Software for Structured Text Entities Dependency Graph Building

ION IVAN, MARIUS POPA, CATALIN BOJA, CRISTIAN TOMA, DRAGOS ANASTASIU
Economic Informatics Department
Academy of Economic Studies
Romana Square No. 6, Bucharest
ROMANIA

Abstract: The current work, presents the concepts for application for structured entities evaluation creation. It describes internal data structure along with the object-oriented techniques used for the implementation. Methods and properties description are also presented. The presentation is tackling software-building process from abstract concepts to concrete implementation.

Key-Words: text entities, graph, data structures, software, statistic analysis

1. The graph structure and its dependency characteristic

The graph is a highly flexible structure with very big data representation potential and it is used as the base technique for data structuring in the presented software. The base component of a graph is the vertex, which is referenced within the application as *GraphNode*. Using template classes so that a high level of extensibility is achieved when using recursion does the implementation of the structure [1], [2].

When *T* is considered to be a predefined or a user defined type, a collection of elements with the mentioned type is called *UnManagedNodeList*. In figure 1, a schematic representation of such structure is presented.

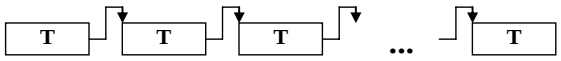

Fig. 1 UnManagedNodeList

A graph node is a container in which reside two types of elements. The first one is the element called INFO, which is a variable of type T representing the informational part of the node. The referencing area represents the second type, which is a dynamically allocated area where pointers for the connected nodes are held. The representation of *GraphNode* with corresponding neighbors is presented in figure 2. The Referencing area is itself a dynamically allocated list, because a node can have none, one, or more than one neighbor.

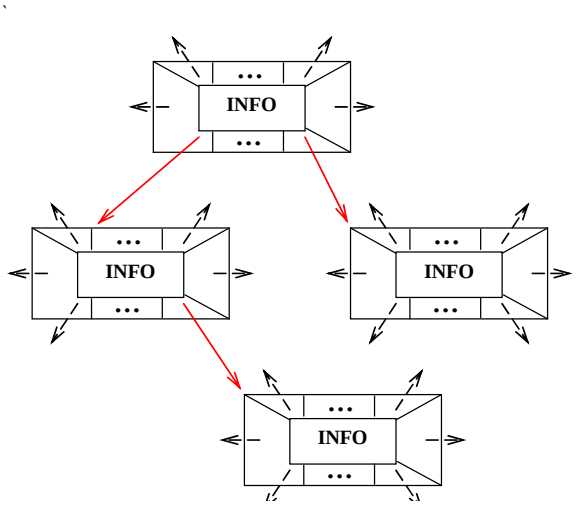

Fig. 2. Schematic representation of a GraphNode with its connections

When building an *UnManagedNodeList* of type *GraphNode*, all the elements in figure 2 are included in every box represented in figure 1. In this way, when accessing the first element of such construction, a *GraphNode* element like one shown in figure 2 is accessed, pointing to the main node, which has connections to all other neighbor nodes. The presented construction is called the *GraphNodeList* and it already represents a graph with *n* entry points, where *n* represents the number of elements that *UnManagedNodeList* has. The implementation of the structure is presented in figure 3.

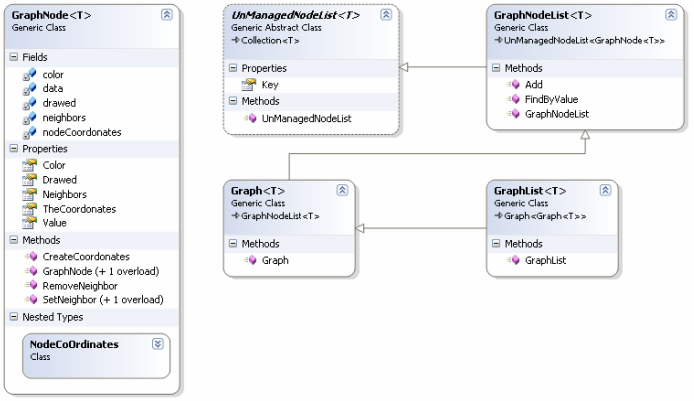

Fig. 3. The graph structure implementation

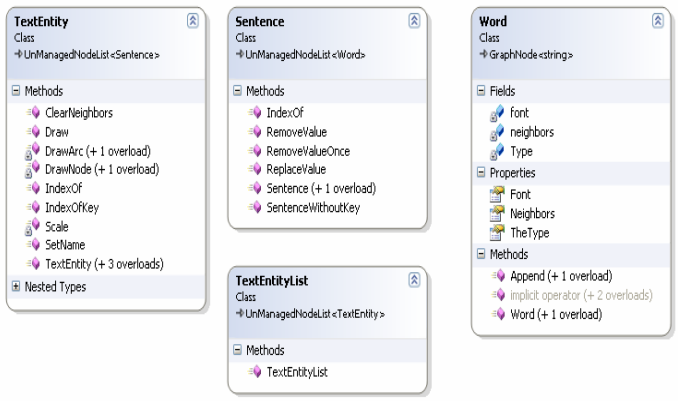

Fig. 4. The TextEntity structure implementation

The structure in figure 3 extends the concepts to a list of generic graph objects. By using the auto referencing technique, the graph structures becomes even more complex and higher flexibility levels are achieved when between the groups of *GraphNode* objects belonging to different entry levels are interconnected, but this comes with an increase in the handling efforts, which are required for keeping the structure solid.

2. Structured text entity

The purpose of the presented software is to analyze and evaluate structured text entities. These are constructions made of text that form from different input sources. In the current work, the input is considered to be a series of files, depending on the type of analysis that is made on the corresponding structure.

The alphabet considered for evaluation is ASCII. The vocabulary is formed out of words, each representing a specific element in the analysis process. The evaluation engine of the current software product considers the word as a list of ASCII characters that is dynamically allocated into the memory so that the exact amount as needed is used [5]. The text entity is structured according to the graph construction presented in chapter 1, so that a word is a *GraphNode* of type string. In the seen reality, a list of words forms a sentence and the list of sentence represents the whole text. In the same manner, the sentence is represented like an *UnManagedNodeList* of words while the text entity is an *UnManagedNodeList* of sentences. The *TextEntity* structure implementation, based on the *Graph* structure, is presented in figure 4.

Even though the graph structure is not presented at its highest capacity, it is even higher then needed to structure a text entity in a construction formed out of words and sentences.

The process of structuring takes place from an abstract level to a more specialized level according to the drawing in figure 5.

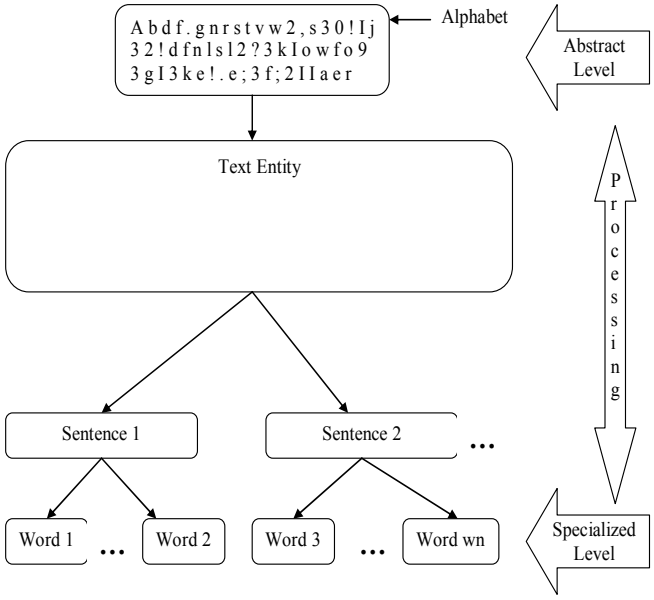

Fig. 5. The process of structuring from abstract to specialized

On the abstract level, there is the alphabet formed out of ASCII characters. The alphabet is then included in a file representing the whole text entity. The structuring begins at this level, from which sentences are formed according to user defined separators and then, in the same manner, the words, which are according to the construction described, the most specialized element that are taken into consideration as a structure.

3. The structure of the software product and the operation of graph drawing

The software product is conceptually built on layers. These are presented in figure 6.

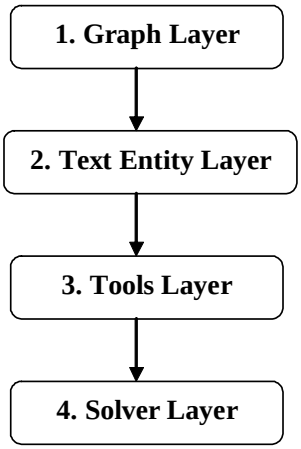

Fig. 6. The layers of the software product

The *Graph* layer represents the structuring mechanism and is the base for developing other layers. It contains specific methods and algorithms for creating and manipulating the complex and highly extensible data structure presented in chapter 1.

On the *Text Entity* layer, the data structure takes a more particular form, so that different text data received as an input is organized according to the structure with the desired level of accuracy. It is at this layer where the classes presented in chapter 2 with specific text manipulation methods and algorithms are implemented.

For other operations that also have a sufficiently big level of generality, the *Tools* layer is created. It represents a container for algorithms like file, data, windows and other elements manipulation, which are generally applicable.

The *Solver* layer handles more problem specific operations. Thus, for every type of problem, as different output is needed, different algorithms must be implemented and at this layer is where those algorithms are created.

By building this software architecture, all the operations are easily maintained, while hiding the complexity by dividing the problem into many modules.

This operation is implemented in the *Text Entity* layer. In this manner, every *Solver* layer offers through the *Text Entity* layer the drawing property for reports generation improvement.

Graph drawing addresses the problem of constructing geometric representations of graphs. Although the perception of how good a graph is in conveying information is fairly subjective, the goal of limiting the number of arc crossings is a well-admitted criterion for a good drawing [8].

Incremental graph drawing constructions are motivated by the need to support the interactive updates performed by the user. It is helpful to preserve a "mental picture" of the layout of a graph over successive drawings. It can be distracting to make a slight modification, perform the graph drawing algorithm, and have the resulting drawing appear very different from the previous one. Therefore, generating incrementally stable layouts is important for many applications. Layout strategies that strive to preserve perspective from earlier drawings are called incremental [1], [2], [8].

The graph drawing property is implemented through the function *Draw* that has the following prototype:

```
public Graphics Draw (ref Graphics g,
                      int    height,
                      int     width,
                      bool
                      horizontal)
```

Where:

- *g* represents the Graphics surface where the graph is drawn. The parameter is transferred by reference, so that the calling object passes the parameter and the *Draw* function modifies it;
- *height* is the variable received by value which shows the maximum height of the drawing;
- *width*, represents an integer value which shows the maximum accepted width for the drawing;
- *horizontal* is a Boolean parameter, which is needed because the drawing algorithm is incremental. When set to true, the parameter shows that the graph has the vertexes displacement horizontally, from left to right, otherwise vertically, from top to bottom.

For the co-ordinates manipulation when it comes to bi-dimensional drawings, the following structure and class are created:

- **Corner structure** for representing a corner, because each node is displayed within the drawing as a rectangle. The structure has a static method for computing the distances between two corners. With this structure, the nearest corners for two rectangles are found, in this way being able to draw the axes without overwriting the nodes. The returned type is a corner, where on *X* property, there is the difference between the *X* parameters of the two corners, and on *Y* property, the difference between *Y* parameters of the corners. The prototype of the function is:

```
public static Corner Difference
(Corner corner1, Corner
corner2)
```

- **NodeCoOrdinates class** for co-ordinates manipulation, with the components description presented in table 1. When instantiating a *NodeCoOrdinates* object, the parameters needed are the left-top corner and the length of the text, while other corner are automatically calculated, considering a default value for the text height.

Table 1 Description of NodeCoOrdinates Class components

Property Name	Property Type	Access Level	Description
CornerLB	corner	public	Gets the left-bottom corner
CornerLT	corner	public	Gets the left-top corner
CornerRB	corner	public	Gets the right-bottom corner
CornerRT	corner	public	Gets the right-top corner
List	Collection< corner>	public	Gets the collection of corners

The *NodeCoOrdinates* class serves as a container for the 4 corners placement into the bi-dimensional space. In this way, the Left-Bottom, Left-Top, Right-Bottom, Right-Top values are assigned and requested by accessing the properties of the presented class. Every corner has 2 parameters, because the drawing space is bi-dimensional, and because of this, the *corner* class is also nested into the *NodeCoordonates* class

4. The operation of statistic analysis and evaluation

The statistic analysis is an operation that takes place on the *Solver* layer. The environment in which the operation takes place is described in [4].

Thus, the **input data** consists of tables of data in which a number of students in the terminal year introduce data representing the order in which they would have wanted to learn the disciplines for better understanding. For this, a number of classes of objects are created:

- **Discipline class** for the disciplines which are analyzed, with the **components** description presented in table 2;

Table 2 Description of Discipline Class components

Property Name	Property Type	Access Level	Description
name	String	private	Holds the name of the discipline
Discipline Name	String	public	Gets or sets the name of the discipline

- **DisciplineSTUDy class** for other algorithms and needed operations implementation, with the class **components** description presented in table 3;

Table 3 Description of DisciplineSTUDy Class properties

Property Name	Property Type	Access Level	Description
Disciplines	BindingList< Discipline>	public	Gets the binding list of the discipline
NoDisciplines	int	public	Gets the name of the discipline
NoStudents	int	public	Gets the number of students
Table	DataTable	public	Gets or sets the input data table

Within the *DisciplineSTUDy* class, the following **methods** are implemented:

- `public TextEntity Compute (Scoring scMethod, ScoringInterpretation scIntMethod)` for computing the input data,

where:

- *scMethod* represents the delegate for establishing a score which is assigned for every discipline, based on the input data;
- *scIntMethod* is a delegate for interpretation of the established score.

The delegates are declared as follows:

- `public delegate int[] Scoring ( int[,] indexes )` for the Scoring type, where *indexes* are the values of the student options from which the score is calculated and

return by the Scoring typed delegate. For example, for the fist discipline in the database, the index is 1, for the second, the index is 2;

- `public delegate TextEntity ScoringInterpretation (int[] scores),` where *scores* are obtained through the *Scoring* function and returns the interpretation whithin a *TextEntity* object that is to be further displayed as a graph.

The *DisciplineSTUDy* class also provides a method for drawing called *Draw*, which actually calls the method with the same name of the resulted *TextEntity* object.

- `public void Save (),` for saving the disciplines at one moment, without the analysis, which is stored in an object that belongs to the *TextEntity* layer.

Output data for the operation is the order in which the students should learn the disciplines for better understanding. In order to show as many information as possible, the output is displayed in a form of a bi-dimensional graph, which shows all the dependencies between the disciplines. In [4] the software is tested with 5 input data sets. The analysis is statistical because the input data is treated statistically, by using the *Scoring* and *ScoringInterpretation* delegates.

5. The operation of referential analysis and evaluation

In the referential evaluation operation, dependencies between entities are established. In [4], the case of a words or terms dictionary is analyzed.

Input data in this case, is represented by one or many definitions, each one including the word or term to be defined and the definition associated to it.

The **output data** consists of the list with general parameters presented in table no. 4 and the graph drawing showing the dependency between the words, as seen in [4].

Table 4 The list with general parameters considered as output

Indicator Name
Total number of words
Total number of defined words
Total number of words used in definitions
Total number of words filtered

For implementing the operation, the following classes are developed:

- **ResultIndicator class** for creation and manipulation of the output parameters. In this manner, the indicators presented in table no. 4 are instances of this class. The properties of the class are presented in table 5.

Table 5 Description of ResultIndicator class properties

Property Name	Property Type	Access Level	Description
IndicatorName	string	public	Gets or sets the name of the indicator
IndicatorValue	int	public	Gets or sets the value of the indicator

- **TermDefinition class** for creation and manipulation of the definitions. Every definition that is received as input data is transformed into an object of this class. The properties of the class are presented in table 6.

Table 6 Description of TermDefinition class properties

Property Name	Property Type	Access Level	Description
Definition	string	public	Gets or sets the section represented by the words that define the term
Word	string	public	Gets or sets the term that is defined

TermDefinition as well as *ResultIndicator* classes are constructed for better input and output data manipulation while developing the computing engine and also for interface implementation improvement, by creating *BindingList* objects that are managed and maintained in the .NET 2.0's *DataGridView* object.

- **DictionaryAnalysis** is the core class for the analyzing process. The properties of the class are presented in table 7.

Table 7 Description of DictionaryAnalysis class properties

Property Name	Property Type	Access Level	Description
computed	bool	private	Variable that is used for determining whether the analyze has been done or not
dict	BindingList<TermDefinition>	public	Variable representing not filtered input data dictionary
dictFiltered	BindingList<TermDefinition>	public	Variable representing the filtered dictionary received as input data
drawnWord	string	private	Variable representing the word for which the analyze is done
result	TextEntity	private	Variable representing the TextEntity that is ready to be drawn
resultIndicators	BindingList<ResultIndicators>	public	variable representing the resulted indicators
TE	TextEntity	private	The TextEntity with input data

The methods used by the objects of the class are:

- `private void Add ( TextEntity.Word wrd, int idxS, bool error ),` for adding a word into the text entity that must be displayed as an evaluation result in shape of a graph. The adding is done according to the way that the graph is prefered to be displayed and because of this, the method is implemented at the Solver level, and not at the TextEntity level. The parameters are:
  - *wrd*, which represents the word that is added;
  - *idxS*, representing the sentence index for the position where the word is added;
  - *error*, which when set to true, indicates that an error node is added and the borders of the cell must be differently colored;
- `public void Compute (BindingList<TermDefinition> theDict, TextEntity.Filter flt),` for analyzing the dictionary received as a parameter named *theDict*, representing the input data, which is to be filtered according to the defined filter *flt*;
- `private BindingList<TermDefinition> Convert2Dict ();` for converting the private property *TE* into a dictionary which is returned by the function;
- `private TextEntity.TextEntity Covert2TE(BindingList<TermDefinition> theDict );` which is a function for converting a dictionary received as a parameter, named *theDict*, into a *TextEntity* object which is returned by the function;
- `public Graphics Draw ( ref Graphics g, string word, int height, int width, bool horizontal, bool compresed ),` for drawing the graf out of the text entity that results after analyzing the term received through the parameter *word*. When the parameter *compressed* is set to true, the resulted graph has only the root and the leaves at the first level. Other parameters are used for sending them to the object implementing the graph drawing operation, which is presented in chapter 4;
- `private TextEntity.TextEntity ResultCompressed ();` which is the function for generating the so called compressed text entity when the draw function is called with the parameter *compressed* set to true;
- `public void Save ()` used for saving the input data into XML files.

The referential analysis operation is tested in [4] on 4 input sets and with different view-points on the graph draw.

6. Conclusions

The presented software product aim is to offer a base implementation of a data structure with large extensibility potential so that any analytical operations made on structured entities to be done in a very robust manner. The presented approach, of considering a structuring process, which integrates at a very low level of the problem, offers the mechanism of solving different types of problems that are though able to be structured at a common level.

Because of the object-oriented programming facilities, like interfaces, the choice of implementing the structure and algorithms in C# language is a very good one.

Through the deriving hierarchy, which travels the distance from concept to example or from general to particular, the possibilities that reveal tend to cover many practical situations.

The presented approach outruns the strict necessities for solving the presented problems and this is done because the main purpose is not that to solve specific problems, but to create a base framework for structured entities research. The software is organized on layers, so that any practical implementation can be done on as many layers as desired. Thus, the graph layer and the text entity layer offer the base, which is able to hold many implementations. Further from this point, the improvement of the base layers is desired, along with many constructions, for which they stand as root, which should be developed to ensure their flexibility and robustness. Also, graph drawing mechanism must be further improved and extended to allow proper analytical operations to be done on different types of structuring models.

Reference:

[1] Michael Goodrich, Roberto Tamassia, David Mount, *Data Structures and Algorithms in C++*, John Wiley & Sons Publishing House, 2003.

[2] Aaron Tenenbaum, Yedidyah Langsam, Moshe Augenstein, *Data Structures using C*, Prentice-Hall International Publishing House, 1990.

[3] Ion Ivan, Cătălin Boja, *Metode statistice în analiza software*, AES Publishing House, 2004.

[4] Dragoş Anastasiu, The Study of Curricular Dependency, *Journal of Applied Quantitative Methods*, Volume 1, No. 1, Sept. 2006 – www.jaqm.ro

[5] Ion IVAN, Catalin BOJA, Cristian CIUREA, Robert ENYEDI, Cristian TOMA, Marius POPA, Collaborative Systems Metrics, *International Workshop in Collaborative Systems*, Mediamira Science, pg. 120 – 146, October 2006.

[6] David BRÜLL, Björn SCHWARZER, Sebastian OSCHATZ, Arnd STEINMETZ, A dataflow graph based approach to web application development, The Proceedings of the 10th WSEAS International Conference on SYSTEMS Vouliagmeni, Athens, Greece, July 10-12, 2006.

[7] Deniss KUMLANDER, Improving the maximum-weight clique algorithm for the dense graphs, The Proceedings of the 10th WSEAS International Conference on SYSTEMS Vouliagmeni, Athens, Greece, July 10-12, 2006.

[8] On-line Paper http://www.uv.es/~rmarti/paper/gd.html