ChyPro* : A Hypermedia Programming Environment for Smalltalk-80

Maurice AMSELLEM

Laboratoire d'Intelligence Artificielle, Université Paris 8,
2, rue de la Liberté, 93526 Saint-Denis.
ams@harald.ai.univ-paris8.fr.

Abstract. Smalltalk-80 has introduced the use of interactive programming tools such as browsers, inspectors and debuggers which brought a major improvement on conventional text files based programming environments. However, they still heavily depend on character string representations of code and data and, as such, inherit all the known limitations of text. In this paper, we describe new versions of the Smalltalk-80 programming tools that transcend these limits, based on hypermedia techniques such as outlines, links, annotations and graphics and on direct manipulation interfaces. Particularly, the new tools allow :

◊ browsing simultaneously scattered pieces of software through multiple points of view,

◊ interleaving graphical and textual representations of Smalltalk code, documentation and data,

◊ structuring, organizing and linking heterogeneous or distant pieces of code or documentation.

1 Introduction

Conventional programming languages such as Pascal or C favour a modal programming style : programs are created and modified in editors, then compiled and linked with compilers and then executed and debugged with run-time debuggers. The editor, the compiler, the debugger and the processed programs are independent and communicate via files. This isolation is not deliberate but is a result of the underlying operating system's lack of integration.

In contrast, everything in Smalltalk, from the programming tools to the libraries, the programs and the operating system, is uniformly represented by objects sending themselves messages. Moreover, the user interface of all these tools consists of a short number of powerful adaptable interface components. While the simplicity and the elegance of Smalltalk's implementation and interface contributes greatly to the integration and homogeneity of this environment, it can become a nuisance to its usability, if it is established as the only directing rule.

In the following, we will first focus on the limitations of the Smalltalk's programming tools (the browsers, the debuggers and the inspectors). Then we will present the

* ChyPro stands for **C**omputed **Hy**permedia **Pro**gramming.

ChyPro versions of Smalltalk's programming tools which intend to go beyond these limitations without sacrificing any of the valuable characteristics of the standard tools.

2 Limitations of the Smalltalk-80 Standard Tools

We assume the reader is familiar with Smalltalk's syntax, with the browsers, inspectors and debuggers[1]. We will therefore focus directly on their limitations.
Despite the fact that Smalltalk-80 is a highly graphical environment and provides a complete library of graphical classes, Smalltalk's tools still heavily depend on textual representations of the objects manipulated. Actually, most of Smalltalk's user interface is based on textual pluggable[2] interface components such as list views, text views, labelled switches and menus. For this reason, the Smalltalk tools are restrained by the inherent limits of character strings[3].

2.1 Limitations of the Browser

◊ The browser can handle only strings of characters. Although the text editor, used in

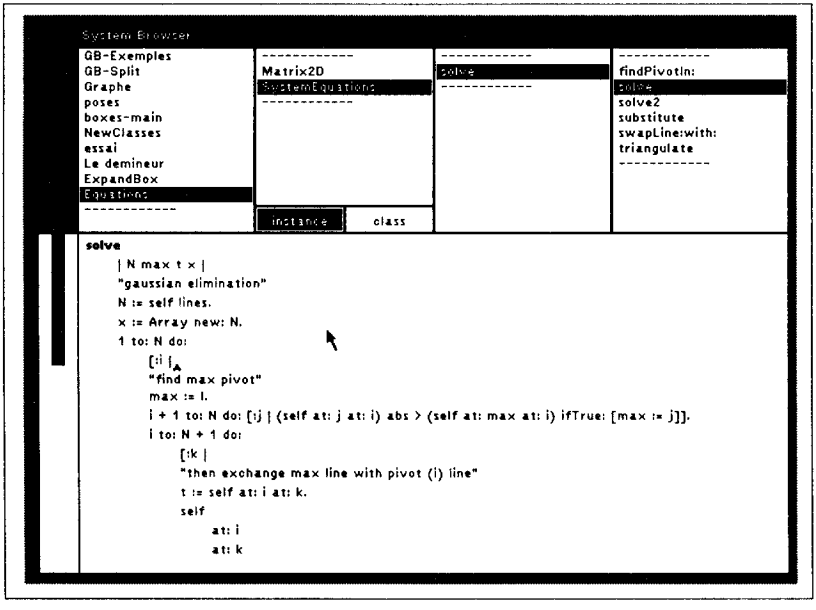


Fig. 1. A System Browser. The Gaussian elimination algorithm is inspired from [13].

[1] The interested reader will find a thorough description of the language and the environment in [7, 5].
[2] The philosophy of pluggable objects [8] is to avoid the creation of a subclass when an application specific behavior is needed by handling the parametrization of this behavior in the class itself. Examples of pluggable objects are TextView, SelectionInListView and BooleanView.
[3] This issue is so crucial that many recent systems such as Object Explorer [2], Track [3] or MPVC [6] have been built to extend the graphical capabilities of Smalltalk's tools.

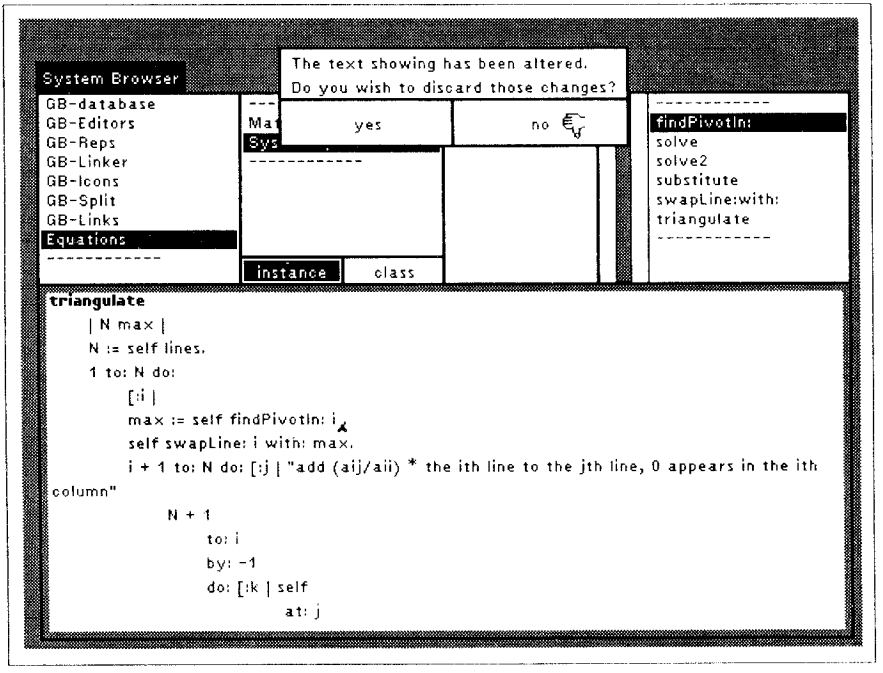


Fig. 2. Browser locking. The programmer instinctively attempted to select the method findPivotIn: to check its content while he was modifying the method triangulate, resulting in a dialogue to prompt him for a validation or cancellation of the modification he did. While this is certainly a good protection against accidental loss of data, it has nothing to do with the user's needs.

all the programming tools, supports multiple fonts and styles, text emphasizes are completely ignored by the compiler and discarded at storage. Thus every other information such as diagrams, software documentation, software organization or graphical annotations of code have to be translated to a string representation so as to be manipulated.

◊ Class and metaclass information are arbitrarily separated and cannot be accessed simultaneously in the browser.

◊ Software elements such as methods' code, class definitions and comments are displayed and manipulated in the browser only one element at a time. Moreover, any modification of the text window's content must be either validated or discarded before the user can make another selection in the browser (cf. Fig. 2).

This limitation is extremely frustrating when methods and classes are still under creation and must be edited all at once or when a large modification that spreads over several methods and classes - and this is often the case - must be performed. It constrains the programmer to open a separate browser window for each modification.

◊ Browsing is limited to 4 levels (categories, classes, protocols and methods[4]) and to

[4] More levels can be simulated with composite names like Collection-Abstract, Collection-Arrayed or private accessing.

physically related elements : the browser allows to jump from a method to another method in the same class and protocol, from a class to another in the same category or from a method to its senders, its implementors or the messages it contains. However, semantically but not physically related elements[5] must be reached manually.

2.2 Limitations of the Inspector

The inspector has the same limits as the browser view:
◊ Access to an object deeply nested in a root inspector view requires opening as many intermediate inspectors as there are levels of nesting. The resulting windows are not connected with each other and clutter the screen (see Fig. 3).
◊ As in the browser, when the value of a variable is altered and the user attempts to

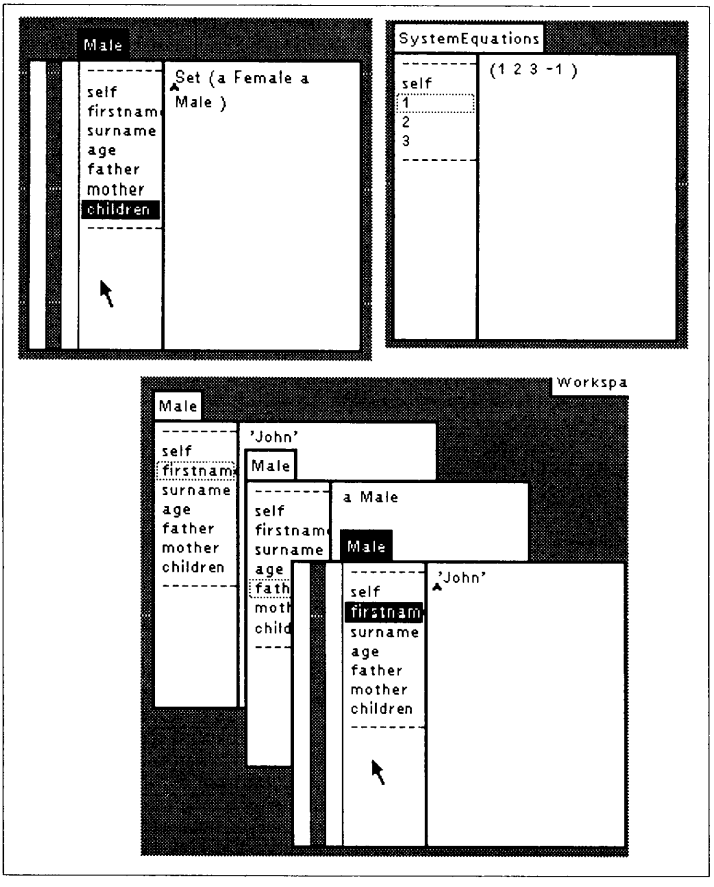


Fig. 3. Inspectors.

[5] For example, the message changed:#text corresponds to some implementation of update: in some class, depending on the dynamic context: these messages are semantically connected but not physically.

inspect another variable in the same window, he is prompted with a dialogue asking him if he wants to accept or discard the modifications.
◊ The values displayed are limited to character strings. Thus even graphical objects or complex data structures can only be viewed and edited via a textual representation. This is quite perplexing with regard to an environment that emphasises its graphical capabilities.

2.3 Limitations of the Debugger

◊ The debugger can be regarded as an inspector of the stack : we know that debugging is a more complex task than a mere inspection of the stack.
◊ In the debugger, it is impossible to inspect more than one context of the stack at the same time, for example to follow the control flow through the stack.

As a conclusion, the Smalltalk programming tools are not adapted to the programmer's needs because the designers privileged implementation issues. In effect, the implementation is a model of elegance. All the programming tool panes consist of only three classes : SelectionInListView handles menus lists, TextView handles text editing and SwitchView handles the Boolean switches of the browser . The panes are synchronised with a dependency mechanism (further details on the implementation can be found in [8]).

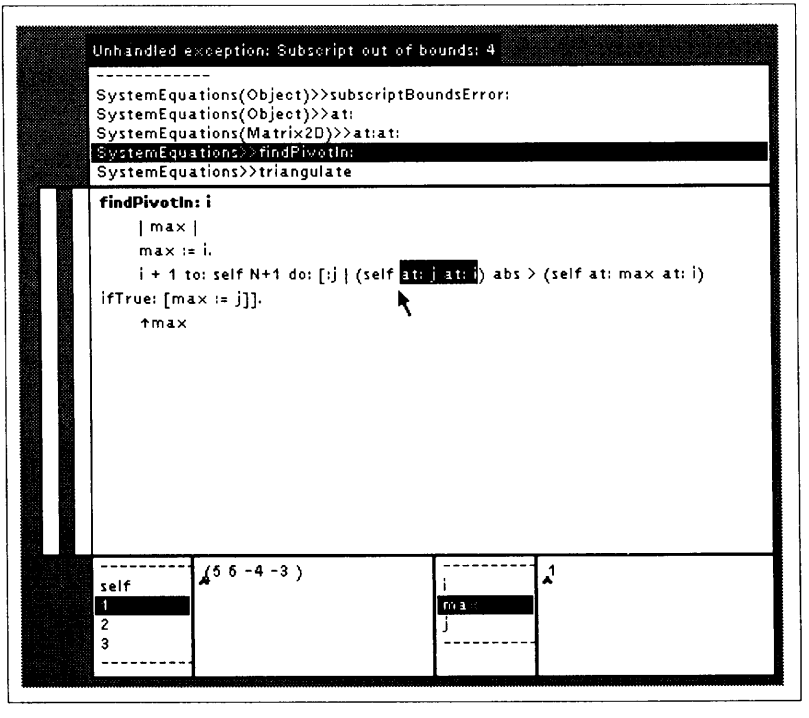


Fig. 4. The debugger.

3 ChyPro: a Hypermedia Programming Environment for Smalltalk

We will now describe the ChyPro versions of the Smalltalk-80 programming tools. We have built new versions of the browser, the inspector and the debugger.
We carefully designed the user interface, implementing direct manipulation interfaces[6] and using abundant visual clues such as animation effects, icons and cursor shapes.

3.1 A Browser Based on Hypermedia

The new browser consists of a three-paned window: a text outline pane, a graphical

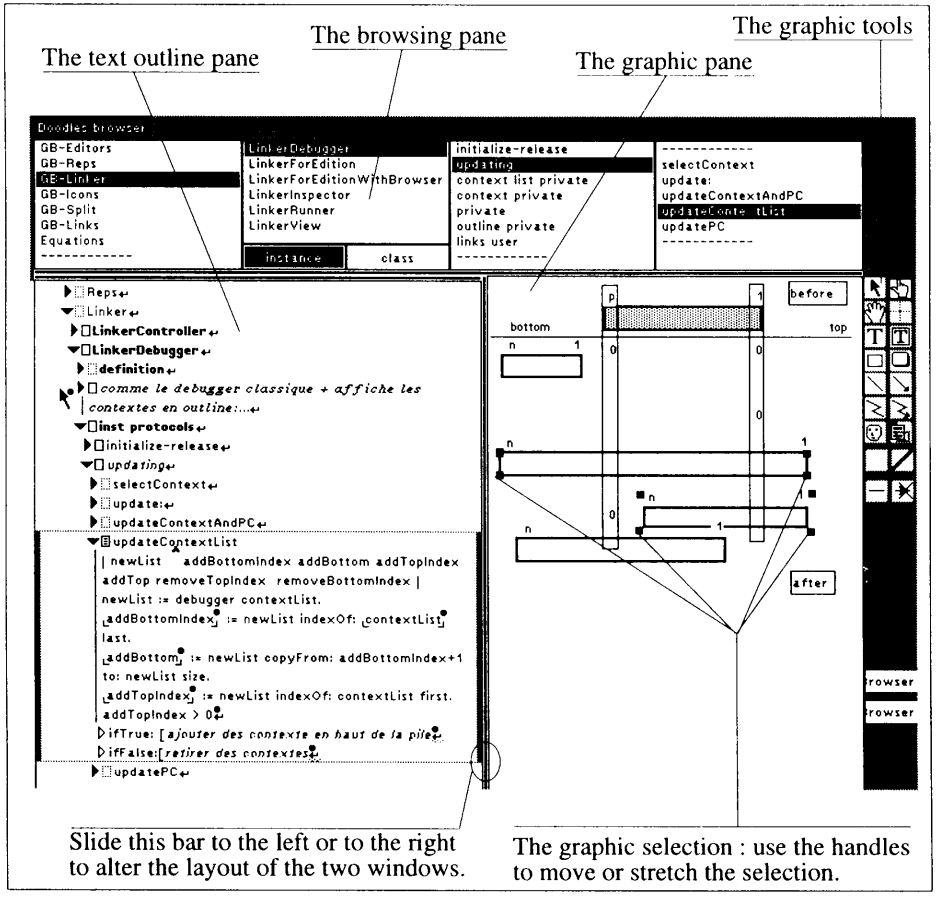


Fig. 5. Overview of the outline browser's interface.

[6] A system is said to have a *Direct Manipulation* user interface if the user feels he is acting directly on the entities involved in his mental plans when he interacts with the objects of the system [10, 12].

pane and a browsing pane (cf. Fig. 5). Depending on the context, each pane can be resized or even collapsed.

Graphics and Text. Programmers think both textually and graphically about their software, so we have included a textual editor and a graphical editor which are available at any time. Some word processors interleave graphics and text but we rejected this solution because we believe that textual and graphical information do not share the same spatial properties[7]. We preferred to juxtapose text editor and graphical editor. The graphical editor handles both pixel maps and geometrical objects such as rectangles, lines, polygons, arrows and blocks of text. The text editor is an extension of Smalltalk's standard text editor. In our system, graphical information can be linked explicitely to text through graphical annotations (see the paragraph on annotations below) or implicitely through computed graphical representations of the semantics associated to the text.

Hierarchical Links. Hierarchical links help in organizing and structuring software objects.
We use *outlines* to implement hierarchical relationships in text.
An outline is a textual representation of a tree structure. Each node of the tree is repre-

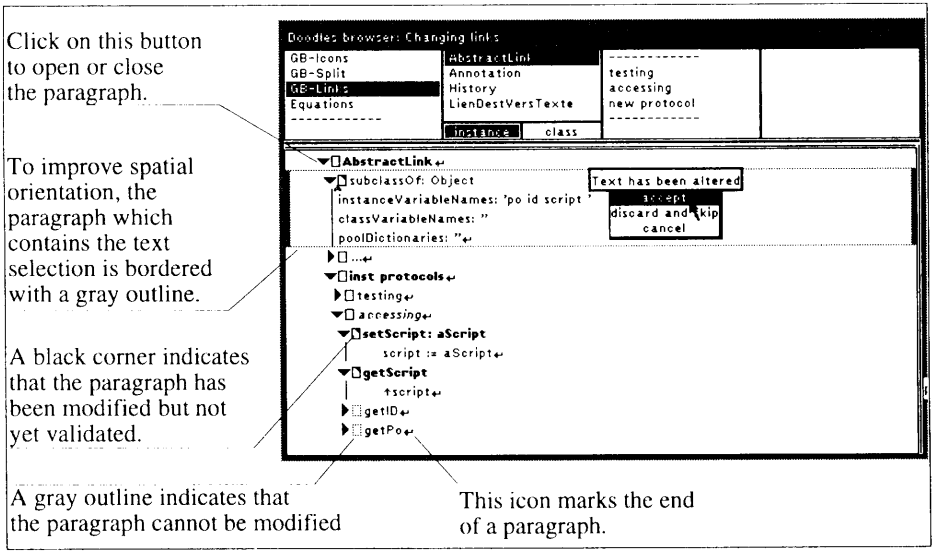


Fig. 6. Modification check. The user has clicked on the button of the first paragraph (AbstractLink) to close it. Since this paragraph contains other paragraphs which have not been validated, the user is prompted with a menu asking him whether he wants to validate the paragraph, discard the modification and skip to the next modification or to abort the checking operation.

[7] Precisely, text is one-dimensional (1.5 in the case of outline) and graphics are two-dimensional (2.5 if we consider graphical layers). Furthermore, graphics take more screen space than text and disturb the flow of text.

sented by a paragraph of text. The level of nesting is indicated by the indentation of the paragraph in a left-to-right ordering.
The user can add new paragraphs, change the indentation of existing ones and open or close paragraphs by simple mouse or keyboard operations (see Fig. 6).
◊ Outlines can be used on a local level to structure long and complex methods by splitting them in paragraphs. Adding a comment at the first line of each paragraph and then collapsing the paragraphs to show only these comments reveals the method's underlying structure. The same can be done to handle long documentation (see Fig. 7).
◊ On a global level, the programmer can use the outlining feature to build a tailored view of his software by opening in one pane the paragraphs of the classes and methods that are of interest for him and closing the others.
◊ Software organization is not limited to 4 levels (categories, classes, protocols and methods) as in the standard browser : the standard classification serves as a basis upon which intermediate arbitrarily nested classifications can be added using the outline feature.
◊ Different methods and classes can be arbitrarily opened and modified on the same window. The editor memorizes all the paragraphs that have been modified but not validated so that no modification will be lost . For example, when the programmer *accepts* a paragraph, the system searches for sub-paragraphs within this paragraph that have

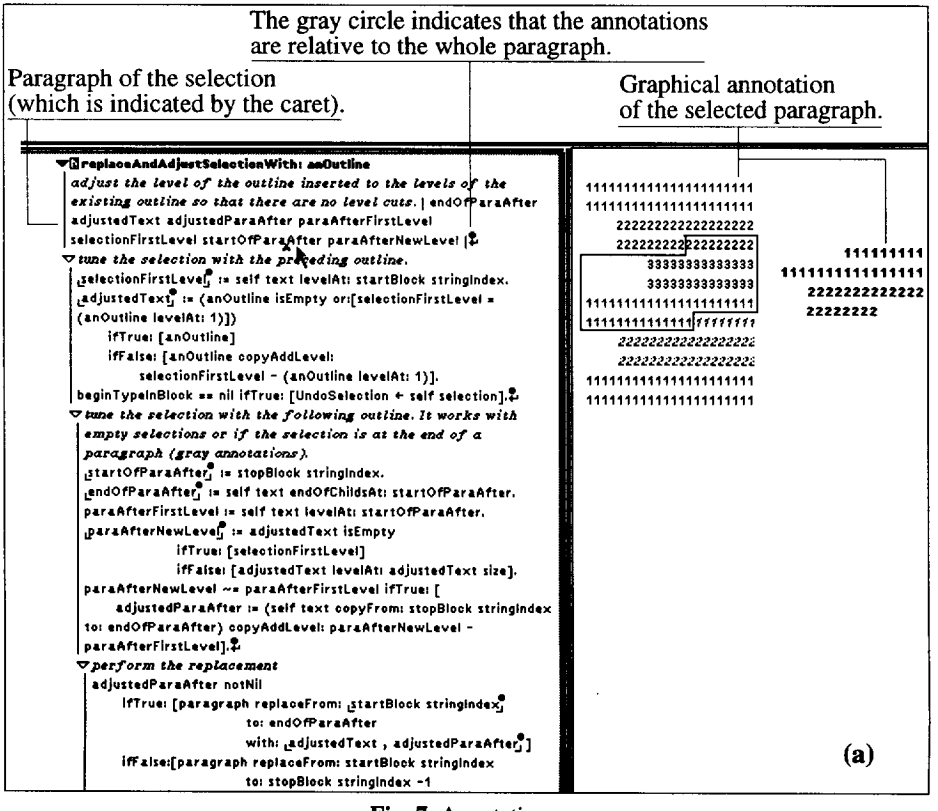
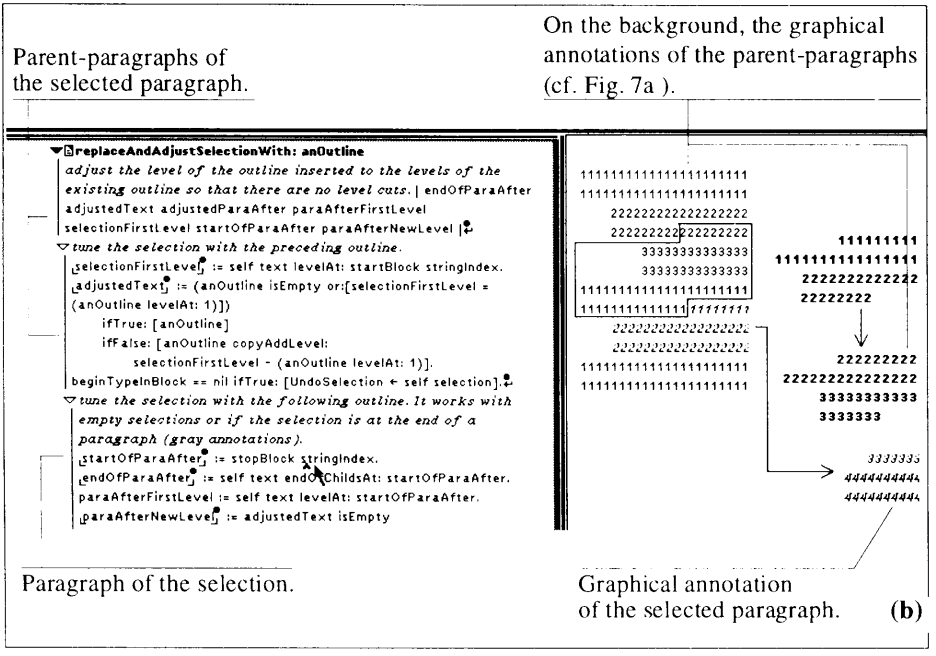


Fig. 7. Annotations.



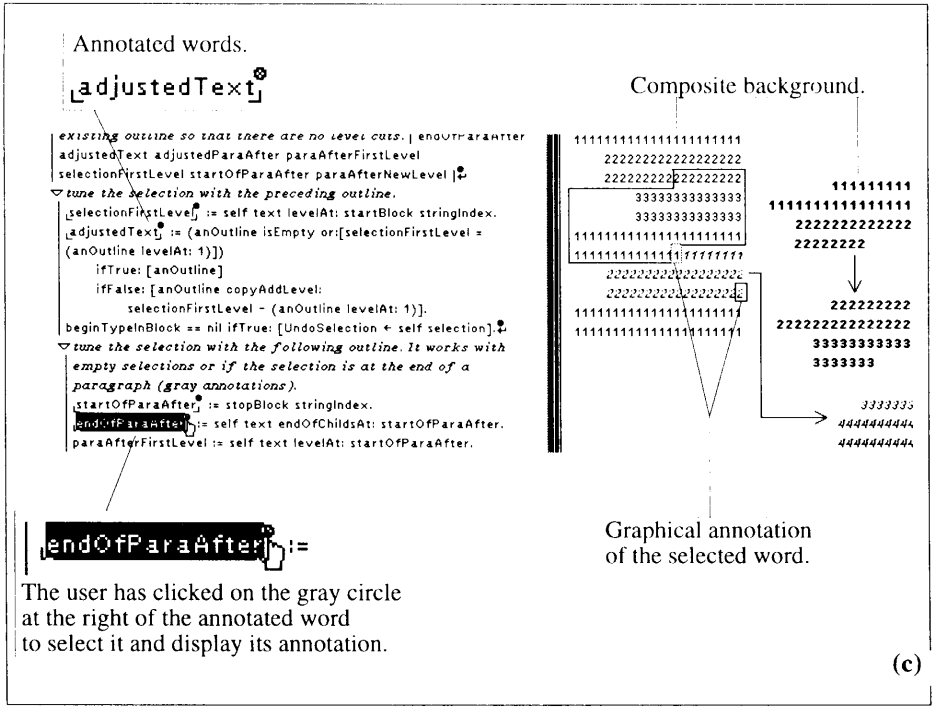


Fig. 7 b and c. Annotations (continued).

not yet been validated and asks for validation and discarding of each paragraph found (cf. Fig. 6).
◊ Multiple representations of software components are allowed : for example, a class can show its hierarchy using an outline, or can show its definition and methods, or can show itself merely as a short string to avoid window cluttering.

Referential Links. Referential links are non hierarchical, usually directed, links. The origination of the link is called the link *source* or *reference* and the other end of the link is called the link *destination* or *referent.* [4, pp. 34].
We have distinguished two kinds of referential links: button-links and annotations, that differ in the way they are created and accessed.

Annotations link text and pictures together: the picture is said to be the graphical annotation of the text. Drawing an annotation in the graphics pane automatically and immediately links it to the text currently selected (which can be a group of words or a whole paragraph) in the outline pane. The link is accessed thereafter - and the annotation is drawn in the graphics window - merely by selecting the annotated text.
If the text selection is nested into other paragraphs, the parent paragraphs of the selected text form a chain of contexts for this selected text. Thus, the annotations of these parent paragraphs are also displayed in the background of the annotation of the selected text: this is useful to mix different annotations or to build graphical backgrounds for an annotation (cf. Fig. 7a, b and c).

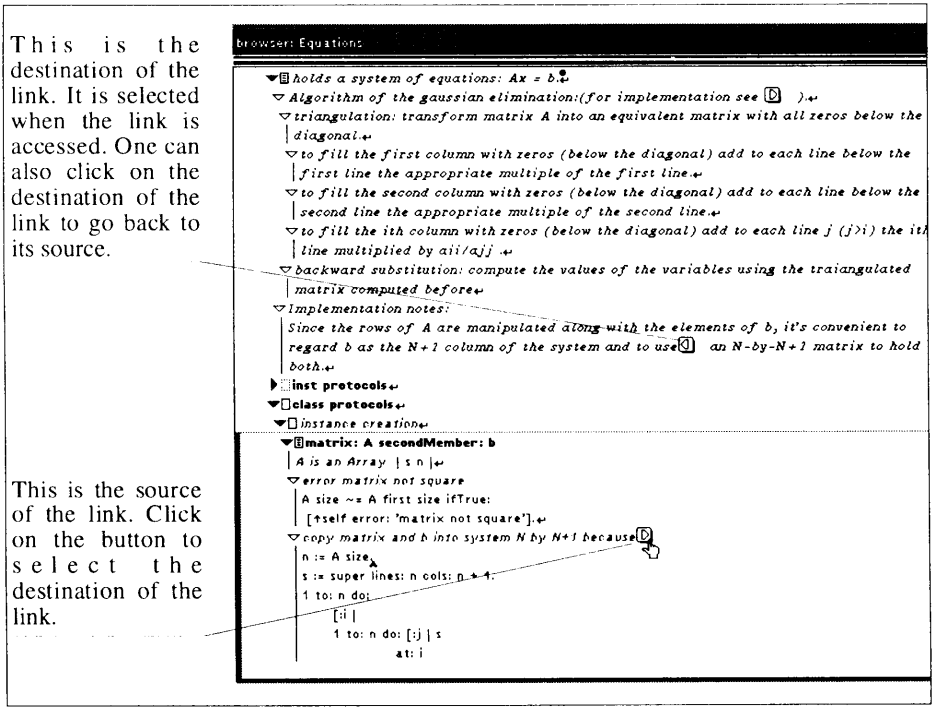


Fig. 8. Browsing buttons-links.

Annotations can be used to add graphical explanations to the code or to link graphical documentation with text. Graphical annotations are extremely useful to help understanding programs. Dynamic composition of annotations renders them even more powerful (see Fig. 5 and 7a, b, c for examples of annotations).

Button-links are used to link together two pieces of text or two pictures. They can be used to handle the connections between components of the software, for example, to connect the code of the methods with their corresponding informal specification in the classes documentation or to connect two pieces of code which are semantically but not physically connected. Button links can be used in the graphical editor to draw road maps or architectural diagrams of the software.
The source of the link is represented as a small button and the destination of the link is accessed by clicking on the button (cf. Fig. 8).
To avoid unnecessary opening of windows, the editor searches for the destination of the link first in the window containing the source link, then in the other opened windows and as a last resort, it spawns a new window to show the destination of the link.

Text Layers. Often Programmers add to their program pieces of text or code which are not involved in the computation. This text or code can be debugging statements, statements to display an animation of the program or to print values of expressions, or merely comments explaining the purpose of some statements. They are all intended to help understanding the program's operations and are usually deleted afterwards, only to be needed six months later.
To handle this issue, we have introduced *text layers*. To affect a layer to a piece of text

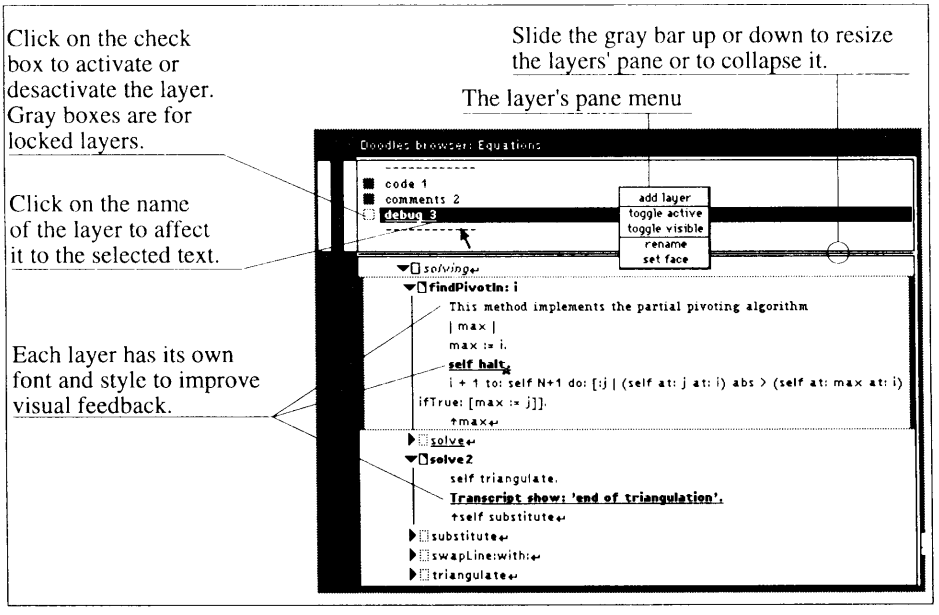


Fig. 9. Text layers.

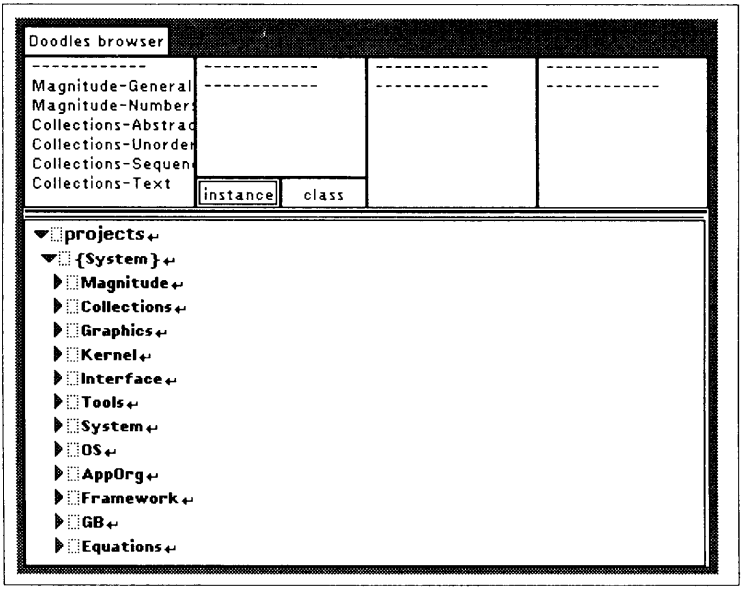
one first selects the text to alter and then chooses a layer in the layers menu (cf. Fig. 9). Each layer has its own font to recognize it and can be made visible or invisible (this is useful to mask cluttering layers). In addition, each layer has an operation associated to it and a number of flags to interactively parametrize this operation. For example, a debugging layer has an activity flag. When it is inactive, all the statements that are included in the debugging layer are not executed. Thus, debugging statements are virtually added or deleted with the click of a mouse (cf. Fig.9) and without recompilation.
The operation associated to a layer is invoked before the compilation: this operation takes some text and transforms it into another text. For example, the comments layer merely encloses the text into quotes (this way, comments do not need to be enclosed in quotes if they are in the comments layers). An activable layer transforms the code into: (Layer code: N) isActive ifTrue:[CODE], where N is the unique identifier of the layer and CODE the initial code.

Coping with Disorientation. A common problem with hypermedia systems is what Conklin calls *disorientation* [4, pp.38]. Since there is a lot of information displayed in a window, the user is likely to feel *lost in space.*
To cope with this problem, we have included the standard browser pane at the top of the window and are currently experimenting with a mechanism for memorizing previously accessed links.
The browser pane operates exactly as in the standard Smalltalk browser and is synchronised with the outline editor (cf. Fig. 10 a, b and c). Furthermore, it does not have the limitation of the standard browser described in Fig. 2 (compare it with Fig. 11).

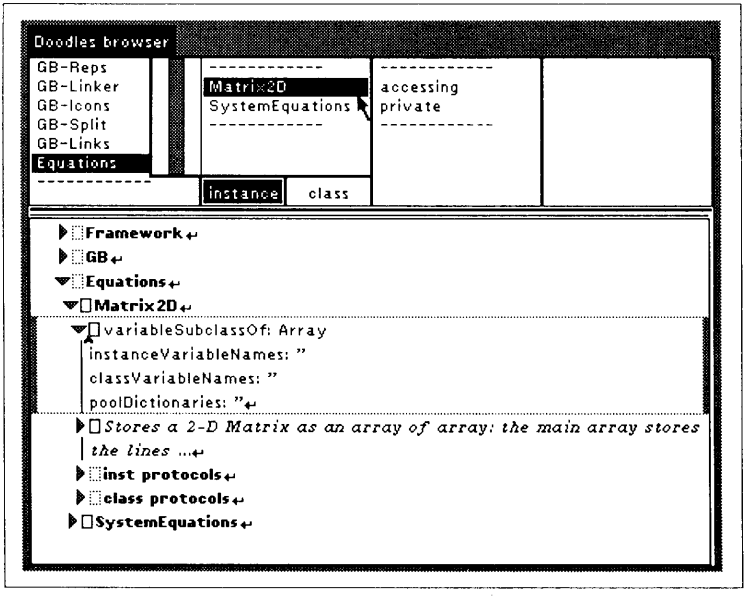
Interactive Compiling. If an error occurs during evaluation of a Smalltalk code (whether in the browser, the debugger or the inspector), an error message is inserted at the location of the error. If the error can be automatically repaired (e.g. undeclared or misspelled variable) the incriminated piece of code is selected and the user can choose within a menu the kind of reparation to perform on it.
In our browser, a few things have changed and must be taken into account:
- The error position is not relative to the beginning of the window but to the beginning of the current paragraph.
- Some text to compile may be invisible in the display: this can be either text in an invisible layer or text in a collapsed paragraph: hidden text containing an error is automatically shown in order to select the corresponding code.
- layers' operations may have transformed the program by adding invisible text. The interactive compiler must know about every transformation made to the text in order to correctly locate and show errors.

3.3 Other Tools

The Outline Inspector. The outline inspector view is composed of two panes (see Fig. 12a): the left pane shows a computed outline view of the objects being inspected and the right view a computed graphical representation of these objects. These representa-



(a)



(b)

Fig. 10. Using the browsing pane.
(a) At start-up, the outline pane shows an outline of the whole class library.
(b) If the users makes a selection in the browser pane (for example, if he selects the class Matrix2D), nested outline paragraphs are automatically opened until the paragraph defining the class appears.

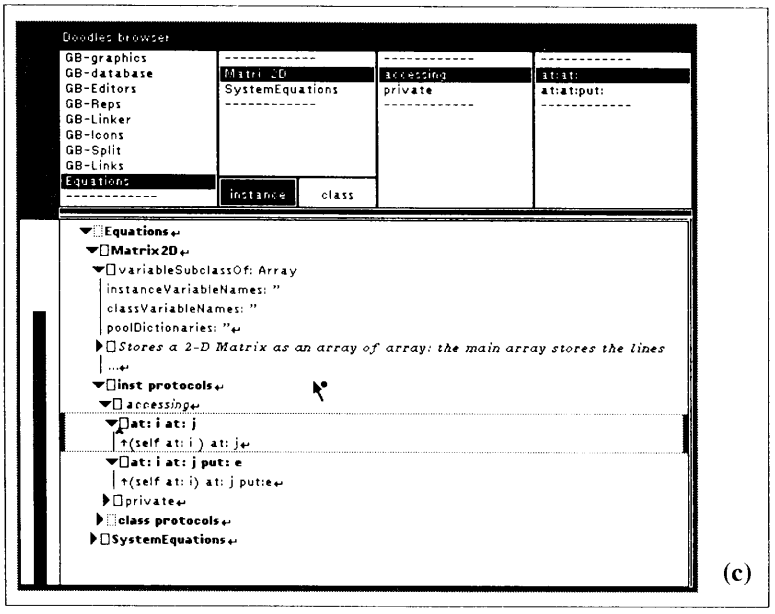


Fig. 10 (continued). Using the browsing pane.
(c) Inversely, if the user manually opens paragraphs in the outline editor (here, the method at:at:) , the selection in the browser changes to correspond to the text selection.

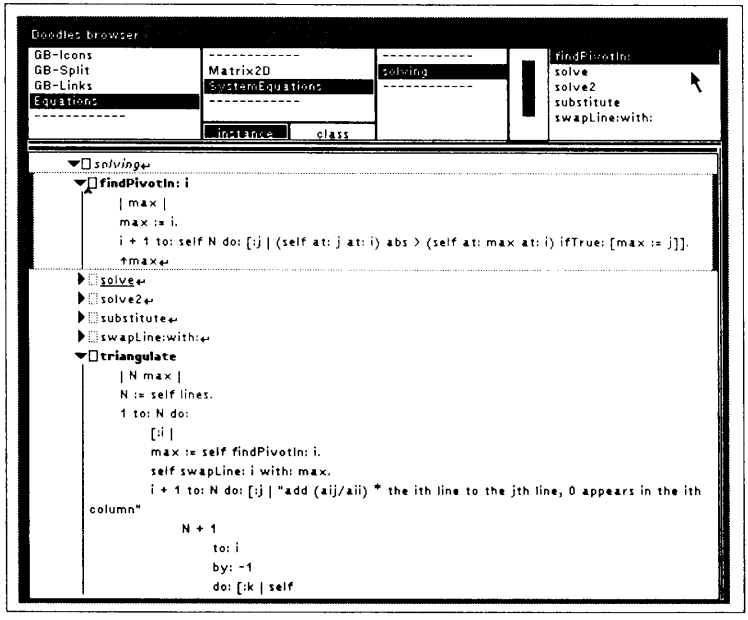


Fig. 11. Multiple browsing. This Figure is the counterpart of Fig. 2. If the user selects the method findPivotIn: while the method triangulate is opened, the paragraph corresponding to the newly selected method is opened, leaving the previous paragraph untouched.

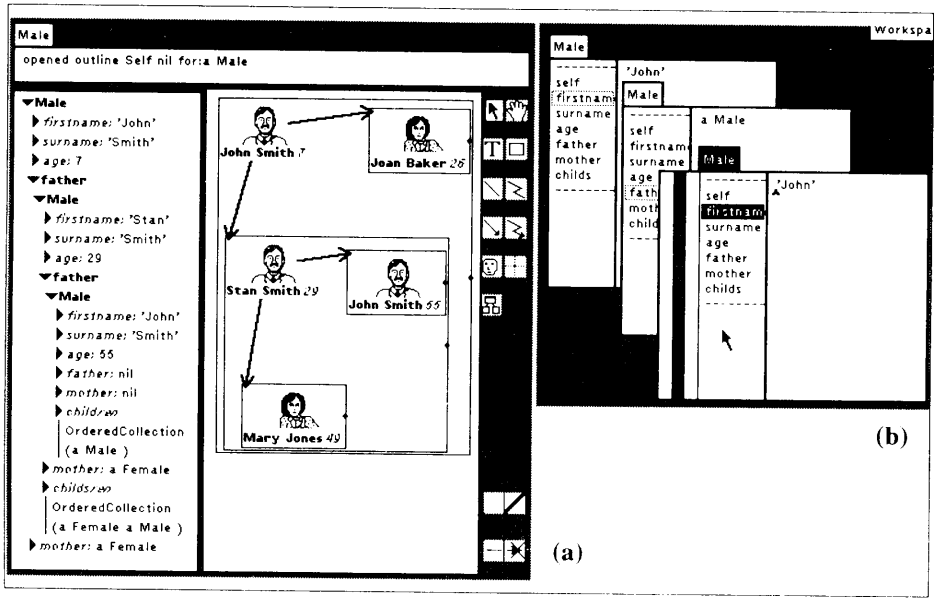


Fig. 12 a and b. Comparing the inspectors. These Figures show how a specific inspecting problem can be addressed by each version of the inspector: We would like to check if a child and his grandfather share the same first name 'John'.
The outline inspector (at the left side) handles the inspection with a single window. Moreover, it shows all the 3 persons' data both as an outline and as a (computed) diagram. The old inspector (at the right side) needs 3 overlapping windows which show only 2 informations and the underlying structure disappears once the windows are moved.

The statements being evaluated are bordered with a black one-pixel outline.

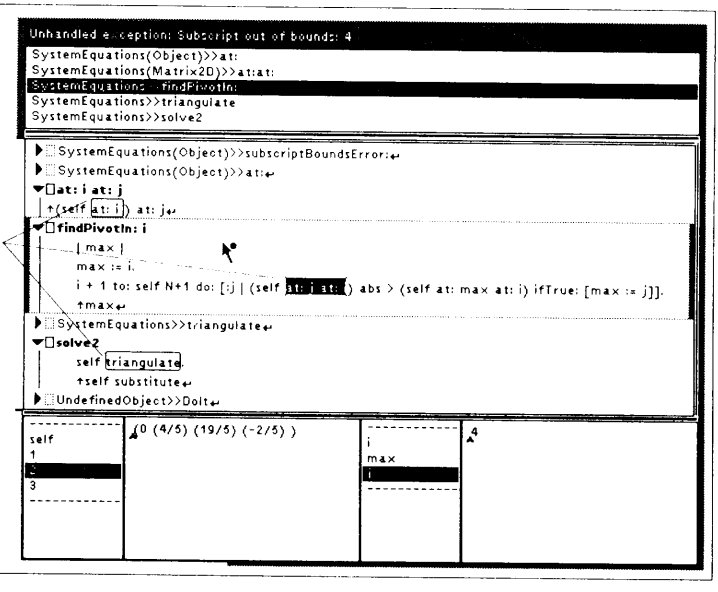


Fig.13. the outline debugger - this Figure and Figure 4 handle the same debugging issue.

tions are obtained by sending appropriate messages to the inspected objects through their semantics plugs (cf. section on semantic plugs below).
◊ The ChyPro inspector allows viewing and modifying of arbitrarily nested objects in a single window.
◊ It combines computed graphical and outline representations.
◊ Informal annotations can be temporarily added to the computed display of objects.

The Outline Debugger. The outline debugger view is composed of 3 panes (Fig. 13): The upper and lower windows are identical to those of the standard debugger (cf. Fig. 4).
The central window is identical to that of our browser (cf. Fig. 5).
◊ The debugger allows to browse in the same window several non-contiguous contexts of the stack. This is useful to follow the control flow over several methods.
◊ As in the browser, annotations and documentation can be retrieved and added to the methods. Annotations are displayed relative to the statement currently executed. So stepping through the code dynamically updates the annotations.

3.4 Implementation Issues and User Evaluation

Semantic Plugs. In the standard browser, the code view holds a computed textual representation of some Smalltalk object, depending on the selection in the browser's lists. In our browser, each paragraph in the outline view is equivalent to the standard browser's code view. Thus, each paragraph must be connected to the object it represents. To implement this connection, we use an intermediate object called a semantic plug : the plug is said to be the semantic of the paragraph.
The plug holds information about the kind of the connection. This information is:
◊ a pointer to a Smalltalk object.
◊ a user-defined pair <part,value> which identifies uniquely the part of the object that is represented by the plug. This allows several plugs to share the same object.
◊ the type of the representation : this allows each paragraph to have more than one representation.
◊ a modification flag that tells whether the paragraph has been modified since its last validation.
Figure 14 shows examples of plugs.
Semantic plugs are specialized in OutlinePlug to handle outline representations, GraphicPlug to handle graphic representations and GraphicPlugWithStack to handle objects receiving messages in an execution context.
All paragraphs need not have a semantic plug: for example, in Figure 7a, the inner paragraphs of the method replaceAndAdjustSelectionWith: do not have any semantics, as indicated by the paragraphs' white buttons.
In order to handle the textual and graphical representations of its semantic object, the plug sends it several distinct messages
◊ to get a menu, which will then be used to perform object specific operations (see Fig. 15).

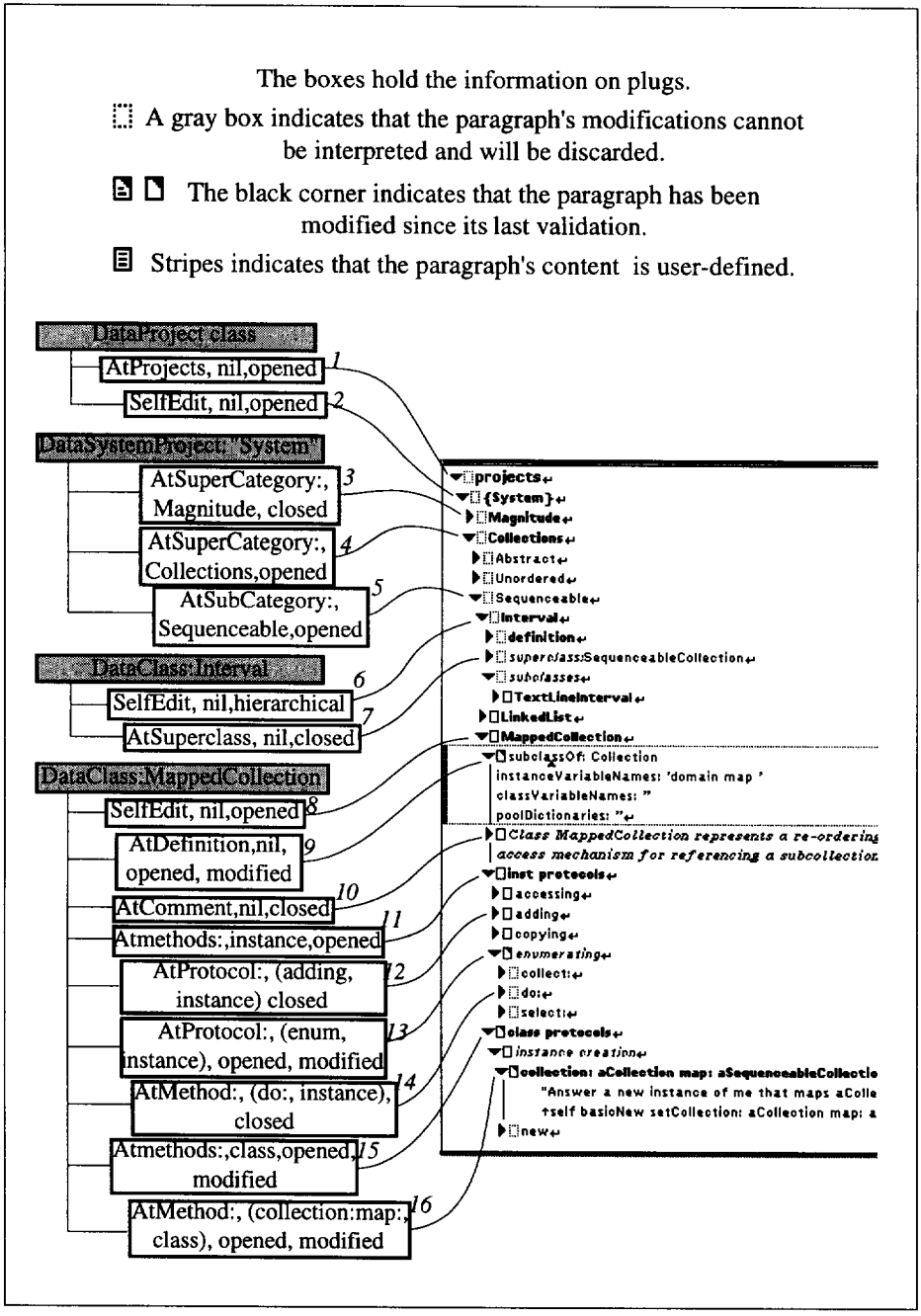


Fig. 14. Plugs and outline mapping. The plugs are in numbered white boxes and the semantic objects are in gray boxes. Each plug is represented by a triplet <part, value, type>. Unlike Smalltalk's standard browser, instance and class methods are plugged to the same object. Thus, we identify a method by an array #(message classOrInstance).

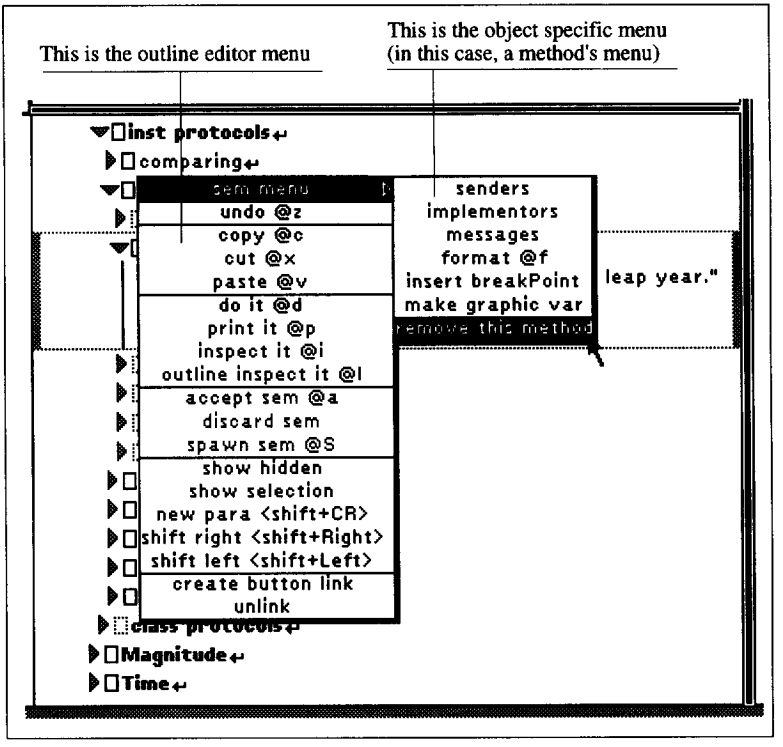


Fig. 15. Object specific menus.

◊ to retrieve an external user-defined representation of the object (typically, an outline representation of a method, with all layers and graphical annotations). If no user-defined representation exists, a generic outline representation is computed instead, which then serves as a canvas upon which the programmer can add informal information.

◊ to interpret the outline representation : an object interprets each of its partial representations in a specific way. For example, a class will focus on the structural information of its protocols outline representations. It will rather focus on the code when interpreting a representation of some of its methods. In the case of a representation of its documentation, a class will merely store the data without tempting any semantic interpretation.

◊ to get a computed graphical representation of some aspect of an object (cf. the graphic pane of the inspector on Fig. 12a). Computed graphical representations can also be used to draw (manually or automatically) architectural diagrams of a software, which each element of the diagram semantically connected to the software object it represents.

◊ we intend to add other messages to handle direct manipulations of graphical objects (i.e. click, move, delete, drag-and-drop and so on), or even semantic interpretation of graphical diagrams.

The messages to be sent, called *representation handling messages,* are computed depending on the plug's type and the operation to perform. For example, The plug #16 <AtMethod:, (collection:map:, class), opened> on Figure 14 will send to the DataClass MappedCollection the message openedOutlineAtMethod:#(#collection:map: #class) In: requestor to get the outline representation of the method, the message openedMenuAtMethod:#(#collection:map: #class) to get the semantic menu, the message accept: someRep openedOutlineAtMethod:#(#collection:map: #class) In: requestor to interpret the representation, or the message openedGraphicAtMethod: #(#collection:map: #class) In: requestor At: aPoint to get a graphical representation of the method.

Chypro's Architecture. ChyPro classes (see Fig. 16) can be grouped in two categories: hypermedia and programming.
The hypermedia classes implement the graphics and outline data structures, the user interface, the links and annotations and the semantic plugs. The graphical and textual classes are symmetrical with respect to the central class LinkerAbstract and its specializations. This class coordinates the outline and the graphic editors and serves as a mediator between the editors, the plugs and the programming environment.
The programming classes implement the representation handling messages for classes, projects, basic Smalltalk classes and object specific methods. Representation handling messages also handle the external storage of outlines and graphics using Smalltalk's 2.5 Binary Object Streaming Service (BOSS) [1].

User Evaluation. The ChyPro tools have an acceptable response time, not much slower than the standard tools on a 68040-based computer.
We have tested it with experienced Smalltalk programmers, so that they can compare the new tools with the standard ones:
• The interface seems confusing at first, but after the user gets adaptated to it, the simultaneous browsing capabilities appear very useful for creating new classes and methods.
• Although adding annotations is quite exciting, users should avoid to graphically annotate every piece of code and restrict annotations to complex methods.
• Text layers are helpful for debugging and exploring implementation alternatives. We are trying to extend text layers to handle versioning.
• Although the debugger updates its display incrementally, it is still slower than the Smalltak debugger on step-by-step execution. We are currently working on a combined debugging, inspection and animation tool with powerful tracing and animation capabilities that should reduce the need for stepping through a program.

[8] The current implementation takes 500k of code and about 75 classes (plus 500k of hypermedia data).

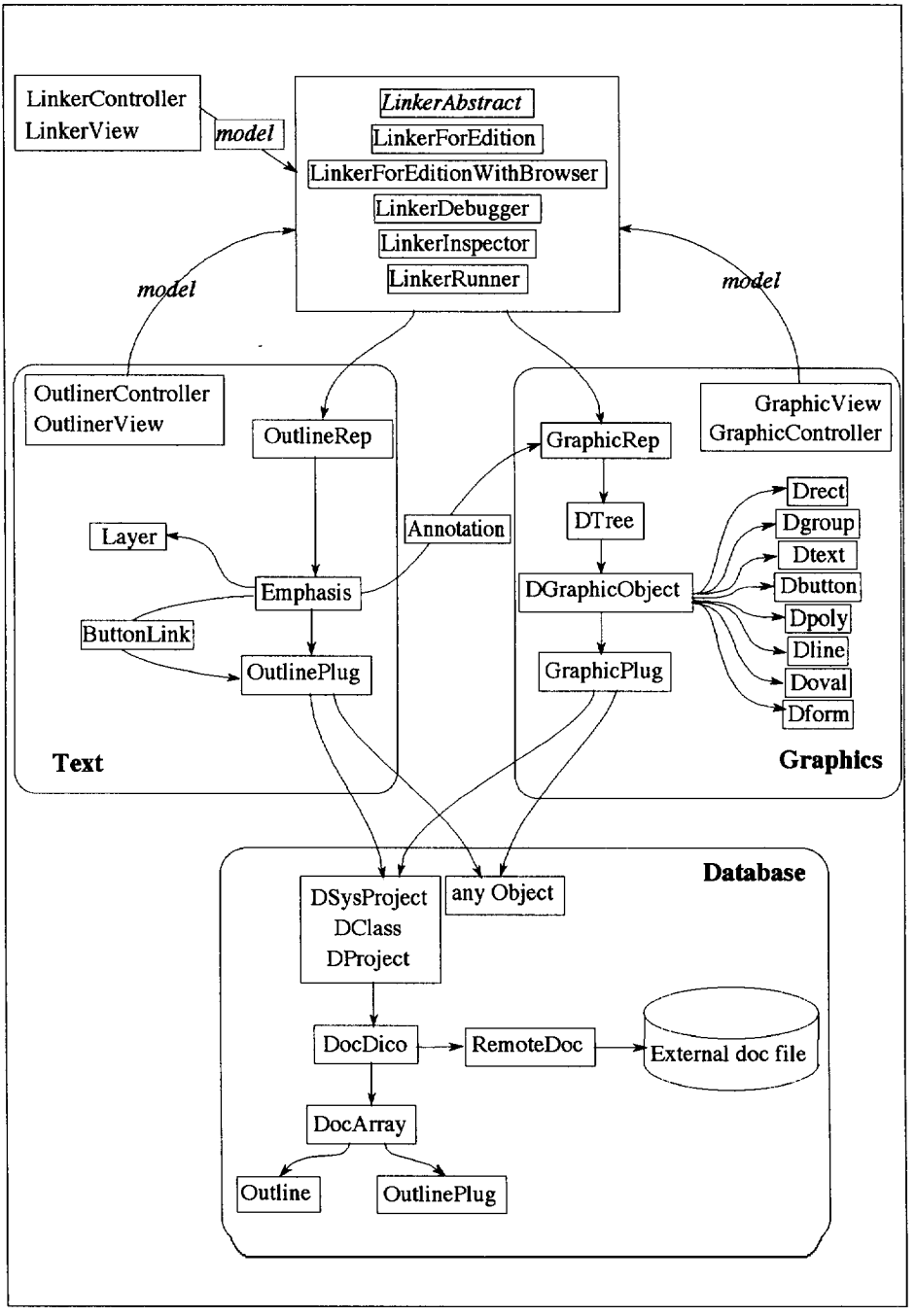


Fig. 16. The ChyPro classes organization. Several classes in the same box are alternatives depending on the tools and the context. Find the symmetry of graphical and outline classes.

4 Conclusion and Perspectives

We have shown how hypermedia techniques can be used to improve the Smalltalk programming environment. Yet, a few important issues remain open :

◊ We started writing ChyPro with Smalltalk-80, release 2.3. We then switched to Smalltalk 2.5 to use the Binary Object Streaming Service. We will soon switch to VisualWorks. Thus we will have to rewrite all the user interface, which represents about half of the code's size.

◊ We have designed ChyPro so that it can handle large scale software. So far, we have used it to write in their entirety small but relevant programs such as the Gaussian elimination. We have also used it to manage several complex issues in the design of a big program such as ChyPro itself[8]. We intend now to experiment it with other experienced programmers in their daily work.

◊ As the power of computers grows, new programming systems using complex techniques such as 3D graphics [9] have appeared. Programming systems based on visual displays and interaction and perhaps virtual reality will certainly manage to break Winograd's *barrier of complexity* [14]. Although virtues of pictorial representations are incontestable, we believe that verbal expression is a necessary part of human reasoning in such complex problems as mathematics, engineering and programming. We believe with Georg Raeder [11] that the programming systems of the future will use appropriate combinations of textual and pictorial representations to handle software. Our proposal for a new Smalltalk environment is a first step in this direction.

References

1. D. Bay & al. : Smalltalk 80 version 2.5, Reference Guide and Advanced User's Guide. Parcplace 1989.
2. S.C. Bilow : Object Explorer for Visualworks. Journal of Object Oriented Programming, Vol. 7, #3, June 1994.
3. H.D. Böcker, J. Herczeg : Browsing Through Program Execution. In: D. Diaper & al. (Eds.) : Proceedings of Interact'90. Elsevier Science Publishing 1990.
4. J. Conklin : Hypertext, an Introductory and Survey. I.E.E.E. Computer, September 1987.
5. A. Goldberg : Smalltalk-80, the Interactive Programming Environment. Addison Wesley, 1984.
6. P. Krief : M.P.V.C, un Système Interactif de Construction d'Environnements de Prototypage de Multiples Outils d'Interprétation de Modèles de Représentation. Technical Report, Université Paris 8, 1990.
7. W.R. Lalonde, J.R. Pugh : Inside Smalltalk vol. I. Prentice-Hall, 1990.
8. W.R. Lalonde, J.R. Pugh : Inside Smalltalk vol. II, Prentice-Hall, 1991.

9. H. Lieberman : a Three-Dimensionnal Representation for Program Execution. In: New Paradigms for Software Development , IEEE Computer Society, 1986.
10. J. Nanard : la Manipultation Directe en Interface Homme-Machine. Université de Montpellier II, Doctorate Thesis, December 1990.
11. G. Raeder : A Survey of Current Graphical Programming Techniques. IEEE Computer, August 85.
12. B. Schneiderman : Designing the User Interface. Addison-Wesley 1992
13. R. Sedgewick : Algorithms. Addison-Wesley, 1984.
14. T. Winograd : Breaking the Complexity Barrier (Again). In D.R. Barstow & al. (Eds) : Interactive Programming Environments. McGraw-Hill 1984.