# IMPULSE-86
# A SUBSTRATE FOR OBJECT-ORIENTED INTERFACE DESIGN

Reid G. Smith  Rick Dinitz  Paul Barth

*Schlumberger-Doll Research*
*Old Quarry Road*
*Ridgefield, CT 06877-4108*
*USA*

## ABSTRACT

Impulse-86 provides a general and extensible substrate upon which to construct a wide variety of interactive user interfaces for developing, maintaining, and using knowledge-based systems. The system is based on five major building blocks: *Editor, EditorWindow, PropertyDisplay, Menu,* and *Operations.* These building blocks are interconnected via a uniform framework and each has a well-defined set of responsibilities in an interface.

Customized interfaces can be designed by declaratively replacing some of the building blocks in existing Impulse-86 templates. Customization may involve a wide range of activities, ranging from simple override of default values or methods that control primitive operations (*e.g.*, font selection), to override of more central Impulse-86 methods (*e.g.*, template instantiation). Most customized interfaces require some code to be written—to handle domain-specific commands. However, in all cases, the Impulse-86 substrate provides considerable leverage by taking care of the low-level details of screen, mouse, and keyboard manipulation.

Impulse-86 is implemented in Strobe, a language that provides object-oriented programming support for Lisp. This simplifies customization and extension.

## 1 INTRODUCTION

Domain-specific interactive editing tools make complex systems easier to construct, modify, maintain, and use. General editing tools, while very useful, are not always sufficient. Interaction with knowledge-based systems in particular can be made more effective if a specialized interface, with knowledge of the domain, is used.

Impulse-86 addresses the problem of building a wide variety of sophisticated editing tools. Our goal is to enable developers and end users of knowledge-based systems to design custom, domain-specific interfaces for their systems, without a requirement for extensive expertise in interactive graphics. Our approach is to provide an extensive set of building blocks and a uniform framework for connecting them.

In section 2 we describe the goals of Impulse-86. In section 3 we discuss an example editor, and present the editing knowledge and organization that give Impulse-86 its power. In Section 4 we present two further editors that illustrate the diversity possible using our substrate. In Section 5 we discuss the effectiveness of Impulse-86 as a substrate for user interface construction. We discuss utility of designing Impulse-86 in an object-oriented paradigm in Section 6. Related work on user interfaces is discussed in section 7.



## 2 BACKGROUND AND GOALS

We aspire towards the longer-term goal of providing a common knowledge-based interaction substrate to underly our application systems. It is well-understood that good interfaces account for a sizable percentage of the overall code and effort in many systems (see [12] for a knowledge-based system example). Hence, if we are to reduce the cost of building interactive systems, and at the same time increase the consistency and power of the interfaces that they present to an end user, then one of the areas in which we must concentrate our attention is user interface design tools.

Throughout this paper we discuss Impulse-86 as a *knowledge base editor.* We take the point of view that the process of editing may usefully be defined as any series of interactions between a user and a system in which the user *views* the state of the system, and *controls* or *changes* the state of the system. An editor is an entity that mediates these viewing and controlling interactions—presenting the user with a view of the system and effecting the desired control. This definition encompasses traditional text editing, browsing, program development, debugging, and end-user interaction.

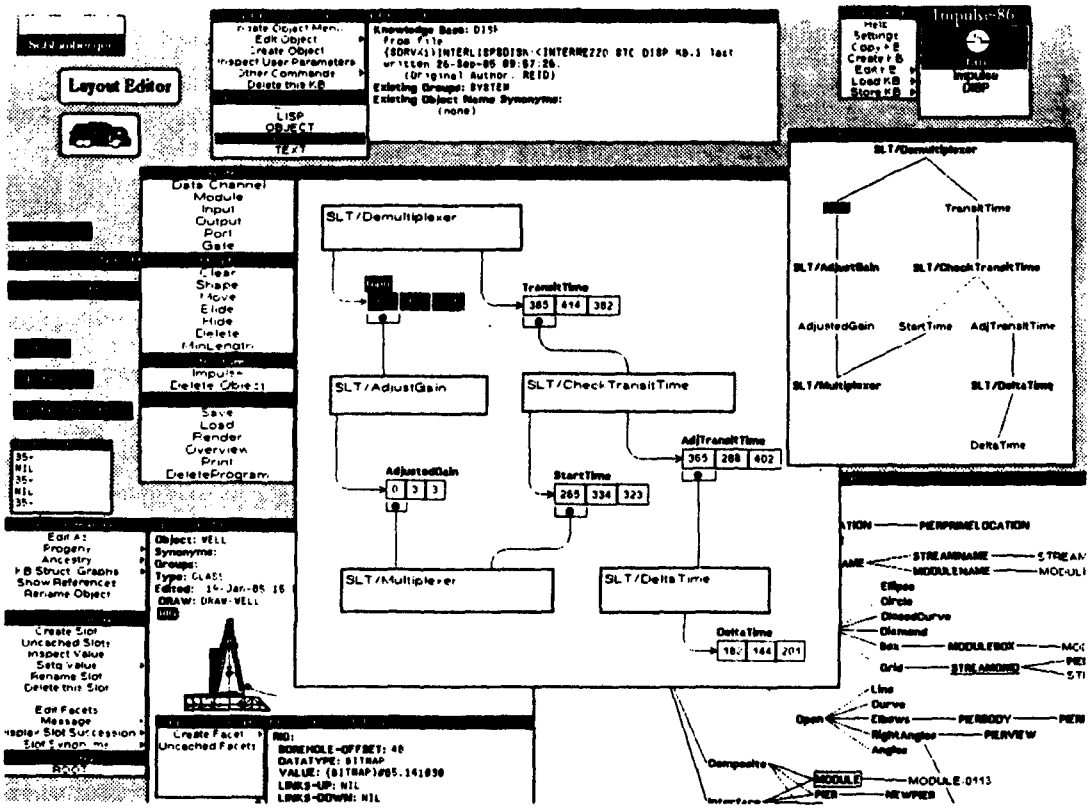


Figure 1: Typical Impulse-86 Editing Session

A knowledge base editor should support different perspectives corresponding to three types of user: developer/maintainer; domain specialist; and end user. An editor must provide a reactive environment for developers; enable domain specialists to focus attention on the encoded domain knowledge (while the editor hides the underlying representational mechanisms); and provide transparent and easy-to-use interfaces for end users.

Knowledge-based systems typically consist of many complex structured objects, connected by several relationships, and are intended to model richly structured domains (*e.g.*, structural geology, process control software). As a result, both the knowledge-based system developer and end user are typically interested in interacting with three major kinds of entities: objects and their internal structure, relationships among objects, and complete systems of objects, relationships, and code—including their dynamic behavior. In sections 3 and 4 we shall exhibit editing tools specialized for each of these entities.

Based on the above considerations we have been led to the following design goals for interfaces constructed with Impulse-86:

- Effective integration and use of high resolution bitmap displays, pointing devices, and keyboards.
- Flexible support for varied methods of user interaction for both viewing and controlling systems (*e.g.*, graphics, animation, menus, pointing and *smart* type-in). The editor should support interaction with domain-specific information in a form *natural* to an end user familiar with the domain.
- Unlimited interaction contexts for viewing and changing different parts of a system simultaneously. (Figure 1 is a snapshot of a typical screen with several interaction contexts active.)
- Organizational support for customization and extension.[1]

[1]Stallman defines extensibility relative to EMACS: "... the user should be able to add new editing commands or change old ones to fit his needs, while he is editing" [16]. We go beyond *command* extensibility, aiming for extensibility with respect to what is viewed, how it is viewed, and how it may be changed.

[2]Strobe [13,14,3] is an extension of Interlisp-D that supports object-oriented programming. It has also been implemented in C and in Common Lisp.

Impulse-86 extends Impulse [11], an editor designed to enhance the productivity of developers of knowledge-based systems. Impulse has been in use at a number of centers for the past three years, and has proven to be an effective tool for creating, extending, and maintaining Strobe knowledge bases and related code in the Interlisp-D programming environment.[2] Impulse-86 is itself implemented as an object-oriented program in Strobe. Like its predecessor, it is the standard editor for Strobe knowledge bases, but now supports development of end-user interfaces as well.

## 3 A BUILDING BLOCKS APPROACH TO INTERFACE DESIGN

To enable easy construction of a wide variety of state-of-the-art editing tools and interface styles, Impulse-86 is conceived as a modular, extensible interface construction kit. We will see below that a user/developer can customize and extend the behavior of an interface based on Impulse-86 by overriding default values and methods, and by specializing existing objects. In addition, Impulse-86 makes use of a well-defined set of message protocols for constructing an interface. This affords a user the flexibility to make quite radical changes in the operation of an interface by modifying methods at different levels in the Impulse-86 taxonomic hierarchy.

In the following we discuss a canonical example. We then present the Impulse-86 substrate, using the example as a point of reference.

### 3.1 THE OBJECT EDITOR

The object editor is used for editing a single Strobe object and its internal structure. Figure 2 depicts a user's view of the standard object editor; the object being edited encodes geologic knowledge taken from the *Dipmeter Advisor*† system [12].

The object is presented to the user in a window with attached menus. The first five lines show the values of properties that are common to all objects; the remaining lines display the object's slots.[3] The name of a property or slot is shown in boldface; if a slot has synonyms, they are enclosed in curly braces; the slot's value (if any) is shown in lightface, following a colon; (↑) indicates that the slot is inherited from a more general object. We have found these conventions useful, but they are only defaults. A user can define other ways to convey the same information (or other information), overriding the defaults.

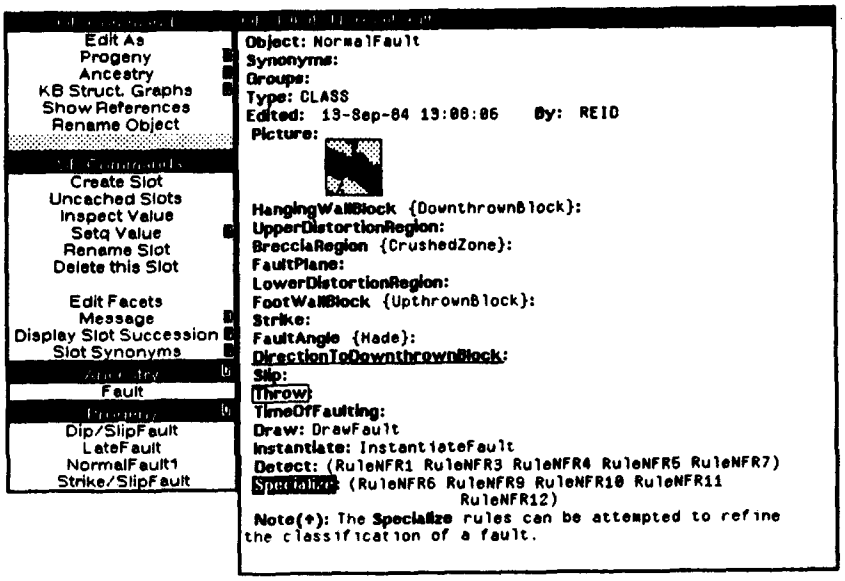


Figure 2: Object Editor

†Mark of Schlumberger.

[3]In Strobe, the Smalltalk concepts of instance variables, class variables, and methods are all encoded as slots.

The command menus enable a user to rename or destroy the object, change any of the slots, create new slots, delete or rename slots--standard editing operations. A small mark to the right of a command indicates the existence of subcommands that extend the functionality of that command. We have adopted the discipline that a *left* mouse button selection invokes the command shown, while a *middle* mouse button selection pops up a menu of subcommands.

Two menus show the objects most closely related (taxonomically) to the object being edited; they can be used to invoke a new editing context in which one of those relatives is the center of attention.

Items may be mouse-selected in the display as arguments to a command. Selections (or foci) are indicated by different styles of highlighting. As with other defaults, the highlight styles may be changed by the user.

### 3.2 THE IMPULSE-86 SUBSTRATE

The substrate contains five major building blocks: *Editor, EditorWindow, PropertyDisplay, Menu,* and *Operations.* Each building block, or part, is a Strobe object in the Impulse knowledge base. Each embodies special knowledge that enables it to fulfill a particular role in an interface (these are described below). The user of Impulse-86 can customize and extend the behavior of an existing interface by modifying or specializing the structure of some of its building blocks.

Some parts are *instantiable*—a new instance is created each time the part is used; *non-instantiable* parts are analogous to re-entrant code—they contain no changeable data, so separate instances would be redundant. Instantiable parts have an *editee*—the domain focus—typically a part of the knowledge base being edited.[4]

**Editor:** The editor is the central object. It mediates interactions between the user and the editee—its parts constitute the interface. Editor *instances* are cloned from an editor *class* by template instantiation. By instantiating a separate editor for each editee, Impulse-86 enables an unlimited number of independent interaction contexts to exist simultaneously.

Editors have components drawn from any of the five major classes (or from additional classes defined by a user). Editors are explicitly permitted to have other editors as components (and so on, in a recursive fashion). This enables the construction of an editor whose subeditor structure parallels the substructure of its editee. Knowledge about the way composites are structured in the application domain may therefore be embedded in the structure of its interfaces. We will later consider some of the ways in which this knowledge can be put to use.

Figure 3 shows the editor structure for the class of object editors. Each object editor instance is generated from this template. Figure 4 shows the instance corresponding to the object editor shown in figure 2. In both graphs, an arc indicates that the object on the right is a *component* of the object on the left. Instantiated components are shown in lightface; non-instantiated components are in boldface.

In our example, `ObjectEditor` and `SlotEditor` are both editors. Each Strobe object has associated properties (*e.g.*, name, synonyms) and a set of slots. The `ObjectEditor` mediates interactions with the object-specific properties, while the `SlotEditor` mediates interactions with the slots.

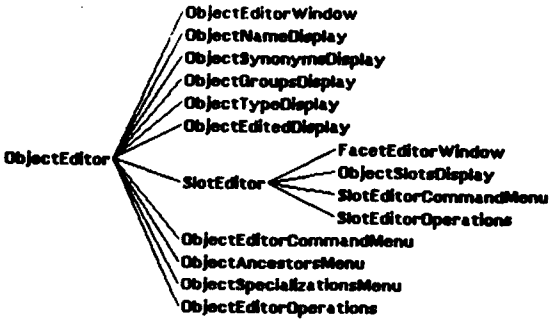


Figure 3: Components Structure of the Object Editor

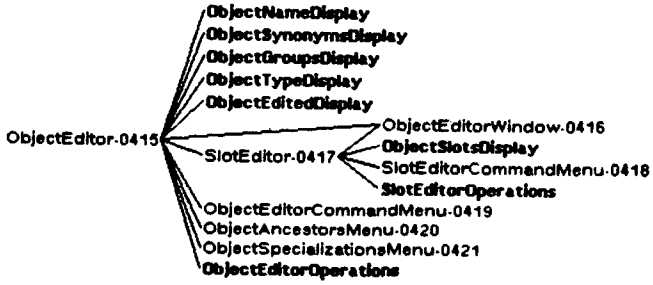


Figure 4: An Object Editor Instance

**EditorWindow:** The editor window manages the screen context of a collection of editors. It is responsible for performing the usual window operations (*e.g.*, scrolling, repainting, reshaping). It also maintains a correspondence between editees and the window regions in which

[4] Analogous to the Smalltalk *model* [1].

the editees are displayed. This enables both data selection by mouse pointing and efficient update when parts of an overall domain structure are changed. The window also records *foci*—items selected by the user as arguments to a command.

Each editor may have at most one window among its components. When an editor has subeditors, some or all may share the same window, or utilize separate windows. Figure 4 shows that `ObjectEditor` and `SlotEditor` instances both share the same instance of `ObjectEditorWindow`.

**PropertyDisplay:** The property display presents a view of an editee in a window. Impulse-86 has a number of different kinds of display, each implementing a distinctive visual style—and a user can define new types of display. A property display sets up the correspondence between its editee and the window regions in which the editee is displayed. As noted above, the correspondence is maintained by the window. A property may also be *active* (*i.e.*, the display that it produces may contain a region (or set of regions) sensitive to mouse selection).

There are six displays in the object editor example. The first five are components of `ObjectEditor`; each displays one of the five properties associated with the object itself. The sixth (`ObjectSlotsDisplay`) is a component of `SlotEditor`; it iterates over all of the slots.[5] Each of these displays prints a single line in the `ObjectEditorWindow`.

**Menu:** Menus are messengers between the user and an editor. In this capacity, menus display information in a restricted format, and may invoke interactions via mouse selection. Menus may be unique to a particular editor or shared among a collection of editors. Impulse-86 provides a large number of built-in menu styles, ranging from the static menus shown in figure 2 to pop-up and pushbutton menus.

The `ObjectEditor` has one command menu and two other menus that display the immediate relatives of the editee object (from which a new editing context can be invoked). The `SlotEditor` has one command menu. When a selection is made in a command menu, the menu sends a message to its associated editor, requesting the selected command. The responsibility for executing the command lies with the editor.

The separation between `ObjectEditor` and `SlotEditor` is used to advantage for indicating which commands are appropriate to a user-selected item in the editor window. For example, when an object-specific property has been selected, Impulse-86 grays over the menu of slot-related commands. This helps to focus the attention of the user on the relevant commands.

**Operations:** Methods that perform the commands defined for an editor are grouped in operations objects. These methods are invoked by a message from the editor.

`ObjectEditorOperations` knows how to execute the commands listed in the `ObjectEditorCommandMenu`; `SlotEditorOperations` knows how to execute the commands listed in the `SlotEditorCommandMenu`. For example, when the `SlotEditor` receives a message that the user buttoned `Rename Slot`, it relays that message to `SlotEditorOperations`, which has a method that actually renames the slot.

In addition to the five major building blocks, there are a few minor building blocks. For example, Impulse-86 provides high-level support for keyboard interaction through *TTY-InteractionWindows*. Audio output, which we have only begun to explore, also falls into this category.

The number of building block classes is less important than the organizing principle: *Partition Responsibility*. While the editor object is responsible for mediating interaction, we distinguish between *view* and *control* (or *change*) activities. We further partition responsibility for view activities into *display* and *locus*, corresponding to the roles of property displays and windows. We partition responsibility for control activities into *select* and *execute*, corresponding to the roles of menus and operations. Separating menus from the operations they invoke permits us to invoke the same operations via menu, typein, function invocation, or message from a remote processor.

Our strategy is to maximize opportunities for sharing user interface code. The correct partition pays off by virtually eliminating duplication of effort. This also encourages consistency among user interface designs [7,15].

## 4 ADDITIONAL EXAMPLES

We mentioned earlier that developers and end-users of knowledge-based systems interact with their systems at the three levels of (*i*) individual objects and their structure, (*ii*) relationships among objects, and (*iii*) the behavior of whole systems. The object editor is one example of a tool for editing at the object level. In this section we illustrate the range of extensibility provided by Impulse-86. We present a graph editor for interacting at the relationship level, and a dataflow editor for the system behavior level.

[5] Each display is typically responsible for one editee. However, displays (indeed, any component) can be iterated over a list of editees.

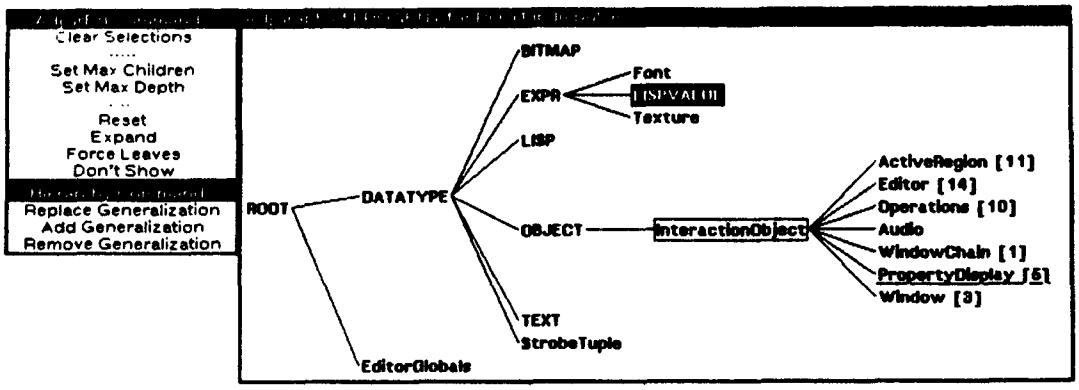


Figure 5: Graph Editor: Specialization Relationships in Impulse

### 4.1 GRAPH EDITOR

Graphs are a natural representation for viewing relationships between and among several objects in a system. Impulse-86 provides a simplified, object-oriented interface to the Interlisp-D GRAPHER package [2], so users can edit relationships from a global viewpoint.

A graph editor displays some or all of the objects tied together by a relationship; the graph editor in figure 5 shows a taxonomic hierarchy, with inheritance flowing to the right. Each node represents one object; numbers in square brackets indicate the number of specializations not shown. Graph editors are like other editors with regard to defaults and foci.

The graph editor enables the user to see and rearrange the global structure of the system as implied by the graphed relationship. The user can add or delete links, and can easily invoke an editing context that shows more detail for any object in the graph.

The graph editor is a specialization of the object editor. It has a new kind of property display (to display the graph itself), two new static command menus (shown on the left of the editor window in figure 5), and a new set of operations. A user may further customize the graph editor with respect to fonts, layout (*e.g.*, vertical vs horizontal), link types (*e.g.*, dashed vs solid), node display (*e.g.* bitmaps vs names), and so on. These customizations require no more than simple declarative changes to default values. In addition, Impulse-86 allows for construction of *KB Structure Graphs*. To construct such a graph the user specifies a function that, given an object, will generate its successor objects one *graphic link* away (*e.g.*, according to a *part-of* relationship as in figures 3 and 4, or a taxonomic relationship as in figure 5). Impulse-86 takes care of the mechanics of displaying the transitive closure of the graph generator function, together with the rest of the interface.

### 4.2 DATAFLOW EDITOR

The dataflow editor (figure 6) is a more radical departure from our original object editor example. It uses animated icons to convey the dynamic behavior of a Stream Machine[6] dataflow program. The user sees all the pieces of a Stream Machine program: large rectangles represent computation modules, rows of small boxes represent data streams, arcs represent read and write connections between modules and streams, dots tag the stream item currently being read, and characters printed inside the small boxes show data values on the streams. Stream Machine programs are created and modified in this graphical language. The user connects modules and streams by connecting their icon counterparts on the screen.

As the program runs, the picture is repeatedly updated to show the new data values on the streams, the positions being read, and which modules are blocked waiting for data. The user can invoke new interaction contexts to see details of any computation module, stream or read/write connection.

The dataflow editor was implemented by creating a number of new property display objects (*e.g.*, lines, boxes), together with a set of menus and operations specific to animation. The Impulse-86 property display and editor window objects already support the notion of a correspondence between editees and two-dimensional screen regions.

[6]The Stream Machine [8] is a coarse-grained dataflow system in which programs consist of several concurrently executing modules communicating via streams of data. The modules are sequential programs.

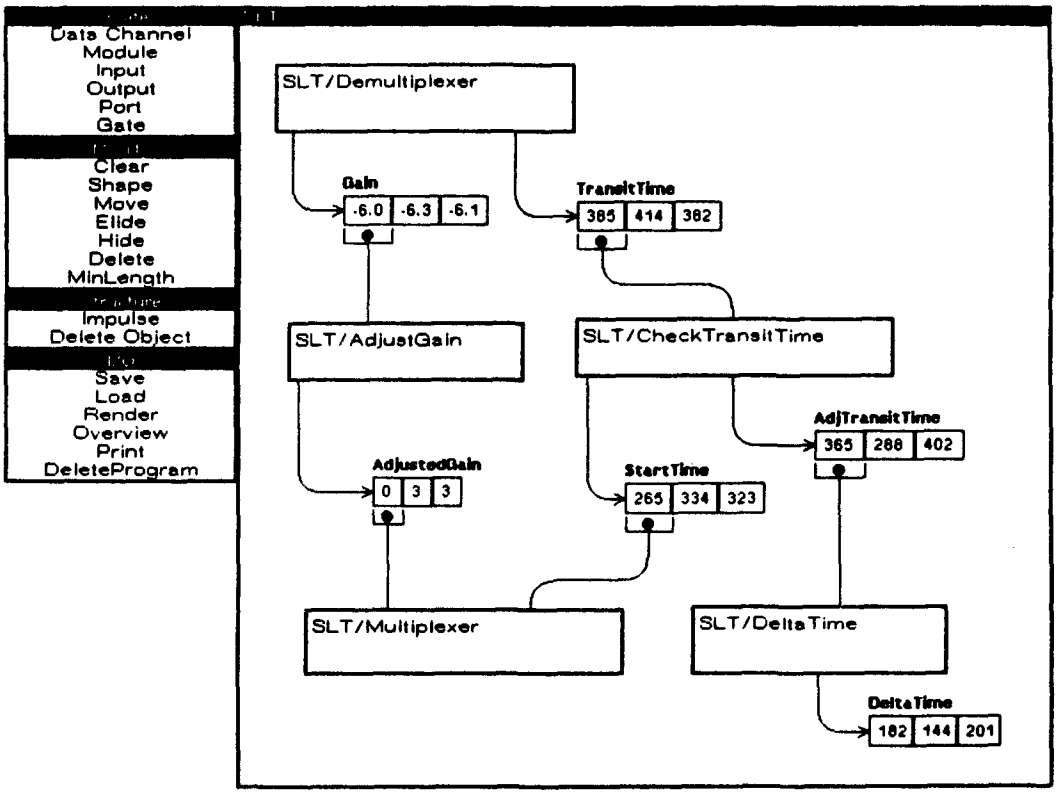


Figure 6: Dataflow Editor

## 5 EFFECTIVENESS OF THE SUBSTRATE

One of our goals for Impulse-86 is that the substrate make it easy to build *many* domain-specific interfaces. The object editor, graph editor and dataflow editor are all implemented on a single substrate. They seem very different on the surface. The object editor looks at a single object, the other two look at several objects as bound by one or more relationships. The graph editor employs a pre-existing package for manipulating graphs, the other two are independent of outside packages. The dataflow editor dynamically animates the functioning of a system, the other two do not.

Yet there is much that all three examples share. Each creates and maintains interaction contexts. Each displays its editee to the user through windows. Each receives commands from the user through menus and/or typein. Each executes commands that control the state of its editee.

These commonalites and differences are captured by Impulse-86. The commonalites are reflected by the major building blocks of the Impulse knowledge base; the differences are evident from the number of objects under each major class.[7]

Impulse-86 users have built a variety of interfaces that fall into three main classes. The object editor is a member of the general class of structured object editors. Other members of this class are specialized for particular kinds of objects in various domains (*e.g.*, a rule editor for rules written for Strobe's rule interpreter [3], a module editor for declaratively specified tasks [15]).[8] The graph editor is the parent of a number of specializations; each implements a different strategy for handling information overload. The dataflow editor was generalized to a subsystem of Impulse-86 called GROW [9]. GROW has been used to build animated editors for other domains.

It is noteworthy that each customized interface typically required only a few hours to build, and none required more than a week. Most were built by modifying existing Impulse-86 editors or by connecting existing parts.

We attribute the short development times for these interfaces to the particular set of behaviors encapsulated in Impulse-86 building blocks. The designer is insulated from the details of manipulating bitmaps, windows, mouse interactions, typein streams, and menus—I/O details which

[7] At the time of this writing, there are approximately 50 objects in the PropertyDisplay hierarchy and approximately 85 objects in the Menu hierarchy of Impulse-86.

[8] Impulse-86 allows a user to indicate a mapping between a class of objects in a domain knowledge base and an editor class. The mapping may also be dynamically determined.

traditionally require painstaking attention. Attention can therefore be concentrated on what the customized interface should look like, what information it should show the user, and what commands the user should be able to request.

The correct design for a specialized interface depends on the specific application system—it determines what is important to be able to view and control. Therefore, the developers and users of Strobe knowledge-based systems are in the best position to design interfaces customized for those systems. Impulse-86 enhances productivity by giving these people the ability to design and implement special purpose interfaces.

Three features of the Impulse-86 substrate contribute to this ability. The five classes of building blocks are quite general, covering a wide range of interaction styles. Implementation as an object-oriented system makes it easy to specialize and modify. Impulse-86 contains a number of archetypical interfaces; modifying one of these is an effective strategy for creating a new interface.

Impulse-86 *is* the standard interface to Strobe-based systems. A community of knowledge base developers and end users at Schlumberger uses Impulse-86 tools every day. In addition, Impulse-86 is used as the interface to its own knowledge base—to customize and extend it for a particular application. The toolkit is mature enough to support the end-user interface for the *Dipmeter Advisor* system (including scrolling log graphics), as well as the control-panel interface to the well-logging environment described in [15].

# 6 IMPULSE-86 AS AN OBJECT-ORIENTED PROGRAM

Impulse-86 is itself a large object-oriented program, written in Strobe. In this section we discuss the features and techniques of the object-oriented paradigm that we found especially useful in our implementation.

## 6.1 INHERITANCE

Without the modularity and inheritance provided by object-oriented programming, Impulse-86 would be inconceivable. A bushy taxonomy embodies the behavioral commonalities and differences among the five major building blocks and their specializations. As expected, inheritance keeps the size of the system manageable while enabling enormous diversity. Inheritance allows us to capture in a unified system a wide range of interaction styles possible with a sophisticated personal workstation.

## 6.2 MIXINS

Mixins via multiple inheritance are the mainstay of the Menu building block. Menu behavior is an amalgam of features that lie on several orthogonal dimensions:

- does the menu remain on the screen, or does it appear and disappear?
- does a selection from that menu indicate a choice, or invoke a command?
- how many items can the user select at once—one or many?
- does the selection remain highlighted when the user releases the mouse button?
- which Lisp package implements the menu at a low level?

Each menu combines mixins chosen from the range of possible features in each dimension. Figure 7 shows a portion of the menu taxonomic hierarchy.

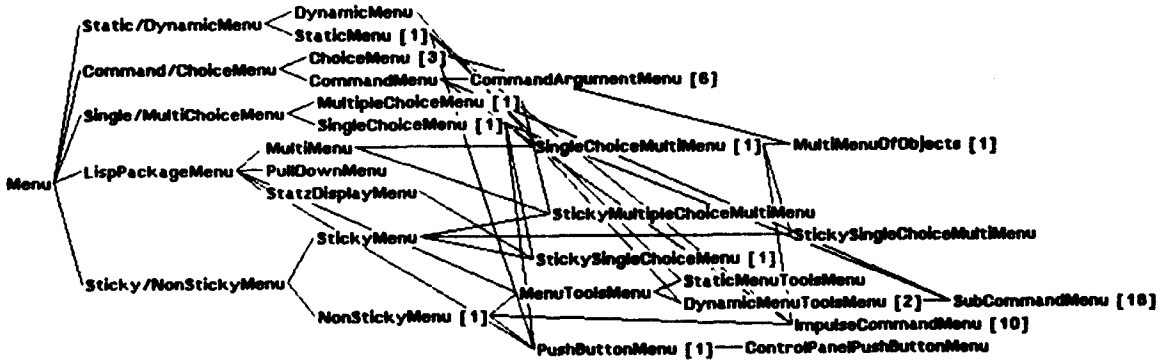


Figure 7: A Portion of the Menu Hierarchy.

Mixins are also used to make PropertyDisplays sensitive to mouse selection, and to attach menus to certain PropertyDisplays.

### 6.3 POLYMORPHISM

The ability of different objects to respond to the same protocols with varying behavior is called polymorphism [6]. Impulse-86 utilizes a uniform set of protocols over all five classes of building blocks. This enables extremely general code for starting up an interaction context, putting windows and menus on the screen, printing information in windows, and so on.

Further polymorphism is evident among the building block classes. There are several dozen kinds of Menu that can be used in Impulse-86 editing tools. Each presents a different interface to the user, but all share the same protocols (*e.g.*, creation, display, mouse selection). Similarly, there are dozens of PropertyDisplays. Each prints information to the screen in a different format, but they too share protocols (*e.g.*, positioning, printing).

Each successive revision of Impulse has had a finer grain size for protocols than its predecessor. As we continue to separate functions that were previously bundled together, more protocols are added to the repertoire. This unbundling increases the generality of the interface tool kit; the number of possible behaviors grows combinatorially. At the same time, inherited default values tame this explosion, so the designers need not face an overwhelming number of choices at every turn.

Consider the subclasses of PropertyDisplays that indent, print a caption in one font, print a value in another font, and finally emit a new line. Originally, these responded to a single protocol that formatted the datum and printed the line in a window. This was unbundled by creating new protocols—one to indent by a specified length, one to print the caption, one to print the value, one to emit a new line if required, and one to invoke some combination of the others. Where there was once a single possible behavior, there are now many; each behavior is specified by a combination of choices from four dimensions (*i.e.*, indent, caption, value, and new line). One result is the ability to more readily handle intermixed single and multiple column displays. Moving to a finer granularity helped expose these further opportunities for polymorphism and extensions to functionality.

## 7 RELATED WORK

Other groups are currently working on systems that support construction of interactive interfaces. The GUIDON-WATCH system [10] demonstrates the utility of a user interface tuned to the operation of a particular class of consultation systems. The authors note that knowledge-base editors originally intended for use by the knowledge base developer/maintainer are typically inappropriate interfaces for the end user. Impulse-86 offers a substrate that bridges the gap between the tools required by a developer/maintainer, by a domain specialist, and by an end user. Its extensibility further enables it to support the construction of specialized interfaces for each type of user.

The SIG system [5] is much closer to Impulse-86. Built on top of Smalltalk-80 [1], it too offers an extensible kernel that supports generation of interactive displays. In the terminology of [5], SIG emphasizes the *view* aspect of interaction; it addresses the *control* aspect in a less structured manner. In contrast, we have found it useful in Impulse-86 to provide a considerable amount of structure to support the control aspect of interaction as well as the view aspect. We have also found it useful to provide a relatively fine-grained structure—numerous behaviors, organized into five categories—to support user extension. Finally, we have found it useful to make the editor the kernel concept that unifies both the view and control aspects of user interaction.

Closest to Impulse-86 in spirit is EZWin [4], an object-oriented editing system which provides three object classes for constructing editors. The *EZWin* class corresponds to an Impulse-86 *Editor* object. This object represents the entire editor structure, including the window, a process for handling interaction, and the various *Command* and *Presentation* objects available in the editor. Command objects correspond roughly to Impulse-86 `Menu` and `Operations` objects; Presentation objects correspond to Impulse-86 `PropertyDisplays`. Although the two systems have similarities, there are several important differences. "EZWin systems are basically editors for graphical objects." [4, p. 186] Systems constructed in Impulse-86 are interfaces for knowledge-based systems; the interaction objects and routines are completely separate from the knowledge-base being edited. This separation allows the interface and application to be modified independently and supports the reuse of interfaces with different knowledge-based systems. Another difference between the systems is the scope of interaction management. EZWin supports interaction within a single editor, while Impulse-86 provides support for an editing session, which includes interaction with (and among) many, varied editors. For example, several editors may view the same object simultaneously (figure 1). Impulse-86 supports connected structures of subeditors and supereditors

that can be managed (i.e., created, deleted, updated) as a whole. A final difference is the separation of `Menu` and `Operations` objects in Impulse-86. Operations may be invoked by any type of interface, including menus, type-in, or even a running program. We have found this quite useful when changing interfaces to a program.

## 8 SUMMARY

Impulse-86 provides a set of building blocks for constructing a wide variety of domain-specific interfaces for knowlege-based systems. The kit contains five major building blocks which implement a full range of interaction activities. The substrate enables developers and end users of knowledge-based systems—who are not interactive graphics specialists—to design specialized interfaces for their systems. Such domain-specific editing interfaces are essential tools for developing and maintaining knowledge-based systems.

One indicator of the ease with which interfaces can be implemented in Impulse-86 is the growing collection of special purpose editors and tools to animate system behavior. We attribute this to a clear organization of interface knowledge, a flexible object-oriented implementation, and a diverse set of archetypical interfaces that a designer can customize or specialize.

## ACKNOWLEDGEMENTS:

Impulse-86 is a direct descendant of the first two versions of Impulse, both of which were designed and implemented by Eric Schoen. Bob Young and Mike Kleyn made numerous suggestions for the new design. David Barstow, Tina D. F. Dinitz, Sol Greenspan, Elaine Kant and Bob Young read drafts of this paper, adding greatly to the coherence and flow of the final manuscript.

## REFERENCES

[1] Adele Goldberg. *SMALLTALK-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA., 1984.

[2] *Lisp Library Packages Manual.* Xerox Artificial Intelligence Systems, Pasadena, CA, December 1985.

[3] G. M. E. Lafue and R. G. Smith. A modular tool kit for knowledge management. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 46–52, August 1985.

[4] H. Lieberman. There's more to menu systems than meets the screen. *Computer Graphics*, 19(3):181–189, July 1985.

[5] D. Maier, P. Nordquist, and M. Grossman. Displaying database objects. In *Proceedings of the First International Conference on Expert Database Systems*, pages 15–30, April 1986.

[6] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: themes and variations. *AI Magazine*, 6(4):40–62, 1986.

[7] D. A. Norman. Design principles for human-computer interfaces. In *Proceedings of the CHI 1983 Conference on Human Factors In Computer Systems*, pages 1–10, ACM-SIGCHI, December 1983.

[8] Paul Barth and Scott Guthery and David Barstow. The Stream Machine: a data flow architecture for real-time applications. In *Eighth International Conference on Software Engineering*, pages 103–110, London, England, September 1985.

[9] Paul S. Barth. Grow: an object-oriented approach to graphical editing and animation. 1986. (to appear).

[10] M. H. Richer and W. J. Clancey. GUIDON-WATCH: a graphic interface for viewing a knowledge-based system. *IEEE Computer Graphics and applications*, 5(11):51–64, 1985.

[11] E. Schoen and R. G. Smith. Impulse: a display-oriented editor for Strobe. In *Proceedings of the National Conference on Artificial Intelligence*, pages 356–358, August 1983.

[12] R. G. Smith. On the development of commercial expert systems. *AI Magazine*, 5(3):61–73, Fall 1984.

[13] R. G. Smith. Strobe: support for structured object knowledge representation. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 855–858, August 1983.

[14] R. G. Smith. *Structured Object Programming In Strobe.* Research Note SYS-84-08, Schlumberger-Doll Research, March 1984.

[15] R. G. Smith, G. M. E. Lafue, E. Schoen, and S. C. Vestal. Declarative task description as a user interface structuring mechanism. *Computer*, 17(9):29–38, September 1984.

[16] R. M. Stallman. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, pages 300-325. McGraw-Hill, New York, 1984.