The Mjølner Environment: Direct Interaction with Abstractions

Görel Hedin & Boris Magnusson
Department of Computer Science, Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
email: gorel@dna.lth.se, boris@dna.lth.se

Abstract

This paper presents the user interface to programs and their execution in the Mjølner Programming Environment. The key idea is to present the programming language abstractions, such as classes and procedures, as individual windows which the user can interact with directly. This approach is used consistently to visualize both a program and its execution. The windows are arranged hierarchically reflecting the static nesting of blocks. The window hierarchy gives powerful support for interaction and navigation in a program. Incremental compilation techniques are used to make a high level of interaction and integration possible.

1 Introduction

In this paper we present the user interface to programs and their execution in the Mjølner Incremental Programming Environment. The objective of the Mjølner project is to provide highly interactive programming environments for industrial use. Mjølner is primarily intended to support strongly typed object oriented languages in the Simula tradition [DMN72]. There seems to be a very rapidly growing interest in this area with many new languages emerging, e.g. Beta [KMMN87], Eiffel [Mey87], C++ [Str86] and Trellis/Owl [Sch86].

The major contribution in the design of the Mjølner user interface is the consistent focus on the abstractions used by the programmer. Thus the language abstraction mechanisms *class* and *procedure* ("method" in Smalltalk terminology) are brought out and presented as windows. A hierarchical window system is used to represent nesting of these constructs. The effect is comparable to the effect of the Macintosh finder for hierarchical file systems with the advantage of direct manipulation of objects on the screen and ease of navigation. We claim that the user interface is *object-oriented.*

The Mjølner environment includes an incremental compiler and a very flexible run-time system which allows modification of a running program. This results in a system that offers unusually close integration of program modification and execution. The object oriented user interface style is used throughout the programming and execution process. The environment is under development and will initially support programming in Standard SIMULA [Sim87].

The Mjølner project [DLMM86] is a Nordic effort in which companies and universities in Norway, Denmark and Sweden participate. The incremental programming environment presented here is only one of

several activities within the project. Other activities focus on program databases, an implementation of the Beta language, syntax-directed editors, and object-oriented specification languages [MBD87].

The rest of this paper is organized as follows: In section 2 we describe the concept of hierarchical windows on which the user interface is built. Section 3 describes the environment from the user point of view and the facilities available. In section 4 some brief comments on the implementation are given. Section 5 concludes the paper.

2 Object-Oriented Use of Hierarchical Windows

The use of personal workstations with high resolution graphics, mice, and window techniques has had a revolutionary effect on the development of programming environments. Windows allow the user to view and interact with several things at the same time, thus getting rid of the "modes" that are inherent in traditional terminal-based mini-computer environments [Tes81]. Windows can be used in different ways. We differ between *activity-oriented* use of windows and *object-oriented* use of windows.

2.1 Activity-Oriented Use of Windows

The Smalltalk environment [Gol84] is a pioneer project in its utilization of windows for observing and manipulating both the program source and objects that are created during program execution. The use of windows in the Smalltalk environment can be characterized as *activity-oriented*, i.e. each kind of window enables the programmer to perform a certain activity. A typical example is the Smalltalk Browser window. The browser serves as a viewport into the underlying source code structure. The user can browse through this structure, looking at a single class or procedure ("method") at a time. In order to look at two procedures at the same time, two browser windows are needed. There is no particular relation between the browser windows. It is e.g. possible to let them show the same procedure body. Other kinds of windows in Smalltalk are Inspectors (used to inspect the contents of an object) and Workspaces (used to type in commands to the system). Magpie [DMS84] and Trellis [BHK87] are other examples of environments using windows this way.

2.2 Object-Oriented use of windows

A contrasting way of utilizing windows can be characterized as *object-oriented*. In this approach each window has a one-to-one correspondence with an object, making it possible for the user to *identify* the window with the object itself. The user performs all activity relevant to the object directly on or in the window. The Macintosh Finder [Wil84] is an example of a system using windows this way. Files and directories are shown as individual windows (or icons). This object-oriented use of windows was pioneered in the Star user interface [SIKVH82]. It lends itself to natural and powerful interaction mechanisms since the objects (windows) can be manipulated directly rather than by commands [Sch83]. E.g. a file can be moved to another directory in the Macintosh Finder by simply dragging its icon to the new directory window.

2.3 Hierarchical Windows

In Mjølner, windows are used in an object-oriented way, primarily to represent abstractions in programs but also on the top level as an interface to the file system. *Hierarchical windows* are used to express containment relations between objects (windows). The Mjølner hierarchical window system [HNRR88], is implemented on top of a kernel of X-window primitives and currently runs on both Sun and Vax workstations. The hierarchical window system allows any window to contain a local set of full functionality windows. Windows can be moved, resized, and iconized. Sibling windows and icons may overlap freely.

Traditional window systems, with a single set of overlapping windows, follow the metaphor of overlapping papers on a desktop. The hierarchical window system implies a generalized metaphor; a paper on the desktop may act as a viewport into a new desktop. When a "desktop" is moved on the screen, all its "papers" follow with it. If a "desktop" is made larger, more of its "papers" come into view.

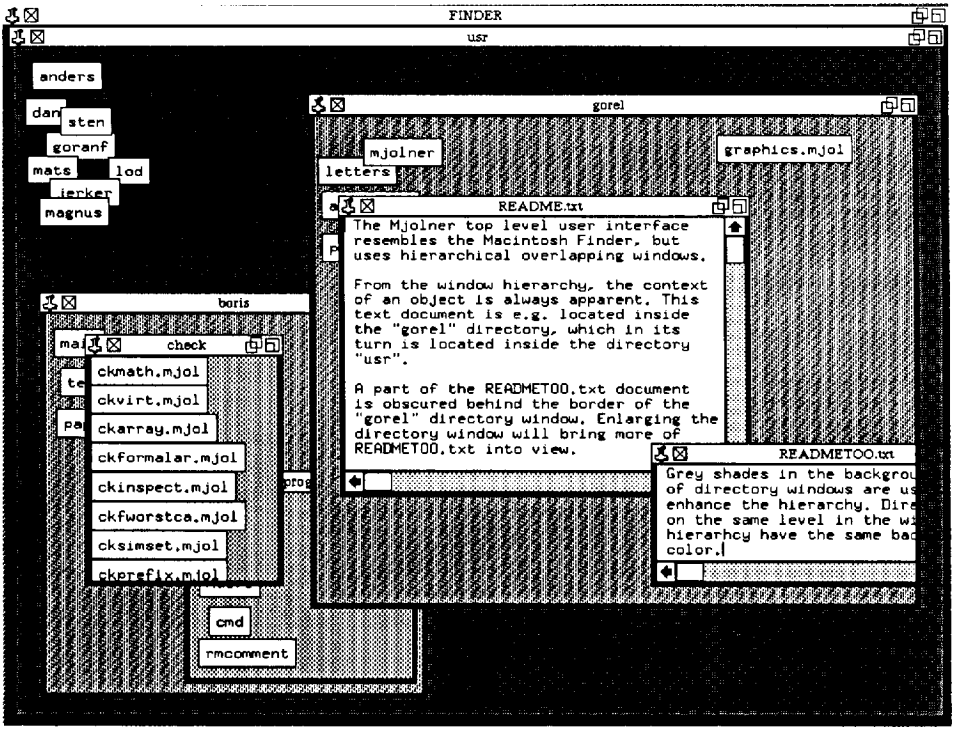

Figure 1. The Mjølner Finder

2.4 Use of Windows in Mjølner

Figure 1 shows an example of how windows are used in Mjølner. The top level window (the Mjølner Environment window) contains the file system. Each directory is shown as a window containing windows and icons for each of the files in it, similar to the Macintosh Finder. Clicking on a file icon expands it to a

window displaying the file in some appropriate way (depending on the file type) and allowing appropriate interaction. E.g. clicking on a text-file icon expands it to a text window with editing facilities. Contrary to the Macintosh Finder however, a Mjølner window which is expanded from an icon keeps its place in the window hierarchy. An object (e.g. a file) is thus always presented in its context (e.g. its surrounding directories) regardless of its current icon/window state.

The full advantage of hierarchical windows is seen when windows are used within applications to present substructures. In figure 2, a program file "graphics.mjol" has been expanded to a Mjølner programming window. This window contains the program represented as a hierarchy of nested blocks (classes and procedures), each shown as an individual window or icon. In this case, the window hierarchy is more than just an organizational device since it actually reflects the static scope rules of the programming language.

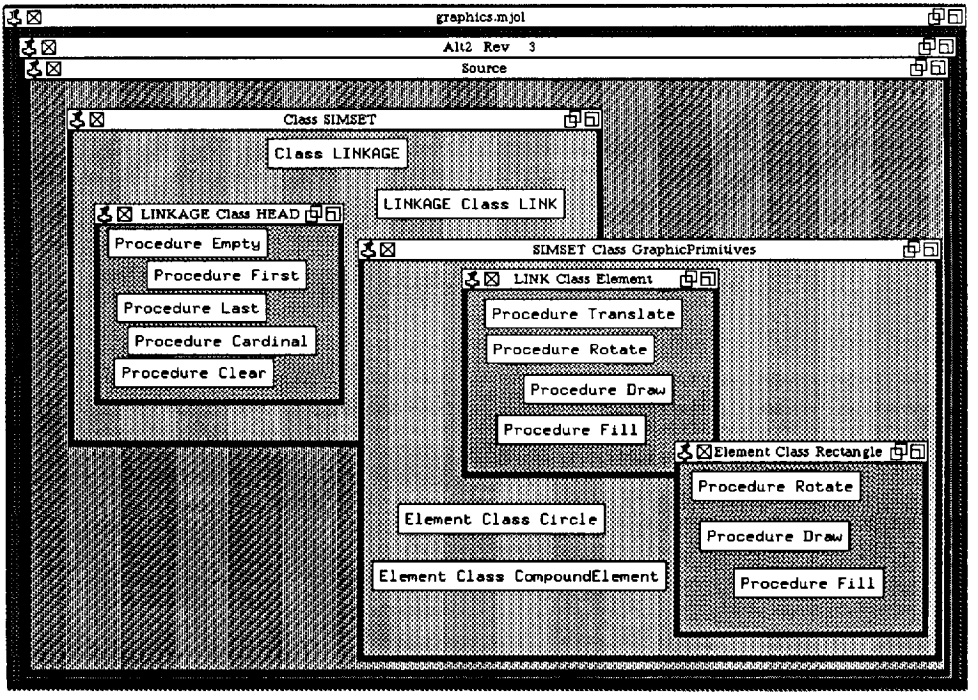

Figure 2. A Mjølner Programming Window

`GraphicPrimitives` is an application class containing classes for different graphic elements. Class `Element` describes the general behavior of a graphics element; translating, rotating, drawing, and filling. This behavior is overridden as needed in the subclasses. `GraphicPrimitives` inherits classes for handling double linked lists from the standard Simula class `SIMSET`. The elements are to be organized into lists, so class `Element` is a subclass to the standard class `LINK` in `SIMSET`.

It is important that the window hierarchy is used for "the right thing". The graphical layout of windows inside windows suggests using the window hierarchy to model containment relations. E.g. in the file hierarchy a directory *contains* its files. In the program block hierarchy, the class `GraphicPrimitives` *contains* the class `Element` and its subclasses `Rectangle`, `Circle`, and `CompoundElement` . The

class `Rectangle` in its turn *contains* the procedures `Rotate`, `Draw`, and `Fill`. In an object-oriented programming language there are many other relations that are important as well, e.g. subclassing, call stacks, and references between objects. In section 3 we will return to how these relations may be modeled in Mjølner.

2.5 Interaction Advantages and Problems

There are several advantages of the object-oriented use of hierarchical windows:

Objects are easy to find, since each object has a natural place in the window hierarchy. The window hierarchy can be used as a built-in browser of objects, and makes it easy to navigate in large systems.

Object context is always shown. The context is often crucial to identify an object, e.g. to differ between procedures `Draw` in different classes. When interacting with an object, it is often valuable to view parts of its context at the same time. E.g. when editing a procedure one usually wants to see the instance variables of the class at the same time. In the hierarchical window system the contextual relation between objects is always apparent.

Good utilization of screen space. Overlapping windows increase the virtual screen space considerably. The usage of hierarchical windows is an aid in better utilization of this space. A single mouse click brings a window and its inner windows to the top of the screen, or iconizes a window freeing the screen space of it and all its inner windows. This allows whole contexts with interior window structure to be brought in or out of view by a single mouse click. It is easy to arrange the windows so that only information of current interest is shown, leaving less interesting information as icons behind the expanded windows.

Interaction is more complex in a hierarchical window system than in a traditional window system. One problem is that the inner windows tend to become small. In order to enlarge an inner window, its enclosing windows often need to be enlarged as well, which sometimes becomes cumbersome. Another problem occurs when the user is temporarily uninterested in the full context of a window. The enclosing window borders then take up screen space which could be used for better purposes. Further experimentation is needed to come up with smooth interaction mechanisms which solve these problems.

2.6 Related Interaction Styles

The idea to use hierarchies of graphical objects to represent programs is also used in e.g. the BOXER environment [SA86]. In BOXER the key graphical structure is a box containing text which in its turn can contain inner boxes. A box is not a window, rather it is considered a special sort of character which is moved by cut-paste operations on the text. The boxes can be iconized and expanded by the user. They can, however, not be moved around freely because of their position in a text. This gives heavy restrictions on what parts of the system can be viewed simultaneously. BOXER is mainly intended to be used in education and to give novice programmers a "what you see is what you have" interface. We agree with the BOXER designers in the merits of such an interface, and we think it is very useful also for a professional programmer.

Fisheye viewing [Fur86] is a viewing strategy which "can show places nearby in great detail while still showing the whole world - simply by showing the more remote regions in successively less detail". Fisheye viewing thus share some properties of hierarchical windows, enabling the user to see both overview and detail at the same time. However, fisheye systems and hierarchical window systems have completely different styles of interaction. Fisheye viewing can be seen as an advanced scrolling technique; when the focus is moved, the context will be changed automatically as a function of the focus point. The hierarchical window system does not have a similar concept of focus. Opening a window does not change the status of other windows.

3 Programming and Execution in Mjølner

3.1 The block hierarchy

A program in Mjølner is primarily considered to be a hierarchy of blocks, where a block is a unit of abstraction such as a class or a procedure. The main characteristics of a block are that it has a local name scope and that it can be dynamically instantiated during execution: classes are instantiated to objects, and procedures are instantiated to procedure activations. The representation of the block hierarchy using a window hierarchy makes it possible for the user to navigate in the program in terms of its abstractions. A new class or procedure is created simply by creating a new window in the appropriate father window. The two-dimensional layout of son-block icons in a block window frees the user from the unnatural ordering between sibling blocks enforced by the traditional representation of a program as linear text. Since the block is the primary unit of abstraction used by the programmer, we find it very natural that blocks play the main role in the user interface.

3.2 Editing and Incremental Compilation

The local variables and the statement body of a block are presented in special subwindows of the block window. This makes it easy to e.g. view the variable declarations of a class at the same time as the body of one of its procedures, as in figure 3. The declarations and bodies are internally represented as abstract syntax trees which are unparsed as text. The editing is done directly on the tree using a syntax-directed editor. Syntax errors are thus impossible to make.

Incremental static semantic checking is performed in response to every editing operation. Static semantic errors such as type conflicts, undeclared variables, etc. are marked on the screen by underlining the erroneous construction. E.g. in figure 3, the name `anElement` is underlined. The user may ask for an explanation of the error, when the erroneous construction is the current focus of attention in the syntax tree. As shown in figure 4, the explanation reveals that `anElement` is not declared. When the declaration is added, the error-markings automatically disappear. The philosophy in Mjølner concerning errors, is that they should be visible but not intrusive. Errors are simply marked on the screen and may be corrected at any time by the user.

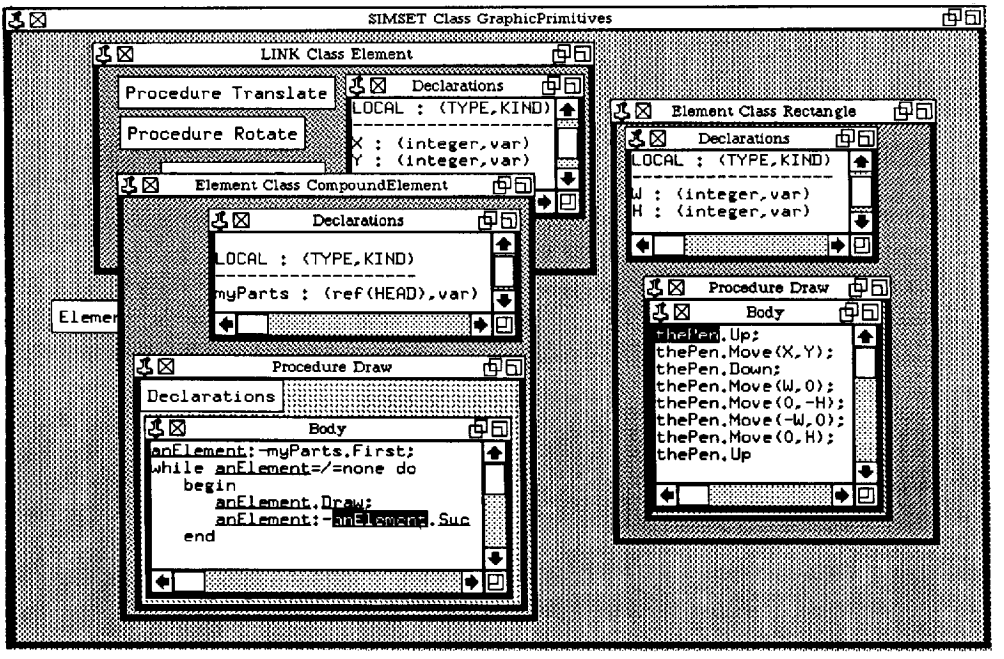

Figure 3. Declarations and Statement Body of Blocks

The class `CompoundElement` has a variable, `myParts`, which is a list of its constituents. It draws itself by walking through the list and drawing each constituent. Class `Rectangle` is represented by its the upper left corner (inherited from class `Element`) and its width and height. It uses a plotter pen to draw itself.

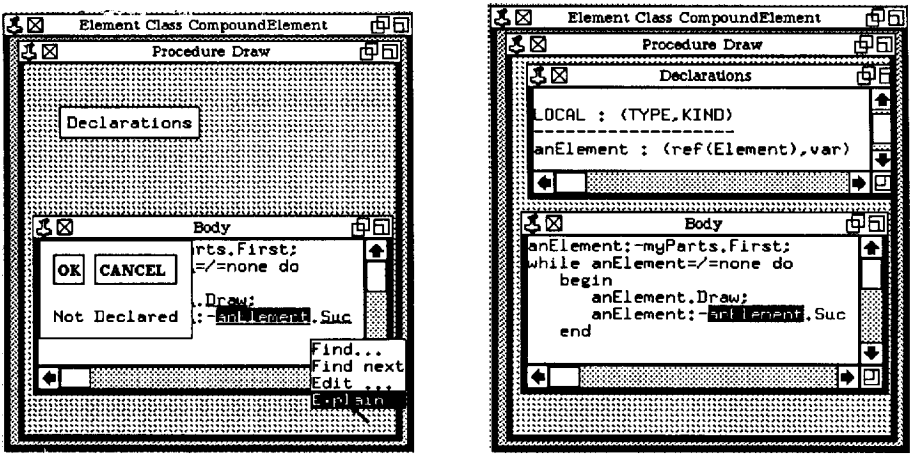

Figure 4. Correcting a Static Semantic Error

The variable `anElement` is underlined because it is not declared. When the user corrects the error, by inserting a local declaration, the underlining disappears.

Code is generated incrementally on a block-by-block basis, and is incrementally loaded into the executing program. This is done automatically as needed by the system, so the user does not have to issue any explicit command.

3.3 Execution

The executing program is presented to the user as a system of block instances (objects and procedure activations), each represented by a window of its own. In analogy to the program definition, the windows are organized hierarchically according to the static scope. E.g. in figure 5, the `Draw` procedure of `CompoundElement` has called the `Draw` procedure of a `Rectangle`. The procedure activation windows are shown inside the appropriate object windows. The graphic element objects are shown inside the `GraphicPrimitives` object.

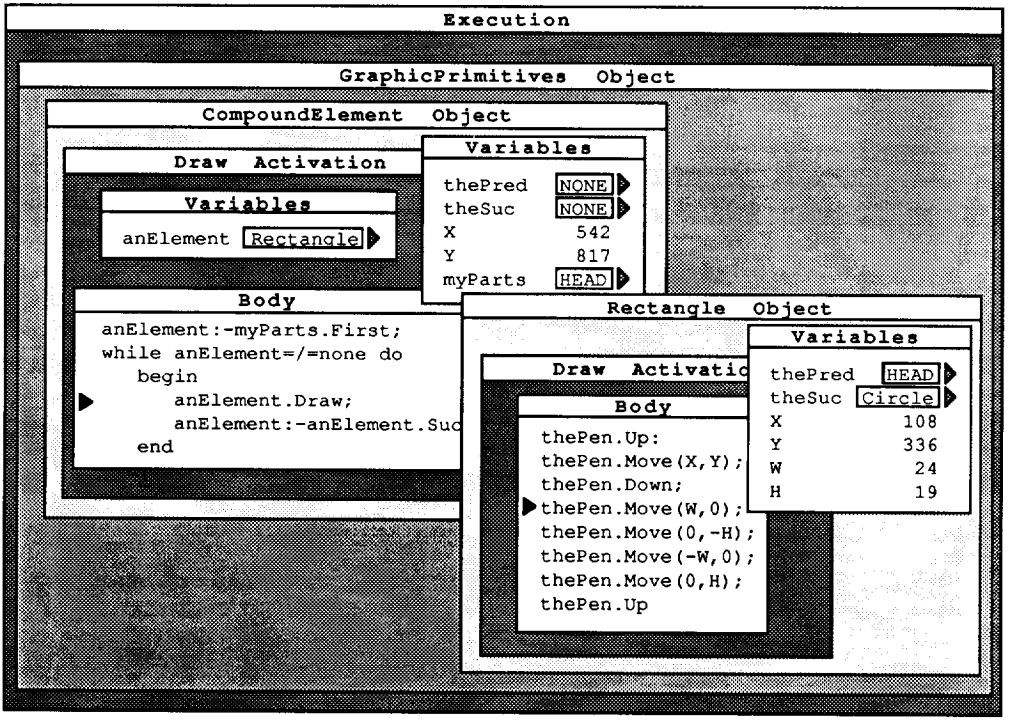

Figure 5. Objects and Procedure Activations in an Execution State.

The procedure `Draw` of a `CompoundElement` has called the `Draw` procedure of one of its constituent elements. This element, a `Rectangle`, is in the middle of drawing itself.

Each block instance window contains subwindows for data and statement body, which parallel the declaration and body windows inside a block window. The data window shows the variable values, and the statement body window shows the current execution point. An object in Simula is considered to be described by a concatenation of its class and superclasses. The data window will therefore contain inherited

variables as well as variables defined in its actual class. E.g. the `Rectangle` object in figure 5 has `thePred` and `theSuc` variables inherited from class `LINKAGE`; `X` and `Y` are inherited from class `Element`; `W` and `H` are defined in class `Rectangle`.

If all existing block instances were shown on the screen, this would lead to a very cluttered view. Usually the user is interested in seeing only very few of the block instances, e.g. the currently executing instance, and some objects involved in the current computation. In Mjølner, the user can interactively open the objects and procedure activations of interest. Reference variables (pointers) in the data windows are presented as buttons. Pressing a button, using the mouse, causes the window of the referred object to appear on the screen. Windows enclosing the referred object are opened automatically when needed. If the referred object is already present on the screen, the user is given visual feedback on its position.

A call stack window presents a list of buttons for the currently active block instances. Thus, when execution is suspended, e.g. at a breakpoint, a procedure activation can be opened via a button in the call stack window. Object windows can then be opened via the reference variables in already opened windows. See figure 6.

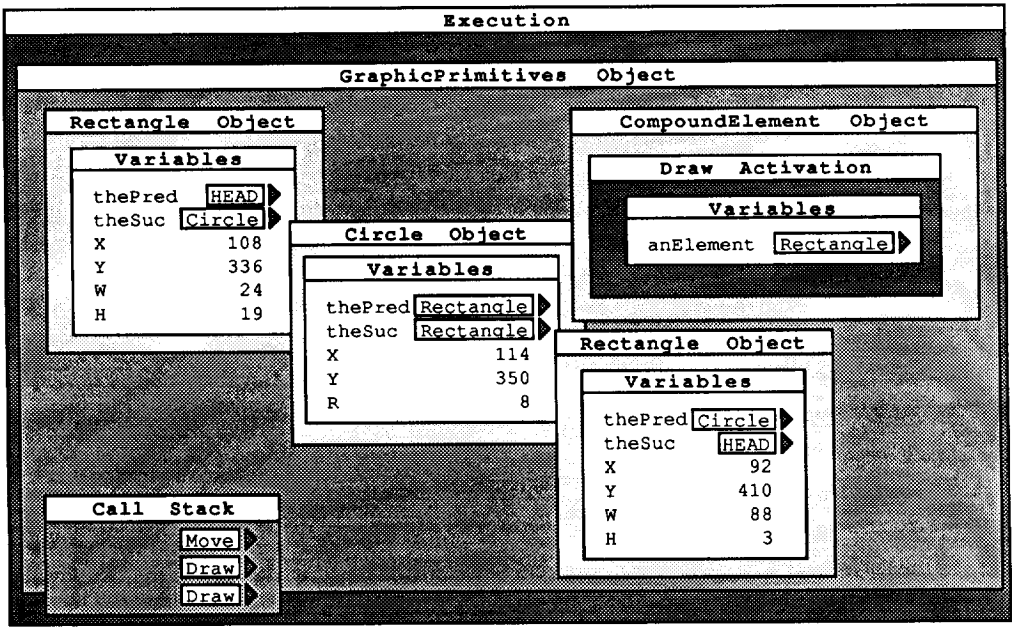

Figure 6. Following References in an Object Structure

Upon execution suspension, the user has opened the `Draw` activation via the lower button in the call stack. This caused the enclosing objects `CompoundElement` and `GraphicPrimitives` to appear as well. The next step was to open the Variables window in the `Draw` activation. The variable `anElement` currently refers to a `Rectangle` object. Pushing the `Rectangle` button, caused the `Rectangle` object to appear (in the top left corner of the picture). The user then follows `theSuc` references to bring the rest of the objects in the list to appear on the screen.

Execution can proceed stepwise or to a breakpoint. If desired, the contents of the block instance windows can be updated continuously during the execution. This makes it possible to e.g. monitor an object structure. More details on observation and debugging facilities are given in [THM87].

3.4 Integrated editing and execution

Although Mjølner is a compiling system, program editing and execution is very closely integrated, even more than in many interpreting systems. The program definition may freely be edited during execution. Errors can be corrected in executing programs, functionality can be added and immediately tested in an execution state with existing object structures, and experiments can be performed with executing programs.

In general, such close integration between editing and execution leads to version and consistency problems. E.g., what should happen if a variable is added to a class and there are existing objects of that class? In Mjølner, these problems are solved by providing two mechanisms; block instances of different versions may co-exist in the same execution state, and block instances may be converted from an older to a newer version. This is described in more detail in [HM86] and [HM87].

In many cases, however, the results of the close integration is very straight forward. E.g. if the implementation of a procedure is changed, Mjølner's default behavior is to let old procedure activations continue to execute according to the old version, but new procedure activations will execute the new version. New procedures and classes can be added and tried out immediately.

3.5 Presenting relations

The hierarchical window system captures the fundamental relation of static enclosure. There are, however, many other important relations in a program, e.g. the subclass-of relation, the instance-of relation (an object is an instance of a class), and the refers-to relation (for reference to objects). In section 3.3 we saw how buttons were used to follow refers-to links. They could be used also for subclass-of and instance-of relations. Buttons make it possible to easily navigate in a complex object structure in a manner similar to the ideas in hypertext systems [Con87].

An attractive extension of buttons would be to actually draw a connector between two windows when a button is pressed. Such window connectors would have to be an integral part of the window system in order to stay connected when windows are moved, resized, and iconized. The use of connectors would make it possible to show the relations between objects and their interiors at the same time, as shown in figure 7.

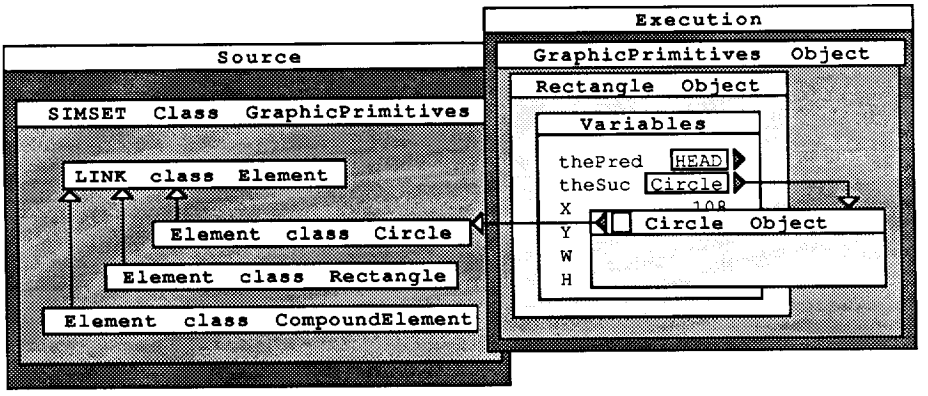

Figure 7. Presenting Relations

Window connectors could be used to show different kinds of relations. In the Source window, connectors are used to show the subclass-of relation. A connector from the `Circle` object to the `Circle` class shows the instance-of relation. A connector from `theSuc` variable to the `Circle` object shows a refers-to relation.

4 Implementation

The Mjølner environment is unusual in that it combines compilation technology with a very high degree of interactivity. Earlier incrementally compiling environments, such as the Cornell Program Synthesizer [TR81], Gandalf [MF81], and DICE [Fri84], have used a more traditional presentation of programs. In this section we will comment briefly on some important aspects of the implementation, and conclude with a report on the current status.

4.1 Incremental Compilation Techniques

To be of use in an industrial setting, an interactive environment must be able to handle large programs without noticeable increases in response times. It has therefore been of major importance in Mjølner to use incremental compilation techniques which can be *scaled up*, i.e. whose performance is relatively independent of program size. A typical bottleneck in this respect is static semantic checking. Changes to declarations may affect the static semantics in places far away from the declaration. E.g. adding a parameter to a procedure inside a class will affect all calls to that procedure. Since the static semantic checking is performed incrementally while editing, it is essential that the affected places are found quickly. In Mjølner, a method based on attribute grammars is used, but allowing side-effects when evaluating the attributes [HDM87]. The side-effects are used to build up information structures which are used to quickly find affected places after a change.

Code generation is not as time-critical as static semantic checking, since the code is needed only when execution is resumed. In Mjølner, code is generated in chunks on a block-by-block basis. To minimize the delay before execution can be resumed, we are considering to implement lazy code generation and code

generation in the background, utilizing the programmer's "think time", similarly to as it is done in the Magpie environment [SDB84].

4.2 Flexible Run-time Environment

An important feature of Mjølner is that execution is allowed to continue after the program has been edited. To accomplish this, Mjølner uses a very flexible run-time architecture. The generated code-chunks are arranged in a "template tree" which parallels the block tree. A template is a data structure which contains information relevant to all instances of a particular block; activation record size, garbage collection information, link to superclass template, etc. Block calls (procedure calls and class "new"s) are performed by indirect jumps via the template tree. The code for a block call is thus independent of the position of the called block. When new code for a block is generated, it is loaded incrementally into the executing program, and linked into the template tree. The new code will be used automatically the next time the block is called.

The template tree is a generalization of the standard way of implementing message sending (virtual procedure calls) in Simula, using a virtual-table per class. The execution overhead is very small; given the static father link of a new block instance, three indirect addressing instructions are needed to find its code and template (StaticFather.Template.SonTemplate(i).Code). This can be compared to two direct addressing instructions in a traditional non-incremental Simula system.

Since the execution overhead is so small, there is no need for separate development and production environments. The executing program and the development system run as separate UNIX processes and communicate via a small kernel embedded in the run-time system. During development, the executing program is normally started via the development environment. However, the program can just as well be run "stand-alone" without the development system. If needed, e.g. at a run-time error, the development system can be attached to the executing program.

More details on the template-tree run-time architecture, and consistency aspects of mixed editing and execution, are given in [HM87].

4.3 Current Implementation Status

Mjølner is implemented in Standard Simula and executes on Sun and Vax workstations. At the time of writing (May 1988), the system is under development. Extensive work remains to be done before it can be used in practice. Editing and incremental semantic analysis supports a substantial subset of Simula. Code generation is currently being integrated into the system. The run-time system is not yet implemented, so at present, programs cannot be executed. However, a prototype run-time system for a toy language has been developed earlier, to evaluate the design [MM85].

In its current state, the system can handle programs of about the same size as the one shown in the examples. There are many rather straight-forward ways of optimizing the system, so we have good hope of being able to handle more realistic programs in a near future.

5. Conclusions

We have presented the user interface of the Mjølner incremental programming environment. The key idea is to focus on the programming language abstractions: classes and procedures. These are presented as nested windows, using a hierarchical window system. An execution is similarly presented as a structure of nested objects and procedure activations. The window hierarchy provides a powerful and natural means of navigation in terms of the abstractions. While navigating and manipulating the program in terms of his abstractions, the user is encouraged to regard the program as a physical structure of nested classes and procedures.

The user interface is object-oriented in the sense that the objects on the screen (windows) can be *identified* with the objects in the program (classes, procedures, objects, activations). This gives the user a sense of directly interacting with the abstractions in the program. Compilation and debugging tools are not explicitly visible to the programmer, but available as functionality in the relevant objects.

The high level of interaction is made possible by incremental compilation techniques. Especially important is the flexible run-time environment which supports incremental updates of the loaded executing program. This allows an unusually close integration of program modification and execution.

Acknowledgements

We are deeply indebted to the rest of our local Mjølner group for taking part in the development and implementation of the ideas presented in this paper: Magnus Taube, Sten Minör, Lars-Ove Dahlin, Anders Gustavsson, Jerker Nilsson, Dan Oscarsson, Mats Bengtsson, and Göran Fries.

The hierarchical window system was implemented by the Norwegian Mjølner group.

We also want to thank Claus Nørgaard and the referees for constructive comments on the paper.

The Mjølner project is partially funded by a grant from the Nordic Fund for Technology and Industrial Development. The Swedish sub-project has an additional grant from the Swedish Board for Technical Development (STU).

References

[BHK87] P. O'Brien, D. Halbert, M. Kilian. The Trellis Programming Environment. OOPSLA '87 Conference Proceedings. SIGPLAN Notices, December 1987. pp 91-102.

[Con87] J. Conklin. Hypertext: An Introduction and Survey. IEEE Computer. Sept. 1987. pp 17-41.

[DLMM86] H.P. Dahle, M. Löfgren, O.L. Madsen, B. Magnusson. Mjølner - A Highly Efficient Programming Environment for Industrial Use, Mjølner Report no. 1. Dept. of Computer Science, Lund Institute of Technology, Sweden, 1986. Also in Proceedings of the 15th Simula Users' Conference, Jersey, 1987.

[DMN72] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula: Common Base Language. Norwegian Computer Center. 1972.

[DMS84] N. Delisle, D. Menicosy, M. Schwartz. Viewing a Programming Environment as a Single Tool. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. SIGPLAN Notices, May 1984. pp 49-56.

[Fri84] P. Fritzson. Preliminary Experience from the DICE system, A Distributed Incremental Compiling Environment. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. SIGPLAN Notices, May 1984. pp 113-123

[Fur86] G. W. Furnas. Generalized Fisheye Views. In Proceedings of ACM SIGCHI Human Factors in Computing Systems Conference. 1986, pp 16-23.

[Gol84] A. Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, 1984.

[HDM87] G. Hedin et. al. Incremental Semantic Analysis in Mjølner. Mjølner Report S-LTH-25.1, Lund Institute of Technology, Sweden. 1987.

[HM86] G. Hedin, B. Magnusson. Incremental Execution in a Programming Environment Based on Compilation. 19th Hawaii International Conference on System Sciences, Honolulu, 1986.

[HM87] G. Hedin, B. Magnusson. Supporting Exploratory Programming in Simula. In Proceedings of the 15th Simula Conference, Jersey, 1987.

[HNRR88] T. Hauge, I. Nordgard, T. Rød, G. Raeder. Gungne, Functional Specification. Mjølner Report, N-EB-4.2, EB Technology, Nesbru, Norway, January 1988.

[KMMN87] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, K. Nygaard. The BETA Programming Language. In B.D. Shriver, P. Wegner (ed.) Research Directions in Object Oriented Programming. MIT Press 1987. pp 7 - 48.

[MBD87] B. Møller-Pederson et. al. Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL. Computer Networks vol. 13, No. 2, 1987, pp. 97-117.

[Mey87] B. Meyer. Reusability: The Case for Object-Oriented Design. In IEEE Software, March 1987.

[MF81] R. Medina-Mora, P. Feiler. An Incremental Programming Environment. IEEE Trans. on Software Eng. Sept 1981. pp 472-482.

[MM85] B. Magnusson, S. Minör. III - an Integrated Interactive Incremental Programming Environment Based on Compilation. ACM SIGSMALL Symposium on Small Systems, May 1985.

[SA86] A. A. diSessa & H. Abelsson, BOXER: a Reconstructible Computational Medium, CACM Sept. 1986, pp 859-868.

[Sch83] B. Schneiderman, Direct Manipulation: A Step Beyond Programming Languages, IEEE Computer, August 1983.

[Sch86] C. Schaffert et al. An Introduction to Trellis/Owl. OOPSLA'86, SIGPLAN Notices, Nov 86.

[SDB84] M. Schwartz, N. Delisle, V. Begwani. Incremental Compilation in Magpie. Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 19, 6, (June 1984).

[Sim87] Data Processing - Programming Languages - SIMULA. Swedish Standard SS 63 61 14. SIS. Stockholm, Sweden, June 1987.

[SIKVH82] D.C. Smith et al., Designing the Star User Interface, Byte Magazine, April 1982.

[Str86] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1986

[Tes81] L. Tesler, The Smalltalk Environment, Byte August 1981, pp. 90-147.

[THM87] M. Taube et. al.,The Mjølner Observation Tool. Proceedings of the 15th Simula Users' Conference, Jersey, 1987.

[TR81] T. Teitelbaum, T. Reps. The Cornell Program Synthesizer: a Syntax-Directed Programming Environment. CACM Sept. 1981. pp 563-573.

[Wil84] G. Williams. The Apple Macintosh Computer, Byte Magazine. February 1984. 30-54.