10.
Editing Dialog Models

As in the chapter on visual dialogs the systems described here are concerned with the problem of data presentation. In particular, they are concerned with how to integrate the presentation of the data with the interactive input. Early UIMS attempts concentrated almost exclusively on the input side of interaction and the handling of input events. The visual dialog systems were concerned with drawing visual portions of the presentation and with establishing relationships between the application data and the visual presentation.

The approach of the systems in this chapter is that the user interface should be determined by a model of the information to be interactively manipulated rather than by the commands that the user will issue. It can be asserted that most interaction consists of either browsing or editing information.[1] This is found to a limited extent in the transition from MIKE to Mickey, discussed in Chapter 8. MIKE was based exclusively on a model of the commands that were to be issued. Data types were represented only by names. In Mickey the type declaration information was used to create text edit boxes, dialog boxes, check boxes, and radio buttons. Each of these various interactive techniques is a form of editor for application data. An editing UIMS is driven by information about application data through the semantic interface and then provides interactive techniques for manipulating information.

An editor combines input handling and data presentation in a unified fragment of interaction. Take, for example, the simple text edit box. The model for a text edit box is that it is editing a string. Each of the user inputs has specific meaning for how the string is to be changed and is tightly integrated with how the string is to be displayed. Radio buttons and check boxes are similar editors for particular kinds of data. A dialog box is a mechanism for compositing editors together to form an editor for a composite piece of information.

Motivation for Data-based UIMS Models

Direct engagement with the information being manipulated is a major key in direct manipulation. In order for a UIMS to support direct engagement it must have a model of the information that it is manipulating. Early UIMS work only had a model of the commands and therefore the interfaces created by such systems were inherently indirect.

There are also a number of features a UIMS can support which are not possible without some model of the underlying application data. An undo facility must know about the application data so that it can put the data back the way they were before the changes to be undone were made. Many text editors and source code control systems provide facilities for managing patches, changes, or versions of text files. Similar facilities are of value in other interactive environments. A UIMS that understands the data that it is manipulating could support such features. Most graphical user interfaces do not provide search facilities. Again, if the UIMS understands the data model then the UIMS can provide such search facilities automatically. These are just a few examples of extended features that can be part of the UIMS and, therefore, be made available to all applications built using the UIMS. Such facilities require, however, that the UIMS knows about the application data.

Cousin

The Cousin system[2] was one of the earliest editing UIMSs. The heart of Cousin is the *environment*, which is a set of named and typed slots through that the interactive user can manipulate variables that communicate with the application. Entering parameters and commands is performed by editing the values in the slots.

The environment is controlled by a *system description* which specifies all of the information that Cousin needs about each slot. Each slot is described by:

- Name
- Data type
- Default value
- Constraints on legal values for the slot
- The syntax for values in the slot
- A textual description for explanations

For each data type Cousin provides an editor for manipulating the values of the slot.

Editing Templates

As the MIKE profile editor was being developed it was recognized that patterns of commands were being repeated over and over again. A list of menu items was manipulated in very much the same way as a list of windows, a list of data types, or a list of commands. All of the information about an application's user interface was stored in a system called STUF[3] (STrUctured Files). Essentially, this facility provided Pascal-style records and unions to be stored on disk files. The various kinds of lists were all stored in STUF structures as linked lists with a great deal of similarity between them. These similarities were exploited in a system called Editing Templates.[4]

An editing template is a particular kind of interactive editor which is parameterized in various ways to create particular application instances of the editor. An editor template has not only a particular interactive behavior but also a particular view of the data that it is editing.

A Linked List Editing Template

The linked list editor is an example of such a template. This template will scroll through a list of data objects that are stored in a linked list form. The parameters to this template would be:

%ObjType — the STUF data type of the objects in the list
%Link — the name of the field that is used to link successive objects together to form the linked list
%CurIdx — this is a *cursor*, or a simple index to the current object tuple that is being edited.
%HeadType — the STUF data type for the object that contains the pointer to the head of the linked list.
%HeadField - the field in the header object that points to the first element in the list
%XExt and %YExt — the X and Y extent of each element in the list's display.
%Window — the window that the list is to be displayed in.
%ObjDisp(Obj, X,Y) — an application routine which will display the Obj in the screen location specified by X and Y.
%ObjDel(Obj) — an application routine to delete Obj.

The parameters represent the information that will tailor a generic linked list editor to a specific application. The linked list editor template would then supply the following command procedures which can be exposed directly to MIKE as part of the user interface.

%WindowUp — move %CurIdx up one item in the list. (Because of MIKE's command orientation all interfaces are key or event driven rather than direct manipulation. There are no scroll bars)

%WindowDown — move %CurIdx down one item.

%WindowLeft and %WindowRight — if the window is wide enough for multiple columns of items, this will move %CurIdx to the item immediately to the right or left of the current one.

%WindowPageUp and %WindowPageDown — moves %CurIdx up or down a full window's worth.

%WindowDelete — deletes the object referenced by %CurIdx.

In addition to the above command procedures there are several additional routines which can be used by the application code to interface with the generated instance of the template.

Restore%Window — will redraw the entire window by means of appropriate calls to %ObjDisp.

%WindowUpdateCur — will update the display of the current object in the event that changes have been made by the application.

%WindowInsert(Obj) — will insert the specified object into the linked list immediately after %CurIdx, make it the current object and update the display appropriately.

%WindowChangeCursor(Obj) — will make Obj the new current object in the list.

The editing template is primarily a piece of code that has been parameterized for a simple macro processor. In early versions of editing templates, a template user would supply values for each of the parameters and then run the template code through the macro processor to produce all of the code for a new instance of the linked list editor. The editor code would handle all of the editing and screen update issues for multiple column presentations of lists of anything. This provided a significant speedup in the creation of similar pieces of interaction. The routines shown above would all have application-specific text substituted for the parameter names (% —).

The application would then expose some of the routines directly to MIKE (as in the case of %WindowUp). The application could provide many other command procedures which would all work on the object referenced by %CurIdx. After changing the object they could invoke %WindowUpdateCur to get the screen updated. Insertion of new objects would be handled in this way since only the application would know how to create new objects of various kinds.

The linked list template is only an example. Several other editor templates were created. A simple extension to the linked list is the sorted

linked list which uses an application-specific compare routine to determine if items are in sorted order. A tree editor provides for traversal and display of tree structured data. A schematic was created which provided editing facilities for nodes that were connected by arrows. The application data structures contain only the node and connection information, the editor provided all of the layout and maintained the node connections.

Architecture of Editing Templates

The original editing templates system consisted only of the code for each template, a set of parameters for each instance of the template, and the macro processor necessary to create an instance of the template from the parameters. This simple system was augmented by the creation of two special purpose applications (TED and ICE) which were built with the editing templates system.[5] Figure 10:1 shows the editing templates architecture.

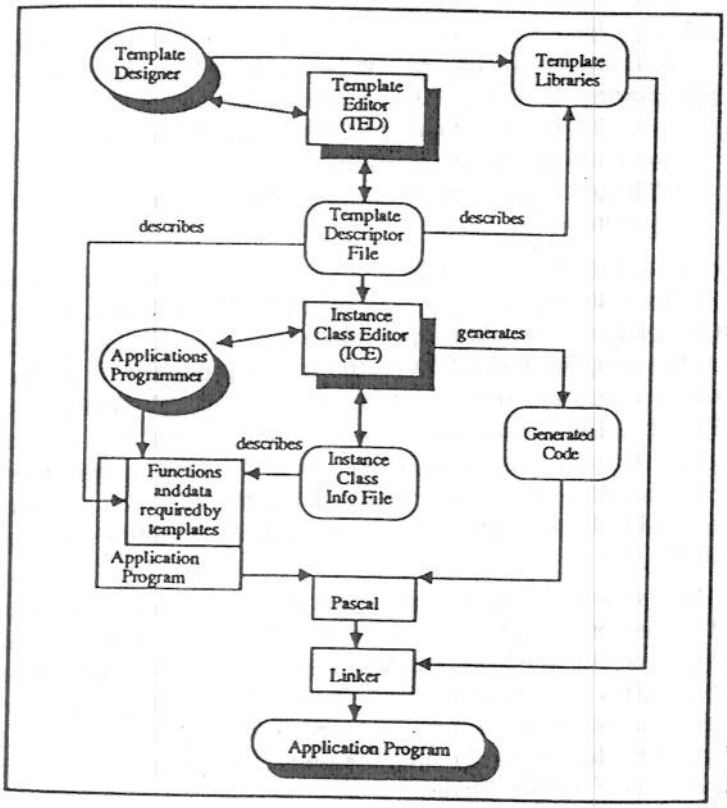

Fig. 10:1 Editing Templates

When one creates a new editing template, one builds a general library that will perform the desired editing functions. A template descriptor is also created which contains all of the information about how the template should be used. A template descriptor is created by TED (Template EDitor) and specifies the parameters to the template as well as the types that it uses, the application routines that it needs from the application, and the routines that the template will generate.

When an application programmer wants to use an editing template they use ICE (Instance Class Editor) to create a new instance class. An instance class is created when a template has parameters substituted into it. An instance class can then be used to create instances of the editor in multiple windows. For example, one could take the linked list editor template and describe it in a template descriptor. One could then create a student editor from the linked list template by supplying the information about linked lists of students. This student editor is an instance class. The student editor can then be used at run time to create instances of the editor in different windows for different lists of students.

Most of the code generated by ICE was to interface between the application and the generic code in the template libraries. This was particularly necessary because of Pascal's excessively restrictive mechanism for handling callback procedure addresses.

ITS

A more advanced model for editors is called Interactive Transaction Systems (ITS).[6] The semantic view of the ITS model is that a user is interactively editing information stored in tables. These tables are similar in nature to relational databases in that they contain tuples with fields of information and fields have types. The prime thrust of ITS development has been in abstracting the style (or presentation) of an interface from the rest of the dialog description and eliminating any syntactic specification issues by using an editing model for interaction.

This section will first discuss the four main components of ITS, followed by a deeper discussion of how dialog information is represented. The generation of a dialog specification tree will then be discussed in conjunction with the issue of style or presentation independence. Finally, some run-time issues will be covered.

Four Parts of ITS

The specification of an ITS dialog consists of a data definition, an interface content definition, style specification, and the run-time structures. All of

these definitions are represented as trees of tagged items, each item having a number of attributes.

Data Definition

The data definition specifies the majority of the semantic interface to the application in the form of data and table definitions. For example:

```
:data type=fullname, structure=disjoint
  :di field=last, type=string
  :di field=middle, type=string
  :di field=first, type=string
:edata

:data type=student, structure=disjoint
  :di field=name, type=fullname, emphasis=special
  :di field=address, type=us_address
  :di field=parent, type=fullname
  :di field=class, type=integer
:edata

:table name=students
  :ti field=std, types=student
:etable
```

The purpose of the data definition is to specify the information to be interacted with. There are a variety of attributes that can be specified with each part of the data definition in order to provide information to the dialog about how this data is to be used and manipulated. Examples of such attributes are "structure=disjoint" and "emphasis=special." Such information is used by the views and the style rules to control the presentation.

Dialog Content

The purpose of the dialog content is to represent the logical (not presentational) form of the dialog and to specify the nonediting actions that the dialog is to take. The dialog content is specified at a relatively abstract level which consists of five basic components: lists, forms, choices, info, and frames. The dialog content specification is a neutral ground between the data definition and the style specifications.

A list is an arbitrary collection of things to be presented to the user, such as a list of all students or a list of files. All items in the list are assumed to be similar in structure and amenable to similar presentations for each element of the list. Note that the concept of a list does not specify

how it is ordered, stored, or otherwise accessed (this is a data definition issue), nor does it specify how the list should be displayed (this is a style issue). The content specification simply indicates that a list of items should appear and specifies what information should appear. It is the business of the dialog content specification to indicate if all of the information about a student should be presented or only the name of the student. For example, a list of student names would be:

```
:list listname=student_names, table=students
  :li field=name
:elist
```

A form is a fixed collection of things. A form corresponds to a dialog box or property sheet. As with a list, a form would only specify what items are to appear in the form. The form would not specify how the items are to be displayed.

A choice represents a set of items from which a user must choose. Choices subsume constructs like buttons, menus, check boxes, and lists of check boxes. The dialog content specifies what the choices are, as well as what is to be done when a choice is made. A choice might be a choice of values such as:

```
:choice message="Class In School", field=class,
      kind=1_and_only_1
  :ci message="Freshman", value=1
  :ci message="Sophomore", value=2
  :ci message="Junior", value=3
  :ci message="Senior", value=4
  :ci message="Graduate", value=5
:echoice
```

A choice might also select among a list of actions to be taken. Such as:

```
:choice kind=1_and_only_1
  :ci message="Apply", action=Apply
  :ci message="Register", action=Register
  :ci message="Expel", action=Delete
:echoice
```

Info items are simply pieces of textual or graphical information that are to be presented to the user, such as help texts.

The dialog content is organized into a set of frames. A frame is a single conceptual unit of dialog. It would correspond to a top-level window on the Macintosh or in X. A frame is also analogous to a card or background in

HyperCard. It is a single, logical screen of information. An example frame might be:

```
:frame table=students
  :choice kind=1_and_only_1
    :ci message="Apply", action=Apply
    :ci message="Register", action=Register
    :ci message="Expel", action=Delete
  :echoice
  :list listname=student_names, table=students
    :li field=name
  :elist
:eframe
```

This frame would present a list of names of students and provide three possible actions to be taken on a student. The entire dialog content specification consists of a list of such frames.

Style Specification

The style specification is what controls the actual presentation of the user interface. The style specification is controlled by environments and rules. An environment is simply a named set of attribute values. Environments can also inherit attribute values from other environments.

The style rules are the key to the style independence of the dialogs. Style rules are basic if/then constructs. The if part specifies a pattern of attributes to which the rule will apply. The then part specifies an environment of attributes to inherit as well as additional units to be instantiated to flesh out the presentation.

Take, for example, the following two environments:

```
define text, view=string, font=times, size=10, justify=left
define special, parent=text, size=15
```

The following style rules could be defined

```
if (TAG=Form) & (Emphasis=special) Then (Match special)
if (type=string) then (Match text)
```

The rules and environments map special cases of the attributes into more extended attribute definitions. The idea is that issues of font size, color, font face, justification, line width, etc., should be specified relative to the abstract issues of what the item is for. A style then, is a mapping from the high level content specification to a particular presentation, as represented by the attribute settings.

The style rules are not limited to simple attribute settings. A style rule can also elaborate a particular item in the tree to a more complex structure. Take, for example, an item in a choice that has "kind=1_and_only_1." The style rule may want to specify that this be implemented as a radio button. To do this it must generate an item for the button and from the message attribute of the choice it must generate the label to go next to the button. The full scope of this rule language is beyond the scope of this discussion.

The most important attribute setting is the view. A view is a generalized editor that controls not only the presentation of the information but also how it responds to input events in manipulating the information.

Run Time

At run time all of the frames are represented as trees of views within ITS. These trees have had all of the style rules applied and are fully elaborated. When a frame is made active, its static representation is copied and the copy is decorated with appropriate information to tie its parts to the appropriate data instances. This forms the actual connection between the ITS dialog tree and the application data in the tables.

Generation of the View Tree

As has been mentioned, the run-time description consists of trees of views which handle the interaction between the user and the application. The key to ITS is in its generation of these trees from the various descriptions.

Mixing of Dialog Content with Data Definition

The first step is to use the data definition information to elaborate the content definition. In our example list only the field was specified for the elements of the list.

```
:list listname=student_names, table=students
  :li field=name
:elist
```

By searching the data definition we can elaborate this item with the type of the name field, which is "fullname," and the "emphasis=special" attribute. We can further elaborate this list item with a form that consists of the last, middle, and first name fields. The result might be:

```
:list listname=student_names, table=students
  :form field=name, type=fullname, emphasis=special
    :fi field=last, type=string
    :fi field=middle, type=string
    :fi field=first, type=string
  :eform
:elist
```

This dialog compilation phase fleshes out the dialog content specification by adding the additional information about the data being presented.

Mixing in of Style

The second step is style compilation. In this phase the style rules are applied to the new tree to add the necessary additional attributes. It is important to note that all elements of the tree inherit attributes from their ancestors in the tree. The fields in our example list would inherit the "emphasis=special" attribute. This would cause the "special" environment to be invoked which would decorate each of the fields with the attributes associated with special text.

Summary of ITS

ITS is based on a model of editing information. It has further refined the traditional UIMS model by factoring out the style rules. In essence, the style rules are those presentation specifications that are independent of a particular application. Mickey forms an instructive counterpoint to ITS.

Mickey used a semantic data definition which had similar content to that of ITS's data definition. The programmer, however, must explicitly control what is presented to the user interface, whereas in ITS the dialog content specification controls what is seen. The dialog compilation phase of ITS exploits the same semantic information that Mickey does in fleshing out the dialog definition. Mickey forces a particular style for presenting the abstract interaction defined by the semantics. ITS, however, has generalized these style issues in the form of style rules and environments rather than hard coding them into the system.

Sushi

Sushi (Raw Editable Objects) is a merge of the concepts found in editing templates and the architecture of Mickey. In Sushi the semantic model is specified by a set of object classes that are defined in COS (C Object System). COS is a C preprocessor which provides a simple class structure and message passing system and has the features needed to support

Sushi. It is possible that object-oriented languages such as C++ or Eiffel could support Sushi if modifications were made to the compiler to provide the needed information.

The interactive model is provided by *object editors*. An object editor is a COS class which is a subclass of ObjectEditor. User interfaces are built by combining application objects with editors appropriate to their class and then instantiating them on the screen.

COS

COS class descriptions are defined in a special language and then run through the generator to create C code for compilation and linking with the application program and Sushi. A COS class consists of a superclass and a list of methods and fields. The following are some example classes defined in COS.

```
Enumerated RegStatus Is {NotRegistered, Registered, OnHold}
Class Student Superclass Obj [ . . . ] {
  Data Field string Name;
  Data Field Address HomeAddr;
  Data Field Address SchoolAddr;
  Data Field Picture Photo;
  Method Field RegStatus Registration
      Read {* ... *}
      Write {* ... *};
  Method void Expel {* . . . *};
  Method void Hire( Wages )
    {* . . .  *};
  }
Class Address Superclass Obj [ ... ] {
  Data Field string Street;
  Data Field string City;
  Data Field string State;
  Data Field long Zip;
  }
Class Wages Superclass Obj [...] {
  Data Field long Hours;
  Data Field long DollarsPerHour;
  }
```

Every class has a superclass and a set of fields and methods. COS defines both data fields and method fields. Normal methods also have their bodies

programmed in C. A method in COS may or may not return a result, and may or may not have a single argument. If multiple values are needed as an argument they must be encapsulated in a COS object. This is somewhat awkward for programming but is natural for direct manipulation user interfaces.

A data field defines an actual field in the class's data definition and then defines two methods (for example Name and Name_) which will read and write that field. A method field, such as RegStatus, allows the programmer to code the read and write methods directly in C. This view of fields and methods allows a COS facade to be placed over any data model that the application programmers desire. The COS classes are only a user interface view of the application.

Aside from the generation of the C code to handle the class declarations and method invocation, COS also generates descriptive information for each class which provides two important capabilities.

1. A description of all methods with their names, argument, types, and result types.
2. A mechanism for formulating object messages at run time.

The class descriptor is attached to all COS objects and is used both by Sushi and by the underlying message-passing mechanism.

Object Editors

The key to Sushi's interactive behavior is the set of *editor classes* that it makes available. An editor class embodies a particular style for interacting with information. An editor class is refined or specialized by its *descriptor*. A descriptor is simply a COS object of whatever class is convenient for that class of editor. The descriptor for long integers, for example, would contain information about how many digits to allow, foreground color, text font, etc. A descriptor for a dialog box is more complex and contains information about where in the box the labels and subeditors should be placed. A descriptor contains information similar to the attributes in an ITS dialog tree. It is by editing descriptors that the interface designer can control the presentation aspects of the user interface. Since all descriptors are themselves objects, editors can be applied to these descriptors in the same way that any other portion of the user interface is defined.

When an editor class is combined with a particular descriptor an *editor* is formed. Every editor will edit objects of a specific class or any of its subclasses. An editor can be combined with an application object of the correct class and a window to form an editor instance. The role of an *editor*

instance is to allow interactive users to browse, modify, and manipulate the application object according to the dialog specified by the combination of editor class and descriptor.

Editor classes generally come in two flavors: class-specific editors and composers. Class-specific editors always apply to a particular class of objects. The descriptor for a class-specific editor is only used to specify the presentation aspects of the editor. Composers are more general editor classes which can be specialized to a wide variety of object classes.

The editors and composers are stored in two lists which belong to the *editor environment object*. The prime purpose of the editor environment object is to maintain these two lists. The editor list contains editors whose descriptors have already been defined and specialized to a particular class. Editors are stored in the list by the name of the class that they edit and as a descriptive name. The descriptive name is important since more than one editor can exist for a given class of object. Composers are listed by name and type form (class, enumeration, union, or sequence) since they have not yet been specialized to a particular object class.

When an application exposes the editor environment object to the interactive user (by using an appropriate object editor for the environment object's class) the user or interface designer will have access to the editors and their descriptors. By selecting editors, editing their descriptors, or creating new editors from the composers, the interface can be interactively modified.

Object Class-specific Editors

The editors specific to particular classes include those defined on primitive types. There are editors provided for long integers, character strings, and Boolean values.

Supplying new class-specific editors is one of the easiest ways to extend Sushi. An example is an image editor. A COS class has been defined which provides access to images stored in a variety of formats with varying numbers of bits per pixel. Based on this class a special editor has been built and placed in the editor list under the BitMap class. Whenever BitMap objects are encountered, this special purpose editor is used. Other such special purpose editors have been built for color look up tables and color selection. The application programmer is completely free to extend this set.

Composers

A prime function of a composer is its GenerateDefaultDescriptor method which generates a default descriptor given a particular object class. As an

example of how this works, consider the DialogBox editor class. Given the class of a particular object the dialog box editor would generate a default descriptor that provides a subeditor for each field in the class and a button for every method in the class. In the case of fields that themselves contain new objects, the default is to generate a button that will open a new editor on the object stored in that field.

The GenerateDefaultDescriptor method will only generate a default descriptor. Once a new editor has been created by generating the default descriptor, that descriptor can be edited to refine the editor. For example, many fields and methods might be removed from a dialog box to create a summary editor for that class. A new button could be added to the dialog box which would invoke the full editor on the same object.

The Application Programmer's View

As an example of how Sushi works we can trace through the process a programmer will take in developing a new application. Suppose that one wanted to build a simple application to store students with their pictures and mailing information.

COS

The first step would be to define COS classes for the information to be edited.

```
Enumerated RegStatus Is {NotRegistered, Registered, OnHold}
Class Student Superclass Obj [ . . . ] {
  Data Field string Name;
  Data Field Address HomeAddr;
  Data Field Address SchoolAddr;
  Data Field Picture Photo;
  Data Field RegStatus Registration;
  Method void Expel {* . . . *};
  Method void Hire( Wages )
    {* . . . *};
 }
Class Address Superclass Obj [ ... ] {
  Data Field string Street;
  Data Field string City;
  Data Field string State;
  Data Field long Zip;
 }
```

```
Class Wages Superclass Obj [...] {
  Data Field long Hours;
  Data Field long DollarsPerHour;
  }
Class Picture Superclass Obj [...] {
  Method Field long PixelIdx
    Read {* . . .*}
    Write {* . . . *};
  Method Field long CurPixel
    Read {* . . .*}
    Write {* . . .  *};
  Method long Xdimension
    {* . . . *};
  Method long Ydimension
    {*. . . *};
  Method void ScanPicture
    {* . . . *};
  Private Field charPtr image;
  }
```

In the class Student, the Expel and Hire methods would have their implementations written in C by the application programmer. In the Picture class the fields PixelIdx and CurPixel would need their Read and Write methods written in C as well as the methods Xdimension, Ydimension, and ScanPicture. The ScanPicture method might run the scanner to enter a new picture.

The application programmer can create and manipulate objects of these classes by using the message-passing macros provided with COS.

EditObject

The programmer's primary interface with Sushi is via the EditObject routine. The programmer passes to EditObject the object that is to be shared with the user and the name of an editor to be used on that object. In addition, the programmer can84

indicate where on the screen the editor should be placed.

EditObject will take the class of the object and search the editor list in the environment object for an editor with the specified name and class. If such an editor is not found, EditObject will select an appropriate composer. EditObject makes a copy of the composer, sets its class name, and then invokes GenerateDefaultDescriptor to create a default descriptor for that class of object. This generation of default editors guarantees that

an editor is always available for any class of object. Figures 10:2 and 10:3 show the editors automatically created for Student and Address.

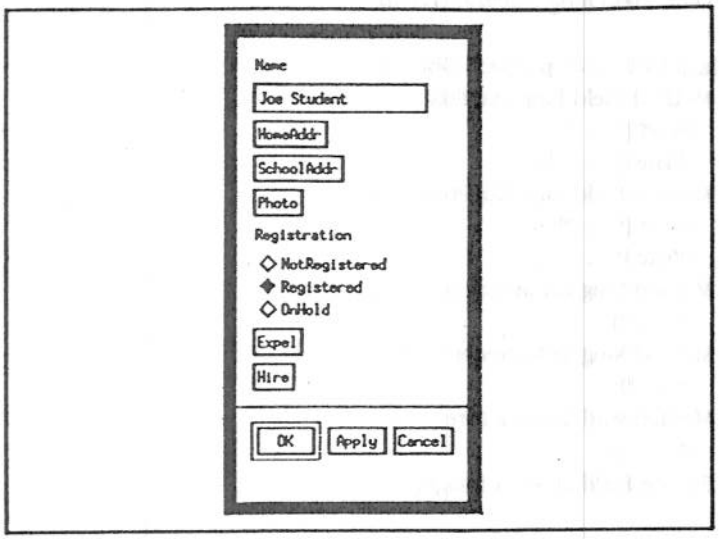

Fig. 10:2
Default Student Editor

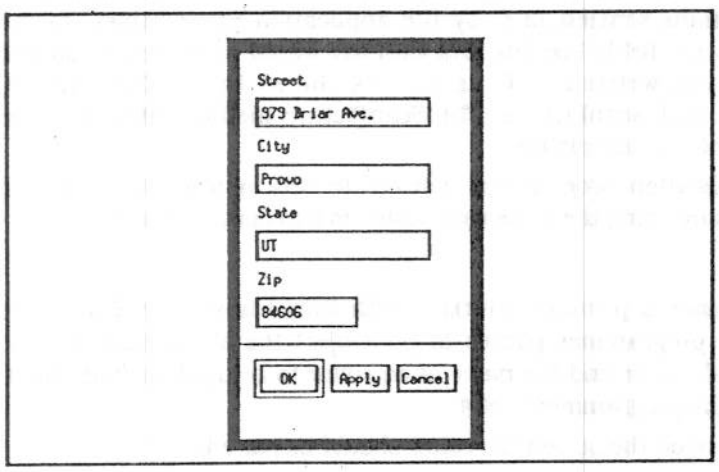

Fig. 10:3
Default Address Editor

Once the new descriptor is generated the new editor is added to the editor list for future use. This eliminates regeneration of the same editor. This new editor is then copied again and instantiated by assigning it the object to be edited and invoking the editor's Create method. If an appropriate editor already exists in the editor list, then it is copied and instantiated

with the object to be edited.

The programmer's view of Sushi can be summarized as: 1) create a semantic model as COS classes and 2) call EditObject to have the user interface objects displayed. Note that the programmer may only need to call EditObject once for a root or environment object which can then have subobjects which will cause new editors to open.

Adding New Editors

In the case of the Picture class the default generated editor would be as shown in figure 10:4.

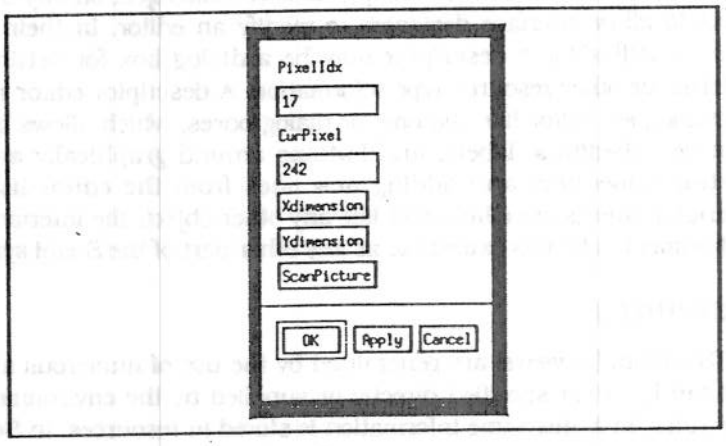

Fig. 10:4
Default Picture Editor

This is not a useful interface to pictures. The programmer could take the COS class ObjectEditor and create a new subclass called PictureEditor. Inside the PictureEditor class the programmer would implement all of the methods necessary to edit pictures graphically. Having written this new editor class, the programmer would add an object of this editor class to the environment object's editor list. Whenever the programmer, or some editor, invokes EditObject on a Picture object, the new editor will be used.

Applications such as a paint program or a word processor frequently have some central editor that characterizes the application. Such a central editor will often need a user interface design that is carefully crafted to that application. The peripheral portions of the interface, like selecting colors, selecting fonts, or opening and closing files can be automatically supported by Sushi with little or no effort. Once that central editor has been created it is easily integrated into other applications that need to edit similar data.

Interface Designer's View

The interface design of a particular application is characterized by the editors in the editor list and each of their descriptor objects. By saving the class name and descriptor object of every editor one can completely capture the interface design. The application programmer can make the interface design available to users by calling EditObject on the editor environment object. The primary purpose of the editor for the editor environment object is to allow interface designers to: 1) create new editors from the composers; 2) copy existing editors to make new editors; or 3) modify existing editors by editing their descriptor objects. The editor for the editor environment object simply invokes EditObject on any descriptor object to allow interface designers to modify an editor. In their simplest form an editor for a descriptor may be a dialog box for setting fonts, patterns, or other resource type information. A descriptor editor may be a more complex editor like the one for dialog boxes, which allows designers to move subeditors, labels, and buttons around graphically as well as deleting subeditors and adding new ones from the editor list. Since descriptor objects are edited just like any other object, the interface design environment is just as extensible as any other part of the Sushi system.

Summary

In ITS editors (or views) are generalized by the use of numerous attributes that can be either specified directly or supplied by the environments and style rules. In X this same information is stored in resources. In Sushi this information is stored in descriptors but the descriptors are themselves editable objects. Sushi does have the capability of some style control in the use of the GenerateDefaultDescriptor message on composers. This is not as powerful, however, as the style rules of ITS. On the other hand, Sushi has a mechanism which neither X nor ITS directly support, which is the ability to apply its own interface model to itself in editing the descriptive information about the interface.

All of these systems support more of the data and dialog presentation parts of the UIMS architecture than the language and automata-based approaches. They accomplish this by explicitly representing the application information that is to be manipulated. None of these systems contain an explicit dialog specification. The dialog specification is implicitly derived from the combination of data to be edited with the selection of a particular editable presentation for that data. Systems like ITS and Cousin represent their presentations in a form suitable only for programmers or programming-oriented professionals. Although such presentation-based

tools are the domain of the graphics designers the tools have not been geared to their particular skills. Sushi does allow the interactive editing of the interface in a way that a graphics designer could use. It suffers from the problem of each design being unique, whereas the style rules of ITS can be applied repeatedly to new applications. The extension of Sushi's GenerateDefaultDescriptor method to include ITS-like rules may provide the best of both worlds.

References

1 Olsen, D.R. *A Browse/Edit Model for User Interface Management.* **Graphics Interface '88.** Canadian Information Processing Society, June 1988.

2 Ball, E. and P. Hayes. *A Test-Bed for User Interface Designs.* **Human Factors in Computer Systems,** March 1982, 85-88.

3 Olsen, D.R. and R.P. Burton. *Structured Files for Interactive Graphics Programs.* **ISECON '88 Conference Proceedings,** 1988.

4 Olsen, D.R. *Editing Templates: A User Interface Generation Tool.* **IEEE Computer Graphics and Applications** 6(11), November 1986.

5 Blackham, G.D. **Editing Templates.** MS Thesis. Computer Science Department, Brigham Young University, Provo, Utah, 1986.

6 Wiecha, C., W. Bennett, S. Boies, J. Gould, and S. Greene. *ITS: A Tool for Rapidly Developing Interactive Applications.* **ACM Transactions on Information Systems** 8(3): 204-36, July 1990.