Displaying Database Objects

David Maier, Peter Nordquist, Mark Grossman

Technical Report No. CS/E 86-001

January, 1986

Displaying Database Objects

David Maier
Oregon Graduate Center
Peter Nordquist
Intel Corporation
Mark Grossman
Oregon Graduate Center

Technical Report No. CS/E 86-001

January, 1986

Presented at the First International Conference on Expert Database Systems, Charleston, South Carolina, April 1986.

Abstract

We outline the requirements for features and construction of interactive displays on complex database objects. Few systems to date meet these requirements, as they either do not support update through the display, or are not generated automatically from a specification. We present a system, SIG, for producing and interpreting high-level display specifications for complex objects. SIG supports ***display types***, which are declarative descriptions of of interactive displays for classes of objects, and ***abstract views***, which decode and implement display types, and which can dynamically change the format of a display to accommodate changes in the structure of an object.

1. Introduction

The study of database systems has largely overlooked the display of data, even though database applications typically contain more code for data display and entry than for data manipulation [Pilo83]. Relational technology provides a very workable abstraction of secondary storage; there is no analogous abstraction for user interfaces. With the advent of object-oriented database systems that support complex objects and multiple connectivity, a fixed format for displaying the results of queries is no longer adequate. We describe the SIG system for automatically generating interactive displays on structured objects. We look at the capabilities of the system, and the main concepts behind it. SIG is currently implemented in Smalltalk-80 (TM Xerox Corp.), but the design could be ported to other systems with window support. We conclude with our vision of how database applications should be assembled.

2. Requirements

We desire a tool for creating ***interactive displays*** (IDs) on complex database objects, in the environment of a personal workstation with bit-mapped graphics. An ID must not only provide a view of its object on the screen, but also allow a user to update the object by manipulating the view with a keyboard and mouse. We wanted a tool close to the level of abstraction that relational query languages provide for data manipulation. In particular, we tried to satisfy the following requirements:

1. IDs should be described declaratively, and generated automatically from their descriptions when needed. Ideally, no modifications to the object being displayed should be needed to define an ID on it.
2. IDs should dynamically reflect the structure of the object displayed. The number and location of the subviews in an ID can depend on the state of the object displayed. For example, Figures 1a and 1b are "before" and "after" snapshots of the same ID on a binary tree when a left subtree is added. Each subtree has its own sub-ID, which can contain sub-IDs for subtrees at lower levels. Four new sub-IDs were added because the added subtree itself had three subtrees.
3. IDs should accommodate arbitrary levels of structure in the objects they display. There should be no ***a priori*** bound on the depth of nesting of sub-IDs of a display. IDs should also accommodate multiple connectivity in objects, in that the same object can appear in multiple IDs, if it is a subpart of several objects. Updates to the object through one of these IDs should be reflected in them all.

4. The system should support multiple display descriptions for a single type of objects. Figures 2a and 2b show two IDs for the same binary tree, but generated from different descriptions. Figure 2a shows a tree in a format similar to that of Figure 1, but with no arrows. Figure 2b portrays a tree in outline form, with subtrees indented below their parents.
5. The system should assist in the creation of display descriptions, and support a design methodology for building up complex IDs from simpler IDs. Once a display description is created on one class of objects, that description should be available for constructing displays for other classes.

3. Related Work

Previous works on displaying database objects and display generation mostly fail to meet our requirements because either

1. update through the display is not supported, or
2. the displays are written ad hoc, not generated automatically from descriptions.

The INCENSE system [Myer83] generates displays on data structures in the Mesa language. Each data type has an *artist* to render instances of the type on the screen, but data cannot be updated through the display. The Application Development Environment (ADE) being constructed at Burroughs [Ande85] supports *perspectives* for defining a printable format of a data item. Perspectives build up a rendering of a complex object from displays of its subparts, but do not provide for update of objects. The specifications of perspectives are stored as part of the database. Several researchers have proposed high-level languages for drawing pictures [Egge83, Hend82, Pere83, VanW82], but again, these languages provide rendering but not updating of objects. Also, certain document preparation systems [Furu82, Kimu83] can be viewed as displaying some document object on screen or paper, but without update capabilities.

One of the first graphical display systems for databases was the Spatial Data Management System [Hero80]. Another such system is included in the Sembase semantic database system [King84]. Sembase provides displays that dynamically keep up with added objects and changed attributes. Both systems, however, have a fixed set of formats for displaying a particular class of objects.

Program visualization systems, such as PECAN [Reis83], PV [Hero82, Kram83] and others [Fisc84], aid program development and debugging by displaying data structures associated with a program and its execution: parse tree, calling stack, variable bindings. Such systems provide dynamically changing displays, but for a fixed set of structures. Algorithm animations display the changes in an algorithm's data structures as the algorithm executes, for purposes of documentation, education and algorithms research. Such systems to date [Baec81, Brow84, Lond85] require that displays on new classes of data objects be generated manually as needed.

The Programming-by-Rehearsal system [Finz84] gives support for constructing an interactive display of a complex object from previously defined displays on its subobjects. However, each such display is constructed anew or copied from an existing display, not derived from a higher-level specification.

One example of where an interactive display has been constructed from a high-level specification is in the Crystal system for interpreting oil-well logs [Smit84]. For that project, the IMPULSE editor [Scho83] (used for editing knowledge bases described in the STROBE object representation language) was modified to accept declarations of specialized editors, and to produce editors from the declarations.

4. The Implementation Vehicle

The Smalltalk Interaction Generator (SIG) [Nord85] is a prototype ID generator for complex objects, as might be found in an object-oriented database system. SIG builds upon the model-view-controller (MVC) mechanism of Smalltalk [Gold83]. An MVC is a triad consisting of three objects: *model* to be displayed and possibly updated, a *view* responsible for displaying the model on the screen, and a *controller* for interpreting user inputs as updates to the model. The model can be an arbitrary Smalltalk object, but the view and controller must be appropriate to the class of the model. For example, different flavors of views exist for text, list and Boolean objects. A simple ID is constructed from a single MVS triad. An ID with sub-IDs uses an MVC triad for each sub-ID. The structure of the ID is maintained by the view objects, which know about their subviews.

SIG actually makes use of an enhancement of the MVC paradigm known as ***pluggable views***. A pluggable view queries its model to determine the ***aspect*** of the model it should display. Pluggable views cut down the proliferation of types of views. Rather than constructing a new view class for each format and class of model, one view class suffices for displaying a certain aspect, such as the text, of any model. Pluggable views also make interactions among displays easier to program, as several displays can be linked by being views on different aspects of the same model. In a complex display, pluggable views cut out unnecessary redisplay, as a subview need repaint only if its aspect of the model changes.

The MVC mechanism, while providing many features needed for the prototype, does not by itself meet all the requirements for an ID generation system:

1. Defining an ID with the MVC mechanism is a procedural, not declarative, task. To produce an ID, the programmer must write code to sire together views of existing types, and possibly code new view classes.
2. IDs in the MVC paradigm usually have a fixed format; they do not dynamically adapt to changes in the structure of the model. Most Smalltalk-supplied IDs have only a two-level hierarchy of subviews, and do not add or delete views while in use. Nothing prevents a Smalltalk ID from changing the number of subviews it has, but no tools are supplied to help construct dynamic IDs.
3. The MVC mechanism does not lend itself to a modular design methodology. Most IDs are a mass of interrelated pieces that are not easily separated. In adapting an existing ID, it is often difficult to locate the right piece of code to change a particular facet of the display. Design questions concerning which part (model, view or controller) should implement which functions are difficult to resolve. In the Smalltalk-supplied IDs, sometimes the view and controller store data to be displayed apart from the model, and the controller does some of the updating of the display.

5. SIG Capabilities

In database terms, Smalltalk objects are non-1NF tuples that have identity and that can share attribute values. ***Classes*** group objects with similar structure and behavior. An object is an ***instance*** of its class. Every Smalltalk object has a ***protocol*** of ***messages*** to which it responds by changing its state or returning information. The protocol to an object serves to encapsulate its internal state. An object may not, as a rule, manipulate or examine the internal state of another object directly, but may only do so indirectly through a message.

SIG uses a display description, called a ***display type***, to define an ID for an object. Display types are associated with classes, and a class can have multiple display types, as witnessed by the various IDs on binary trees in Figures 1 and 2, which were defined in SIG. The IDs generated by SIG can adapt to changes in the state of the object displayed, as shown in Figures 3a and 3b. Those figures show an ID on an object of a class BooleanListTest. An instance of BooleanListTest contains a Boolean value and a list of strings. The list is only displayed when the Boolean value is true.

The next examples involve a class Employee. Instances of class Employee have a name and address field, a social security number field, a projects field, and a manager field. The first two fields hold strings, the project field holds an object from the class ProjectList, and the manager field holds another Employee object. A ProjectList is an array of projects.

Figure 4 shows a SIG-generated ID on an Employee object. It has four sub-IDs, one for each field, plus some labels. The labels are also IDs, but with no update capabilities. The sub-IDs on project and manager are further structured. The sub-ID on project was generated from a display type for the class ProjectList, and the sub-ID on manager was generated from the same display type that was used for the entire ID. Thus, display types can reference other display types. In particular, a display type can mention itself recursively.

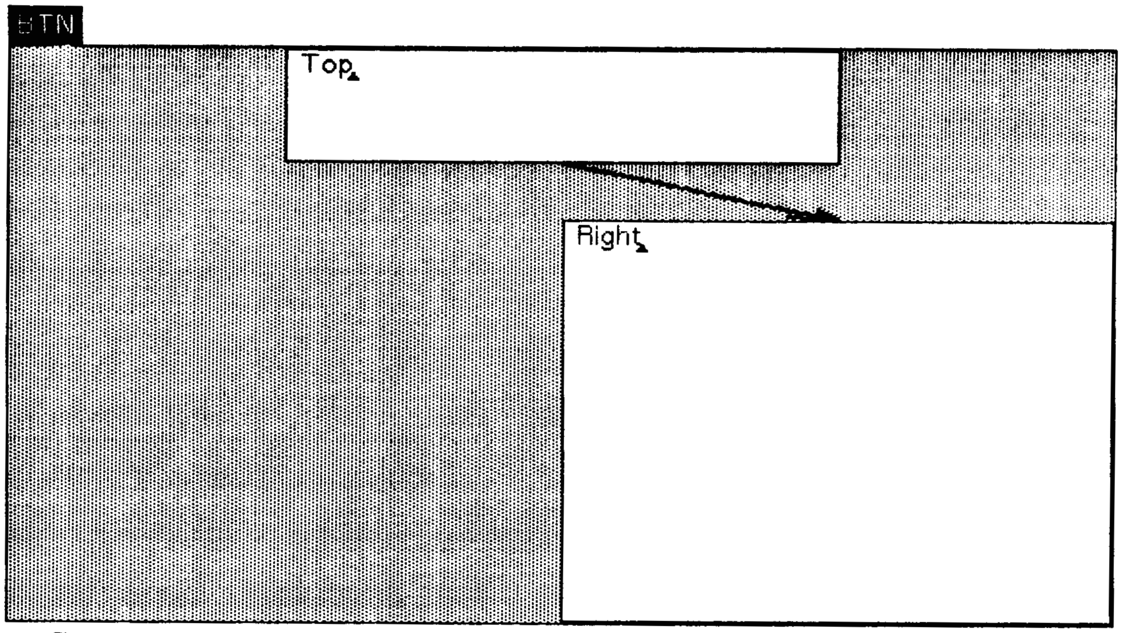


Figure 1a. Tree ID before adding left subtree.

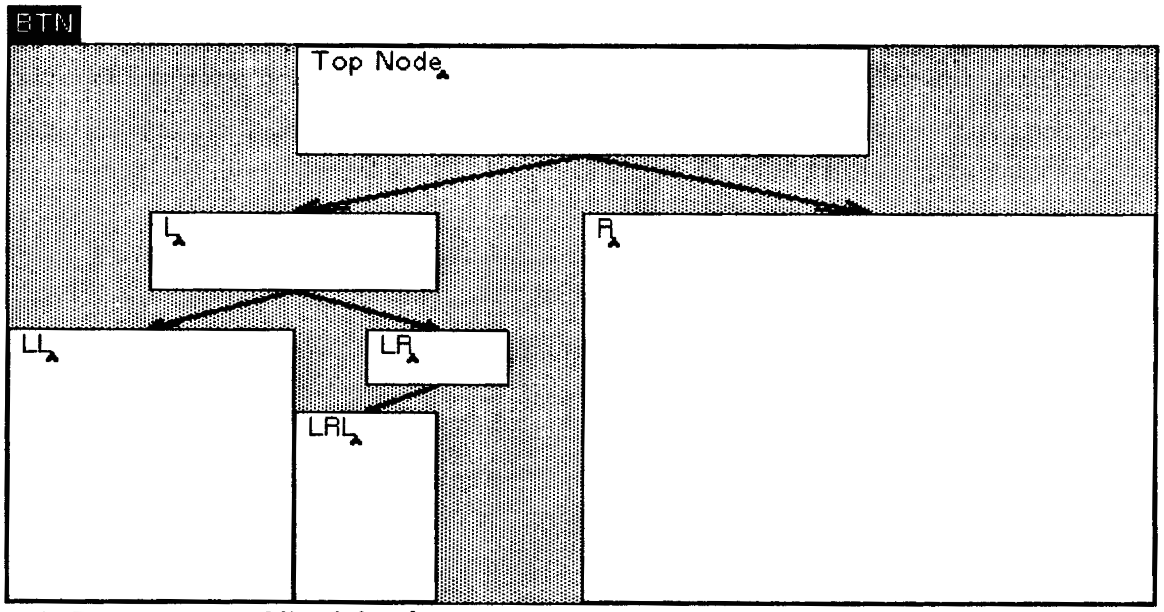


Figure 1b. Tree ID after adding left subtree.

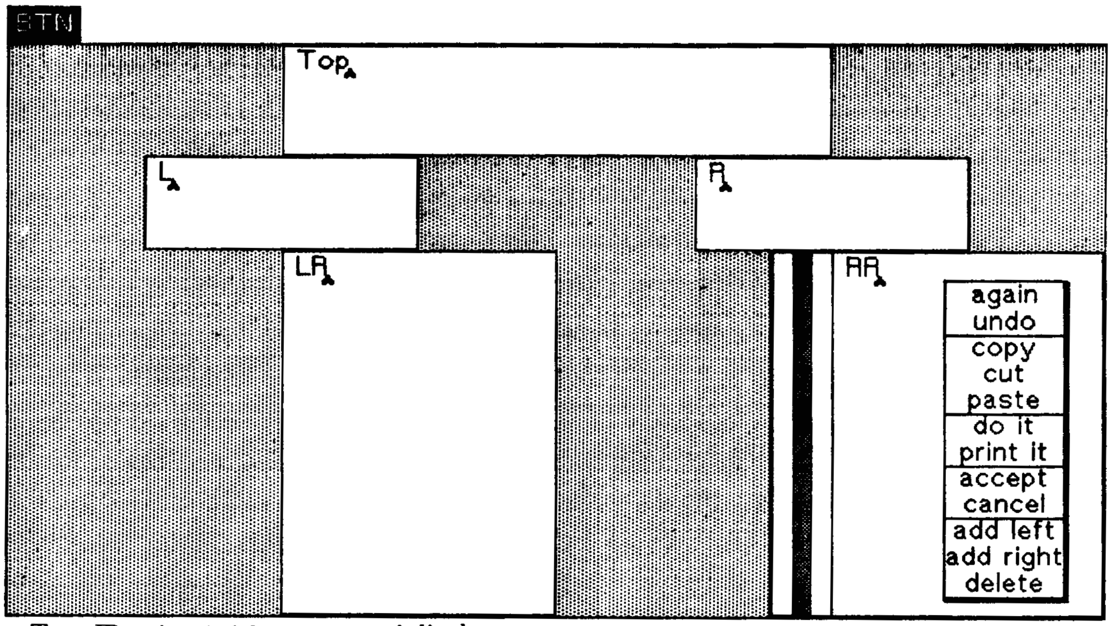


Figure 2a. Tree ID using 'without arrows' display.

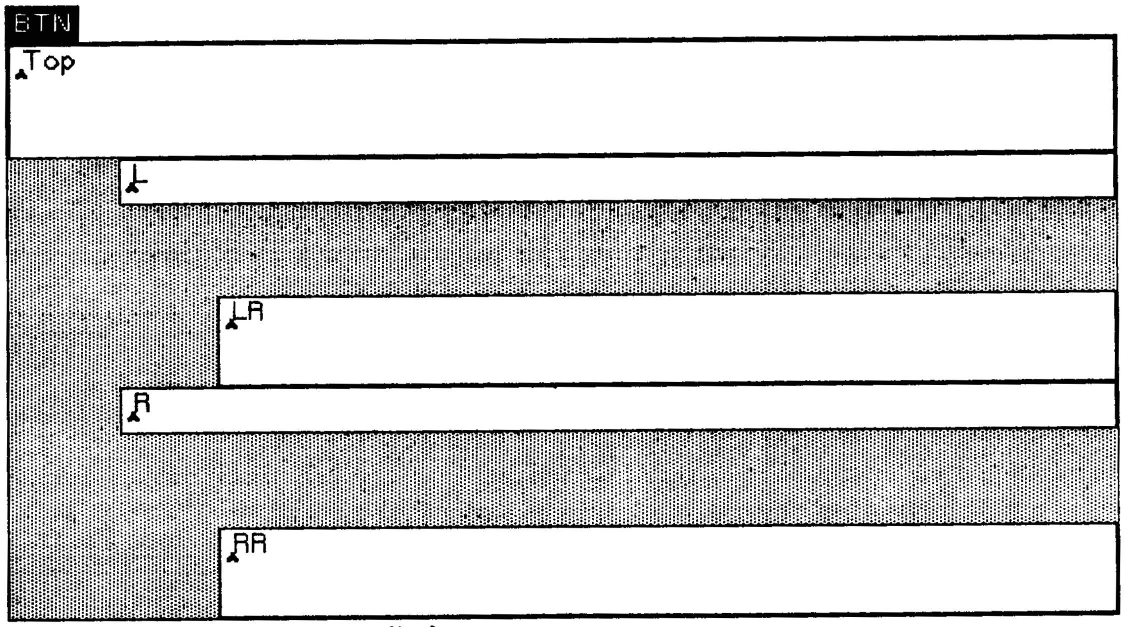
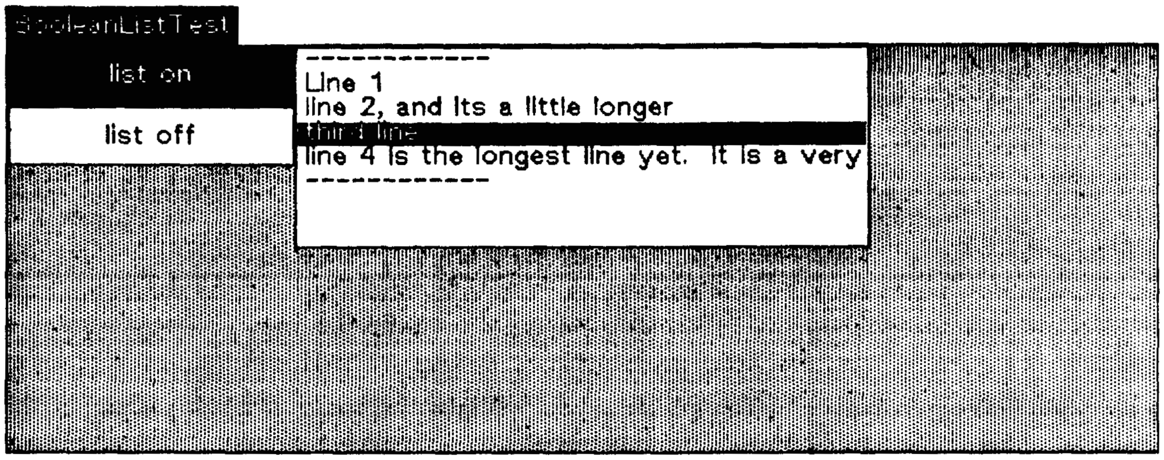


Figure 2b. Same tree ID using 'outline' display.



Line 1
line 2, and its a little longer
third line
line 4 is the longest line yet. It is a very

Figure 3a. BooleanListTest ID with list on.

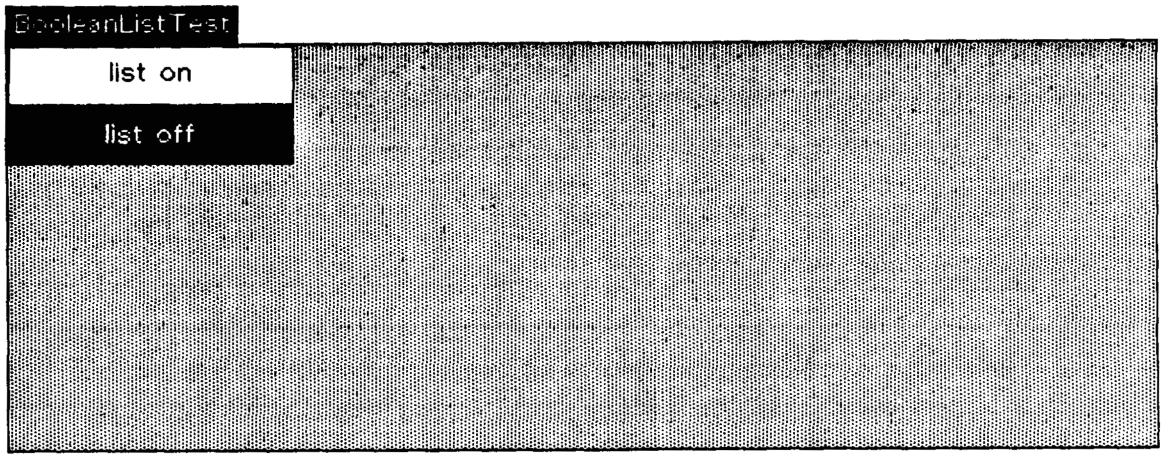
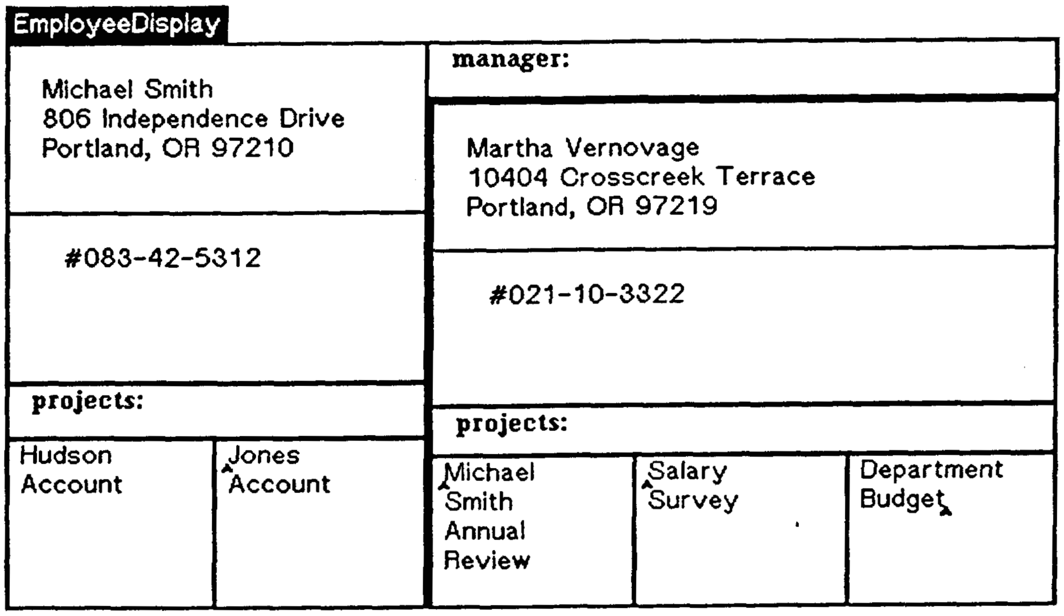


Figure 3b. BooleanListTest ID with list off.



Michael Smith
806 Independence Drive
Portland, OR 97210

#083-42-5312

Hudson Account

Jones Account

Martha Vernovage
10404 Crosscreek Terrace
Portland, OR 97219

#021-10-3322

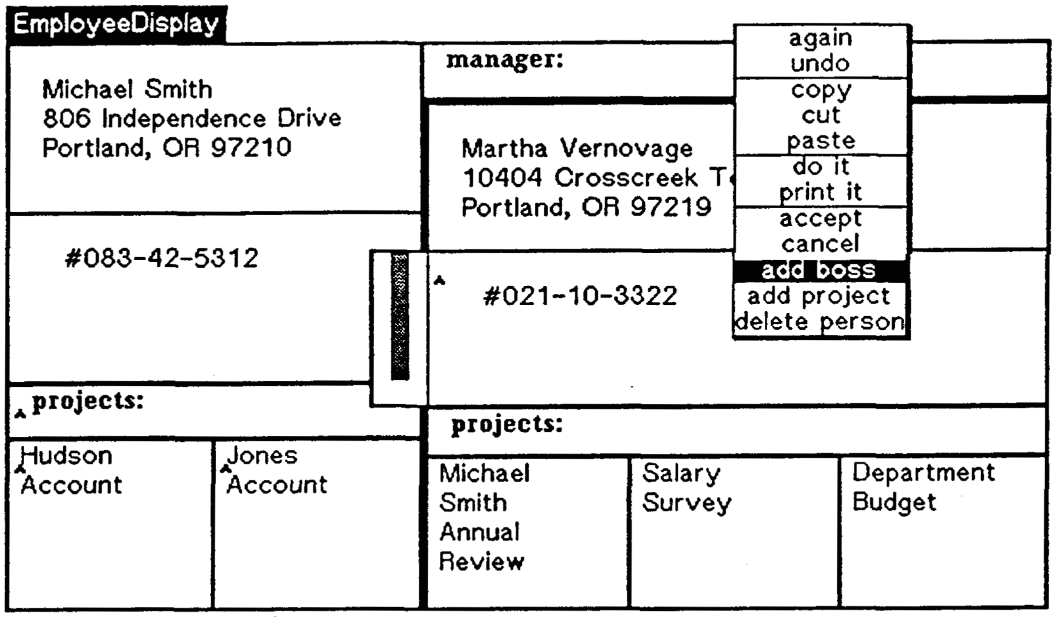
Michael Smith Annual Review

Salary Survey

Department Budget

Figure 4. ID on an employee object.

Part of a display type is a specification of a menu of update commands on each sub-ID. By associating a menu with sub-IDs rather than with the display type itself, one display type can support different kinds of update behavior in different contexts. In Figure 5a we see a menu in use to add a manager for the manager, thus updating the underlying employee object. That update to the employee object in turn causes the ID to change by adding a new sub-ID. In Figure 5b the new manager appears, and we can then add new information about him (Figure 5c).



Michael Smith
806 Independence Drive
Portland, OR 97210

#083-42-5312

Hudson Account

Jones Account

Martha Vernovage
10404 Crosscreek T
Portland, OR 97219

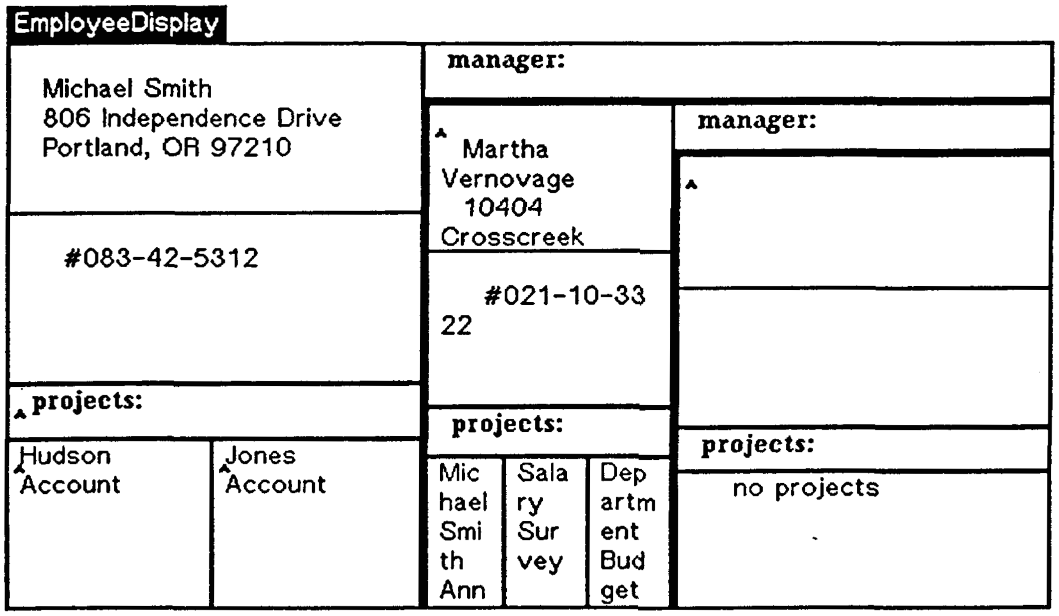
#021-10-3322

Michael Smith Annual Review

Salary Survey

Department Budget

Figure 5a. Adding the manager's manager.



Michael Smith
806 Independence Drive
Portland, OR 97210

#083-42-5312

Hudson Account

Jones Account

Martha Vernovage
10404 Crosscreek

#021-10-33
22

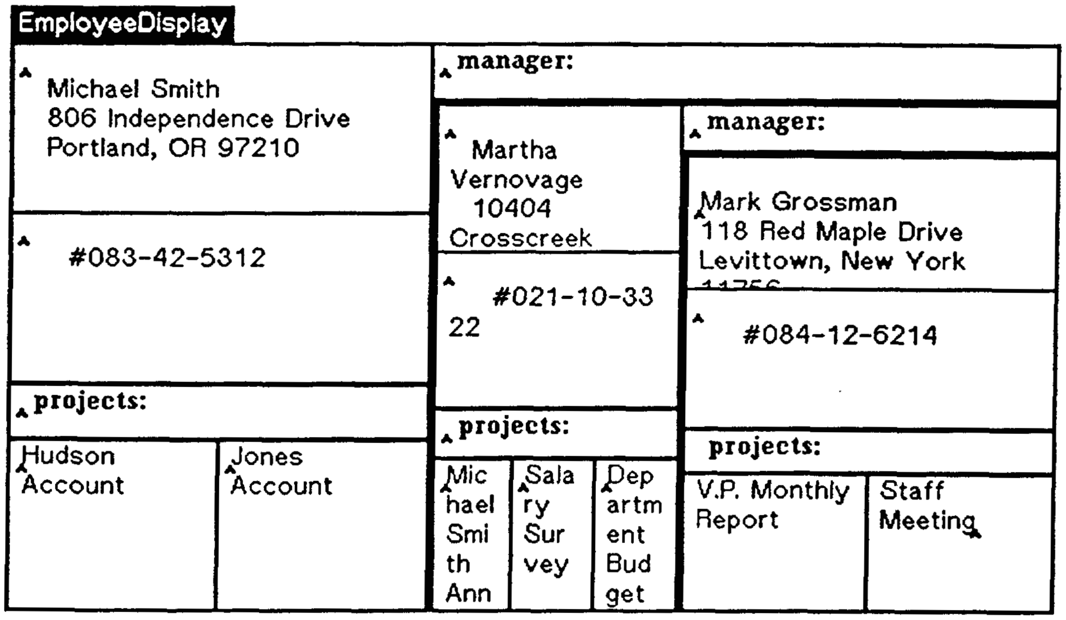
Mic hael Smi th Ann

Sala ry Sur vey

Dep artm ent Bud get

no projects

Figure 5b. New manager, uninitialized.


EmployeeDisplay

Michael Smith
806 Independence Drive
Portland, OR 97210

#083-42-5312

projects:

Hudson Account

Jones Account

manager:

Martha
Vernovage
10404
Crosscreek

#021-10-33
22

projects:

Mic
hael
Smi
th
Ann

Sala
ry
Sur
vey

Dep
artm
ent
Bud
get

manager:

Mark Grossman
118 Red Maple Drive
Levittown, New York

#084-12-6214

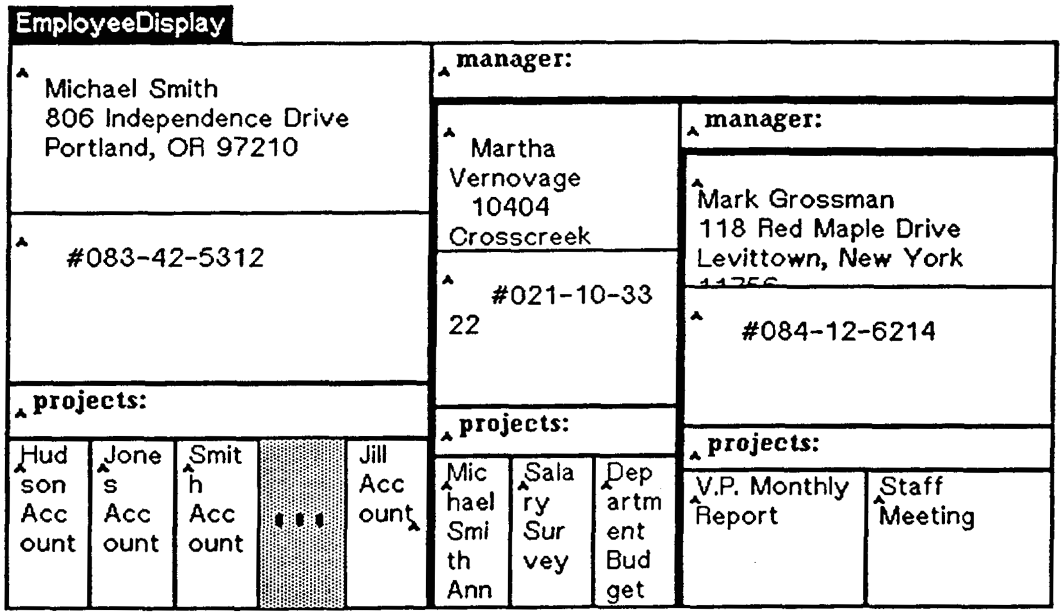
projects:

V.P. Monthly Report

Staff Meeting

Figure 5c. New manager with information filled in.

SIG IDs can handle iteration in limited forms. The sub-ID on projects can adapt to different numbers of projects, but once the number grows past four, some of the projects are replaced by ellipses (Figure 6). Figure 7a shows that we can have two IDs open on the same object. (One of the IDs is a sub-ID of the ID in Figure 6.) If the underlying object is updated through one of the IDs (Figure 7b), the update also appears in the other ID (Figure 7c). The change is not propogated immediately, as the sub-ID involved supports a kind of commit-abort mechanism on changes. The changes are actually made to a copy of the model. The model is altered only when the user commits these changes.


EmployeeDisplay

Michael Smith
806 Independence Drive
Portland, OR 97210

#083-42-5312

projects:

Hud
son
Acc
ount

Jone
s
Acc
ount

Smit
h
Acc
ount

...

Jill
Acc
ount

manager:

Martha
Vernovage
10404
Crosscreek

#021-10-33
22

projects:

Mic
hael
Smi
th
Ann

Sala
ry
Sur
vey

Dep
artm
ent
Bud
get

manager:

Mark Grossman
118 Red Maple Drive
Levittown, New York

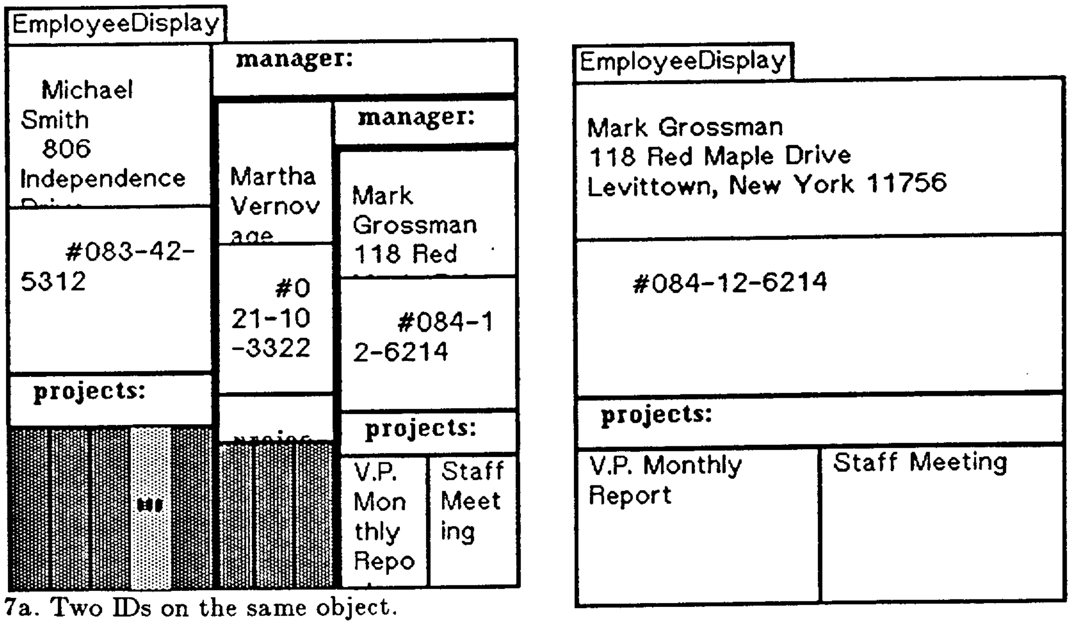
#084-12-6214

projects:

V.P. Monthly Report

Staff Meeting

Figure 6. Elision of projects.



EmployeeDisplay

Michael Smith 806 Independence

#083-42-5312

projects:

manager:

Martha Vernov

#0 21-10 -3322

manager:

Mark Grossman 118 Red

#084-1 2-6214

projects:

V.P. Mon thly Repo

Staff Meet ing

EmployeeDisplay

Mark Grossman
118 Red Maple Drive
Levittown, New York 11756

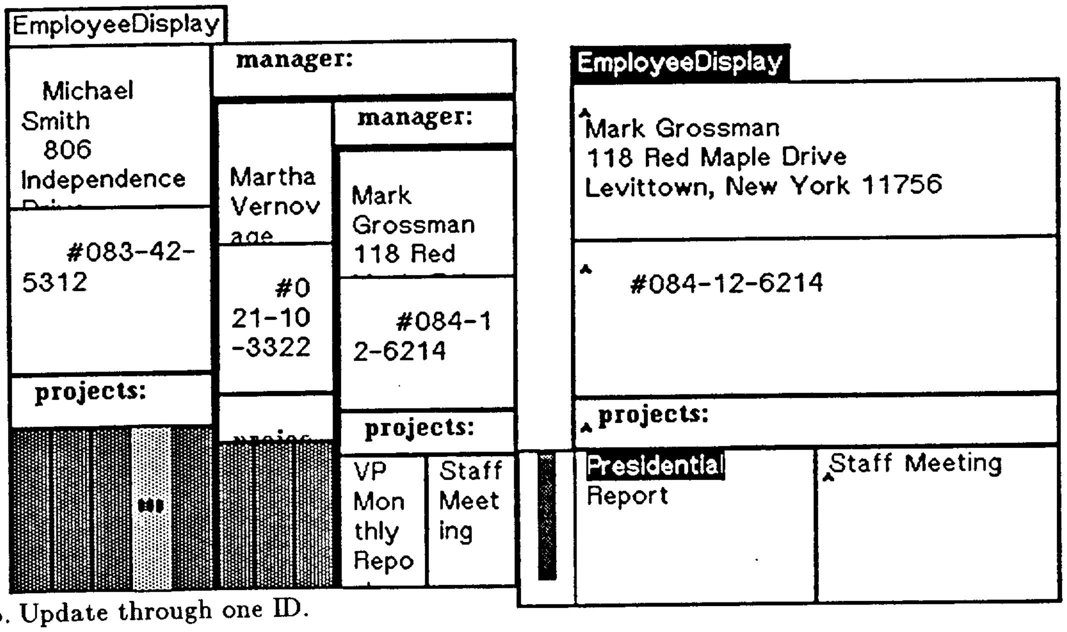
#084-12-6214

projects:

V.P. Monthly Report

Staff Meeting

Figure 7a. Two IDs on the same object.



EmployeeDisplay

Michael Smith 806 Independence

#083-42-5312

projects:

manager:

Martha Vernov

#0 21-10 -3322

manager:

Mark Grossman 118 Red

#084-1 2-6214

projects:

VP Mon thly Repo

Staff Meet ing

EmployeeDisplay

Mark Grossman
118 Red Maple Drive
Levittown, New York 11756

#084-12-6214

projects:

Presidential Report

Staff Meeting

Figure 7b. Update through one ID.

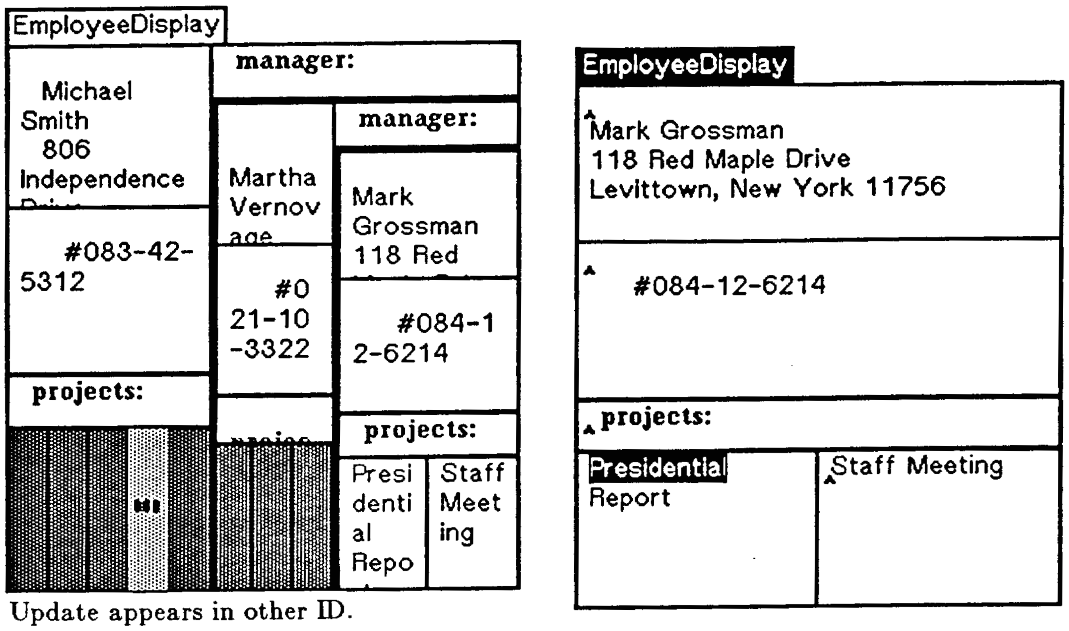


Figure 7c. Update appears in other ID.

A SIG ID can detect if the screen space allotted it is too small in which to display (Figure 8a). In that case the screen space is painted dark gray, and there is a menu option to spawn a new ID on the obscured object (Figure 8b).

Display types in SIG are created with the aid of a display-type editor, which guides a display developer through the various steps in describing a display. The display-type editor suggests a design methodology for IDs, by progression through its panes left to right, top down, filling in requested information. The editor has two modes: *display mode* (Figure 9a), for inspecting previously created displays, and *edit mode* (Figure 9b), for creating new displays, or modifying existing ones. Our experience is that adding a new class and defining the first display type on that class takes several days, for a Smalltalk user who has never constructed a display before. After that, additional display types on the same class take a half hour to four hours to develop. The display-type editor supports copying at various levels of structure in a display type. All or part of a new display type often can be concstructed by modifying a copy of an existing display type. Generating an ID from a display type is a matter of seconds.

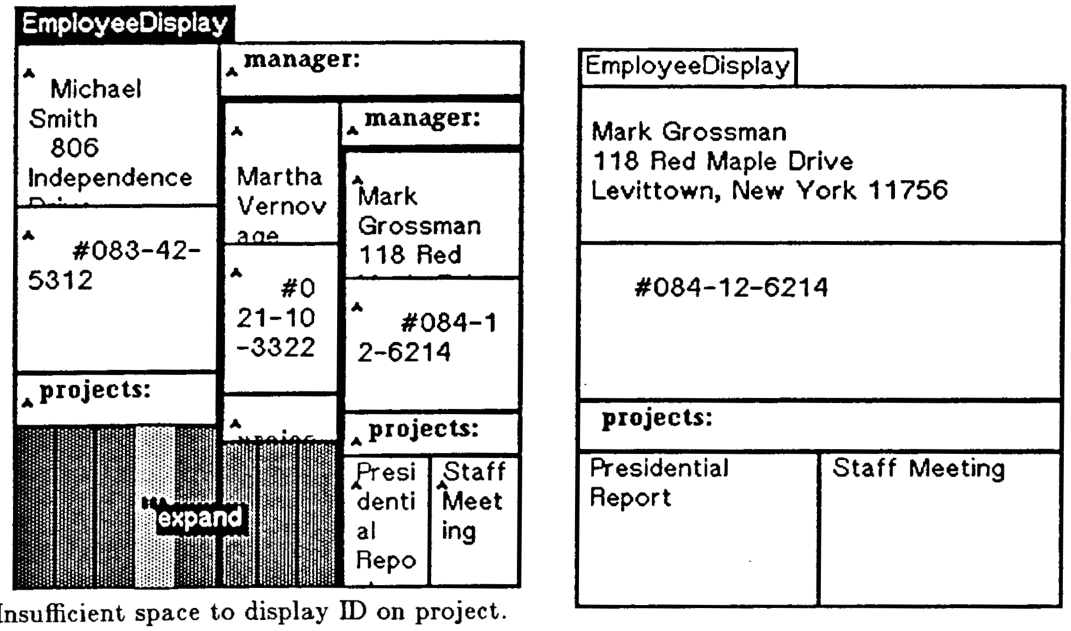


Figure 8a. Insufficient space to display ID on project.

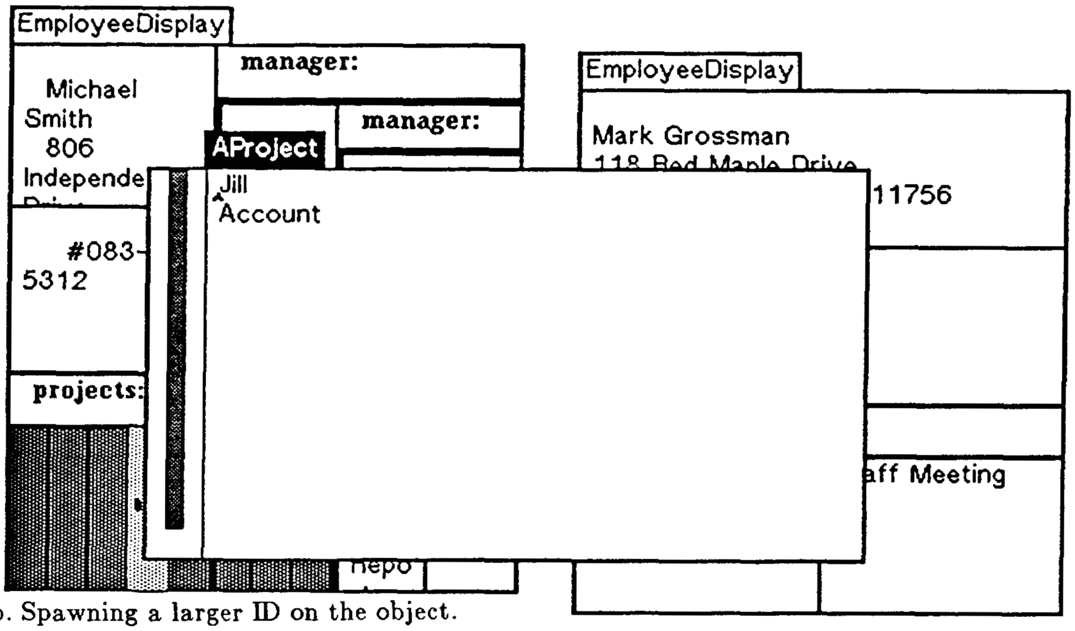


Figure 8b. Spawning a larger ID on the object.

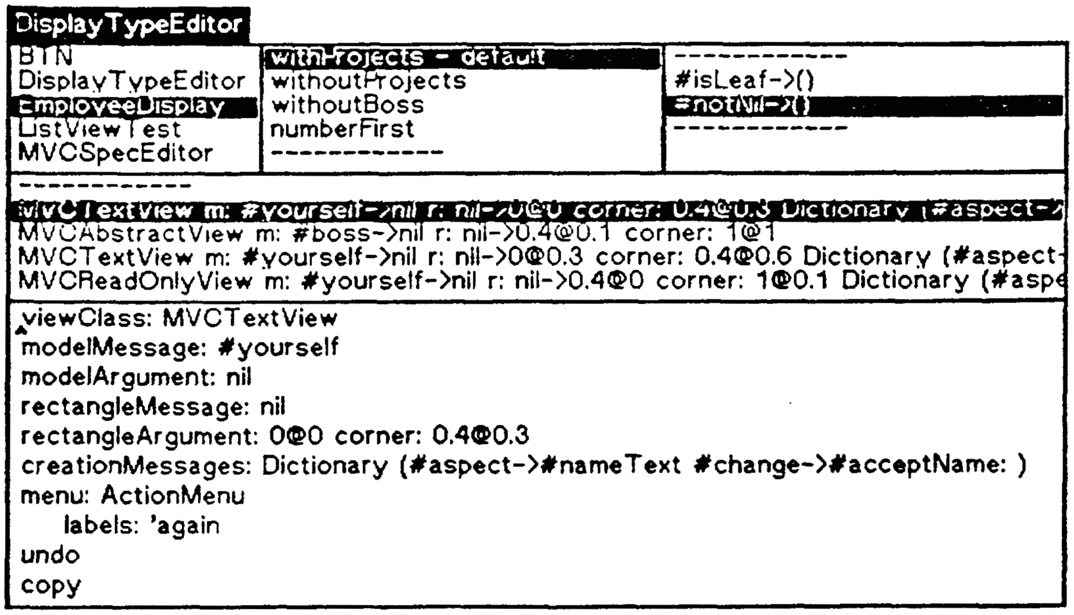
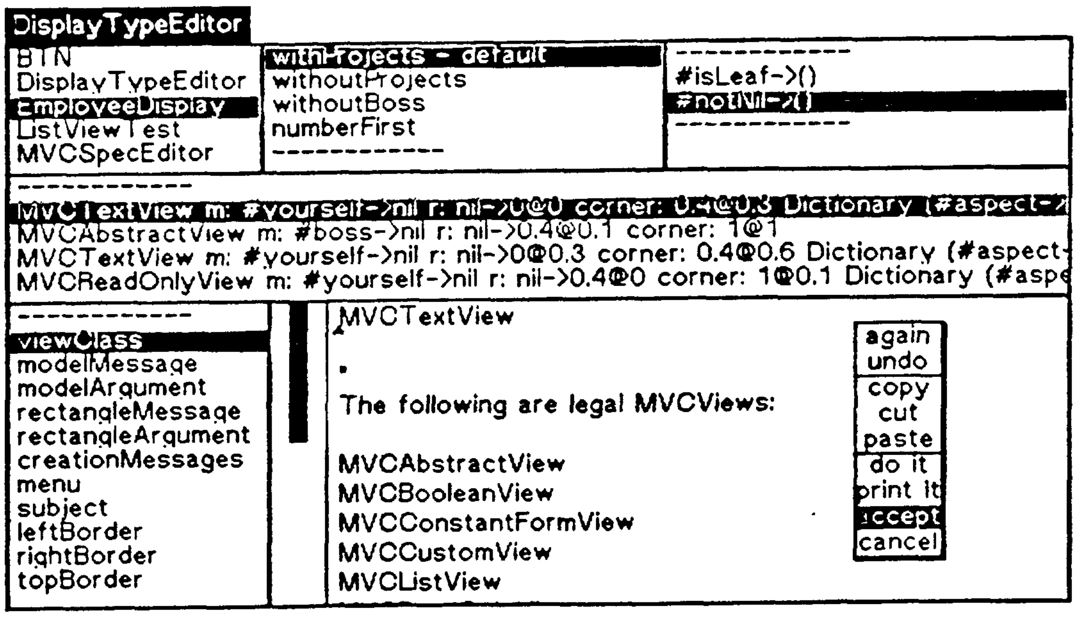

```
DisplayTypeEditor
BTN                   withProjects - default   ------------
DisplayTypeEditor     withoutProjects          #isLeaf->()
EmployeeDisplay       withoutBoss              #notNil->()
ListViewTest          numberFirst              ------------
MVCSpecEditor         ------------
------------
MVCTextView m: #yourself->nil r: nil->0@0 corner: 0.4@0.3 Dictionary (#aspect-
MVCAbstractView m: #boss->nil r: nil->0.4@0.1 corner: 1@1
MVCTextView m: #yourself->nil r: nil->0@0.3 corner: 0.4@0.6 Dictionary (#aspect-
MVCReadOnlyView m: #yourself->nil r: nil->0.4@0 corner: 1@0.1 Dictionary (#aspe
viewClass: MVCTextView
modelMessage: #yourself
modelArgument: nil
rectangleMessage: nil
rectangleArgument: 0@0 corner: 0.4@0.3
creationMessages: Dictionary (#aspect->#nameText #change->#acceptName: )
menu: ActionMenu
    labels: 'again
undo
copy
```

Figure 9a. Display-type editor in display mode.


```
DisplayTypeEditor
BTN                  | withProjects - default | ------------
DisplayTypeEditor    | withoutProjects        | #isLeaf->()
EmployeeDisplay      | withoutBoss            | #notNil->()
ListViewTest         | numberFirst            | ------------
MVCSpecEditor        | ------------           |

------------
MVCTextView m: #yourself->nil r: nil->0@0 corner: 0.4@0.3 Dictionary (#aspect-
MVCAbstractView m: #boss->nil r: nil->0.4@0.1 corner: 1@1
MVCTextView m: #yourself->nil r: nil->0@0.3 corner: 0.4@0.6 Dictionary (#aspect
MVCReadOnlyView m: #yourself->nil r: nil->0.4@0 corner: 1@0.1 Dictionary (#asp

------------         | MVCTextView
viewClass            |
modelMessage         | .
modelArgument        |
rectangleMessage     | The following are legal MVCViews:
rectangleArgument    |
creationMessages     | MVCAbstractView
menu                 | MVCBooleanView
subject              | MVCConstantFormView
leftBorder           | MVCCustomView
rightBorder          | MVCListView
topBorder            |
```

Figure 9b. Display-type editor in edit mode.

6. System Concepts

The two major contributions of this research are ***display types*** and ***abstract views***. Display types gather together the design decisions in constructing a new ID into high-level specifications, separate from the ID itself. An abstract view can interpret a display type, and accommodate its display to changes in the state of the object displayed.

The structure of a display type is best understood through the display-type editor, which reflects that structure (Figure 9b). Each display type is associated with a class of objects. All classes that have display types are listed in the upper left pane. Selecting a class in that pane produces a list of display types for the class in the upper center pane. Each display type is composed of one or more *recipes*. Selecting a display type in the upper center pane produces a summary of its recipes in the upper right pane. Recipes correspond to different states of the object to be displayed, and allow different configurations of the ID for that object. A recipe has a *selection condition* and a list of *ingredients*. The selection condition is a message sent to the displayed object to determine if a particular recipe applies. The ingredients specify the position, contents and rendering of subregions of an ID. It is the selection conditions for recipes of the selected display type that are shown in the upper right pane of the editor. Picking one of the conditions causes the corresponding ingredients to be summarized in the middle pane. Selecting the summary of an ingredient in the middle pane produces a detailed description of that ingredient in the bottom panes.

Each ingredient is a sub-ID specification. SIG supplies specifications for several kinds of views commonly used in describing sub-IDs:

1. text views, which display and edit text,
2. read-only views, which display text, but do not support update,
3. list views, which support scrolling and selection in a list of items,
4. Boolean views, for displaying and toggling a Boolean value, and
5. constant form views, for scaling and displaying a fixed graphical image (a *form* in Smalltalk parlance).

These SIG-supplied view specifications are templates that must have some additional information filled in to describe the sub-ID fully. The lower left pane lists the slots that must be filled in to complete the specification of a sub-ID. Selecting one of those slots causes its current value to be listed in the lower right pane, along with information on permissible values for the slot. SIG fills in the slots with initial values that are appropriate for the kind of view selected. There are slots for the position of the view within the ID, what object is being displayed in the view, widths of borders, a menu for updates, and more. The right pane also provides a pattern for filling in a new value for the selected slot. An ingredient can also specify a user-supplied *custom view* for producing special graphic images. Custom views are used for graphic details that are not produced well by scaling a constant form, such as the arrows in Figure 1.

The last possibility for an ingredient is that it specify an abstract view. Specifying an abstract view defers most decisions on the format of a sub-ID to a display type on another object, usually a subpart of the object being displayed at the highest level. In the display type we have been using for the Employee ID, abstract views were used for the projects and manager fields. (Note that not all the ingredients are visible in the center pane of Figure 9b.)

When an ID is created for an object from a display type, an abstract-view object is created by SIG. The abstract view has the object as its model, and also holds a copy of the display type to be used. The proper display type to be used is found in a dictionary that SIG maintains, which is keyed on class and names of display types. The abstract view generates an ID for the object by interpreting the display type. The abstract view queries the object about its state, to determine which recipe in the display type pertains. The query is performed by sending the selection condition messages in the recipes to the model until one returns true. Once the recipe is determined, the abstract view creates sub-IDs based on the kind of view and corresponding slot values for each ingredient in the recipe. It does so by creating a view object of the kind specified, and selecting an appropriate controller to go with it, and binding that view and controller to the correct model. If the ingredient calls for an abstract view on a subobject with a different (or the same) display type, the ID-generation process is repeated recursively. Thus, abstract views are the mechanism that permits modular description of an ID based on IDs for subparts of an object.

An abstract view interprets a display type dynamically. An ID using an abstract view monitors the object being displayed for significant changes. If changes occur and another recipe is called for, the abstract view reconstitutes itself with sub-IDs based on the ingredients in the new recipe. Parts of the ID that correspond to parts of the objects that did not change usually do not have to be repainted.

7. Work Outside of SIG

There are a few aspects of display development that currently must be dealt with outside of SIG. If a display designer wants to use a custom view, then he or she must implement a routine that draws the desired graphics in a given area. The designer then includes a message to invoke that routine in a display description.

For the class being displayed, the designer must write routines so that objects can process certain messages issued by the ID. Some messages request updates corresponding to items on the ID's menu. Many of those update messages will be implemented in any case when constructing a new class of objects. Usually the subregion of a display area that a sub-ID occupies is fixed in the ingredient for the sub-ID. If that subregion is to depend on some property of the model (such as the number of nodes in a subtree), the model must provide a *rectangle message* that returns that proper subregion in relative coordinates. Other messages are needed to support the pluggable view mechanism. The most important is the *aspect message*, which is sent by a view to a model to determine which aspect of the model to display. A model may require several aspect messages, if different views display different aspects of it.

While some Smalltalk code must be written to produce a display with SIG, that code need only be written once for a class, even if the class has multiple display types. Furthermore, all the code is associated with the class of the object being displayed. No view or controller code need be written, as contrasted to using the MVC mechanism directly.

8. System Construction

As mentioned before, SIG builds upon the MVC mechanism of Smalltalk. Models use a broadcast message to let views know that they have been updated. Smalltalk maintains a *dependencies list* telling which views are concerned with which models. When a model changes, the list is scanned, and the appropriate views are notified. If the model changes in only one aspect, it broadcasts that that aspect has changed. Only views concerned with that aspect respond. The views and controllers SIG uses to build IDs are adaptations of views and controllers supplied with Smalltalk, except for abstract views.

The display-type editor is itself a SIG-generated ID on an object of class DisplayBuilder.

9. What Next?

We believe more of the construction of a display type can be done graphically, rather than lexically. For example, the subregions of a display corresponding to different ingredients could be sketched on the screen, rather than specified via relative coordinates. Other parts of a display type, such as border width and the kind of view for an ingredient could be chosen off menus. We also need better support for iterative sub-IDs, such as a view that can scroll a sequence of subviews. Also, while we can spawn a new ID on a sub-ID that is too small to display, we have no facility to zoom in on one region of an ID.

We would like to specify the routines for messages to the model at a high level in a display type, rather than as Smalltalk code. Display types would then be self-contained descriptions of displays, which would make it easier to implement SIG on systems other than Smalltalk.

Ideally, we would like to create display types for a class with no modification of the class itself. The data modeling and computation portion of application development could proceed separately from the display design. We have experimented with an architecture for interactive applications called Humanizer [Gros85] that cleanly separates display modules from data access and computation modules in a database application. Figure 10 shows the layout of Humanizer. Display modules and other application modules communicate through shared objects held in a Data Access Manager (DAM). Display modules continually monitor the objects in the DAM for changes made by the other modules. In response to user input, display modules modify objects that the other application modules are monitoring. The DAM is an in-core database system that ensures serializable access of objects, enforces constraints, imposes authorization conditions, and maintains queues of interesting events for the various application modules. With Humanizer, it is easy to have two displays communicating with one computation module, or one display communicating with two computation modules, or even switch displays while an application is running.

Acknowledgements

We thank Harry Porter and Mark Ballard for help with the software for producing hard copies of screen images. This work was supported in part by NSF grant IST 83 51730, co-sponsored by Tektronix Foundation, Intel, Digital Equipment Corporation, Servio Logic, Mentor Graphics, Xerox, IBM and the Beaverton Area Chamber of Commerce.

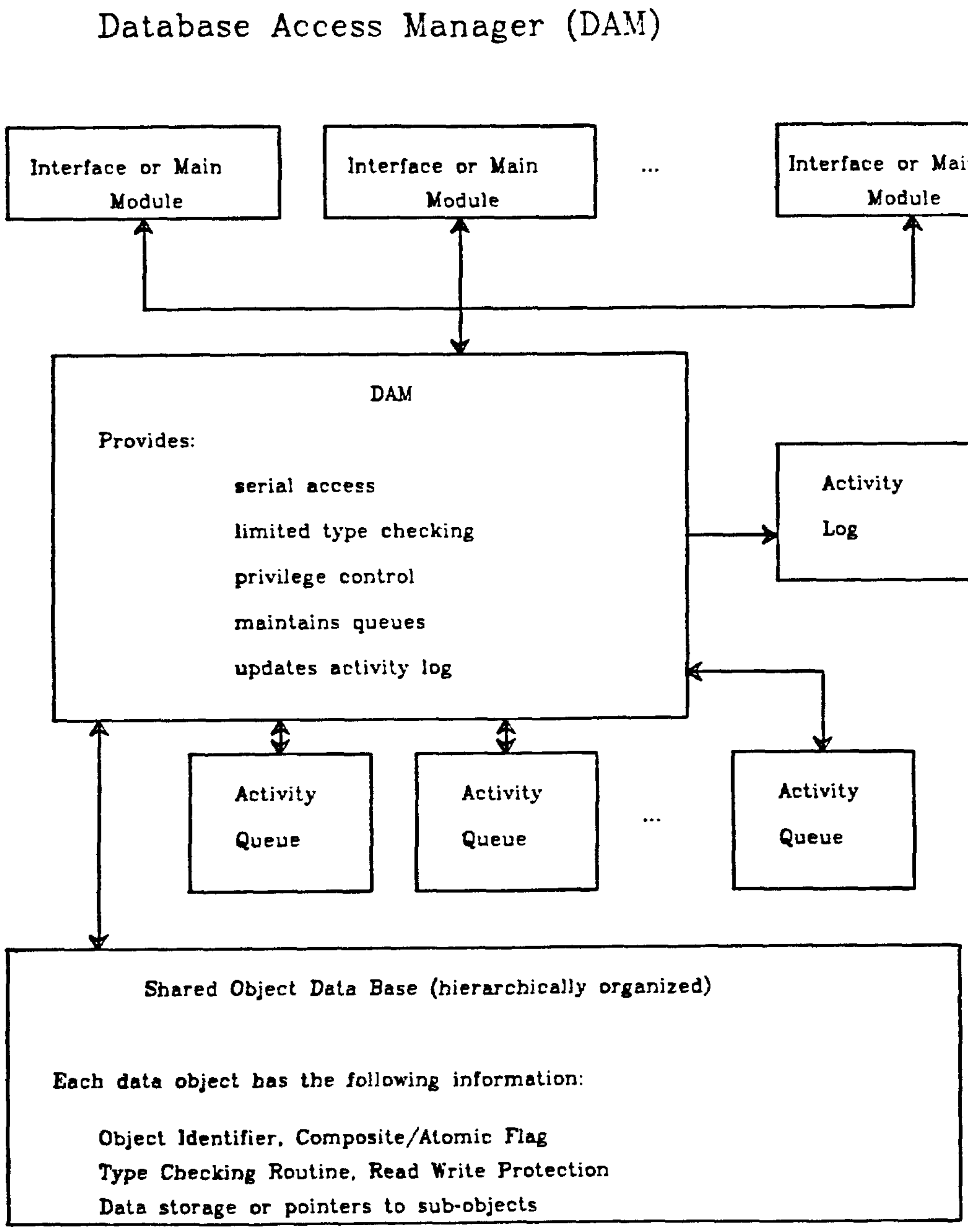

Database Access Manager (DAM)
Interface or Main Module
Interface or Main Module
...
Interface or Main Module
DAM
Provides:
serial access
limited type checking
privilege control
maintains queues
updates activity log
Activity Log
Activity Queue
Activity Queue
...
Activity Queue
Shared Object Data Base (hierarchically organized)
Each data object has the following information:
Object Identifier, Composite/Atomic Flag
Type Checking Routine, Read Write Protection
Data storage or pointers to sub-objects

Figure 10. Humanizer layout.

10. Bibliography

[Ande85] T. L. Anderson and B. B. Claghorn. ADE: Mapping between the external and conceptual levels. In *Information Systems: Theoretical and Formal Aspects,* (Proceedings of the IFIP WG 8.1 Working Conference on Theoretical and Formal Aspects of Information Systems, Sitges, Barcelona, Spain, 16-18 April, 1985), A. Sernadas, J. Bubenko, and A. Olive, eds. North Holland, 1985.

[Baec81] R. Baecker. *Sorting Out Sorting,* 16mm color sound film, 25 minutes, SIGGRAPH '81, 1981.

[Brow84] M. H. Brown and R. Sedgewick. A system for algorithm animation, *Computer Graphics 18* (3), July 1984.

[Egge83] P. R. Eggert and K. P. Chow. Logic programming graphics and infinite terms, Department of Computer Science, UC Santa Barbara, June 1983.

[Finz84] W. Finzer and L. Gould. Programming by Rehearsal, *BYTE Magazine,* June 1984.

[Fisc84] G. Fischer and M. Schneider. Knowledge-based communication processes in software engineering, Proceedings 7th Int. Conf. on Software Engineering, March 1984.

[Furu82] R. Furuta, J. Scofield, and A. Shaw. Document formatting systems: survey, concepts, and issues, *ACM Computing Surveys 14* (3), September 1982.

[Gold83] A. Goldberg. *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, 1983.

[Gros85] M. B. Grossman. Humanizer—A framework for implementing flexible human-machine interfaces, unpublished manuscript, Department of Computer Science & Engineering, Oregon Graduate Center, May 1985.

[Hend82] P. Henderson. Functional geometry, Proceedings ACM Conference on Lisp and Functional Programming, August 1982.

[Hero80] C. F. Herot, R. T. Carling, M. Friedell, and D. Kramlich. A prototype spatial data management system, Proceedings SIGGRAPH '80, 1980.

[Hero82] C. F. Herot, G. P. Brown, R. T. Carling, M. Friedell, D. Kramlich, and R. M. Baecker. An integrated environment for program visualization, Proceedings IFIP WG8.1 Working Conf. on Automated Tools for Information System Design and Development, 1982.

[Kimu83] G. D. Kimura and A. C. Shaw. The structure of abstract document objects, TR 83-09-02, Computer Science Department, Univ. of Washington, September 1983.

[King84] R. King. Sembase: A semantic DBMS, Proceedings First Int. Workshop on Expert Database Systems, October, 1984.

[Kram83] D. Kramlich, G. P. Brown, R. T. Carling, and C. F. Herot. Program visualization: Graphics support for software development, Proceedings 20th IEEE Design Automation Conference, 1983.

[Lond85] R. L. London and R. A. Duisberg. Animating programs using Smalltalk, *Computer 18* (8), August 1985.

[Myer83] B. A. Myers. INCENSE: A system for displaying data structures, *Computer Graphics 17* (3), July 1983.

[Nord85] P. R. Nordquist. Interactive display generation in Smalltalk, TR 85-009, Department of Computer Science & Engineering, Oregon Graduate Center, March 1985 (Master's Thesis).

[Pere83] F. C. N. Pereira. Can drawing be liberated from the von Neumann style?, Artificial Intelligence Center, SRI International, March 1983.

[Pilo83] M. Pilote. A data modeling approach to simplify the design of user interfaces, Proceedings 9th VLDB, October-November 1983.

[Reis83] S. P. Reiss. PECAN: Program development systems that support multiple views, Proceedings 1983 International Conf. on Software Engineering.

[Scho83] E. Schoen and R. G. Smith. IMPULSE: A display oriented editor for STROBE, AAAI '83.

[Smith84] R. G. Smith, G. M. E. LaFue, E. Schoen, and S. C. Vestal. Declarative task description as a user-interface structuring machanism. *IEEE Computer 17* (9), September 1984.

[VanW82] C. J. Van Wyk. A high-level language for specifying pictures, *ACM Transactions on Graphics 1* (2), April 1982.