Computer Science Journal of Moldova, vol.**2**, no.3(6), 1994

Object class hierarchy for an incremental hypertext editor

A.Colesnicov M.Colesnicov L.Malahova

Abstract

The object class hierarchy design is considered due to a hypertext editor implementation. The following basic classes were selected: the editor's coordinate system, the memory manager, the text buffer executing basic editing operations, the inherited hypertext buffer, the edit window, the multi-window shell. Special hypertext editing features, the incremental hypertext creation support and further generalizations are discussed.

1 Introduction

We are considering here the engineering problem of the object class hierarchy design due to the implementation of text, hypertext and structured text editors into object-oriented environments with incremental development support.

The discussed problem had arisen when we had implemented a WYSIWYG system of incremental hypertext creation and editing in the MS-DOS operating system textual screen mode on IBM PC compatible computer. The resulting hypertexts were supposed to be used as help subsystems in applications. On the one hand, the screen mode restriction had excluded the complications connected with the graphical interface. On the other hand, the restrictions of the underlying operating system had forced us to the very careful selection of object classes. E.g., the necessity of the separate memory management subsystem became especially clear.

The system was implemented in Turbo Pascal 6.0 using Turbo Vision.

The proposed object hierarchy was successfully used afterwards in the implementation of the token-oriented keyboard script editor. Right now other projects are developed using the same technology.

Detailed discussions on hypertexts may be found in [3, 4, 8]. The structured text is the text with a superimposed structure. In the case of a hypertext it is a graph, see [2].

Token-oriented editors deals not with separate characters but with tokens, see [1]. We are to note here that in the former case the internal representation of tokens and their screen (visual) representation may differ. As a generalization, some tokens may be invisible on the screen.

The object-oriented design is perfectly discussed in [5].

The preliminary variant of this article was published as [10]. See also [11] for related topic discussion.

Another approach to the structured text editors was developed as a part of the Mjølner project at Lund University, Sweden. See [6, 7]. Their approach differs mainly in two aspects: they deal with a highly structured text (programs and grammars), and they had developed their system top-down. Their system highly supports the incremental development.

2 Features of hypertexts and hypertext creation systems

There are a lot of MS-DOS hypertext system: Norton Guide, MultiEdit Help (part of the text editor), Turbo/Object Professional Help, Tech Help etc. We had tried 6 or 7 systems and had rejected all, see [9]. The main consideration was that we found all of them unsuitable on variable reasons.

The external representation of most existing systems does not conform to both CUI standard and the design style of the product which our system was to be integrated into. E.g., some of them do not use mouse, or restrict its movements on the screen. Some are not state-of-

the-art hypertexts, e.g., they not allow hyperlinks embedded into text but only in menu, some keep hyperlink pointers visible in the text, most do not permit to execute MS-DOS command through hyperlink etc.

The main lack of all studied system was the absence of the WYSIWYG hypertext editor. In most cases the single supplied tool was the help compiler. The hypertext creation process supposed the preparation of the text by the usual text editor, when hyperlink and emphasize markers are to be issued as special command sequences, and then the labelled text is passed through the hypertext compiler. Those special command sequences are invisible in the resulting hypertext but they are visible when preparing the hypertext source with the text editor, and you would see the text not in its final allocation. It is especially difficult to prepare in such a way tables with hyperlinks and emphasize inside.

Hypertext and other structured text creation by a conventional text editor may also imply the temporary defectiveness of the resulting text. The solution supposes the incremental creation, see sec. 8 below.

When this problem was formulated for the practical development, it had presented natural to produce the system from a text editor. So the next section is dedicated to the text editor objects classes.

3 Text editor basic objects classes

The difficulty of the considered task is usually underestimated. We had got some tutorial examples, e.g. that one supplied with Turbo Pascal 6.0, but we could not find anything useful for our task.

Then we were to start from the beginning and to separate abstract object classes of the text editor. It was found that the functions of events handling and of text editing must be separated in different objects. In the examples mentioned above all operations on the text, screen presentation and event handling were mixed together. It had lead to the program which can not be developed to real software product.

We propose a text buffer as the basic object of the text editor. The

buffer has the internal, invisible for the descendent objects memory management subsystem and exports methods performing the elementary operations on characters, lines and blocks. Those operations are inserting, deleting, moving, repositioning the current cursor location etc.

The memory manager was constructed in such a way that it can be replaced by another without changing the upper level interface of the text buffer. In fact, we had changed two variants of the memory manager.

One of the memory management functions is the pre-estimation of memory volume needed for the operation. E.g., to insert a line we need memory for its contents and, optionally, the memory to increment internal line tables; undo buffers grow when we delete information, etc. This mostly depends of the selected memory organization scheme. The detailed memory management description is not the aim of the paper.

When the memory manager pre-estimation shows that there is not enough memory, the operation will not be performed. This approach was selected after testing of another one, when the operation starts and may be finished partly, which was found unsuitable.

The lowest level includes such aspects as the coordinate system for the editor and block representation in those coordinates. We use 32-bits (long integer) numbers for line and position in the line, starting from 0. The line is restricted up to maximum long integer to the right, and the edited text may potentially contain as many lines. The block is represented by the coordinates of the starting character and of the character after block's last one in the same line. We tried also the Turbo Vision approach not using coordinates of characters but of points between characters, which also was found unsuitable.

To deal with the coordinates we use two low-level objects – interval of long integers represented by the first number and the number next to the last, and the block, with appropriate methods. As an example of frequently used coordinate system methods, I would mention here the "relative position" function `TInterval.RelPos(I: Integer)`. The function returns -2, if the argument is lower than the left interval bound, -1, if the argument is equal to the left interval bound, 0, if the

argument is strictly inside the interval, +1, if the argument is equal to the right interval bound, and +2, if the argument is greater than the right interval bound. For a point P, we have `TBlock.RelPos(P: TCoord)` function which yields a similar result. The necessity to differ on-bound and strictly internal position is not obvious.

The undo subsystem is a major separate part of the editor. To implement it we need memory for undo buffers and we are to realize each operation in such a manner that it can be reversed.

Having the coordinate system and memory manager, we can produce the text buffer. Then we are to include the buffer in the visual (screen) object – the edit window. This object handles external events from keyboard and mouse and interprets them using text buffer methods.

The next level is the shell performing the control on many editing windows and text buffers, file loading and saving etc.

We have then the following five levels of the text editor object classes (from the lowest): coordinate system; memory manager; text buffer; edit window; multi-window editor shell.

The accurate distribution of functions between these levels may be a difficult problem. E.g., we insert in each window the change and position indicators which are separate objects. Being visual they belong to edit window level, but they reflects the status of underlying text buffer. Another example: the shell uses change flags to decide does it need to save the contents of the text buffer in the file when closing a window etc. The abstraction layer concept means that we are to use on a level only the methods of immediate preceding one, but in this and some other cases we are to transmit information through three or more levels.

Other examples. Loading and saving edited text files – which level (buffer, window, or shell) do they belong to? When generalizing the text character to tokens, where are we to obtain token's visual representation from, buffer or window? (In the keyboard script editor mentioned above, `GetTokenImage` was made the text buffer's method, but it may be implemented on window level as well).

The difference between the hypertext and the usual text is that

some parts of the edited source have their visual representation and some (formatting, emphasizing and hyperlinks) have not.

This situation can be generalized to one mentioned above when we have a token-oriented edit buffer whose tokens are translated into different visual representation but are processed like characters, i.e., they are organized into lines, may be inserted, deleted and moved token by token, or in lines, or in blocks. The separation of the edit window and the text buffer allows us to perform the case as well.

4 Hypertext editor basic objects classes

Our main principle for hypertext was that the hypertext is the text with some marked blocks. The defining property of the marked block in a hypertext is that every character of this marked block has the properties of the whole block. E.g., we can click with the mouse at any point in the hyperlink, and we will follow this hyperlink.

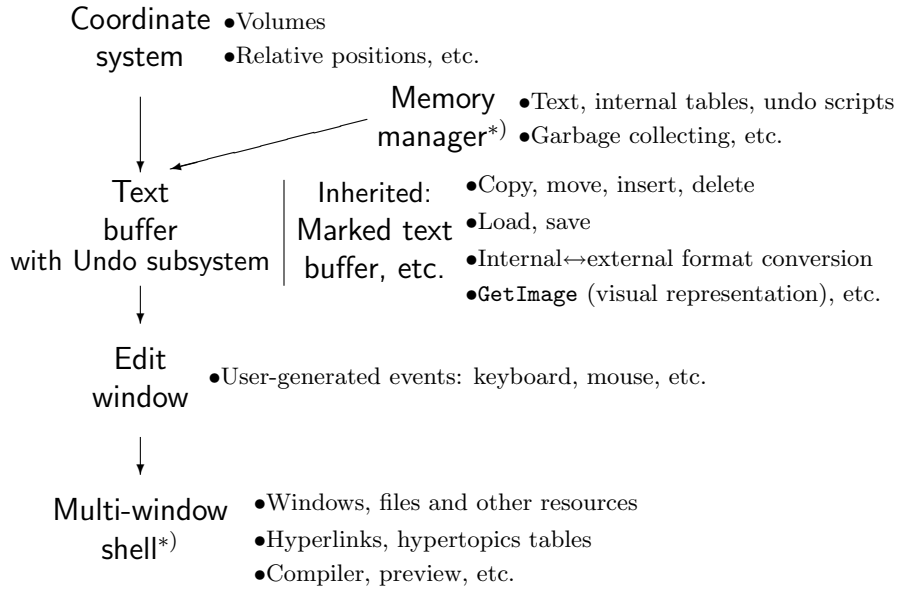
Then we had generalized the text buffer and descend from it the marked text buffer which is the basic object of the hypertext editor. The memory manager was also forwarded to control both text and markers. The marked text buffer performs the same operations as the text buffer taking in account the above mentioned property of the marking. E.g., when we copy the part of the hyperlink we produce in the corresponding place the new hyperlink to the same point etc.

The resulting hierarchy is illustrated by fig. 1.

5 Auxiliary hypertext processing programs

We need also the hypertext compiler, the linker and, may be, the decompiler. Those programs may be included into the hypertext creation environment or used as batch utilities. The desired property of the environment is the hypertext preview, which leads us to problems concerning the hypertext engine (see sec. 9).

The above mentioned problems of function distribution include in our case the need to register hyperlinks, especially when editing many-



*)May be used those of the application

Figure 1:(Hyper)text editor object class hierarchy

part source, to transform external (file) hypertext formatting commands representation to and from internal representation, to search a window and a line in it where the desired hypertopic starts, to avoid hypertopics of being duplicate etc. In fact, we are to realize parts of the hypertext compiler, and it is only the natural step to realize the whole compiler as a part of the hypertext editor. In our particular case, however, the hypertext compiler and the hypertext engine remained the separate batch utilities. See sec. 6 below.

6 The hypertext engine

The hypertext engine is the program navigating through the hypertext. Its desired features were widely discussed in the literature. In our case we needed, e.g., that the hypertext engine should be able to execute any command of the operating system, which was implemented as a special hyperlink.

The hypertext engine design concepts include the question of the single-window versus multi-window hypertext engine. In the single-window case (most existing help systems), each next hypertopic is shown in the same window as the previous one. In contrast, we can create a new window for each new hypertopic. This allows user to compare different topics, to backtrack through topics in any sequence etc. The third approach is to create new window only when the user wants this. Multi-window implementation may be restricted by the memory.

Software engineer implementing the hypertext engine encounters problems similar with ones when implementing the hypertext editor. In fact, perfectly constructed hypertext editor must be able to perform all functions the engine does, that is, in essence, it must contain the hypertext engine as one of its functional parts. Moreover, the hypertext engine needs at its lower level of functional structure the same class levels as edit window class in the editor structure. To implement the hypertext engine we need coordinate system and memory manager also, although only subset of their functions is used. Hypertext format conversion from the external file format to the internal one is necessary too.

Formats of both internal and external representation may differ when we implement the editor and the hypertext engine, particularly because between preparation and presentation the hypertext is compiled and its format is changed.

Hypertext engine class hierarchy is as follows: the presentation buffer based upon the coordinate system and the memory manager; the marked presentation buffer inherited from it; the show window and the multi-window shell. See fig. 2.

We see that at lower levels of the hierarchy are situated similar or

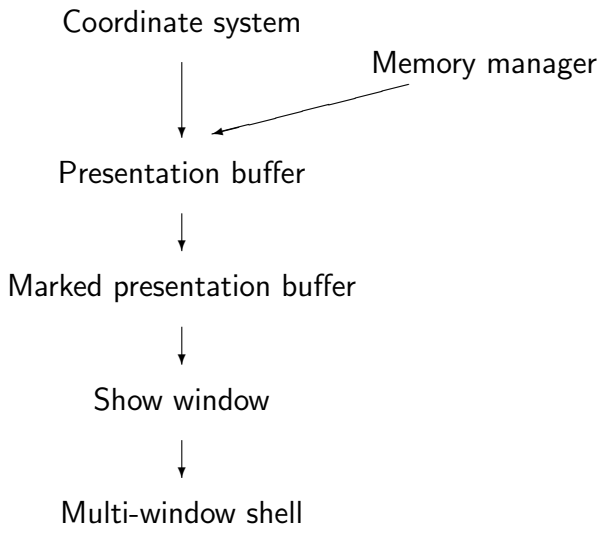


Figure 2:Hypertext engine object class hierarchy

the same coordinate system and memory manager classes. Presentation buffer and marked presentation buffer classes are similar but not the same in their functionality correspondingly with text buffer and marked text buffer classes of the editor.

Having unlimited computer resources one can quite well create an universal tool which would effectively perform both hypertext editor and autonomous hypertext engine functions. If the resources are limited, we are to specialize the correspondent tools and their functional subparts to respond these limitations. This causes that in similar tasks even the levels performing identical functions are to be implemented in different ways.

E.g., the memory manager for the hypertext engine performs only subset of functions needed for the editor. General memory requirements are lower than ones for the editor. In the same time demands for response time and available memory limitations are more severe. The

hypertext editor can be intended for use at more powerful computer models while the hypertext engine must work also at minor models. While hypertext editor is addressed to professional users who can reconcile themselves to small delays if they understand what causes them, the hypertext engine is addressed to end users.

7 Reasons for functional structure similarity

The same or similar class structure encounters in more wide conditions. Let us try to characterize these conditions.

Particularly structure similarity is caused by similar implementation base. In a narrow sense it means similar software creation tools such as programming language or software development tools; more widely it means the same target platform. In our case these are IBM PC compatible personal computers and MS-DOS operating system. E.g., the necessity to implement specialized memory managers for each task is caused both by relatively small size of operative memory at personal computers and by absence of ready-made effective universal memory management tool in MS-DOS operating system.

But the most essential reason for revealed functional structure similarity lies in the similarity of implemented tasks as it is shown above.

8 The incremental development support

We understand incremental computing as mode of task setting and implementation when the whole computational algorithm is performed step by step presenting current results in some (preferably visual) form to the user and waiting for user's intervention into the computation process. Incremental computing implies short response time for each step, flexibility of the process under user's control, keeping resulting structure unmistakable at any time and obtaining reasonable and proper responses to user actions as fast as possible, as a rule at the current step. The general result of the computation thus is accumulated while each step increments it.

Restricting the notion of incremental computing to various classes of tasks we obtain incremental text processing, incremental compilation etc.

In our case, all operations on hypertext creation are automatically executed supporting the right resulting source text structure. E.g., the user can select a large block of text and declare it to be a new hypertopic. Then the system inserts the appropriate marks both at the beginning and at the end of the block, and if the user exits the system right at the moment, the resulting text will be saved in proper external format which would be processed by the hypertext compiler without errors. Step by step marking of hyperlinks, special hyperlinks, emphasizing etc. is treated in a similar manner.

9 Hypertext integration into the application

In the case we want to use a hypertext as the part of an application (e.g., as the help subsystem), it is to be integrated into application. Methods of such integration depend mostly on underlying platform. There exists also an organizational consideration, namely, does hypertext designer supply to application designer a portion of the source code.

Having hypertext engine source code, application designer may directly include it into his system. As a result of the solution, you must provide as many variants of this source code as many programming systems use your customers.

Hypertext engine may also exist as an independent executable module. Then a problem of its call arises. Desired starting point of hypertext show may be passed as the call parameter.

Some advanced systems allow simply to send a corresponding event which is broadcasted over the system and activates hypertext engine.

In MS-DOS-like systems we are to use specific methods because of the strict memory limits. E.g., the hypertext engine or its small part may be resident in the memory, or the engine executable module may be run as the child process of the application, using the memory swapping when necessary.

There exists also a problem of the hypertext status saving between calls if it is deleted from memory etc.

10 Conclusions

Two main defining points of our approach are: the down-top development, and the maintenance of "poorly structured" texts, when the prevailing part of the text remains unstructured. Except hypertexts, another good example of the sort are TEX sources. This case is opposite to such highly structured texts as program sources.

Our approach to hypertext editor implementation consists of the selection of the following abstraction levels of object classes: coordinate system; memory manager; text buffer and inherited marked text buffer; edit window; multi-window editor shell. The resulting system supports the WYSIWYG incremental hypertext creation.

This approach was successfully used in implementing of a hypertext creation system, a token-oriented editor and is now the subject of further development.

References

[1] R. Bilos. A Token-Based Syntax Sensitive Editor, in: Proceedings of the Programming Environments – Programming Paradigms Workshop, Roskilde University Center (1986).

[2] J. Conklin. Hypertext: An introduction and survey, Computer, **20**(9) (1987) 17–40.

[3] Hypertext '87: Proceedings of the conference, Chapel Hill, NC (1987).

[4] Hypertext '89: Proceedings of the conference, Pittsburgh, PA (1989).

[5] Communications of the ACM, Special Issue on Object-Oriented Design, **33**(9) (1990).

[6] Boris Magnusson et al. An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development, in: Proceedings of TOOLS '90, Paris, France (1990).

[7] Sten Minör. On Structure-Oriented Editing, PhD thesis, Department of Computer Science, Lund University, Lund, Sweden (1990).

[8] Emily Berk and Joseph Devlin, eds. Hypertext/hypermedia handbook, McGraw-Hill, New York (1991).

[9] M. Colesnicov. Peculiarities of hypertext technology application to homœopathic expert systems, in: Proceedings of the Scientific-Technical Conference on Computer Science and Computing, Chişinău, Moldova, 1993 (Russian).

[10] A. Colesnicov and L. Malahova. An Object-Oriented Environment for Hypertext Creation, in: Proceedings of the International Conference on Technical Informatics, vol. **5**, Timişoara, România (1994).

[11] C. Groza. The Interaction between the Components of a Hybrid Editor, in: Proceedings of the International Conference on Technical Informatics, vol. **5**, Timişoara, România (1994).

A.Colesnicov, M.Colesnicov, L.Malahova,
Institute of Mathematics,
Academy of Sciences, Moldova
5, Academiei str., Kishinev,
277028, Moldova

e-mail: *kae@math.moldova.su*,
mek@evs.moldova.su,
21*mal@math.moldova.su*

Received 14 April, 1994