SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(4), 371–391 (APRIL 1989)

The ECLIPSE User Interface

IAN SOMMERVILLE
Department of Computing, University of Lancaster, Bailrigg, Lancaster LA1 4YR, U.K.

RAY WELLAND
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, U.K.

STUART POTTER
Software Sciences Ltd., Deanway Technology Centre, Wilmslow Road, Handforth, Wilmslow, Cheshire SK9 3ET, U.K.

AND

JOHN SMART
CAP Group Ltd., 40–44 Coombe Road, New Malden, Surrey KT3 4QF, U.K.

SUMMARY

This paper describes the user interface facilities of the ECLIPSE integrated project support environment. This interface is based on a consistent metaphor called the 'control panel' metaphor and includes standard help and message-handling systems. The paper describes these as well as some of the interface standards which have been developed. The interface has been implemented on top of the 'applications interface', which provides a portable, hardware-independent interface for software tools.

KEY WORDS Software engineering environment User interface ECLIPSE Help system

INTRODUCTION

ECLIPSE is a large-scale integrated project support environment (or IPSE) which has been built in the U.K. by a consortium of universities and systems companies. It is a database-centred environment with the database provided by PCTE (portable common tool environment), an emerging European standard on which environments can be built. ECLIPSE is intended to be a general-purpose environment which may be tailored to support a variety of application domains, but the initial versions of the system will concentrate on supporting real-time systems development using MASCOT and Ada and data processing systems development using a method called LSDM.

An early design decision made in the ECLIPSE project was that integration should not simply be at the database level but that the user interface (UI) should also present an integrated system view. As ECLIPSE IPSE products are intended for release in the late 1980s, it was decided that the user interface should be tailored exclusively to bit-mapped high-resolution workstations. This decision allowed us to assume that each engineer's workstation would have significant local processing power and meant that

0038–0644/89/040371–21$10.50 *Received 7 December 1987*

we could design an interface which is based on graphics rather than on command language interaction.

When this work began in 1984, no other IPSEs had been developed, and bit-mapped workstations were only just becoming available. There was no directly relevant literature on which we could base our work. However, concurrent with the work described here, the ASPECT IPSE[1] adopted a different, more fundamental approach to user interface construction.[2] Rather than build on top of existing facilities (and therefore live with their limitations), the ASPECT project developed a complete user-interface system from scratch, including a window manager. This allowed a very consistent model of user interaction to be presented. This approach was not adopted in ECLIPSE partly because we felt that the practical advantages (cost, reduced risk, portability) to be gained by reusing existing software (such as the workstation window management system) outweighed the limitations (lack of consistency and predictability) this sometimes imposed.

The lack of directly relevant work in this application domain also posed difficulties when we came to consider fundamentals of cognitive science which might be relevant to our work. Although work by Price and Cordova[3] on the use of mouse buttons, Tullis[4] on menu design and Marcus[5] on screen design was taken into account, those cognitive scientists with whom we discussed the general problem had difficulty in understanding the concept of an IPSE, and ultimately presented advice which was so contradictory as to be useless. However, more recent work (not available to us when ECLIPSE was designed) such as that described by Dix *et al.*[6] may be more relevant to environment-interface design, as these workers are considering this application domain.

Unlike development environments such as Interlisp,[7] Cedar[8] or Smalltalk,[9] ECLIPSE is an open environment and is not integrated around a single programming language. The tight tool integration between language and support tools offered in single-language environments could not be provided. Rather, we adopted an approach which was pioneered in the Xerox Star system and made generally known in the Apple Macintosh, whereby all tools adopted a common interface style and made use of a set of standard routines to create their user interface. A detailed description of the approach adopted, called Control Panels, is given by Reid and Welland.[10]

As part of the provision of a consistent framework of interaction, we decided that all messages, including help texts, from the environment to the user should be presented in a similar fashion. This can be accomplished by defining standards, but we decided that a more reliable and cost-effective approach was to provide a standard messaging system which is used by all ECLIPSE tools. In practice, we discovered that help texts are quite different from other messages, so we have built a separate handler for help texts and have a mandatory requirement that all tools should provide help frames describing their use.

This paper is exclusively concerned with a general presentation of the ECLIPSE user interface and does not describe specific tools developed during the ECLIPSE project. We present a brief overview of ECLIPSE and describe the user interface metaphor which is a standard approach for interacting with all ECLIPSE tools. The interrelated help and message systems are then described and, finally, we discuss our approach to implementation.

The initial prototype implementation of the ECLIPSE UI was as a set of C programs which interacted directly with the facilities provided on Sun workstations. However,

the decision to base ECLIPSE on PCTE, which provides its own set of UI primitives, and the commercial requirement of portability meant that we needed an alternative approach which was less closely tied to Sun library software.

Thus we have based the final UI implementations on what we call the 'applications interface'. This is an interface layer between ECLIPSE applications and the UI primitives provided by the workstation. In ECLIPSE, the applications interface has been mapped onto the underlying SunView primitives and, at the time of writing, an implementation using the PCTE UI facilities is under way.

This paper describes a development project rather than a research project. Our objective was to build an interface to a very complex system which helped the user to master that complexity; we did not set out to carry out fundamental research in UI design nor to build a generalized user-interface management system. All of the software described here has been written in C, is currently operational on Sun workstations and is now shipped as part of the ECLIPSE IPSE product marketed by the industrial collaborators on the ECLIPSE project.

THE ECLIPSE ENVIRONMENT

ECLIPSE is a software engineering environment which has been developed in the U.K. It is a large-scale project (150+ person-years) which has been implemented by three systems companies and three universities with funding support from the Alvey Directorate. The Alvey Directorate manages the U.K.'s national research programmes in software engineering, VLSI, artificial intelligence and user-interface design.

ECLIPSE runs on Sun workstations and provides an object-management system built on top of PCTE along with a number of tools which are geared towards the support of software design. The system provides access to fine-grain data by making use of a two-tier database.[11] The structure of ECLIPSE is illustrated in Figure 1.

The tools which are available in the initial version of ECLIPSE include a generic design-editing system which may be tailored to support any design method based on diagrammatic techniques,[12] an Ada cross-compilation system for Intel 286 processors

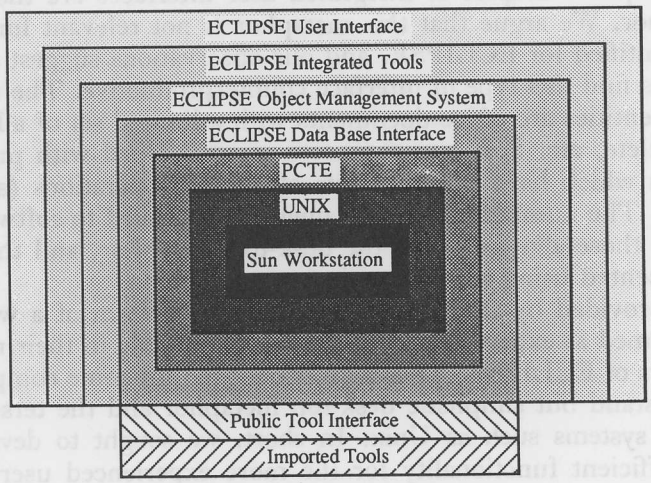

Figure 1. The structure of ECLIPSE

and associated host tools to support that Ada system,[13] a design-support system for MASCOT and a design support system for LSDM. MASCOT[14] is a design method for real-time systems which is widely used in the European defence community. LSDM is a development of SSADM[15] which has been adopted as a U.K. Government standard and which is used, fairly widely, in European and in some U.S. organizations.

As well as the work described here, other work in the general area of user interaction with software engineering environments has been carried out as part of the ECLIPSE project. This has involved investigating the use of iconic interfaces to complex environments and developing a generic graphical editing system. However, as this work is peripheral to the main UI programme, we have not covered it here. An overview of the ECLIPSE system is given by Alderson *et al.*[16] and a comprehensive description by Bott.[17]

ECLIPSE is built on top of the portable common tool environment, PCTE,[18, 19] which was developed by a consortium of European countries as a common basis for environments. PCTE provides an object-management system based on an entity relationship model that manages a database distributed over a network of workstations. It also provides primitives for user-interface support and process management. It is compatible with Unix System V so that Unix applications may run without change under PCTE.

METAPHORS AND STANDARDS

In designing an integrated interface to ECLIPSE there were two distinct problems to consider. How should the interfaces of individual tools be represented and what should the overall environment in which ECLIPSE tools execute look like? Systems such as the Apple Macintosh have an instantly recognizable 'house style' which is used for all tools. An application has a menu bar across the top, giving access to pull-down menus, each window used by the application has a title bar, which can be used for moving or closing the window, scroll bars are standardized, etc. We wished to design a similarly recognizable house style for ECLIPSE and to ensure that tools behaved consistently during common operations such as start-up and shut-down.

The best developed examples of integrated user interfaces are those based on the 'desk-top' metaphor. We argue that this metaphor is not relevant for the majority of the user roles identified for ECLIPSE, and our observations suggest that experienced software engineers find this type of interface counterproductive. The reason for this is that its principal entities are office documents with a limited set of allowed operations such as move, delete, etc. Software engineers are concerned with programs, designs and specifications which have different sets of allowed operations (such as compile, link, check, etc.). The number of possible operations applied to software is, typically, much larger than those allowed using the desk-top metaphor, and the operations are not readily represented using simple iconic representations.

The facilities provided by the window-management system of a workstation, such as the Sun, are aimed at expert users. They are not suitable in their raw form for the less technical users of ECLIPSE. Therefore, we looked for some compromise between the easy-to-understand but simplistic desk-top metaphor and the terse command language offered by systems such as Unix. In short, we sought to develop a graphical interface with sufficient functionality for the more experienced user. The approach which was finally adopted is termed the 'control panel' metaphor.

Control panels

The concept of control panels was introduced to provide the unifying metaphor for the ECLIPSE UI. The objective is to stimulate the control panels which are available with many complex pieces of equipment, to provide both status information and controls. Analogous concepts are provided within ECLIPSE software control panels. One lesson to be learned from hardware control panels was to avoid information overload, the extreme example of which is probably an aircraft cockpit. Fortunately, much of the information can be hidden using menus, thus avoiding having a screen full of control and status information.

We identified five basic types of objects which are sufficient to provide the necessary functionality in our control panels.

1. *Button*. A button is an object which when picked ('pressed') always initiates a single action, such as screen dump. Every time a button is picked it initiates the same action, its effect is not context dependent.
2. *Menu*. A menu displays a list of objects one of which can be chosen by the user to initiate some action; a typical use is command selection. The contents of menus may depend on the context in which the user is working.
3. *State selector*. A state selector is a composite symbol consisting of a menu and a value. The value displays the current state and the menu provides a mechanism for changing that state. An example of the use of a state selector is in representing multiple modes; the value part of the state selector shows the current mode; this value can be changed by selecting the new value from the associated menu.
4. *Sign*. A sign is a two-part object which has a fixed title and a value. The value part may be static, dynamic (changed by the system) or user-changeable. A static sign might be used to display the name of the current user, whereas a dynamic sign could be used for the display of error messages. A user-changeable sign might be used in an editor to display the name of the file being edited. To save work to a different file, the user simply overtypes the old file name in the sign with the new.
5. *Light*. A light is an indicator with a binary status, either on or off. Lights are commonly used to indicate whether a window is current, that is, the window to which the keyboard is attached, or a process is busy.

Once we had identified these basic objects, we designed a set of icons to represent them. The set of shapes shown in Figure 2 is the result of experimenting with different possibilities. As the project has progressed various minor changes have been made to these icons. The guiding principles for designing the shapes were that they needed to be easily distinguishable and sufficiently regular that they could easily be fitted into a reasonable area and could contain text labels. Although we accept that the icons chosen may be less than ideal from a graphical design point of view, the overriding pragmatic consideration was the need to make most effective use of scarce screen space. The user-changeable sign labelled 'Title' in Figure 2 includes the visual cue '->' to indicate that the user may provide an input by typing into the value part of it.

Provision is made for a two-state light to reassure the user that the system has not crashed when a long operation is taking place. Tools should arrange for the light to 'flash' during time-consuming operations to indicate to the user that the system is still active.

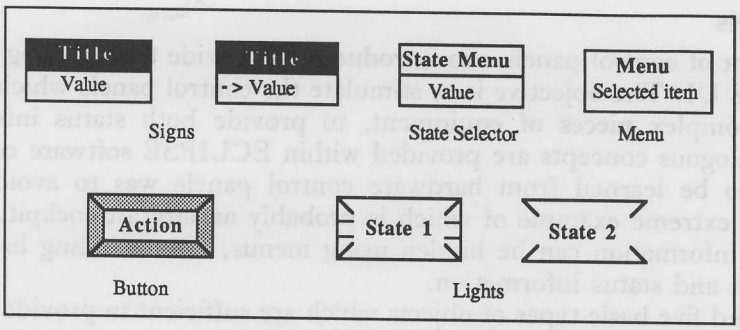

Figure 2. Control panel object representations

The ECLIPSE UI uses the concept of a control panel as the principal means of controlling the major functions of both ECLIPSE as a whole and individual applications, such as a design editor[12] that run under ECLIPSE. However, the control-panel metaphor, like most metaphors, is not the answer to all problems. The control panel is primarily of use for selecting options, setting contextual information, invoking infrequently used major actions and invoking critical actions. Frequently used commands that cannot be easily grouped are not sensible contenders for panel buttons since the user would spend a great deal of time 'mousing' around the control panel.

For example, consider a text editor. The general movement and editing commands within the text should be accessed via some other mechanism, such as keyboard commands, function keys etc. Conversely, file selection, editing mode, display mode etc. would be suitable actions for invocation via the control panel using, for example, menus or buttons. Similarly, actions such as invoking the help system, closing the window, aborting or terminating the application should be provided using control panel facilities.

Control panel layout standards

Once we had settled on a set of concepts and symbols we needed to ensure consistent use of these objects by defining control panel layout conventions. It was in this area that fundamental work such as that of Marcus,[5] Price and Cordova[3] and Engel[20] was most useful. Based on this work, we formulated a number of outline standards regarding the design of menus and the use of buttons in ECLIPSE.

1. Menu options should be centred on a block basis rather than on a per-line basis.
2. The ordering of options within a menu should reflect a logical grouping appropriate to the task at hand.
3. Frequently used or safe actions should be placed at the top of a menu, less frequently used or unsafe actions should be placed at the bottom.
4. ECLIPSE buttons are provided for both unsafe actions, such as 'exit', and actions which do not fall naturally into a logical grouping, such as 'help'.
5. A user selection from a menu should be confirmed visually by inversion.

To achieve visual consistency we recommend relative positions for the various objects within the control panel. Signs should appear to the left of the control panel, followed by state selectors and menus, then buttons and finally lights to the right of the control

panel. However, the designers of control panels often found good reasons (or excuses!) to vary the positions of the objects. An example of a typical control panel taken from the ECLIPSE help system is shown in Figure 6.

The details of application control panels vary, but certain objects appear in nearly all control panels. For example, an 'Exit' button should always be provided and every tool should use this button in a consistent way. The implementation of exit is obviously application-dependent but confirmation of exit without updating is always required.

In general, confirmation of destructive actions must be provided via a pop-up selector containing visual targets. The action must be confirmed by positioning the cursor over a target and clicking a mouse button. The wording of the targets must be unambiguous and they must be far enough apart to prevent accidentally hitting the wrong target.

The Sun software automatically includes a 'name stripe' at the top of every window created; it cannot be removed without modifying the window manager, therefore we decided to standardize its use. In ECLIPSE, the name stripe is used to display the name of the component associated with the window and the component version details.

Screen organization standards

At the beginning of the project we attempted to define a general screen organization, suitable for a variety of applications. Our objective was to establish a convention for screen organization so that different tools presented a consistent window interface to the user. Thus we hoped to recommend window sizes and layouts but toolmakers or end-users could change these if warranted by particular circumstances.

In principle, the workstation's 19 inch screen was capable of holding two windows side by side, where each window is the size of a normal (A4) piece of paper. We thought it desirable to be able to view two large windows simultaneously, as well as smaller windows, but the limited screen size meant that a double A4 display left no room for console windows, status displays or 'parking' of icons.

Experiments were carried out using a number of different founts available on the Sun to try and achieve a compromise between a comfortable fount size and a usable display area. The recommended aspect ratio for windows is that the sides should be in the ratio one to root two or 'golden rectangle'.[21] An attempt to define landscape windows to fit this ratio, in the same orientation as the screen, led to windows which were either too shallow to be usable or so large that only one would fit on the screen!

Our preferred layout was to have two portrait windows, in the recommended aspect ratio, juxtaposed, but the only fount which would allow this was too small to read. Therefore, a readable fount was chosen and windows defined to be the maximum length which can be obtained, producing an aspect ratio of about 1 to 1·23. These window dimensions are intended as guidelines for applications but some ECLIPSE tools, especially those using graphics, require a larger working area.

Initially, we sought to avoid overlapping windows for the reasons set out by Cohen *et al.*[22] Overlapping windows can be awkward to manage, waste screen space and difficult to find. Overlapping windows tend to increase the memory load on users and require them to be familiar with the window manipulation facilities. A tiled solution was considered, but our need for large windows (particularly for editing) made us reject this approach. In practice, users of ECLIPSE use overlapping windows extensively because many applications seem to need larger windows than can comfortably be accommodated in half of the workstation screen.

In summary, we failed to establish a workable standard for screen layout which would be applicable to all ECLIPSE users. Given that screen sizes are unlikely to increase significantly (because of cost and the space occupied), we believe that it is probably impossible to achieve an acceptable standard where the user community is experienced and diverse and where the applications require large window sizes.

THE MESSAGE SYSTEM

As well as the information displayed as part of the normal running of an ECLIPSE tool, there are occasions when a tool wishes to output messages to the user for information, errors, diagnostics etc. The ECLIPSE message system provides the tool builder with the software to handle these messages and provides the user with some ability to choose how and when these messages are displayed.

One of the characteristics of a project support environment like ECLIPSE, is the wide variety of potential users and their knowledge and skills. A great deal of thought was given to how a message system could accommodate this variety and still maintain a consistent approach. Other systems have adopted a number of approaches in an attempt to solve this problem, including classifying users on skill level such as novice, experienced, expert, or by layering messages so that the user can repeatedly ask for more levels of detail. It is very difficult to tune this sort of system as a user's skill level may be much higher in one task than another and he or she may get frustrated with messages which are too terse or verbose.

The method chosen for ECLIPSE involved three main design decisions:

1. Separate the message texts from the code.
2. Structure the message texts.
3. Link messages into the help system.

The first is fundamental and involves grouping the message texts into message sets. These sets are not built into a tool but are read at run-time. This allows alternative message sets to be provided without rebuilding the software. Thus message sets in languages such as French or German can be produced or special, more explanatory messages created for training purposes.

The objective of the message is to be positive and guiding rather than negative and accusatory so each message contains a brief summary of the problem and a suggestion of what to do about it. The experienced user who recognizes the problem may read no further than the first few words whereas the novice, or indeed the expert encountering a message for the first time, may read the whole message.

The structure of the message is equally important. The messages are single lines allowing parameter substitution so that run-time specific information, such as the name of a database object, can be incorporated. For example, a message in the system might be:

Interaction: "Writing to file $1 is not allowed; Check file protection":
Help-> Write protection.

This specifies the message type (Interaction), the message text where the parameter $1 can be substituted with the file name which the user has tried to access, and a link to the help system.

If the message itself is not sufficiently explanatory, the user can call for help. Each message can be linked to a frame of help information as described in the following section. To display the relevant frame the user simply selects the 'Help' button, which is available from every ECLIPSE tool. The user then has access to all the help frames relevant to the current context plus as much detail as has been stored in the help database. Displaying this information in a separate window of the workstation enables the help text to be viewed alongside the problem.

The message system and the help system are closely related. Their relationship is illustrated in Figure 3.

Message types

All ECLIPSE messages are typed and the tool builder decides at which points in the code a message should be generated. Generating a message simply involves specifying a message identifier plus any substitution parameters and a message type.

Message types fall into three classes:

1. *Interaction messages*. These are messages which provide details of the status of an ECLIPSE interaction. The interaction message types are *information, warning, error* and *fatal*. Messages of this type are generated by tools for users, although it must be emphasized that there is no need to indicate the message type to the tool user.
2. *Input messages*. These are messages of type *comment* which allow users to provide feedback to tool builders.
3. *System messages*. These are messages which provide system information and which are generated by tools for tool builders and system administrators. The types in this class are *metric, diagnostic, tracking, console, security* and *system*.

Comments are a way of capturing users' observations about ECLIPSE and any of its tools. During any session, a user is able to record comments about any problems encountered or about any feature of the system which he or she likes or dislikes. Comment messages are logged for subsequent analysis by tool builders.

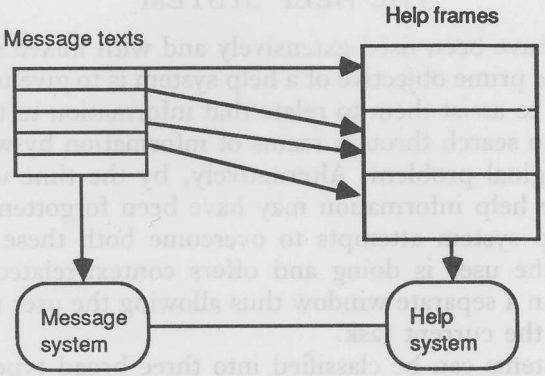

Figure 3. The help and messaging systems

Having assigned the type, the tool generating the message calls the message system, which extracts the required message text, prepends the type descriptor and routes it to a number of possible destinations depending upon the type of the message.

Message routeing and logging

One of the most irritating features of some software systems is the generation of unwanted messages which interrupt a user's work. One of the design objectives of the ECLIPSE message system was to allow users to control the messages generated by ECLIPSE tools.

Each ECLIPSE user has an associated message log in which all generated messages may be retained. This facility was included to allow an evaluation of ECLIPSE usage and to record, in an unambiguous way, problems which users encountered with ECLIPSE software. Messages may be directed to the log as well as the user's screen. In fact, the message system recognizes three possible destinations:

(i) the user's current window
(ii) the console
(iii) the user's message log.

The destination of each message type is governed by a routeing table which allows users to change the destination of messages between sessions or between tool invocations. The table assigns to each message type zero or more of the possible three routes (to ignore a class of messages, zero routes are specified). For example, a user might wish to suppress all information and warning messages but see errors. He or she might choose to display fatal messages (messages which result from an unrecoverable error) in the current window and on the console as well as recording them in the message log. All other types might be routed to the message log for later analysis. Any messages logged are time-stamped and include extra information about the tool involved.

Many of the system functions in ECLIPSE use the message log to record events such as logging on and off, tool invocation and deletion, hence providing an event log for a users' ECLIPSE session. The log is simply a text file and can be processed by any of the text handling utilities to produce reports or statistical information.

THE HELP SYSTEM

On-line help systems have been used extensively and with mixed success in a number of environments.[23] The prime objective of a help system is to give users the information they need quickly and to assist them to relate that information to the task in hand. All too often, users have to search through reams of information by which time they may have forgotten the original problem. Alternatively, by the time users return to their task, the details of the help information may have been forgotten!

The ECLIPSE help system attempts to overcome both these problems. First, it keeps track of what the user is doing and offers context-related help; secondly, it displays the help text in a separate window thus allowing the user to compare that text with the output from the current task.

In general, help systems can be classified into three broad types:

(i) tutorials
(ii) on-line manuals or reference volumes
(iii) memory job/contextual help.

The ECLIPSE help system concentrates on providing context help together with a general browsing facility of available functionality. No attempt is made to provide an interactive tutorial system.

It has been recognized that ECLIPSE will include a growing number of tools which may be developed explicitly or may be brought in as foreign tools. The foreign tools may have their own help systems, and the ECLIPSE help system is designed to enable these 'alien' systems to be, at least partly, integrated. As an example, ECLIPSE includes tools which are documented by UNIX manual pages, and the help system provides a hook facility by which the relevant manual pages can be collected and displayed.

The help information architecture

It is important to structure help information so that users can see the level of detail appropriate to their current context and, at the same time, have the freedom to navigate around the complete help database. The help database is held as part of the ECLIPSE database and is organized to reflect the way ECLIPSE software is structured.

For each ECLIPSE tool, a help frame-set is produced to describe that tool. The unit of help is the frame, and each frame has a name and title plus a page of information. The frame-set has a hierarchic structure in which progressively more detail is provided as the user descends the hierarchy. Links can also be added to refer to related frames, both within the current frame-set and to frames in other frame-sets. The result is a network database where frames are logically grouped with tools or components but where it is possible to interlink the groups as desired. The help system then provides the access points to this database plus the tools to navigate it. This is illustrated in Figure 4.

Figure 4 is a simplified diagram of part of the help frame network. It shows two frame sets, one for editing in general and another for editing software designs. By following links, the user may easily move between these. Of course, the practical link structure is much more complex than that shown in Figure 4.

As a software tool is used, the user's context within that tool changes as different phases or levels of processing are accessed. At each change, the tool records the help context by maintaining a stack of references to help frames in the database. Each ECLIPSE tool has a 'Help' button and, if this button is pressed, the context stack is

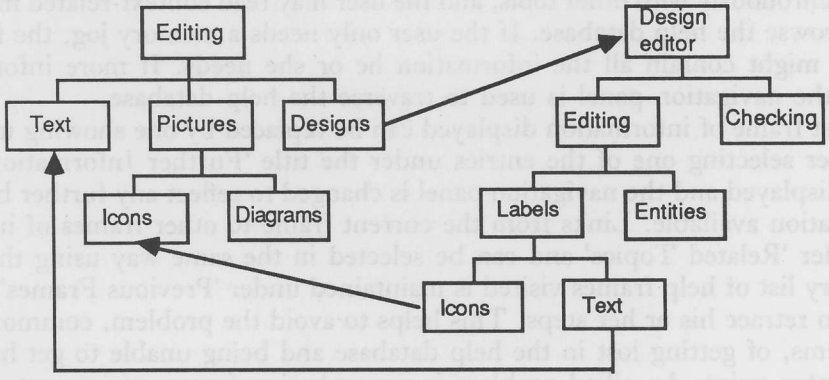

Figure 4. The help information architecture

transferred to the help tool, which retrieves and displays the topmost frame on the stack and stores the other frame references for subsequent viewing.

Figure 5 shows the stack structure where the user has invoked a tool (the design editor), chosen a particular step in the tool (select text) and then encountered an error. If, at this point, the 'Help' button is pressed, the stack contains three help frames, the frame describing the error, the frame describing text selection and the high level frame for the design editor. Initially, the frame showing the information most relevant to the user's current context (the error) is displayed. If this information is inadequate the user can then call up further frames for display as required.

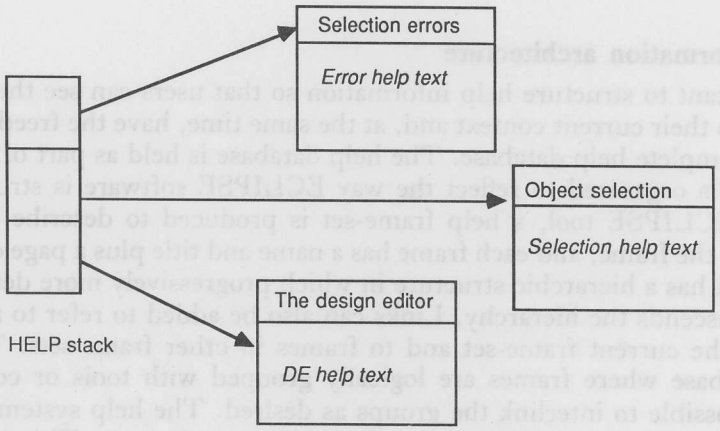

Figure 5. The HELP stack structure

Help display and navigation

The help tool displays help frames in a separate window as illustrated in Figure 6. The top two panels provide control facilities for the help tool, whereas the lower two contain the page of help information and the primary navigation facilities.

The help tool is normally invoked from another tool with relevant context as described above but it can also be invoked by command with suitable parameters to identify the required access point into the help database. In each case, the help tool runs asynchronously with other tools, and the user may read context-related information or may browse the help database. If the user only needs a memory jog, the first frame displayed might contain all the information he or she needs. If more information is required the navigation panel is used to traverse the help database.

The first frame of information displayed can be replaced by one showing more detail by the user selecting one of the entries under the title 'Further Information'. A new frame is displayed and the navigation panel is changed to reflect any further breakdown of information available. Links from the current frame to other frames of interest are listed under 'Related Topics' and can be selected in the same way using the mouse.

A history list of help frames visited is maintained under 'Previous Frames', enabling the user to retrace his or her steps. This helps to avoid the problem, common to many help systems, of getting lost in the help database and being unable to get back to the original entry point. An allied problem is remembering frames of interest, and users sometimes have to note frames to which they may wish to return. To avoid the chore

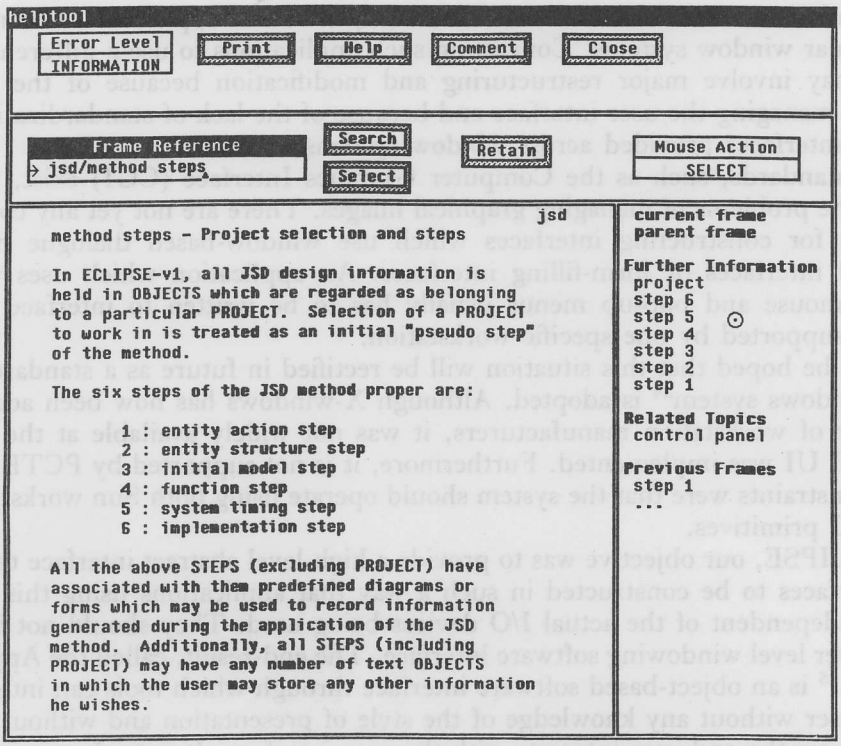

Figure 6. The ECLIPSE help tool

of writing down frame references, the help system provides a marking facility whereby the user flags (or marks) a frame as being of interest. The system then maintains this list which is displayed in the navigation panel under 'Marked Frames'.

If the user is browsing, select and search facilities on the help database are available through the lower of the two control panels. If the name of a frame is known, this is typed into the sign entitled 'Frame Reference', and 'Select' is invoked. That frame is retrieved and displayed. If help is needed on a particular topic, a topic keyword is typed into the sign, and 'search' is invoked. This compares the string against frame names and titles in the database and each one containing the keyword will be recorded. If a single match is found, that frame will be displayed. If more than one match is found, their names and titles will be shown and the user may select one for display or may mark a number of them for subsequent viewing.

The top control panel enables users to suppress certain types of message if they wish ('Error Level'). 'Print' allows a hard copy of the currently displayed frame to be made and, if necessary, the user may seek help ('Help') on the help system. The comment facility ('Comment') allows user remarks to be passed to tool builders and 'Close' closes the window to an icon.

THE APPLICATIONS INTERFACE

Although various window-based systems can present comparable visual interfaces and styles of interaction to the end-user, the underlying interfaces and mechanisms available

to applications are different. This has the consequence that applications become bound to particular window systems. Converting such applications to use a different window system may involve major restructuring and modification because of the pervasive nature of managing the user interface and because of the lack of standardization in the software interfaces provided across window systems.

Some standards, such as the Computer Graphics Interface (CGI) exist, but these address the problems of managing graphical images. There are not yet any comparable standards for constructing interfaces which use window-based dialogue managers, command interfaces or form-filling interfaces. An application which uses windows, icons, a mouse and pop-up menus usually has to be written to interface with the facilities supported by one specific workstation.

It is to be hoped that this situation will be rectified in future as a standard such as the X-windows system[24] is adopted. Although X-windows has now been accepted by a number of workstation manufacturers, it was not widely available at the time the ECLIPSE UI was implemented. Furthermore, it is not supported by PCTE, and our design constraints were that the system should operate using both Sun workstation and PCTE UI primitives.

In ECLIPSE, our objective was to provide a high-level abstract interface that allows user interfaces to be constructed in such a way that applications using this interface remain independent of the actual I/O devices being used. They should not be bound to the lower level windowing software interface. The end-result, called the Applications Interface,[25] is an object-based software interface through which tools can interact with the end-user without any knowledge of the style of presentation and without knowing in detail how the end-user interacts with the user interface. It provides an implementation of control panels but is actually of broader utility and has been used in other software products.

The Applications Interface is implemented by providing a textual description of the interface objects and operations using a description language called FDL (frame description language) and by providing an abstract procedural interface for applications. Tools interact with the user via this procedural interface which is mapped onto the underlying workstation software.

The abstract user interface

The basis of the high-level abstract user interface supported by the Applications Interface is a hierarchy of objects representing the various classes of images that a tool may need to construct its user interface. There are five levels to the hierarchy, denoted as the screen, windows, frames, panes and fields. At the frame and field levels there are several different classes of object, as shown in Figure 7. The object at the root of the hierarchy, which cannot have a parent, is called the screen and represents the whole output area available on a workstation. Within the screen, any number of window objects can be defined although only one of these can be completely visible at any one time. This restriction has been imposed in order to enhance the portability of the user interface.

Each window can be subdivided into a set of areas called frames, of which there are four different classes which support different functional uses of their areas:

1. A graphic frame supports the use of graphical primitives for drawing diagrams and pictures.

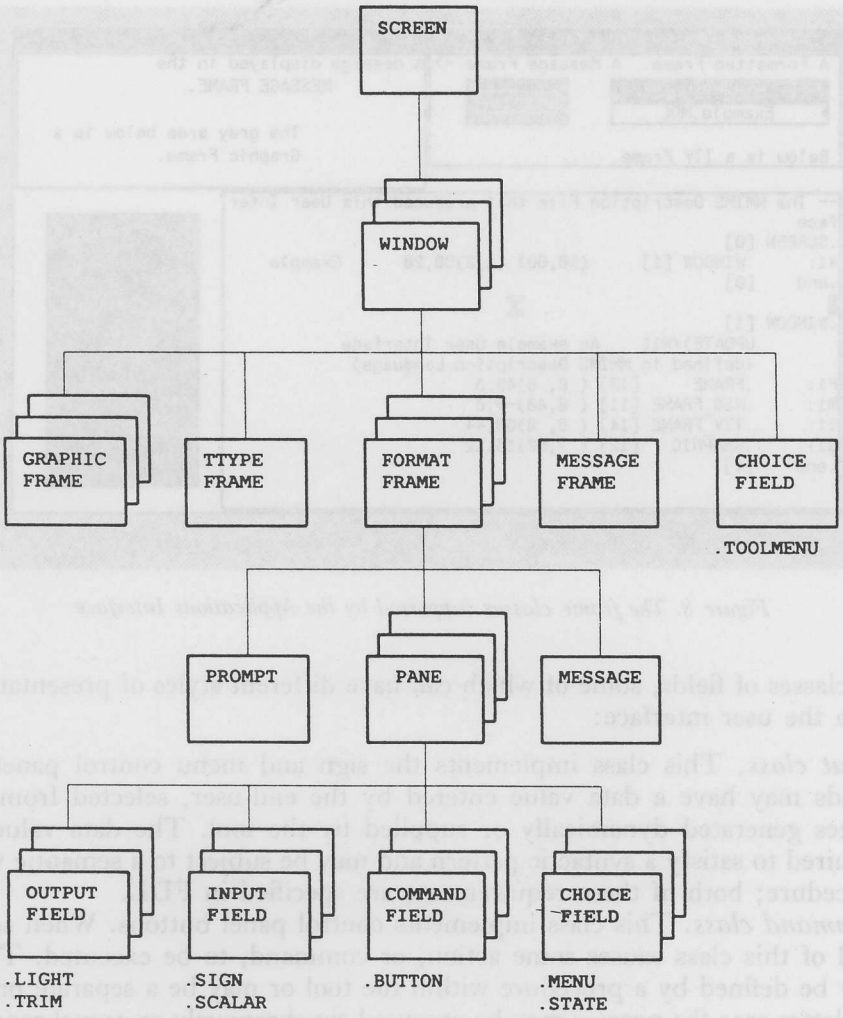

Figure 7. The hierarchy of user interface objects

2. A TTY frame emulates a normal character terminal so that tools which interface to a character terminal can run unchanged on a workstation.
3. A formatted frame supports the control panel mode of tool interaction as discussed earlier.
4. A message frame provides a textual output area used for all messages generated by tools using this interface.

An example of a window containing one of each of the above frame classes is given in Figure 8. This example also shows the FDL text which is used to generate the interface.

Only a formatted frame can be subdivided. These subdivisions are called panes, and consist of a set of fields which a tool wishes to manage as a group.

Fields are the lowest level objects in the hierarchy and are generalizations of the elements of a control panel through which a tool can interact with an end-user. There

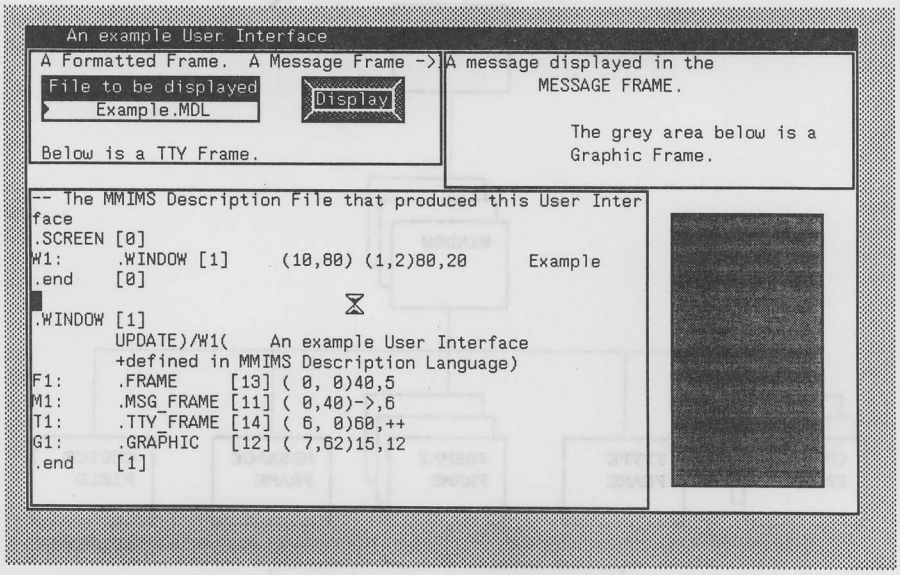

Figure 8. The frame classes supported by the Applications Interface

are four classes of fields, some of which can have different styles of presentation when visible on the user interface:

1. *Input class*. This class implements the sign and menu control panel objects. Fields may have a data value entered by the end-user, selected from a list of values generated dynamically or supplied by the tool. The data value may be required to satisfy a syntactic pattern and may be subject to a semantic validation procedure; both of these requirements are specified in FDL.
2. *Command class*. This class implements control panel buttons. When selected, a field of this class causes some action, or command, to be executed. The action may be defined by a procedure within the tool or may be a separate process. In the latter case the process may be executed synchronously or asynchronously and may use the TTY frame as its I/O device. The action, its context and messages, are all specified in FDL.
3. *Choice class*. This class implements the state selector control panel object. Fields have a fixed list of possible values, one of which may be chosen by the end-user or the tool. The list of possible values is specified in FDL and is displayed as a pop-up menu. The former style is used to define a state or context value, whereas the latter is used to provide a list of actions from which to chose.
4. *Output class*. This class implements control panel lights. Fields of this class may be used to display fixed test and graphical images on the user interface as commentary to assist the end-user.

Each object is defined to represent two rectangular areas which are always constrained within the object's parent in the hierarchy. Each rectangular area has a position and extent relative to its parent and is specified in character units to avoid dependence upon the pixel resolution of any actual workstation. This spatial relationship determines where each object will appear when it becomes part of the visible user image.

The frame description language

In order to provide a machine-independent way of describing user interfaces, we have developed a notation, called FDL (frame description language) which is used to set out a textual description of the interface. This description is interpreted at run-time by the interface management system to generate the actual interface displayed on the user's workstation.

The relationships between the tool, the FDL interpreter and the FDL description are shown in Figure 9.

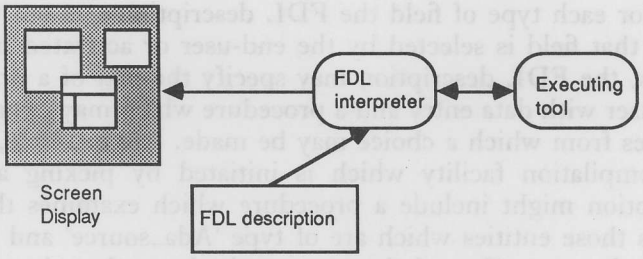

Figure 9. FDL and tool interaction

We have deliberately excluded a detailed description of FDL here as it can be considered as the 'object code' driving the applications interface. As discussed later, neither tools nor interface designers need have detailed FDL knowledge.

FDL is used to define the hierarchy of objects that are required by a tool, together with their relative positions and extents. The positions and sizes of the objects are not available to the tool which uses the FDL, so that the layout of the objects may be changed without affecting the logic and code of the tool. The FDL definition of the objects may also specify the initial values of an object's attributes, such as its visibility and selectability.

The text which labels an interface object must be specified using FDL since it may not be defined by the tool. The label part of a field allows that field to be selected by the end-user typing the label name; if text labels are ambiguous the end-user has to type a sufficient number of characters to identify the required field uniquely. Field selection using a mouse is controlled entirely by the Applications Interface; the tool has no knowledge of the mechanisms used and is not concerned with the input devices available to the end-user.

The text associated with an object may be changed in the FDL without any effect on the tool. The textual values of the choices in a fixed list must be specified using the FDL; each choice has an integer value specified with it in the FDL definition, and this integer is used by tools to reference that choice. This approach allows the order and textual values of a choice list to be changed without affecting the logic or code of the tool.

In order to decouple a tool from the FDL specification of its user interface, each interface object has a name and a set of allowed operations associated with it. Objects may be referenced with this name by the tool, and by operations to access its attributes. Thus, tools have a completely procedural interface to an FDL specification and need not be aware of any details of the interface description language.

Dynamic control of the user interface

At any one time in a system interaction only a subset of all possible interface operations are valid. If the user attempts an invalid operation, the conventional response is to generate an error message and allow the user to retry. However, a better approach is to modify the interface dynamically so that only valid operations are presented to the user. The interpretative implementation of FDL makes such dynamic interface modifications possible.

As well as allowing the position and extent of user interface objects to be described, FDL allows the dynamic response of the user interface to end-user and tool operations to be specified. For each type of field the FDL description sets out the actions to be carried out when that field is selected by the end-user or activated by a tool.

For input fields, the FDL description may specify the text of a prompt, a message to assist the end-user with data entry and a procedure which may dynamically generate a list of data values from which a choice may be made. For example, say an interface offers an Ada compilation facility which is initiated by picking an 'Ada' button. The FDL description might include a procedure which examines the user's current workspace, selects those entities which are of type 'Ada_source' and presents a menu of their names to the user. One of these names is then selected as the entity to be compiled.

The FDL description may also include details of a syntactic pattern which the input value must match, a procedure (or process) which can validate the input value, an error message which is displayed if validation fails and a set of actions to be obeyed when the field's value is successfully updated. These actions can include changing the values of the attributes of any of the user interface objects, including the visiblity and selectability attributes. Thus, if an operation is invalid at some point in a user interaction, selection of that operation in the interface may be disabled.

An example of where this facility is used is in the ECLIPSE MASCOT design editor. Once an entity has been registered in a design database its semantics may not be changed, although its position on a design diagram may be modified. Thus, once registered, the operations available to the editor user are restricted so that destructive operations are simply not presented in the editor operations menu.

For command fields the FDL can be used to specify the procedure (or process) that provides the required operation, a message to be displayed before the command is obeyed, whether the end-user is to confirm the action before the command is obeyed, the context in which to obey the command, an error message to be displayed if the command fails and a set of actions to be obeyed when the command is successful.

The user interface generator

FDL is a powerful notation for specifying interface objects and actions but is a fairly low-level notation, and consequently interface definitions take some time to write and to debug (see the FDL example in Figure 8). This is a particular problem when designing interfaces, as this activity is one which requires a great deal of experiment. Interfaces are generated, tested and then modified during development, and we discovered that writing FDL code was an expensive business.

To speed up the process of interface generation, we have developed an interactive interface construction tool.[26] Given graphical representations of the basic interface object classes, the interface builder can simply pick these and position them as required

on an interface template. There is no need for the user to know the details of the underlying language, indeed, the interface designer need not even know that such a language exists.

The tool which we have devised (Figure 10) provides the interface designer with the ability to design ECLIPSE control panels. Basic control panel items (buttons, menus, signs, etc.) may be selected and positioned on a display. The advantages offered by this approach are

1. The designer is given assistance in laying out the interface.
2. End-users can become involved in interface design.
3. The time needed to construct new interfaces is reduced dramatically. The difference is perhaps the difference between carrying out computations using a spreadsheet and writing a FORTRAN program to carry out the same computations!

CONCLUSIONS

The objectives of the user interface project were to construct a portable, consistent, appropriate interface to a project support environment which improved user's pro-

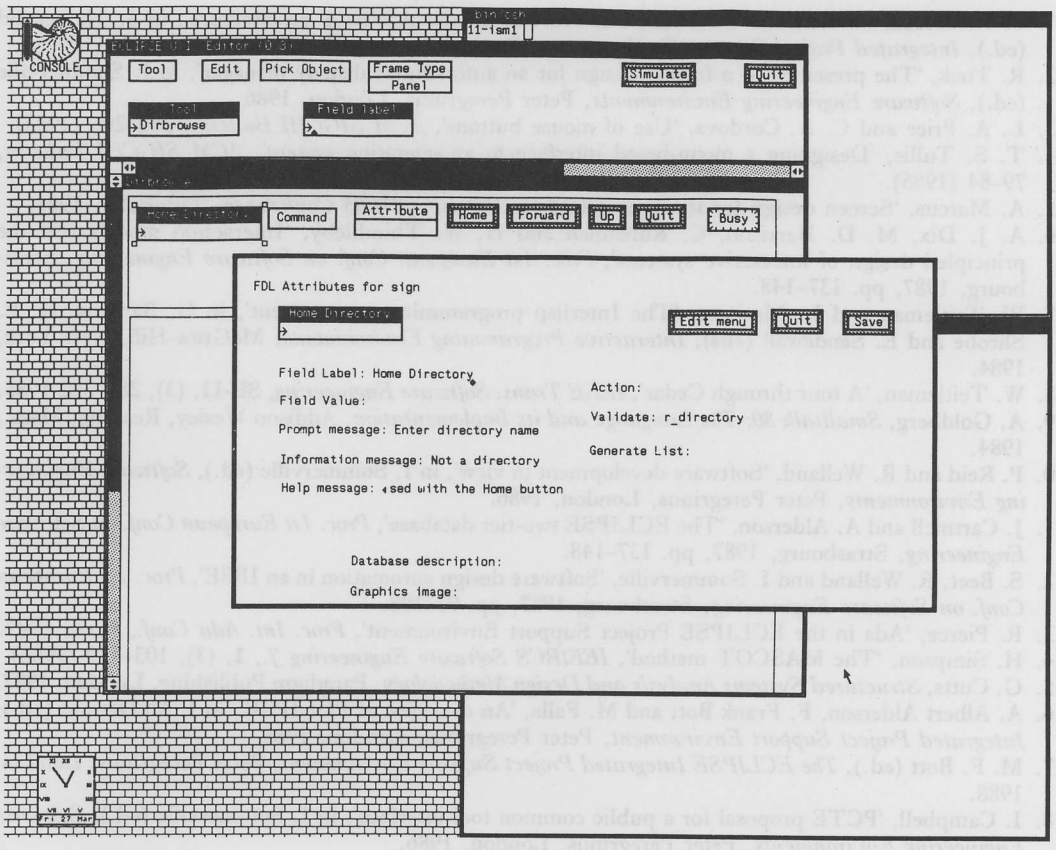

Figure 10. The interface creation tool

ductivity by speeding up system interaction, reducing learning time and reducing user errors. The system which has been built offers a consistent interface (the 'control panel'), includes facilities to help users learn about the system, and is portable inasmuch as it can be moved to another machine by rehosting the applications interface software.

We believe that we have been successful in achieving our initial objectives but, at the time of writing, we have not yet carried out any formal evaluations of the ECLIPSE UI. However, hooks and tools for evaluation have been provided with the interface and it is intended to start an evaluation program in 1988. This will involve theoretical interface modelling, user observations and questionnaires and analyses of the message logs collected by the system.

ACKNOWLEDGEMENTS

The work described here was carried out as part of the ECLIPSE project, sponsored by the Alvey Directorate, U.K. Thanks are due to all of the collaborators in that project, namely Software Sciences Ltd., CAP Group Ltd., Learmonth and Burchett Management Systems Ltd., University of Strathclyde, University of Lancaster, and University of Wales, Aberystwyth.

REFERENCES

1. P. Hitchcock, A. Hall and R. Took, 'An overview of the ASPECT architecture', in J. McDermid (ed.), *Integrated Project Support Environments*, Peter Peregrinus, London, 1985.
2. R. Took, 'The presenter — a formal design for an autonomous display manager', in I. Sommerville (ed.), *Software Engineering Environments*, Peter Peregrinus, London, 1986.
3. L. A. Price and C. A. Cordova, 'Use of mouse buttons', *ACM SIGCHI Bulletin*, 262–266 (1983).
4. T. S. Tullis, 'Designing a menu-based interface to an operating system', *ACM SIGCHI Bulletin*, 79–84 (1985).
5. A. Marcus, 'Screen design for iconic interfaces', *ACM SIGCHI'85 Conference*, Tutorial 4, 1985.
6. A. J. Dix, M. D. Harrison, C. Runciman and H. W. Thimbleby, 'Interaction models and the principled design of interactive systems', *Proc. 1st European Conf. on Software Engineering*, Strasbourg, 1987, pp. 137–148.
7. W. Teitleman and L. Masinter, 'The Interlisp programming environment', in D. Barstow, H. E. Shrobe and E. Sandewall (eds), *Interactive Programming Environments*, McGraw-Hill, New York, 1984.
8. W. Teitleman, 'A tour through Cedar', *IEEE Trans. Software Engineering*, **SE-11**, (3), 285–302 1985.
9. A. Goldberg, *Smalltalk 80: The Language and its Implementation*, Addison Wesley, Reading, Mass., 1984.
10. P. Reid and R. Welland, 'Software development in view', in I. Sommerville (ed.), *Software Engineering Environments*, Peter Peregrinus, London, 1986.
11. J. Cartmell and A. Alderson, 'The ECLIPSE two-tier database', *Proc. 1st European Conf. on Software Engineering*, Strasbourg, 1987, pp. 137–148.
12. S. Beer, R. Welland and I. Sommerville, 'Software design automation in an IPSE', *Proc. 1st European Conf. on Software Engineering*, Strasbourg, 1987, pp. 97–108.
13. R. Pierce, 'Ada in the ECLIPSE Project Support Environment', *Proc. Int. Ada Conf.*, Paris, 1985.
14. H. Simpson, 'The MASCOT method', *IEE/BCS Software Engineering J.*, **1**, (3), 103–120 (1986).
15. G. Cutts, *Structured Systems Analysis and Design Methodology*, Paradigm Publishing, London, 1987.
16. A. Albert Alderson, F. Frank Bott and M. Falla, 'An overview of ECLIPSE', in J. McDermid (ed.), *Integrated Project Support Environment*, Peter Peregrinus, London, 1985.
17. M. F. Bott (ed.), *The ECLIPSE Integrated Project Support Environment*, Peter Peregrinus, London, 1988.
18. I. Campbell, 'PCTE proposal for a public common tool interface', in I. Sommerville (ed.), *Software Engineering Environments*, Peter Peregrinus, London, 1986.
19. F. Gallo, R. Minot and I. Thomas, 'The object management system of PCTE as a software engineering database management system', *ACM SIGPLAN Notices*, **22**, (1), 12–15 (1987).

20. S. E. Engel and R. E. Granda, 'Guidelines for man–display interfaces', *Technical Report TR00.2720*, IBM, Poughkeepsie, NY. 1975.
21. M. M. Danchak, 'Alphanumeric displays for the man–process interface', *Advances in Instrumentation*, **32**, (1), 197–213 (1977).
22. E. S. Cohen, E. T. Smith and L. A. Iverson, 'Constraint-based tiled windows', *IEEE Computer Graphics and Applications*, **6**, (5), 35–45 (1986).
23. J. Walker, 'Implementing documentation and help online', *Tutorial 9, ACM SIGCHI'85*, San Francisco, 1985.
24. R. W. Scheifler and J. Gettys, 'The X window system', *ACM Transactions on Graphics*, **5**, (2), 79–109 (1986).
25. J. D. Smart, 'A man–machine interface management system for Unix', *Proc. Uniforum 1986 Conf.*, Anaheim, CA, 1986.
26. D. England, 'A user interface design tool', *Proc. 1st European Conf. on Software Engineering*, Strasbourg, 1987, pp. 119–126.