str_Ident_
    Partial << "SizeX}{\\widthof{" << str_
    Max_Width_Label_Fragment << "} + \\
    fboxsep*\\real{2.0}}" << endl;
//
cout  << "\\setlength{\\" << str_Ident_
    Partial << "SizeY}{\\heightof{" << str_
    Label_First_Line << "} + \\depthof{" <<
    str_Label_Last_Line << "} + \\
    baselineskip*\\real{" << i_Number_of_
    Lines - 1 << ".0} + \\fboxsep*\\real
    {2.0}}"<< endl ;
//
cout  << "\\setlength{\\StandardWidth}{\\
    maxof{\\StandardWidth}{\\" << str_Ident_
    Partial << "SizeX" << "}}" << endl ;
}
```

This function results in following XY-pic-code for Entity `PARAGRAPH`.

```
\newlength{\ParagraphSizeX}
\newlength{\ParagraphSizeY}
\setlength{\ParagraphSizeX}{\widthof{
    PARAGRAPH} + \fboxsep*\real{2.0}}
\setlength{\ParagraphSizeY}{\heightof{
    PARAGRAPH} + \depthof{PARAGRAPH} + \
    baselineskip*\real{0.0} + \fboxsep*\real
    {2.0}}
\setlength{\StandardWidth}{\maxof{\
    StandardWidth}{\ParagraphSizeX}}
```

This function results in following XY-pic-code for Relation `SEC_SUBSEC_SUBSEC`.

```
\newlength{\SecSubsecSubsecSizeX}
\newlength{\SecSubsecSubsecSizeY}
\setlength{\SecSubsecSubsecSizeX}{\widthof{
    SUBSEC\_} + \fboxsep*\real{2.0}}
\setlength{\SecSubsecSubsecSizeY}{\heightof
    {SEC\_} + \depthof{SUBSEC} + \
    baselineskip*\real{2.0} + \fboxsep*\real
    {2.0}}
\setlength{\StandardWidth}{\maxof{\
    StandardWidth}{\SecSubsecSubsecSizeX}}

\newlength{\SecSubsecSubsecSizeRhombic}
\setlength{\SecSubsecSubsecSizeRhombic}{\
    maxof{\SecSubsecSubsecSizeX}{\
    SecSubsecSubsecSizeY}+\minof{\
    SecSubsecSubsecSizeX}{\
    SecSubsecSubsecSizeY}}
\setlength{\StandardWidth}{\maxof{\
    StandardWidth}{\
    SecSubsecSubsecSizeRhombic}}
```

Because of the lack of space some pertinent fragments of code such as implementation for `Entity::Draw()`, resulting XY-pic-code for invocation of `Paragaraph.Draw()`, etc. were omitted.

3.3 Some Observations on Practical Use of Developed Techniques

Techniques to draw ER diagrams, studied in this paper in the context of development of GVM, were applied practically as a tool to generate figures for publications, related to relational knowledge representation of publishable materials.

An example of visualizations, which can be developed, using techniques of this paper, is shown in Fig.1.

Acquired experience demonstrated, that use of OOP paradigm (C++ programming language) is more efficient comparing to development techniques, based on imperative programming paradigm (C programming language).

In the case of OOP visualizations are generated one or two orders quicker, process at whole is less error prone, resulting visualizations themselves are more aesthetically appealing.

4 Conclusions and Recommendations

Purpose of this paper is development of GVM to elaborate further a PTGVM by examining a particular case of ER diagrams and applying generic programming techniques and tools.

This paper can be considered to be a step towards building-up a generic library for information visualization.

Fragments of software implementations are given for the stage of development, exploiting OOP with inheritance.

Hierarchy of classes was designed and implemented, programs to draw ER diagrams were developed, and a set of exemplary ER diagrams was generated.

In order to position a visualization inside a PTGVM properly, at first a generic implementation of the visualization should be completed. After that it is necessary to analyze complexity of classes themselves, their combinations, and complexity of commands of an underlying graphic language, used for the methods of the classes.

Signature (classes `Entity`, `Relation`, `Arrow`, commands `\POS`, `\txt`, `\ar`, `\underline`, `\frm`) for this particular representative visualization was defined.

Examined case study corresponds to the cell for a group 5, period 5 of prototypical PTVM of [1], which is also ER diagrams. According to classification scheme of [2] case study falls into Graph Generic ADT.

Next step in research is to produce a genuine generic implementation of this type of visualizations. Following techniques are planned to be used:

1. OOP with polymorphism (to realize polymorphic method `Draw`);
2. OOP with parametrized templates (using BGL) to realize a graph data model for the visualization and apply graph traversal generic algorithm to a visualization graph invoking polymorphic method `Draw`.

References

[1] R. Lengler and M. J. Eppler, "Towards a periodic table of visualization methods of management," in *Proc. Int. IASTED Conf. on Graphics and Visualization in Engineering*, Clear Water, Florida, USA, 2007.

[2] B. P. Leontyev, "Enhancement of a periodic table and generic visualization methods," in *Proc. 10th IASTED International Conference Computers and Advanced Technology in Education (CATE-2007)*, Beijing, China, Oct. 2007, pp. 96–101.

[3] J. G. Siek, L.-Q. Lee, and A. Lumsdale, *The Boost Graph Library: user guide and reference manual*, Addison-Wesley, 2002.

[4] C. J. Date, *An Introduction to Database Systems*, 7th ed. Pearson Education Asia Pte Ltd, 2000.

[5] K. H. Rose, "How to typeset pretty diagram arrows with TEX-design decisions used in XY-pic," in *Proc. 7th European TEX Conference*, Prague, Czechoslovakia, 1992, pp. 183–190.