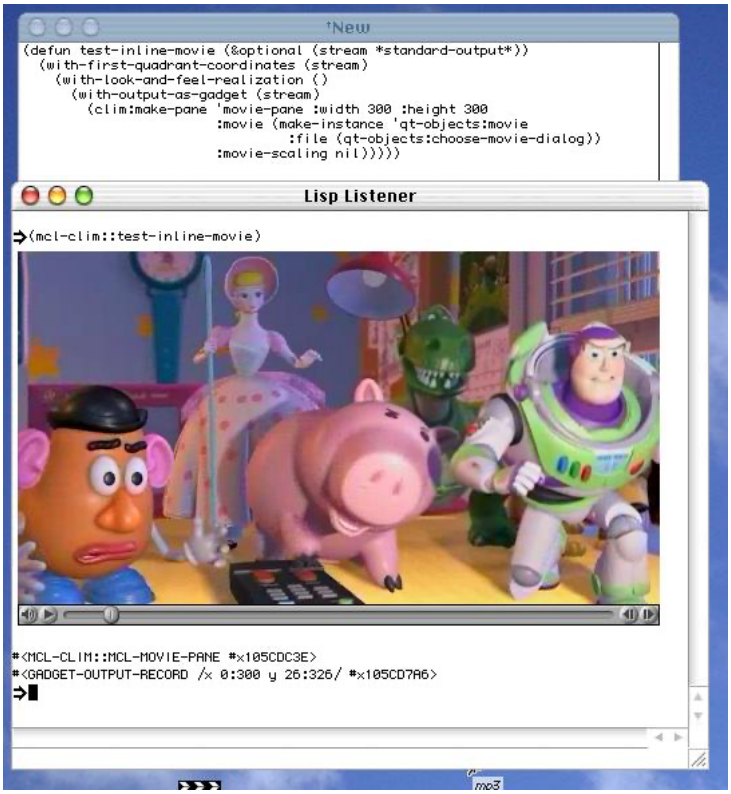
User Interface Management Systems: The CLIM Perspective

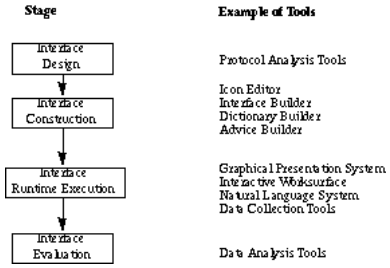


Developing user interfaces is one of the most interesting parts of application development though it can be a tough business. Often contradictory goals must be fulfilled. On the one hand, interface code should be portable, i.e. there should be a way to compile the same code for different host systems such that different look and feels of different platforms can be supported. On the other hand, for commercial user interfaces, it is most important to exactly match the look and feel and to exploit the most recent features of the host system. Another dimension to be considered is the way, interfaces are specified: with direct-manipulative interface builders or with languages that support high-level programming abstractions.

This document gives an introduction to *some* research projects that are concerned with UIMS development. The survey provides the motivation for some of the design decisions of CLIM, the Common Lisp Interface Manager. For a short introduction to CLIM see also the summary presented in [26]. This summary is written by Scott McKay who is one of the authors of the CLIM Specification (see here or here) and one of the implementors of the system. One of the main ideas of CLIM is to specify the *function* of interface components rather than the *form*. Thus, some features are reminiscent of markup languages. The abstractions of the CLIM UIMS are presented in the second part of this document.

Research on User Interface Development Environments

In a narrower sense, user interface management systems (UIMS) are sophisticated software libraries with high-level abstractions for implementing user interfaces. Considering the whole spectrum of interface development activities (see the Figure below), Myers also talks of *user interface development systems* [1] which consist of various tools for rapid user interface development. Foley et al. use the term *user interface design environment*. For historical reasons, I will continue to use the term user interface management system.



Different phases of interface development (adapted from [10]).

The research area of user interface management systems has seen a lot of different systems and approaches. In [2] Foley et al. give an extensive literature survey which is not repeated here (see also Myers, [3]). In the following sections, I will focus on introducing some of the main UIMS trends, models and abstractions.

Because interfaces usually contain graphical elements that can be directly manipulated, it seems to be apparent that the best way to construct them is to use interactive interfaces with palettes for prototypes of graphical objects. These systems are known as *interface builders*.

Interactive Interface Builders

Interface builders provide libraries for standardized dialog objects like buttons, list-gadgets, text-gadgets, gauges (or interactors in Myers' terminology [4]) which can be interactively composed inside a dialog window. Commercial products are available for different languages and program development environments, e.g. for Common Lisp (Harlequin's Interface Builder, Franz' Interface Builder as well as Digitool's Interface Builder), C++, Smalltalk (VisualWorks). Examples for systems are KEEpictures [23], Pogo (a subsystem of HITS [10]), and G2 from Gensym Inc. Simple form-based interfaces can be built with minimum effort. However, when more flexibility is needed, there are some problems with this approach. While application-specific graphics inside a window pane can also be defined using interactive drawing techniques, this is useful only for static data. But what happens when an application uses time-varying data with complex structure? Though it is possible to bind, for example, the position of graphical objects to application data, there are still problems when the number of objects and their constellation varies at runtime. Program code has to be written to insert or delete objects and to map application data to drawing attributes.

Graphical objects shown inside a pane must be automatically made mouse-sensitive. While, for arbitrary application-specific graphical objects (instead of icons), this is neglected in some interface builders, the system ClassWorks supports this. In this system, special "actions" are predefined for graphical objects (cut, paste, movement within certain constraints, etc.). Actions are realized as generic functions using the underlying object system CLOS. Additional (application-specific) actions can be declared and application-specific methods can be written for predefined actions. Currently, no feedback mechanism is supported to indicate what action (if any) can be applied to an object the mouse is pointing at (e.g. through highlighting a graphical object).

The layout of the items inside a dialog window must be adapted when the window is resized. Because size and position of dialog elements depend on the size of the dialog window (application frame) itself, declarative layout specifications are required. User interface management systems manage pane layout by inserting special panes (layout manager panes) into the pane hierarchy. Usually, in user interface management systems, a linear textual notation can be used to specify layout constraints (with ratios, minimum, maximum and average (or default) values for width and height specifications). Some interface builders allow simple specifications to be defined interactively through graphical examples. However, for some pane types (e.g. text panes) the height should not be given in pixel values but in the number of lines (for text). Depending on the font chosen for displaying text, different heights will be actually chosen. See [5] and [6] for a discussion about layout specifications. Note that sometimes, it should also be possible to compute the actual size of a pane with respect to pane contents (e.g. for menu panes). Thus, pane contents can have influences on the constraint solving processes.

There are some proposals to define layout constraints between graphical objects presented inside a pane with direct-manipulative demonstrational tools (e.g. Lapidary [7], OPUS [8] or the commercial ClassWorks from Harlequin). While box-oriented layout descriptions for panes can be specified interactively, there are still problems in interactively specifying, for example, relative position constraints of graphical objects inside a pane (e.g. horizontal or vertical movement only, bounding boxes for movement, etc). In most contexts, the expressiblity of interactive specification mechanisms either for pane constraints or for constraints between graphical objects is too limited. While the intention of these systems was to move up expressiveness while retaining usability, actually, the systems are not very easy to use and to understand. Szekely, one of the authors of such a system, comes to the same conclusion [9].

Interface development involves more than interface builders currently support. Programming is still required for serious applications. Much information used to define graphical mappings and many design decisions are left implicit and cannot be referenced by other subsystems. For example, in most systems, application functions, applied either via menu-interaction or via direct manipulation gestures, are not integrated into a coherent command table mechanism. When a "command" is disabled, all possible invocation methods should be disabled, etc. Similar consistency considerations can be enumerated. Furthermore, specifying interfaces interactively can cause portability problems because the form is specified rather than the function. To overcome the limitations of interface builders, several research programs, considering so-called model-based interface tools, have been initiated.

Model-Based Interface Tools

Model-based interface tools supply explicit models of how an interface should look and behave. The following dimensions are considered in Humanoid [11] [12]:

1. Application semantics: objects and operations of the domain of discourse,
2. Presentation templates: visual appearance of an interface defined by widgets (line, icon, text, menu, button, column, row, table, graph).
3. Behavior: input gestures to be applied to presented objects for specialized interaction styles (e.g. new-point-interactor, angle-interactor, etc.),
4. Dialog sequencing: ordering constraints for commands,
5. Action side-effects: actions executed automatically after a command (e.g. making a new object the "current" object).

A system with similar features is UIDE [13] whose emphasis lies on describing effects of commands (used to automatically generate help facilities). The central component is the design knowledge base. UIDE models simple actions that can be applied to standard interface components: creation, deletion, attribute-modification (e.g. change-thickness), cut, copy, paste. Basic object classes are lines, shapes and text. A frame system is used to model attributes of graphical objects and possible actions defined on them. Systems like these are now state of the art and commercial interface-builders are already available and provide similar or even more extended features (e.g. movement constraints).

In UIDE, actions are modeled with preconditions and postconditions that refer to and modify the world state, respectively. New object classes are introduced by extending the predefined class lattice. It is also possible to define new actions. Models for objects and actions are used to invoke consistency and completeness checks on the knowledge base. Note that actions are defined with respect to graphical objects, not application objects. For interface analyses, a set of completeness and consistency rules is provided with the system. Rules are created by an experienced UIDE developer and are written using a combination of the frame system ART and Lisp. The action models are also the basis for an automatic help system.

Currently, the model-based interface tools UIDE and Humanoid are merged into the system MASTERMIND. One of the main goals of MASTERMIND is to use explicit models to map low-level user gestures onto high-level semantics embodied in the design models and, as in UIDE, to generate automated (and animated) help facilities. The research proposal [21] now comprises ideas to add models for task analysis (and user monitoring). Currently, task modeling approaches from cognitive psychology are considered: e.g. the GOMS method (goals, operators, methods, selection rules). The GOMS method has been introduced by Newell [22] to define a model for performance prediction of an experienced user of different text editing systems. However, even for performance prediction GOMS models are contested. For a discussion of the problems of GOMS and a survey of some of its successors see the discussion on user modeling in the context of human-computer interaction reserach.

UIMS Abstractions from the Viewpoint of CLIM

Abstractions used to model applications and interfaces being introduced by various user interface management systems are quite similar. However, they differ in their concrete nomenclature. No standards for describing UIMS abstactions across language boundaries currently exist and concepts can be described only with respect to specific UIMSs. I will briefly introduce the main concepts and vocabulary of CLIM, the Common Lisp Interface Manager, that is the quasi-standard interface development system for Common Lisp. When differences to abstractions used in other systems (e.g. in Smalltalk) are apparent, these will be pointed out.

CLIM manages screen requirements and mouse-sensitivity areas for graphical output. In addition, several high-level programming techniques are provided: an elaborated graphics model with affine transformations and color support, formatted output, processing of textual commands (with context-sensitive input of parameters), application-building facilities (integration into a host window system, layout specifications for windows and gadgets). Many of the ideas from UIDE and Humanoid are built into CLIM. It also helps to make interface programming portable. Programs are structured so that they can be adapted automatically to the look-and-feel of a specific host system, i.e., declarations are given at the functional level, not at the level of appearance (in terms of form and position).

Structuring the Interface: Frames, Panes, Layouts and Pane Classes

An application frame is the main application window. Usually, the screen area of a frame is managed by the window manager of the host system. A frame consists of several panes which are laid out by declarative descriptions. The main idea of the layout system is to use horizontally and vertically constrained boxes with minimum and maximum specifications for width and height parameters of a pane. The size of a pane (and therefore, the size of a frame) can also depend on its contents. An application frame can have several different layout configurations. Similar structures can also be found in Smalltalk systems.

The contents of a pane are specified by a drawing function (or method) given with the pane declaration. Depending on the class of a pane, different drawing and updating strategies are available. For example, panes can be command listeners (called interactors in CLIM), simple application panes or so-called accepting-values panes (a pendant to dialogs available in other UIMSs, but more flexible because the number of gadget shown in a dialog need not be fixed).

Commands as a Unified Abstraction Mechanism

In CLIM, a command is an object which is generated in order to handle a menu item selection, a mouse gesture (e.g. click) on a graphical object, a key press or textual input. The different interaction techniques are interpreted as different invocation methods for commands. Commands are characterized by names and parameters (and their types) which can be required or optional (keyword parameters). At runtime, CLIM uses hierarchical command tables to find a handler for a command object: a body with a sequence of code statements. When a command (or a complete command table) is disabled in a specific context, CLIM automatically grays out corresponding entries in menus and CLIM's command processor refuses to parse textual input of the command name (and parameters). This is a great advantage over the non-integrated mechanisms of other systems which provide separate invocation methods for menu item or standardized dialog item functions (usually called "callbacks" in other systems) or functions associated with "actions" on graphical objects. However, because of this level of indirection, CLIM cannot be easily integrated with standard interface building ideas (e.g. for interactively specifying menus and submenus). An interface builder for CLIM has been developed by Cramer. It is called Mirage.

Since many applications deal with (dynamic) geometric objects, facilities offered by interface builders for drawing graphical objects (like those provided by ClassWorks or Lapidary [7]) are not required or not directly applicable. The unified command abstraction is unique to CLIM (and its predecessor Dynamic Windows, see below).

In order to parse commands and to accept only valid arguments, CLIM needs several declarations. A command is specified by declaring argument types for formal parameters, prompting strings, default values for arguments and other options such as menu names, invoking keys, etc. The body of a command is defined by writing standard programming language code. Because names and types of parameters (as well as prompts, documentation strings and default values) can be specified in a declarative manner, CLIM can automatically generate form-based completion dialogs and context-sensitive help. Prompting, completion and command parsing is done by CLIM's command processor and not by application functions. This ensures a coherent interaction style. Note again that there are several ways to invoke a command (by typing in textual command strings, by mouse gestures or by keyboard accelerators, see above).

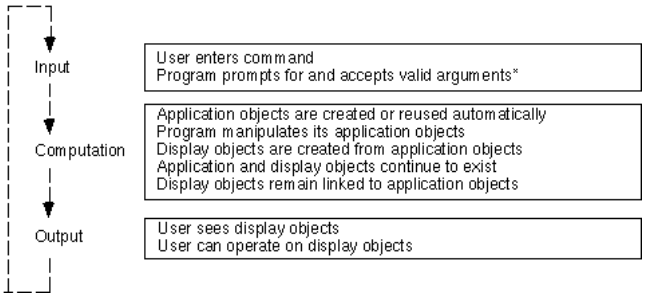
After a gesture (e.g. a drag and drop gesture or a mouse-click gesture) has been completed and a handler for a command object is found in a command table, the corresponding command body is applied to the actual command arguments. The body is executed and application data are updated. As a direct consequence, the presentation of application objects on the screen will have to be redrawn in the input/computation/output cycle.

Input/Computation/Output Cycle

Basically, there are two approaches to handle redrawing of application panes. The bottom-up approach records changes to application objects and redirects these changes to update requests to associated graphical objects shown in different panes. Smalltalk supports this process with its Model-View-Controller scheme [25] (other systems use similar approaches). An application object is called a *model* which is associated with one or more *views*. Controller objects manage the manipulation of graphical objects and translate changes to these into corresponding *change messages* sent to model objects (application objects). Though the bottom-up approach is not directly supported by the CLIM framework, it could also be implemented with minimal effort. However, the approach requires that modification requests to application objects be recordable (which is not alway the case for low-level data structures necessary for some appliations).

The other way to handle redrawing uses a top-down strategy. This is the supported way of CLIM. Each pane of an application frame is associated with a drawing function (or drawing method). The drawing model of CLIM allows elementary output (e.g. with `draw-line`, `draw-rectangle`, etc.) to be recorded (in so-called *output records*). Recorded output is automatically replayed (e.g. when a pane is scrolled by the user). Output records may be arranged into a part-of hierarchy (spatial decomposition). Specialized output records can be used to cache previous output. CLIM automatically computes what has to be erased and what has to be redisplayed when cache values change. This mechanism is called *incremental output*. The caches contain references to attributes of domain objects and, when these change, the output record will be replayed. The part-of hierarchy of output records can be used to control the search space for cache-changes. Other user interface management systems also provide a facility to arrange graphical output using hierarchical structures (e.g. ClassWorks).

For instance, the replay facility of output records can be used to sort output into different overlay levels. Output records can be used like tapes that record drawing instructions. For example, the output records can be replayed in another sequence than originally recorded. The basic input/computation/output cycle implemented by the CLIM runtime system is sketched in the following figure.



CLIM approach to the input/computation/output cycle (from [20]). The items in the top and bottom boxes (input and output) are supported by CLIM rather than have to be redesigned for every new application.

User interface toolkits provide libraries for standardized interface elements such as buttons, scroll-bars, gauges and others. Each of these components can be manipulated with the mouse. The components have been hand-coded in order to handle mouse events like button clicks and also to update the screen. However, when an application draws graphical elements in its main application window, the material is usually "dead". Expensive low-level code (event handlers) has to be written in some systems to breathe life into application-specific graphical elements. An important contribution of CLIM is, among others, automatic screen and mouse-sensitivity management right from primitives like `draw-line`, `draw-rectangle`, `draw-text` and similar basic drawing functions. While ClassWorks also allows graphical objects (which are modeled explictly using geometrical primitives) to be made mouse-sensitive, CLIM is much more flexible and its architecture is much clearer. Note that in applications built with interface builders, graphical object structures are not necessarily identical to the corresponding application-specific object structures, and therefore, information is represented twice. With CLIM, there is no need to create separate graphical object structures as copies of application data structures (using special graphical primitives)

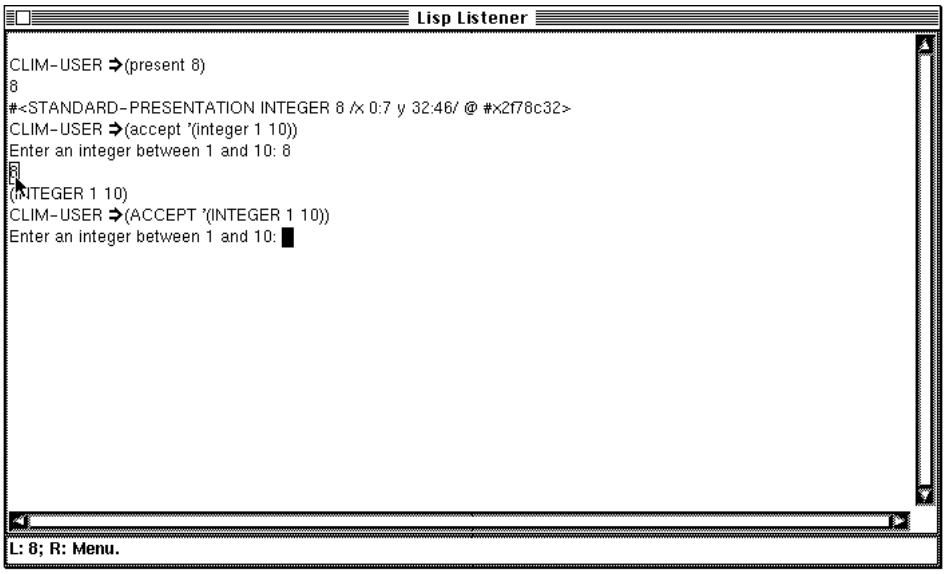
only for getting graphical objects mouse-sensitive. Interfaces are built in terms of application objects rather than special graphical objects or system-dependent toolkit objects. The association between an application object and its graphical equivalent(s) is called *presentation*.

The main contribution of CLIM is that a presentation can be associated with type information (*presentation types*). Types are useful to determine whether a certain graphical operation is applicable when an object of a certain type is required for a certain input request.

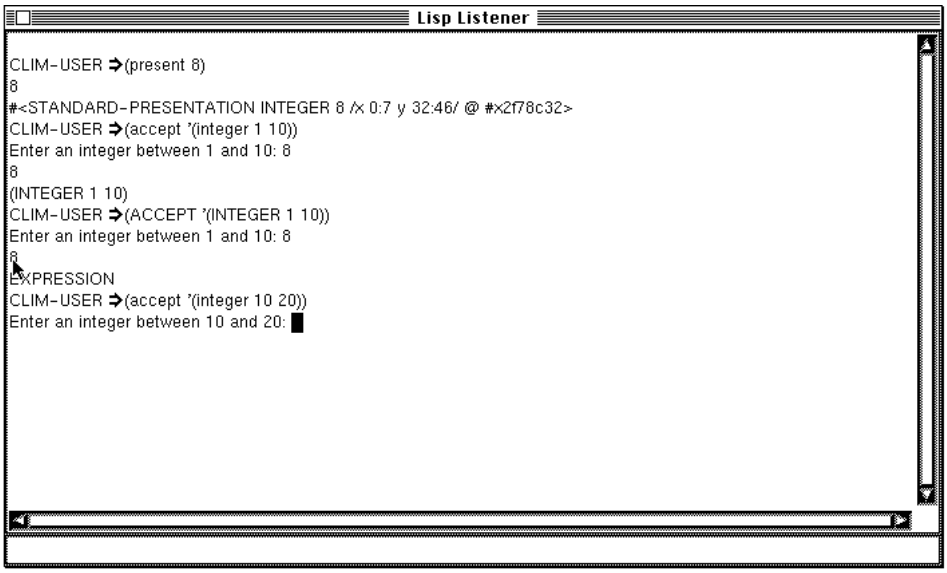
Associating Type Information with Graphical Output

Presentation types are an extension to the type system of Common Lisp. In particular, they can be seen as parameterized CLOS classes, i.e., by definition every CLOS class is a presentation type. In CLIM, argument types for commands can be defined. The types that can be given are presentation types. Due to type information, CLIM can automatically make previous output mouse-sensitive whenever a command argument is required that matches the type of previous output. Inheritance mechanisms are used in this process as usual for object-oriented systems.

Parameters for classes are needed to describe, for example, a range of integers (written as `(integer 1 10)`). Presentation types are arranged in a hierarchy (single-inheritance only, due to performance considerations?). Parameters with types are treated adequately by the inheritance mechanism, i.e., `(integer 1 10)` is a subtype of `(integer 1 100)`. Let us assume a number is printed, say `8`. If another number is to be read, e.g. an integer between 1 and 10 (`(integer 1 10)`), the previously printed `8` is mouse-sensitive and can serve as an input source. Consequently, due to type incompatibilities, an `8` is not mouse-sensitive when an integer between `10` and `20` is requested (see the screen snapshots in Figures below). The basic functions for output and input are `present` and `accept`.



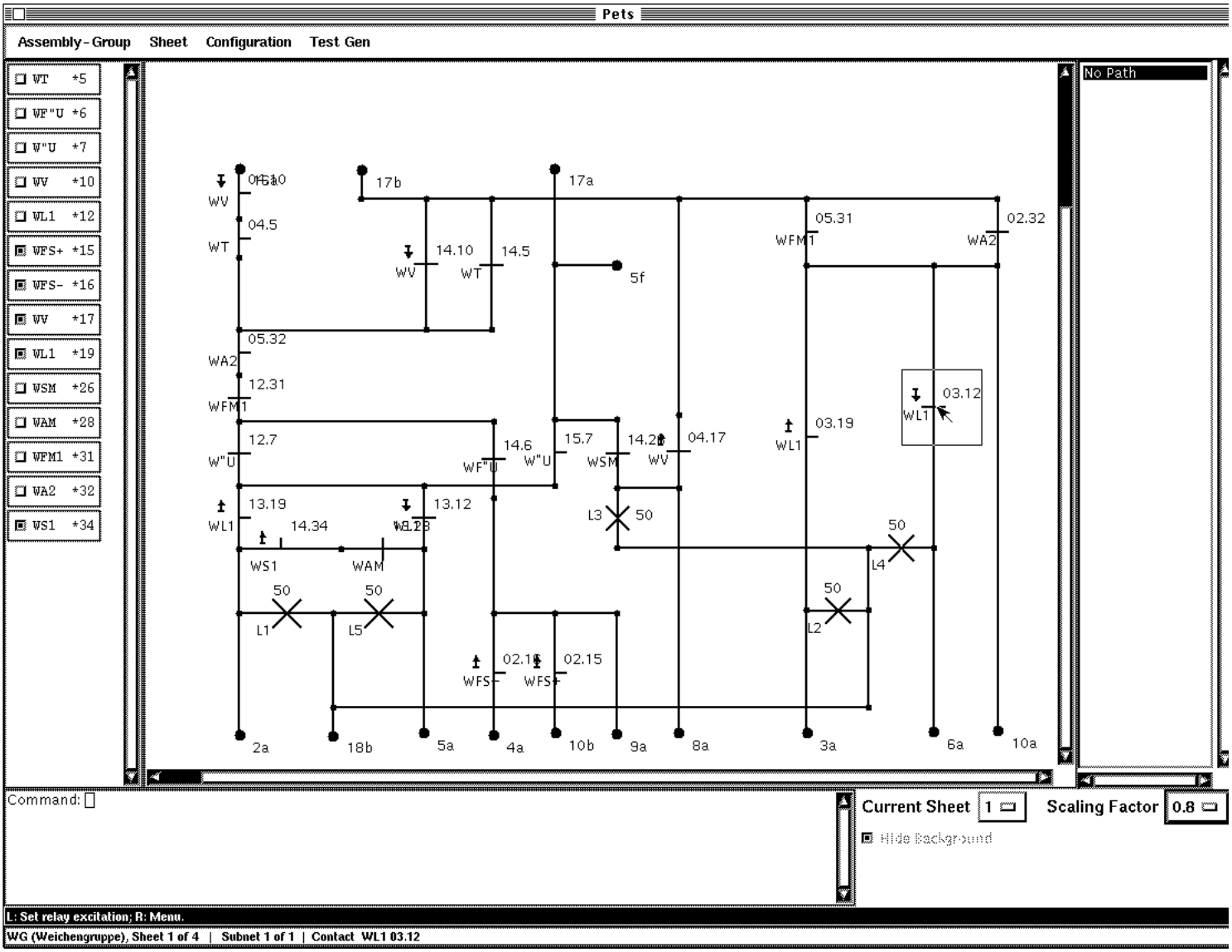
Accepting an integer between `1` and `10`. The `8`, printed previously, is mouse-sensitive.



Accepting an integer between `10` and `20`. The `8` is not mouse-sensitive.

The `accept` mechanisms are not only applicable to textual output, but also to graphical objects. This is presented with another application example.

The figure below shows the main window of an application developed at the LKI. The application is called PETS and is used to generate test plans for relay assembly groups used by railroad companies (a Postscript description of PETS can be found here). CLIM has been proven to be extremely useful for this application.



Snapshot of a graphical editor for relay circuits.

In the main window, `circuit-elements` like `relay-contacts` (small bars crossing or touching the wires), `lamps` (crosses) and `terminals` (large dots with labels) are presented. In this application, every `circuit-element` is represented as a CLOS object and is drawn using the associated presentation type. As can be seen in the right part of the circuit, the output is mouse-sensitive (indicated by a rectangle) because certain commands are applicable. In the lower part, there is a command listener for giving textual commands. When the end-user types a command (e.g. *Delete Object*) and this command requires a parameter of a certain type (e.g. `circuit-element`) all objects presented with this type will become mouse-sensitive. When a more specialized type is required (e.g. for the command *Switch Contact*), the type will be restricted to `contact-elements` (a subclass of `circuit-element`) and only the objects presented with this special type will be made mouse-sensitive by CLIM. Note that the concrete graphical appearance itself is not important for type compatibility and mouse-sensitivity reasoning of the accept-mechanism. Only the occupied space and type information associated with graphical output is considered.

Translators

When some kind of file information is printed (with file name, creation date, size, etc.), a presentation type `file-info` might be used. If, in some other context, a `string` is accepted, the names of the files can provide adequate input. Thus, output of type file-info can be translated into a string which is the file name only. CLIM allows an application programmer to define mapping functions from one presentation type to another (called `presentation-translators`). Declaring a translator suffices to make a `file-info` mouse-sensitive when a string is accepted.

In order to allow the user to invoke a command, another kind of translator may be used (`presentation-to-command-translator`). When the user clicks at a presentation, the application object behind the presentation is used as the first argument of the command given in the `presentation-to-command-translator` definition. A special kind of `presentation-to-command-translator` is a `drag-and-drop-translator`. The first argument of the associated command is the dragging source and the second is the destination of the drag-and-drop gesture. The definition of a translator is very easy (basically, just the command names and presentation types have to be declared), there is no need to define low-level code for mouse events since this is automatically handled by CLIM. When a command has been disabled, no command translators are activated, i.e., objects are not mouse-sensitive. While required arguments for a commands are gathered, clicking on presentations can cause application functions to be applied if so-called presentation-actions are declared.

Portability Considerations

In some systems, interface builders are use to define, for example, a dialog with radio buttons as dialog items. Each radio button represents an application object (or an attribute of an application object). Using an interface builder requires exact coordinates and alignments to be specified. However, when the interface is displayed at runtime in an environment that uses a different style (maybe with larger buttons or larger fonts for the buttons' labels) than initially anticipated, the radio button array will be distorted. Later in the development process it might become clear that the end-user has to select between too many items. Then, radio buttons cannot be used anymore and,

if an interface builder will be used, for example to integrate a list dialog item, complicated manual redesign operations and changes to the interface code are the consequence.

CLIM uses a more sophisticated approach. Instead of specifying the interface in terms of toolkit objects, the interface programmer can use a "deep-level declaration". All that is required is to define a selection of set of application objects using the member presentation type constructor. For the surface level, CLIM chooses a radio box, if appropriate. The interface designer can overwrite CLIM's decision by specifying a certain view to be used for the input operation (see below). Other presentation type constructors offered by CLIM are `member-sequence`, `alist`, `completion`, etc. With the introduction of these concepts, CLIM is enabled to produce (more) portable interfaces.

Views

Another important abstraction found in many user interface management systems is the notion of a view. In CLIM, view classes can be arranged in a taxonomic hierarchy. A view is an instance of a view class. Depending on the view, presentation of values and accepting them can be done with different style or geometric perspectives.

Predefined views in CLIM are `textual-view` and `gadget-view` (e.g. with `slider-view` as a subclass). Thus, CLIM supports more than one view of application data. Note that views are not important for the type inferencing mechanisms of CLIM for accepting values (s.a.), but affects the visual appearance of application objects presented in a pane. For standard user interface elements the views provided with CLIM ensure that the appearance of corresponding dialog items is adapted to the look and feel of the host window system. To implement the domain-specific appearance of objects, an interface designer can define present and accept methods adapted to the class structure of his application. This will be discussed in the next subsection.

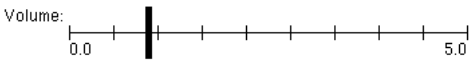
Presenting and Accepting Values

A presentation associates graphical output with application objects and is defined by a presentation type. A specific presentation type context (represented by a specialized presentation output record) can be set up dynamically (`with-output-to-presentation`) or by writing so-called `present` methods using an extension to CLOS. Present methods dispatch on the application object, the presentation type, the stream (pane) type, and the view.

The input pendant to output mechanisms like `present` methods are `accept` methods. In the example application shown in the figure above, a pane (to the left) is used to display the status of the relays that have contacts in the current circuit shown in the main pane. A relay status is a boolean value (indicating that the corresponding relay is either excited or not). If the relay is excited, the small button will be pressed (black) otherwise it will be released (white). However, instead of using present, the pane (an instance of a special pane class) is displayed with calls to accept. Thus, the status of the relays (given as a default value to the call to accept) can be changed interactively. No special view is given and therefore, CLIM uses a check box that is adapted to the look an feel of the host window system. The number of required check boxes (i.e., calls to `accept`) depends on the circuit being edited. Therefore, placing check boxes interactively using techniques found in interface builders are not applicable here (the pane used in the application frame even allows check boxes to be scrolled if not all check boxes can be made visible at a time). CLIM provides standard `accept` methods for various built-in presentation types (and presentation type constructors).

The ideas behind `accept` can be extended to domain-specific presentation types by writing `accept` methods (dispatching on the presentation type, the stream and the view).

```
(accept 'number
        :view '(simple-slider-view
                :width 300
                :tick-number 10
                :min-value 0.0
                :max-value 5.0)
        :prompt "Volume"
        :default (volume ...)
        :stream ...)
```




```
(accept 'number
        :view +text-field-view+
        :prompt "Volume"
        :default (volume ...)
        :stream ...)
```

History, Status and Further Developments of the Ideas Behind CLIM

Inspired by the work of Zdybel et al. on the AIPS (Advanced Information Presentation System [14]), Friedell [15] and Shneiderman [16], Cicarelli developed the first ideas about presentations (published in his dissertation from 1984 [17]). While AIPS uses modeling structures of KL-ONE to explicitly represent domain structures (physical objects, vehicles, ships, etc.) and display structures, the presentation system of Ciccarelli tries to be completely domain-independent.

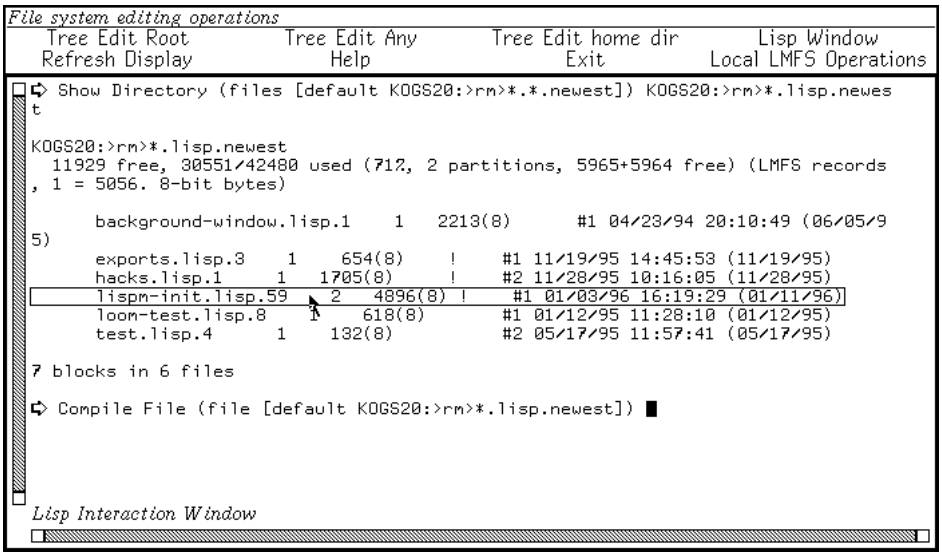
Ciccarelli developed the idea that the user manipulates an explicitly represented presentation structure. As a direct consequence, the manipulation operations are &recognizedÔ and translated into commands for the application (a database system in Ciccarelli's work). After a command has been executed, the presentation structure and, therefore, the screen is updated, i.e., the presentation data base is mapped to the graphics package (the system has been implemented on a Lisp machine). He also introduced the notion of a presentation style (presentation template) that largely corresponds to views in CLIM now.

Lieberman, in his 1985 SigGraph paper [18], presented an object-oriented implementation of an interface construction system (EZWin) which uses presentation objects for a visual representation of things in the problem domain. Operations on presentation objects cause command objects to be created. Lieberman developed the idea that previous output should be usable for arguments of subsequent commands. The old question whether to first select the operation and then the object or vice versa has been shown to be ill-posed since clicking on a presentation object can cause the presentation of a command menu that contains only those command that are applicable to the type of the information (nowadays called presentation types). If further arguments are required, more objects can be clicked on. If the operation is given first, e.g. by typing a command, the possible arguments are mouse-sensitive and can be selected afterwards.

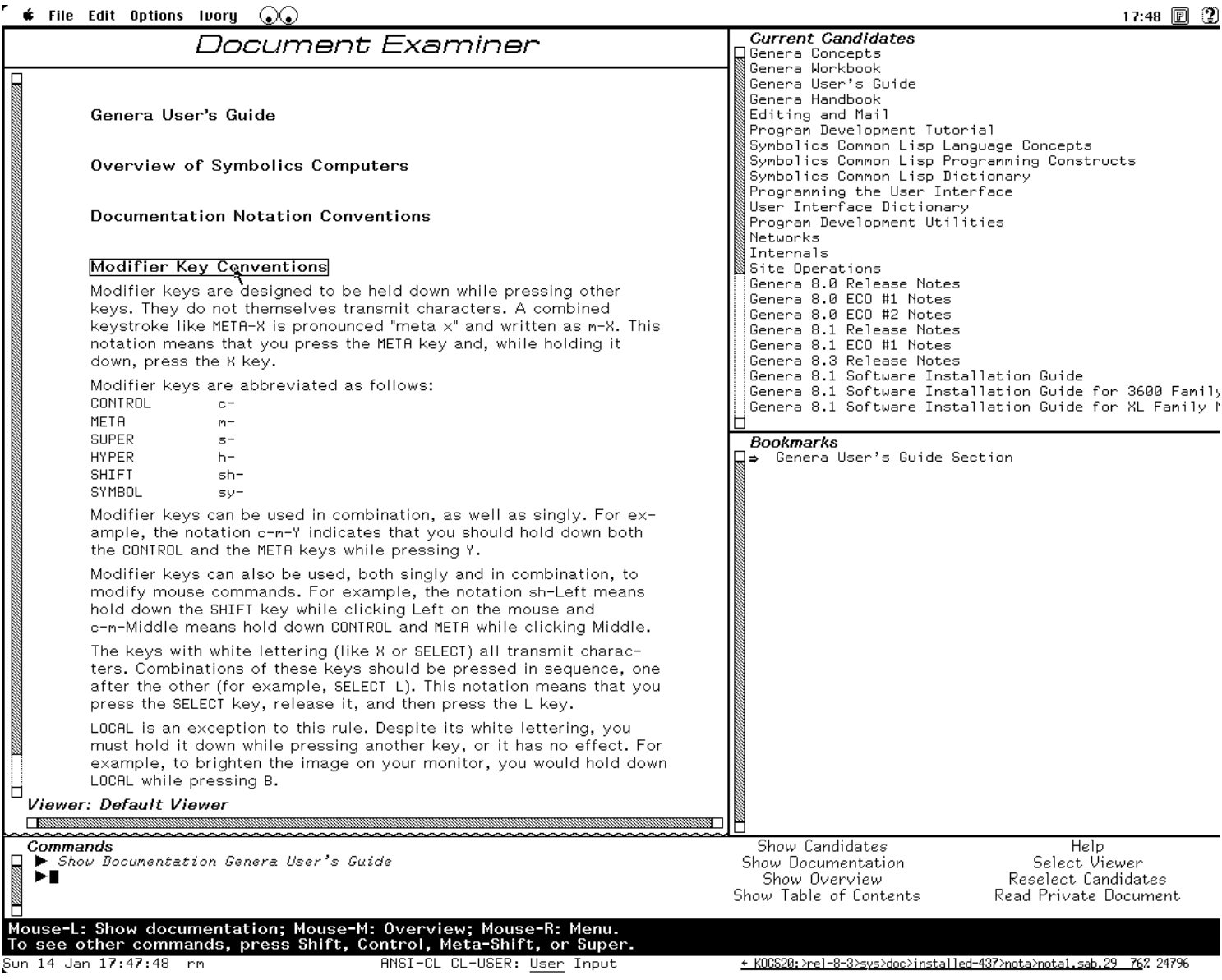
In order to show what can be done with the mouse, EZWin automatically maintains mouse-sensitivity information in order to highlight a presentation object (or its visual equivalent) when the mouse passes over it.

A pioneering system which uses these ideas was *Dynamic Windows*, the high-level window system of Genera, the operating system of the Symbolics Lisp Machine. The first versions of *Dynamic Windows* appeared with Genera 7.0 in 1986 [19]. Presentations have been used for the Lisp program development environment and for a command processor (nowadays called shell). A command like *Show Directory* `host:>dir>*.lisp` prints file information into a window. When, afterwards, a command

Compile File is given, all the Lisp files are mouse-sensitive and can be clicked on. It is also possible to directly click onto the file name and select, for instance, *Edit File* via a pop-up menu.



The management of mouse sensitive regions is also used for application programs. The next figure present the Document Examiner, an ancestor of all hypertext systems (was it possible to refer to files on the net?).



Pictures of Symbolics machines can be found at here. See also an overview of the Symbolics Technical Summary, a presentation of the Genera development environment and a presentation of the Document Examiner.

CLIM is a successor of *Dynamic Windows* but is also available for Lisp compilers on UNIX and Windows (Franz Allegro CLIM and Harlequin CLIM, Symbolics CLIM) as well as Macintosh computers (Macintosh Common Lisp CLIM from Digitool Inc.). Some other systems have adapted the ideas of *Dynamic Windows* (see the presenting listener for the XEmacs editor as part of Franz Allegro Common Lisp development environment and also the XEmacs itself) but none of these replicas manage type information. Furthermore, only textual output is made mouse-sensitive. The CLIM architecture is far more consistent. However, currently, CLIM is not suitable for high-performance graphics of the sort needed for sophisticated animation or video production applications (multimedia applications). Recent approaches try to incorporate ideas of presention types into remote forms sent out via information systems like WWW to prevent incorrectly filled out forms to be returned to the server. (see the work of John Mallery and his Common Lisp HTTP server). If type information were available, parsing of input could be done by the local client program, rather than would require communication with the server! Furthermore, when an object is not suitable for the current input context, it will not be mouse-sensitive at all.

Conclusion

It should be clear now that, from a bottom-up perspective, a sophisticated system like CLIM is perfectly suited as a basis for interface and application development. The features of CLIM are very useful in applications with dynamic data where interface builders fail. However, as can be expected, there is no guarantee that user interfaces built with interface builders, model-based interface tools or extended user interface management systems like CLIM, are really usable for end-users.

References

1. Myers, B.A., User Interface Tools: Introduction and Survey, IEEE Software, January 1989, pp. 15-23.
2. Foley, J.D., Wallace, V.L., Chan, P., The Human Factors of Computer Graphics Interaction Techniques, IEEE Computer Graphics, Vol. 4, No. 11, November 1984, pp. 11-44.
3. Myers, B.A., User Interface Tools: Introduction and Survey, IEEE Software, January 1989, pp. 15-23.
4. Myers, B.A., Encapsulating Interactive Behaviors, in: Proceedings Human Factors in Computing Systems, Bice, K., Lewis, C. (Eds.), Austin, Texas, 1989, pp. 319-324.
5. Haarslev, V., Möller, R., A Declarative Formalism for Specifying Graphical Layout, in: Proc. 1990 IEEE Workshop on Visual Languages, Skokie/IL, Oct. 4-6, 1990, IEEE Computer Society Press, 1990.
6. Möller, R., Haarslev, V., Layoutspezifikationen für komplexe graphische Objekte, in German, in: Proc. Graphik und KI, GI-Fachgespräch, Kansy, K., Wißkirchen, P. (eds.), Königswinter, April 1990, pp. 78-91.
7. Myers, B.A., Vander Zanden, B., Dannenberg, R.B., Creating Graphical Interactive Application Objects by Demonstration, in: Proceedings UIST'89, Nov. 1989, pp. 95-104.
8. Hudson, S.E., Mohamed, S.P., Interactive Specification of Flexible User Interface Displays, ACM Transactions on Information Systems, Vol. 8, No. 3, July 1990, pp. 269-288.
9. Szekely, P., Luo, P., Neches, R., Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. in: Proceedings of CHI'92, The National Conference on Computer-Human Interaction, May 1992, pp. 507-515.
10. Hollan, J., Rich, E., Hill, W., Wroblewski, D., Wilner, W., Wittenburg, K., Grudin, J., An Introduction to HITS: Human Interface Tool Suite, in: [24], pp. 293-337.
11. Szekely, P., Luo, P., Neches, R., Beyond Interface Builders: Model-Based Interface Tools, USC/Information Sciences Institute.
12. Szekely, P., Luo, P., Neches, R., Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. in: Proceedings of CHI'92, The National Conference on Computer-Human Interaction, May 1992, pp. 507-515.
13. Foley, J., Kim, W.C., Kovacevic S., Murray, K., UIDE: An Intelligent User Interface Design Environment, in: [24].
14. Zdybel, F., Greenfeld, N., Yonke, M, Gibbons, J., An Information Presentation System, in: Proceedings of IJCAI 81, IJCAI, Vancouver, Canada, August 1981, pp. 978-984.
15. Friedell, M., Automatic Synthesis of Graphical Object Descriptions, Computer Graphics, Volume 18, Number 3, July 1984.
16. Shneiderman, B., Direct Manipulation: A Step Beyond Computing, IEEE Computer, August 1983.
17. Ciccarelli, E.C., Presentation Based User Interfaces, MIT Ariticial Intelligence Laboratory, Technical Report 794, August 1984.
18. Lieberman, H., There's More to Menu System than Meets the Screen, in: Proceedings ACM SigGraph, Vol. 19, No. 3, 1985, pp. 181-189.
19. Symbolics, Programming the User Interface Vol A, Symbolics Inc., September 1986.
20. Symbolics, Common Lisp Interface Manage (CLIM): Release 2.0, Symbolics Inc., Jan. 1994.
21. Neches, R., Foley, J., Szekely, P., Sukaviriya, P., Luo, P., Kovacevic, S., Hudson, S., Knowledgeable Development Environments Using Shared Design Models, in: Proceedings ACM/AAAI International Workshop on Intelligent User Interfaces, Orlando, Fl. Jan., 1993.
22. Card, S., Moran, T., Newell, A., The Psychology Of Human-Computer Interaction, Erlbaum, Hillsdale, 1983.
23. Wolf, S., Setzer, R., Wissensverarbeitung mit KEE - Einführung in die Erstellung von Expertensystemen, in German, Oldenbourg, Muenchen, 1991.
24. Sullivan, J.W., Tyler, S.W. (Eds.), Intelligent User Interfaces, ACM Press,1991.
25. Krasner, G.E., Pope, S.T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, ParcPlace Systems, Smalltalk Manual, Palo Alto.
26. McKay, S., CLIM: The Common Lisp Interface Manager, CACM 34(9), pp. 58-59, 1991.