AUTOMATIC GENERATION OF USER INTERFACES

by

Prasun Dewan

Computer Sciences Technical Report #666
September, 1986

AUTOMATIC GENERATION OF USER INTERFACES

by

PRASUN DEWAN

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1986

ABSTRACT

In traditional interactive programming environments, each application individually manages its interaction with the human user. The result is duplication of effort in implementing user interface code and non-uniform—hence confusing—input conventions. It would be useful if the user interface of an application could be *automatically* generated. This idea requires an application-independent model of user interaction together with a programming environment that supports the model.

Recent work in user interface design has suggested that *editing* can be used as a general model of interaction. This dissertation presents an approach that supports this model.

This approach allows applications to be created as *objects*. An object is an instance of a *class* that describes the data encapsulated by the object and the *methods* to manipulate them. Between each object and the user is a *dialogue manager*. The dialogue manager receives *messages* from the object, which name variables that can be edited by the user. It displays the variables using the data definition in the class of the object, offers the user a structure editing interface to modify them, and sends new values back in messages to the object. The object can then execute methods to make its internal data consistent with the displayed data. Thus, from the point of view of the objects, the user appears to be another object that can send and receive messages. From the point of view of the user, the objects appear to be data that can be edited. The dialogue manager acts as an intermediary between the object and the user, translating between the languages of object interaction and user interaction. A dialogue manager is provided automatically by the environment.

The utility of our approach is demonstrated through discussion, examples, and implementation of its major components.

ACKNOWLEDGEMENTS

First, and foremost I would like to thank my advisor, Marvin Solomon. His constant support, guidance, and ideas helped make this dissertation a reality. I would also like to thank Charles Fischer for influencing almost all aspects of my graduate career. His superlative teaching contributed to my interest in and knowledge of programming languages and environments. Raphael Finkel and he, as members of my committee, provided invaluable help in the writing of this dissertation. Many thanks go to Tom Reps, Shaike Artsy, Matthew Thazhuthaveetil, Bruno Alabiso, David Notkin, Ed Canavan, Michael Cockrem, Hari Madduri, Michael Scott, and Ravishankar, who provided constructive comments on the presentation of my ideas. I would also like to thank Bill Kalsow and Bryan Rosenberg for suggesting that I pursue Fraser's idea of a generalized editor. I would like to thank Anoop Gupta for constantly sending me interesting technical reports from CMU. I am grateful to Allan Bricker and Greg Myrdal for being ever willing to help me with the Dandelions. I am also grateful to the members of the Crystal and Charlotte Projects, through whose efforts this research was supported in part by NSF grant MCS-8105904 and Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4095, monitored by the Naval Research Laboratory under contract No. N00014-81-C-2151.

Special thanks go to Anil Pal for being a constant friend and source of information on almost all topics in which I have been interested. Several other friends have also contributed to making my graduate career a pleasant one. Current and past residents and guests of the Pitz including Cathy, Madan, Monica, Allan, Patti, Chou, Kapoor, Kathy, Mohan, Pappu, Greta, Greg, Mary Beth, and Jim provided company and support, as did Ed, Valentina, Jorge, Mike, Ann, Cheryl, Beth, Victor, Michiko,

Angela and other past and current Knappers. David Aslakson, Viggy and other members of the Squash League made sure I maintained my sanity, while Yang, Artsy, Hari, Dan, Carolyn, Michael and other current and past officemates made it easy to sacrifice play for work. Toby and Kishore, fellow participants in the race to finish Summer '86 Dissertations, helped break the monotony of thesis writing by providing much needed company, ring toss competitions, and other fun diversions. Allan, Tad, Udi, Sheryl, Lorene, Henry, Donde, Chahal, Kelkar and other current and past members of the Computer Sciences Department made this place such a great one to be in.

Finally, I would like to thank my family for everything.

CONTENTS

Chapter 1

Introduction

In traditional interactive programming environments, the responsibility of providing the user interface of an interactive program is divided among the program itself, subroutine libraries, the programming language, and the operating system. The interactive program, however, shoulders most of the burden of providing its user interface. This allocation of responsibility has three drawbacks.

First, interactive programs are **bulky**. Typically, an interactive program is concerned with scanning and parsing input, reporting errors, converting correct input into variable values, and displaying results. The code to perform these tasks can be the major portion of an interactive program. A survey of commercial programs showed that display generation and management code constituted 40-60 percent of the source text of the programs sampled [43].

Second, there is **inconsistency** in the software environment. Different interactive programs usually offer different ways to enter operations, and often the same operation is called by different names. This problem is illustrated by considering how the 'delete' operation is invoked through different interfaces of a Unix environment. Assume that the environment comes with a window manager that can be used for interaction through a bit-mapped display. Deleting a window is typically done by using the mouse to select the appropriate operation name from a pop-up menu, deleting a file from a directory is done by typing the operation name 'rm' and then the file name, and deleting a character from a file is typically done by using the cursor keys to select the character and typing the operation name 'x'.

Third, most of the interfaces are **primitive** because they do not offer several 'friendly' features that are hard to implement. Examples of these features are menus and templates for input data, incremental feedback, operations to view information at various levels of detail, and operations to redo or undo other operations.

These three problems can be corrected if the user interface of an interactive program is automatically generated. Several previous approaches have attempted to provide this solution. However, these approaches have one or more of the following limitations, which are detailed in chapter 4:

- A large class of applications cannot use the approach.
- A programmer has to create *duplicate* descriptions of data structures in two different languages. As a result, the programmer has to learn multiple description languages, and is responsible for keeping duplicate descriptions consistent.
- A user is forced to interact with a *single program* at a time and a program is forced to interact with a *single user* at a time.

This dissertation presents a new approach that overcomes the above drawbacks. It consists of four components: an *editing* model of interaction, *objects*, *dialogue managers*, and a set of *input/output primitives*. The following subsections describe these components briefly.

1. Editing Model of Interaction

Our approach supports the *editing* model of interaction, which has been shown to be suitable to interact with general applications [6, 7, 28, 30, 38, 3, 4]. The model allows the user to view all applications as data that can be 'edited'. We illustrate the model through Fraser's example of a directory manager that allows the user to manipu-

late a directory by editing its listing [6].

Figure 1 illustrates how a user may edit a Unix-style listing of a directory. Figure 1(a) shows an initial listing of a directory composed of two entries. A user can edit the listing to modify the directory. For instance, he can edit a name field to change the name of an entry (figure 1 (b)), edit an access field to change an access right to an entry (figure 1 (c)), or delete a line to remove an entry from the directory (figure 1(d)).

Thus the interface presented by the directory manager to manipulate a directory is similar to the interface presented by a text editor to edit a text file. There is, however, an important difference between the two interfaces. The directory manager cannot allow the user to make *arbitrary* changes to a directory listing. For instance, in the above example, it cannot let the user insert the character 'q' in an access field, it cannot allow the editing of the date and size fields, and it cannot let an unauthorized user change the editable fields of a directory.

It is easy to see how other applications may present editing interfaces. A 'process manager' can allow a user to edit a visual representation of the current processes to insert a process, delete it, or change its status. A 'printer manager' can allow a user to edit a representation of the printer queue to submit and remove files for printing. An executive may allow a user to enter commands by editing a representation of the history of previous commands. In general, any interactive application can present an editing interface by displaying a visual representation of its data, allowing the user to edit the representation in a syntactically and semantically consistent fashion, and reacting appropriately to a change in the representation.

ACCESS	LINKS	OWNER	SIZE	DATE	NAME
drw-rw-r--	1	joe	512	Jun 23 1985	src
-rw-rw-r--	1	joe	111	Oct 13 1984	todo

a) Initial Listing

drw-rw-r--	1	joe	512	Jun 23 1985	src
-rw-rw-r--	1	joe	111	Oct 13 1984	done

b) User Edits File Name

drw-rw-r--	1	joe	512	Jun 23 1985	src
-rw-rw-rw-	1	joe	111	Oct 13 1984	done

c) User Edits Access Field

drw-rw-r--	1	joe	512	Jun 23 1985	src

d) User Deletes File

Figure 1: Editing a Directory

The editing model is not only general, but also has the following pleasant properties:

- It has been *successfully used* by several applications such as text editors, spreadsheets, language-oriented editors [5, 44, 11, 32, 50, 51], form editors [37, 35], and document editors [41, 20].
- It leads to *uniformity* since the different interfaces share a common set of editor commands. For instance a single 'delete' command can be used to delete a file from a directory, a process from the list of active processes, a file from a line printer queue, and a user from the list of current users.
- It removes the traditional distinction between 'edit time' and 'run time'. In traditional environments, the interaction with a typical application can be divided into two phases. During the first phase, a text editor is used to compose the input of a program. Subsequently, during the 'run time' of the program, the input is checked for errors, and output is produced. The editing model combines these two phases into a single phase. As a result the user receives *incremental feedback.*

2. Objects

Our approach augments a traditional programming environment with extensions to support the encapsulation of processes and the files they manipulate into indivisible units called **objects.** Objects acquire properties of both processes and files. Like processes, they are *active* entities capable of executing code. However, unlike processes, but like files, they can become part of the *permanent memory* of the system.

3. Dialogue Manager

Between each object and a user is a **dialogue manager**. A dialogue manager implements the user interface of an object. It displays one or more visual representations or *presentations* of the data in the object and provides a user an editing interface to modify these presentations. It announces changes to a presentation in messages to the object. Similarly, it updates a presentation in response to messages from the object.

Thus from the point of view of objects, the user appears to be another object that can send and receive messages. From the point of view the user, the objects appear to be data that can be edited. The dialogue manager acts as an intermediary between the object and the user, translating between the languages of object interaction and user interaction (Figure 2).

A dialogue manager is provided *automatically* by our approach. As a result, an application programmer is concerned only with the specification of the user interface of an object and not its implementation.

4. Input/Output Primitives

Our approach replaces traditional input/output procedures supported by conventional programming languages with a new set of input/output primitives. Traditional input/output procedures have several deficiencies that make them incompatible with an approach that supports both *editing* and *automation*. We illustrate these deficiencies by taking the example of the **read** and **write** procedures of Pascal.

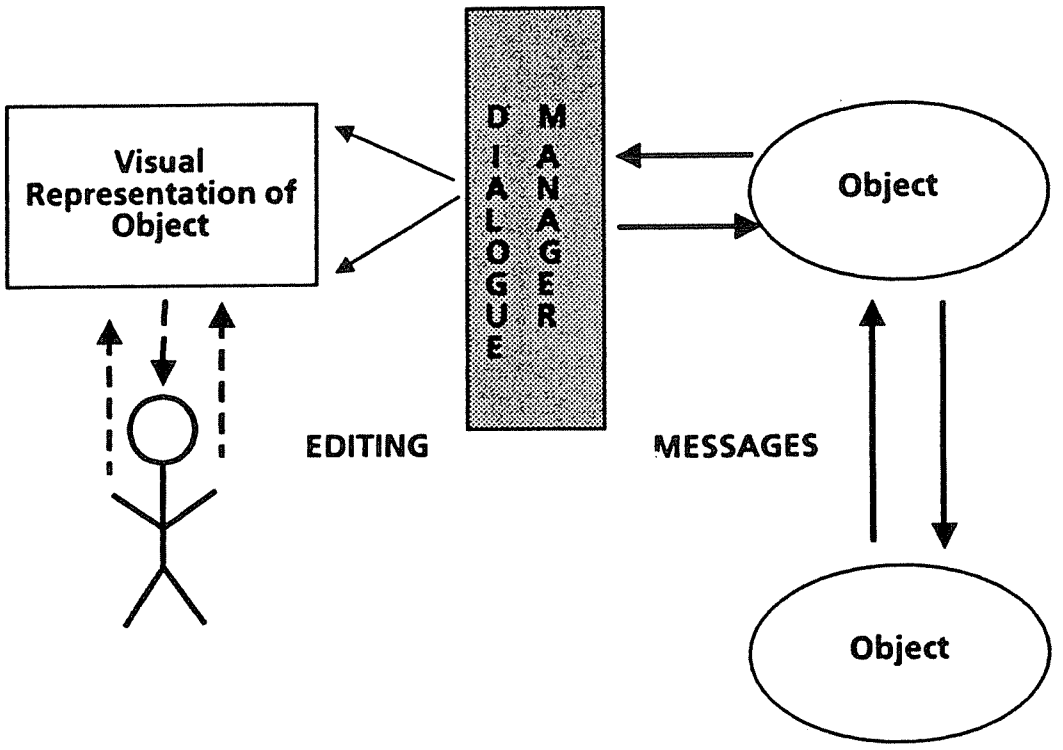


Figure 2: A Dialogue Manager Translating between the Languages of Object Interaction and User Interaction

The **read** procedure in Pascal forces a user to input data items *sequentially*. The editing model, however, allows a user to edit several data items non-sequentially.

The **read** procedure advances the input buffer of an input stream. As a result, a user cannot edit previous input and resubmit it to the program as new input. Similarly, **write** advances the output buffer of an output stream. As a result, a program cannot display new results by changing previous output. Finally, read and write streams are disjoint. Therefore, a user cannot edit output and present it as input, nor can a program update an input value displayed on the screen.

The parameters to **read** and **write** are predefined values like integers, reals, and Booleans. Input and output of programmer-defined values is achieved by *casting* them into these predefined values. As a result, the programmer has to be concerned with the substantial work of 'unparsing' programmer-defined values into predefined values, and 'parsing' predefined input values into the corresponding programmer-defined values.

The input/output primitives supported by our approach overcome the above drawbacks of conventional input/output procedures by providing the following:

- *Modification of Previous Input/Output*: A user can enter a new value for a variable by editing its previous value. Similarly, an object can output a new value of a variable by updating its previous value.
- *Non-Sequential Input*: A user can edit the variables displayed in a presentation in any order. Our approach allows an object to react to changes to any one of the variables displayed in a presentation.
- *Input/Output of Programmer-Defined Values*: An object can input and output values of programmer-defined types such as arrays, records, variant records, and recursive structures referred by pointers.

- *Display Properties*: A variable is associated with display properties, which an object or user may change to control the format and other characteristics of the variable.
- *Inheritance Tree*: The display properties of variables are arranged in an inheritance tree. This tree provides a powerful mechanism to define default formats.

5. The Thesis

The thesis of this dissertation is two-fold: first, a *general* approach for automatic generation of user interfaces exists; second, the approach can be effectively implemented on top of existing software environments.

The rest of the dissertation is organized as follows. Chapter 2 describes the approach through the example of the *Dost* environment. Dost is an extension of the Mesa-based Xerox Development Environment (XDE) [48, 49], and its major parts have been implemented on the Xerox 8010 (Dandelion) workstation.

Chapter 3 describes how Dost may be used, both to create classes and to interact with their instances. It describes in detail six classes, representing a diverse range of applications. For each class, it describes the user interface available to interact with instances of the class, and the code required to create the class.

Chapter 4 describes the role played by different components of our approach, discusses some useful features missing from it, describes our experience with Dost, compares our work with related work, and presents conclusions and directions for future research.

Chapter 2

Dost

In this chapter, we describe our approach through the example of *Dost*[1]. Dost is an extension of the Mesa-based Xerox Development Environment (XDE), and its major parts have been implemented on the Xerox 8010 (Dandelion) workstation. We first give an overview of the environment. Next, we describe the four components of our approach: objects, dialogue managers, the input/output primitives, and the interaction model. Finally, we discuss the main features of the implementation.

In this dissertation we shall use the term Dost to express both the environment and the approach it illustrates.

1. Overview

Dost differs from XDE and other traditional environments in several important ways. Application programmers do not write programs; instead they create **classes.** Similarly, users of the system do not 'run' programs, instead they 'edit' **objects.** An object is an ***instance*** of a class. The class describes the data encapsulated by the object and the **methods** to manipulate them. The methods of an object are invoked in response to **messages** from other objects and users (through dialogue managers).

An object is associated with one or more **presentations** that display the data encapsulated by the object. Changes in the presentation cause corresponding changes

---

[1] *Dost* is a word in Urdu meaning *friend.* The name expresses our hope that both users and application programmers will regard Dost as a friend.

in the object. Similarly, changes in the object due to internal changes or messages from other objects cause its presentations to be updated.

Between each object and a user is a **dialogue manager**. A dialogue manager handles user interaction on behalf the object. It offers the user a *structure editor* interface to modify the presentations of an object. It announces user-caused changes to a presentation in messages to the object. Similarly, it updates a presentation for the user in response to messages from the object.

Thus from the point of view of objects, the user appears to be another object that can send and receive messages. From the point of view the user, the objects appear to be data that can be edited. The dialogue manager acts as an intermediary between the object and the user, translating between the languages of object interaction and user interaction.

A dialogue manager is provided *automatically* by the environment. As a result, an application programmer is concerned only with the specification of the user interface of an object and not its implementation.

1.1. Example

We illustrate the main features of Dost through a sample class 'Bibliography', which defines bibliography databases. Each instance of the class manages a database, and allows a user to add, delete, and modify entries. We first describe how a user interacts with an instance, and then how a programmer defines the class.

To interact with an object, the user first loads its presentation(s) into a Dost *window*. A Dost window is like an XDE text window except that it is managed by a dialogue manager instead of a text editor. An object of an arbitrary class can be edited in

such a window.

Figure 1 shows the structure of an empty Dost window. Let us assume the user wants to create a new instance of the class 'Bibliography'. He fills the class and instance ('myBib') names in the appropriate fields of the window, and selects the 'load' command. Figure 2 shows the result of executing the command. A new object called 'myBib' is created, and its presentation is loaded in the window. Initially, the object contains no entries. Therefore its presentation contains the *placeholder*

<ReferenceList>

which indicates that a new list of reference entries may be added to the presentation.

Figure 3 shows the sequence of actions a user may employ to replace this placeholder with a list of references. Each box displays the current presentation of the object. An arrow indicates an editor command invoked by the user, and the shaded text in a box indicates the operand of the editor command.

The user can *expand* the initial placeholder to get a template for a new reference. The template contains placeholders for the fields of the entry. In our example, these may be replaced to enter the author name, title, and the kind of entry desired. An entry may be a reference to a journal, book, or technical report.

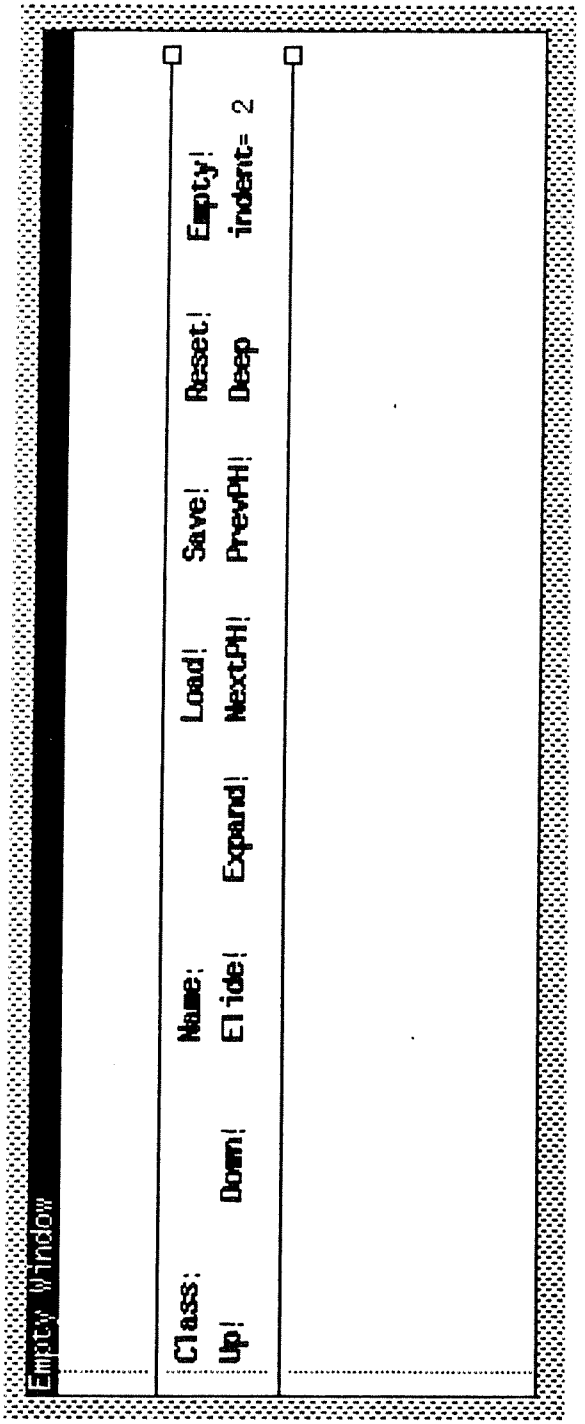


Figure 1: An Empty Window

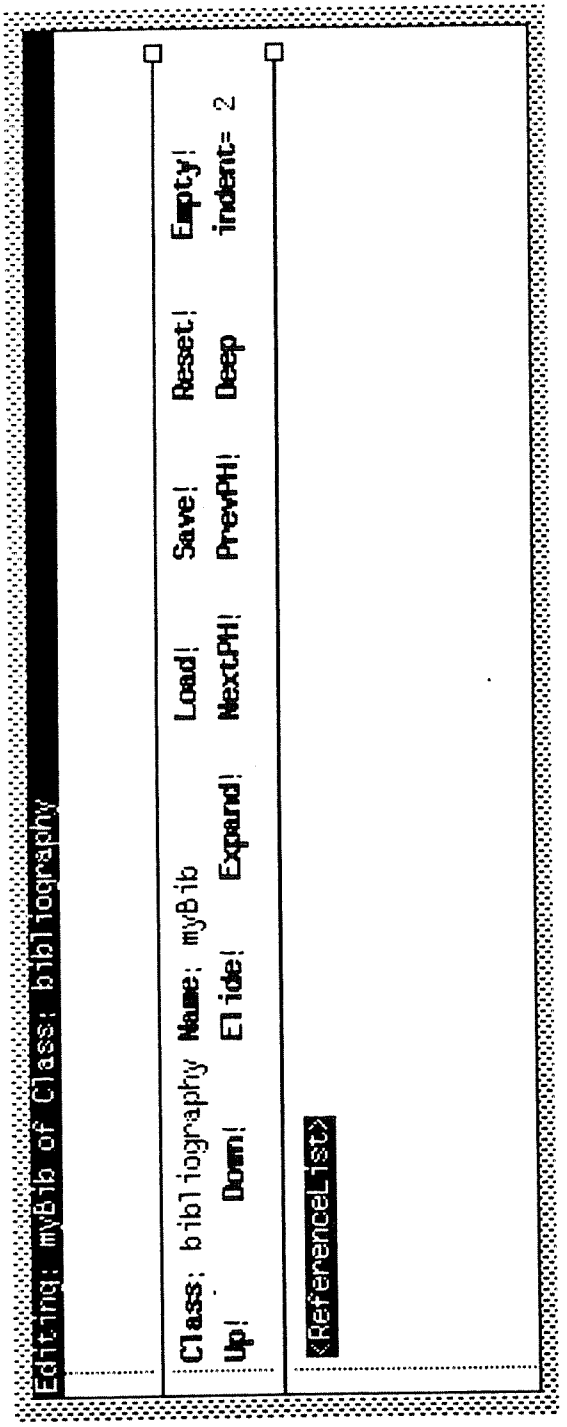


Figure 2: A Loaded Instance of 'Bibliography'

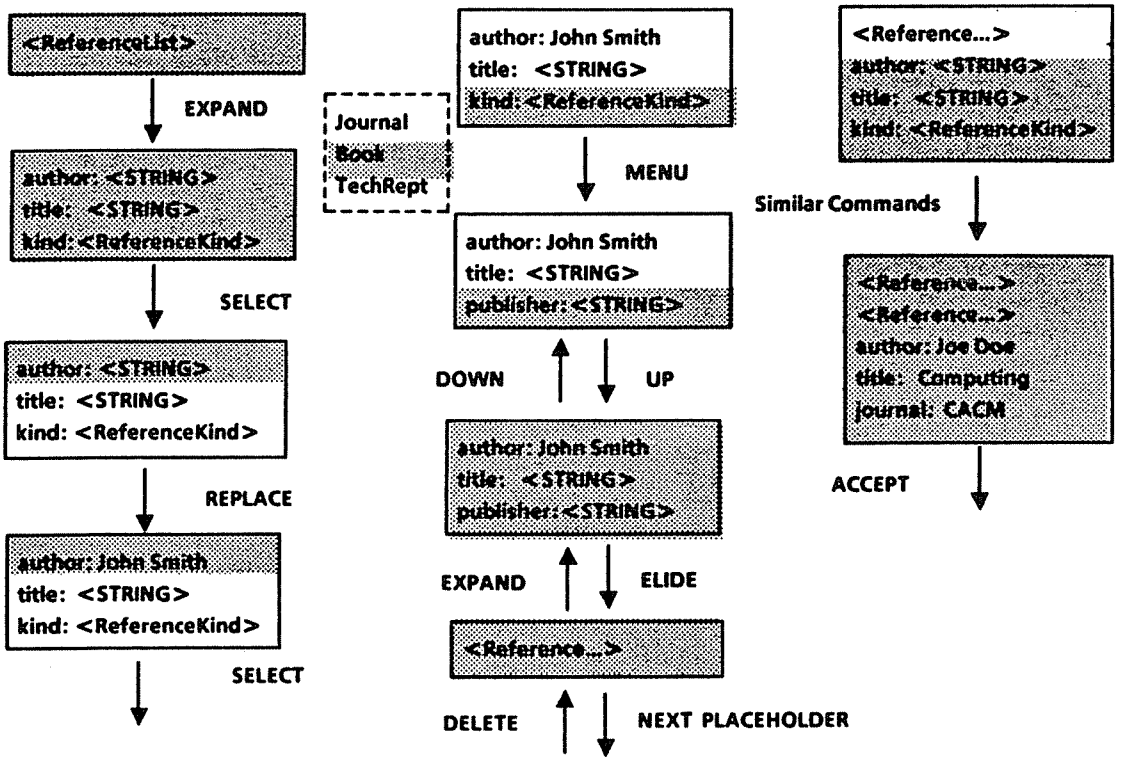


Figure 3: Editing a New Instance of "Bibliography"

To enter the 'author' field, the user first *selects* it, and then *replaces* the placeholder with the appropriate name. The 'title' field may be entered in a similar fashion. The user chooses to consult a menu to enter the 'referenceKind' field. In the example, the user selects a 'Book'. In response, the dialogue manager replaces the field with fields appropriate to a book reference. In this example, there is only one such field, which prompts the user for the publisher of the book. If the user were to select 'Journal' or 'TechRept', the dialogue manager would prompt the user for a journal name or a university name.

Figure 3 illustrates other commands available to the user. The user can select the whole entry by executing the *up* command, and then hide its details by executing the *elide* command. The effects of these two commands may be reversed by executing the *expand* and *down* commands respectively. The *next placeholder* command may be executed to insert a template for a new entry. (This entry may be removed by executing the *delete* command.) The user may now fill this entry, and add other entries using similar commands. After specifying all the elements of the list, the user may execute the *accept* command. At this point, the new list of entries is sent to the object.

Figure 4 shows the text of the class 'Bibliography', which is written by the application programmer to support the user interface we have just seen. The class declares the variable 'refList', which stores the database of reference entries. The variable is a pointer to a record containing a Mesa *sequence* of 'Reference' records. (A Mesa sequence is an array of variable size). A 'Reference' is a variant record discriminated by the 'referenceKind' field. These declarations are used by the dialogue manager as a guide for the user interface of instances of the class.

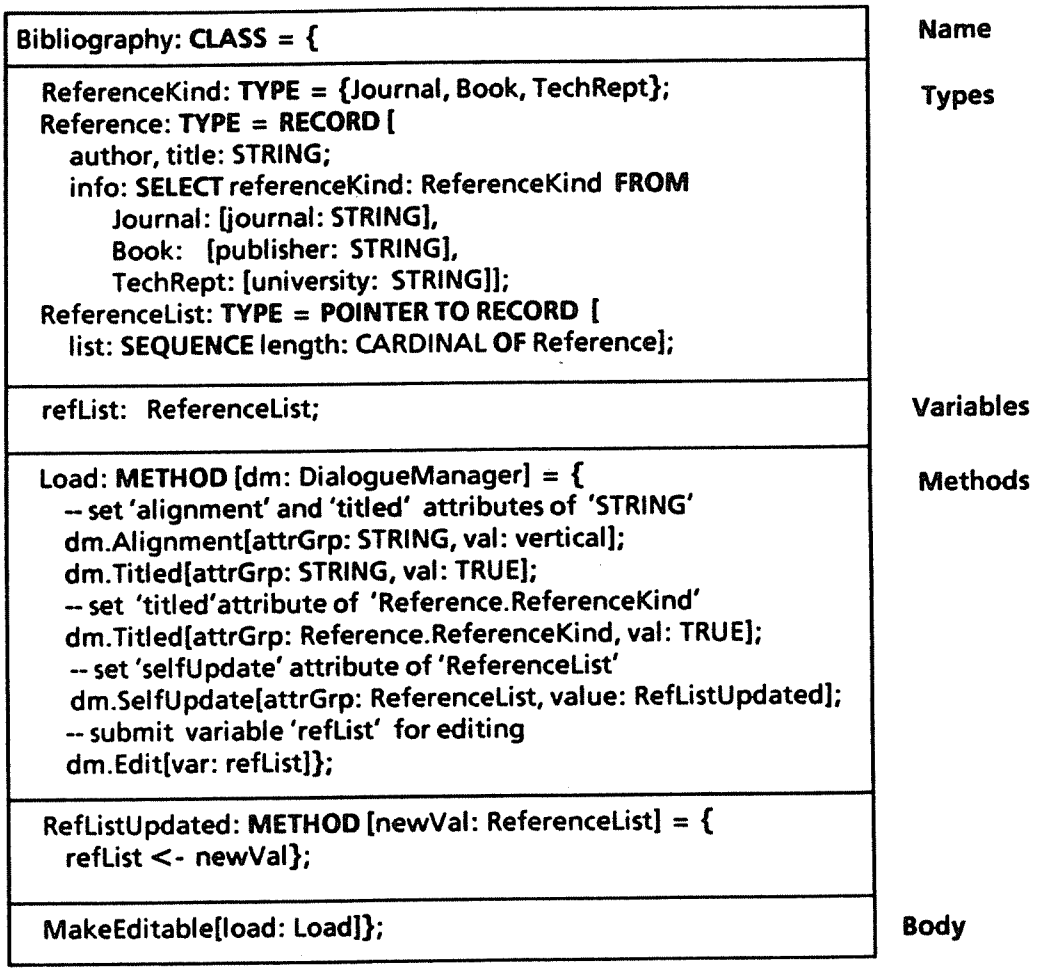


Figure 4: The Class 'Bibliography'

The body of the class (which is executed by each new object) is the call *MakeEditable*, which expresses the caller's willingness to interact with users through dialogue managers. The call registers with the system the method 'Load' as the *load* method of instances of the class. This method is invoked when the user asks a dialogue manager to load an instance in its window. Its parameter 'dm' contains the name of the dialogue manager. The object uses this name to send messages to the dialogue manager.

The first two messages sent by 'Load' ask the dialogue manager to associate certain display properties with all variables of type 'STRING' in the object. The first asks the dialogue manager to align these variables vertically, and the second asks it to 'title' them. (If a variable is 'titled', its name precedes its value. For instance, if an 'author' field of a 'Reference' record is 'titled', the name 'author' precedes the display of the string value of the field). The third message asks the dialogue manager to 'title' the discriminant 'referenceKind' in variables of type 'Reference'.

The fourth message tells the dialogue manager about the method 'ReferenceListUpdated', which is to be called when variables of type 'ReferenceList' are updated. The last message in 'Load' asks the dialogue manager to display variable 'refList' as part of the presentation of the object. The dialogue manager uses the information in the type declarations of the class and the initialization sent by 'Load' to allow structure editing of the variable in the manner described earlier. Later, when the user executes the *accept* command, the dialogue manager calls 'ReferenceListUpdated', passing the new list of reference entries in the 'newVal' parameter. The object uses this value to update its version of the list stored in 'refList'.

Figure 4 demonstrates the automation in the environment. The class declaration contains very little code to handle interaction with the user: the call 'MakeEditable' and the 'Load Method'. This code 'drives' the dialogue managers of instances of the class, which implement the user interfaces of these instances and are provided by the environment.

2. Objects

2.1. Overview

Our approach augments a traditional programming environment with extensions to support *classes* and *objects*. These extensions support the editing model of interaction, as discussed in chapter 4.

Classes in Dost are a cross between Smalltalk [12] classes and traditional programs. Similarly, objects in Dost are a cross between Smalltalk objects and traditional program instances. By a program we mean a separately compiled and linked unit of code, and by a program instance we mean an execution instance of a program.

We illustrate the nature of Dost classes and objects by comparing them with their counterparts in Smalltalk and traditional environments respectively.

Comparison with Smalltalk

A Dost class is similar to a Smalltalk class in that it defines the data encapsulated by its instances and the operations to manipulate them.

A Dost object is similar to a Smalltalk object in two respects. First, it is part of the permanent memory of the system. Thus it has a *permanent name*, and can be *activated* and *passivated*. Second, it can *communicate* with other objects through messages.

However, there are several key differences between Dost and Smalltalk, which stem from the fact that Dost is an extension of a traditional environment.

First, Smalltalk is a *total-object* system, that is, each entity in the system is an object. Dost on the other hand, is a *partial-object* system. Objects in Dost are special entities that coexist with 'smaller' entities such as integers, reals, and other data described by Pascal-like type declarations.

Second, Smalltalk classes share code through the mechanism of *inheritance*. In Dost, classes are extensions of Mesa programs, and share code by using Mesa constructs for importing and exporting declarations. Thus, the destination of a message is determined at execution time in Smalltalk, but is bound at compile time in Dost.

Third, in the Smalltalk environment all objects are in a single virtual address space and are activated and passivated *automatically* by the paging system. In Dost, objects are not forced to share a single virtual address space and the system activates and passivates objects in response to *explicit* requests from the active objects and dialogue managers in the system. Moreover, an object is responsible for reading its data structures from permanent storage when it it is activated and writing its data into permanent storage when it is passivated. This feature keeps the design and implementation of Dost simple. Moreover, it allows our approach to be applied to (more or less) traditional environments.

Comparison with Traditional Environments

We now describe the nature of Dost objects and classes by comparing them with their counterparts in a traditional environment. Since the Dost environment is an extension of XDE, we use the latter as a representative example of a traditional environment.

A Dost object is similar to a Mesa program instance except for two differences. First, an object is part of the permanent memory of the system, much as a file is a part of the permanent memory of a traditional system. Thus it has a permanent name, and can be activated and passivated.

Second, an object can receive messages from other objects. Sending a message to an object in Dost is similar to invoking a procedure in a Mesa program instance. The difference is that messages can be exchanged between arbitrary objects in Dost, while in XDE arbitrary program instances cannot name and communicate with each other.

A Dost class is like a Mesa program except that it contains constructs to support objects. These constructs are described in detail below.

2.2. Constructs to Support Objects

Naming an Object

Object names are used by other objects and users. The cognates of objects in traditional systems are files, so their naming scheme is borrowed from files. Each object has an ascii *permanent name*, which is global to all users and objects. A user may use this name to specify an object for editing. An object may use the name to get a pointer to the named object, as described in the next section.

Object Pointers

Dost associates a type with each class in the system, and allows variables of these types to point to instances of the classes. We call such variables *object pointers*. The following is an example of a declaration of an object pointer to an instance of class 'Bibliography':

```
bibPtr: CLASS Bibliography;
```

An object pointer, like a Pascal file pointer or a Mesa file handle, may be bound to an existing object by a call to the predefined procedure *OpenObject*. The procedure takes the name of the object and returns a pointer to it. Thus the call:

```
bibPtr <- OpenObject ["myBib"]
```

returns a pointer to the object 'myBib' in 'bibPtr'. Dost creates a new object of the current class, if one with the specified name does not exist. (The body of the new object is executed before the pointer is returned, allowing it to make itself editable for example)

Activation and Passivation of Objects

The interaction model supported by Dost allows each object to represent *permanent* data that can be saved between editing sessions. Therefore, it is important that the environment provide support for permanence in objects.

A programmer may make a class of objects permanent by including the parameter *dataFile* (of type 'STRING') in the declaration of its parameter list. Dost associates each instance of such a class with a unique file called its *data file*, which may be used by the object to store its stable data structures. When an existing instance is activated, or a new instance is created, the 'dataFile' parameter contains the name of the data file of the instance. The object can read its data structures from this file.

The object may ask Dost to unload it from memory by calling the predefined procedure *Passivate*, which takes no arguments. It is expected to save its data structures in its data file before it calls the procedure. Moreover, it is responsible for ensuring that other objects and dialogue managers do not refer to it. (Our approach assigns

these responsibilities to an object to keep the design and implementation simple.)

The following class fragment demonstrates how objects may be activated and passivated:

```
ActivePassive: CLASS [dataFile: STRING] = {
    -- declarations of variables
    ...
    -- a procedure that saves the 'permanent' data structures into a file
    SaveDataStructures: PROCEDURE [fileName: STRING] = {
      ...};
    -- a procedure that reads the 'permanent' data structures from a file
    ReadDataStructures: PROCEDURE [fileName: STRING] = {
      ...};
    -- a procedure that decides to passivate the object
    CallPassivate: PROCEDURE = {
      SaveDataStructures [dataFile];
      Passivate []};
    -- main body, called on each activation of the object
    ReadDataStructures[dataFile]};
```

Methods and Messages

A class may contain method declarations, which define the methods that can be invoked in its instances. Method declarations are similar to Mesa procedure declarations. The differences are that the keyword **PROCEDURE** is replaced with **METHOD**, a method can be invoked from another object, and methods may be declared only in the outermost scope of the class. Figure 4 illustrates how methods are defined; it contains declarations of the methods 'Load' and 'RefListUpdated'.

A method is invoked in response to a message from an object or a dialogue manager. The following is the syntax for sending a message:

<ObjectPointer>.<Message>

where <ObjectPointer> is a pointer to the destination object, and can be omitted if the

source and destination objects are the same. The syntax of <Message> is the same as the syntax of a Mesa procedure call. The semantics of sending a message are similar to the semantics of a Mesa procedure call except that a message can be sent to another object.

(A Mesa procedure call is similar to a Pascal procedure call. The main differences between the two are that in Mesa (1) parameters may be specified using keyword or positional notation, (2) arguments not explicitly specified may be supplied by default, and (3) all arguments are passed by value. For instance, given the procedure definition

```
ExampleProc: PROCEDURE [p1: INTEGER <- 0, p2: BOOLEAN <- TRUE] =
    {...};
```

the following are valid calls:

```
ExampleProc [1, FALSE]; -- positional notation
ExampleProc [p2: FALSE, p1: 1]; -- keyword notation
ExampleProc [p2: FALSE]; -- the default value '0' is assigned to 'p1')
```

3. Dialogue Manager

An object interacts with the user through one or more *dialogue managers*. A dialogue manager defines two interfaces: one for the user, and the other for the object. The former describes the user interface of the object, and the latter describes the set of input/output primitives available to the object.

At any moment, one or more dialogue managers may be active in Dost. Each dialogue manager displays a *window* on the screen. A user starts interaction with an object by asking a dialogue manager to display the presentations of the object in the window. In response, the dialogue manager asks the object for the names of the vari-

ables that need to be displayed in the presentations of the object (by calling the load method). The object responds by sending the appropriate information. The dialogue manager uses this information and its knowledge about the *types* of these variables to present to the user a *default* interface to edit the variables. An object may use the input/output primitives provided by the dialogue manager to *receive* values of variables edited by the user, *update* the values of displayed variables, and *override* the interface presented by the dialogue manager.

4. Input/Output Primitives

4.1. Overview

The choice of our input/output primitives was governed by our goal of automating the generation of editing interfaces while giving a programmer flexibility to choose the characteristics of an interface. We describe below the main features of these primitives:

Non-Sequential Input

The editing model allows a user to edit the variables displayed in a presentation in any order. Our approach supports this model by allowing an object to react to changes in any one of the variables displayed in a presentation.

Display Structure

A dialogue manager *saves* the values input and output by an object in a display structure. The user inputs new values by *editing* the values stored in this structure. The object outputs new values by *updating* the values stored in the structure.

Input/Output of Programmer-Defined Values

An object can input and output values of *programmer-defined* types. These include arrays, records, sequences, records, variant records, and structures referred by pointers. The dialogue manager converts user input into the internal representation of these values. Conversely, it converts the internal representation of these values into images on the screen. Thus a program is not concerned with the task of *parsing* input to determine a programmer-defined value. Similarly, it is not concerned with the task of *unparsing* programmer-defined values into output on the screen.

Display Properties

A variable is associated with display properties, which an object or user may change to control the format and other characteristics of the variable.

Inheritance Tree

The display properties of the variables displayed by an object are arranged in an *inheritance tree*. The nodes of the tree *inherit* display properties from their parents, which they can *override*. The leaf nodes of the tree are associated with the displayed variables. The inheritance tree provides a powerful mechanism to define default formats.

The following subsections discuss the Dost input/output primitives in more detail. In particular, they answer the following questions:

- What are the primitives provided to support object-dialogue manager communication?

- How does an object display variables in its presentations?
- How can an object control the format of a displayed variable?
- How are user changes to a presentation buffered?
- How does an object receive changes to its presentations?
- How does an object specify the 'granularity' of the change transmitted to it?
- How are user errors handled?
- How can an object control the set of commands available to edit a variable?
- How can *automation*, which frees an object from the details of user interface tasks, and *flexibility*, which allows an object to tailor the interface according to its needs, *both* be provided?

4.2. Object-Dialogue Manager Interaction

An object communicates with users through one or more dialogue managers. In this section we discuss the basic primitives provided in Dost for object-dialogue manager communication.

Naming a Dialogue Manager

In Dost, an object names a dialogue manager the same way it names other objects. A predefined type *DM* is provided, which may be used to declare pointers to dialogue managers and send messages to them. The syntax for sending messages to dialogue managers is the same as the syntax for sending messages to ordinary objects. The 'Load' method in figure 4 shows how an object names a dialogue manager and communicates with it.

Naming Presentations

Dost allows an object to display several presentations in different *presentation subwindows* of a window. Thus an object can present several 'views' of itself. For instance, a spreadsheet manager can show one view displaying the values of the spreadsheet and another view displaying expressions that define the relationship between these values.

An object names a presentation by using the *subwindow number* of the subwindow in which the presentation is displayed.

Setting up Object-Dialogue Manager Communication

An object is connected to a dialogue manager when a user loads its presentations in a window. This section discusses the primitives to set up communication between an object and a dialogue manager.

An object can call the procedure *MakeEditable*, to express its willingness to interact with users through dialogue managers. Through this call an object tells Dost its *load* method and (optionally) the number of presentations it has. (This number is fixed in Dost to keep the implementation simple. Its default value is 1.) The load method is invoked in an object when a new dialogue manager is attached to it, and receives in its parameter a pointer to the dialogue manager. An object may use this pointer to send messages to the dialogue manager.

The class 'Bibliography' shown in figure 4 illustrates how object-dialogue manager communication is set up. The main body calls the procedure 'MakeEditable', which submits the method 'Load' as the load method.

An object may be attached to several dialogue managers at the same time. Thus an object may be edited in several windows simultaneously. The load method is called each time a user loads an object in the window managed by a dialogue manager.

4.3. Displaying Variables

An object displays a variable by sending an *Edit* message to a dialogue manager, which asks the dialogue manager to append a presentation of a variable to the contents of a presentation subwindow. The arguments of the message name the subwindow number and the variable to be displayed. The subwindow number is an optional argument and defaults to zero.

An object may use 'Edit' messages to display variables of predefined and programmer-defined types. As a result an object may include in its presentations the display of not only integers, reals, and strings, but also, arrays, records, sequences, variant records, and pointers.

4.4. Display Nodes and Attributes

A presentation of an object is composed of presentations of one or more **display nodes**. A display node is associated with a variable displayed by the object and contains editing information about the variable. A presentation of a display node displays a 'formatted' value of the variable. A variable may be associated with *several* display nodes, each providing a different 'view' of the variable.

A display node acquires the type of the associated variable, which determines its various characteristics or **attributes**. One of the attributes is the *value* attribute, which contains the value of the variable associated with the node. Another attribute is the *initialized* attribute, which determines if the node is considered initialized or

uninitialized. This attribute influences two decisions: first, if the value of the node is displayed to a user (§ 4.6), and second, if the value is sent to the object (§4.7 & §4.8).

The other attributes, discussed later, determine:

(1) the presentation of a display node.

(2) the mechanism by which an updated value is communicated to the object.

(3) the commands available to the user to edit the presentation of the node.

A display node is created by a dialogue manager either in response to an 'Edit' message from the object or as a result of the execution of an editor command by the user. Each 'Edit' message creates an initial **display tree** consisting of display nodes for the variable and its components. The tree reflects the current value and structure of the variable, and may be modified by the user commands that add or delete nodes from the tree. The following discussion uses the 'Bibliography' example to illustrate the nature of display trees of enumerations, records, variant records, and sequences, and structures referred by pointers.

Figure 5 shows initial display trees rooted by display nodes of of type 'ReferenceKind', 'Reference', and 'ReferenceList'. Each node is labelled by its type name, if defined, or the constructor used to create the type.

Simple Type

Figure 5(a) shows an initial display tree created in response to an 'Edit' message that submits a variable of the enumeration 'ReferenceKind' for editing. The tree consists of a single node, which is initialized with the value of the variable. The structure of this tree, like the structure of the corresponding variable, is fixed, and no nodes may be added or removed from the tree.

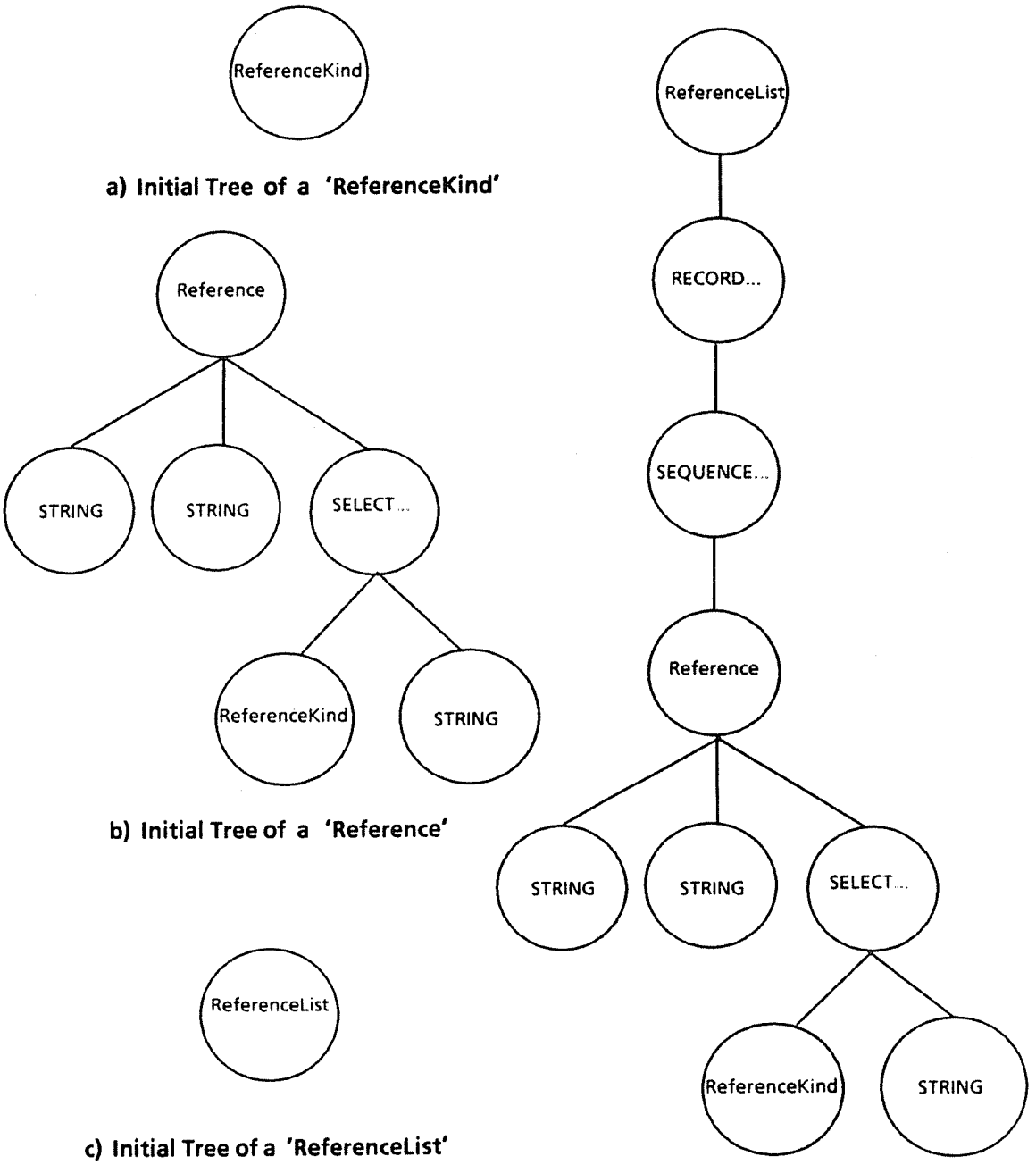


Figure 5: Display Trees

Variant Record

Figure 5(b) shows a display tree created in response to a message that submits a variable of the record type 'Reference' for editing. The tree contains display nodes for the fixed fields, the tag field, and the variant fields of the record. These are initialized with values determined by the value of the variable. In this example, the structure of the tree is independent of the value of the tag field of the record. In general, however, the structure of a display tree reflects the structure of the selected variants. A user may alter the tree by changing the tag values.

Sequence Pointers

The description of the display tree for a variable of type 'ReferenceList' requires a discussion of Mesa sequences. A Mesa sequence type is considered to be a union of some number of array types, just as the variant part of a variant record type is a union of an emumerated collection of record types. This has the following consequences:

> A sequence type can be used only to declare a field of a record. At most one such field may appear within a record, and it must occur last.
>
> A sequence-containing record has a tag field that specifies the length of a sequence, and thus the set of valid indices for its elements.

Mesa also places the restriction that sequence-containing records have to be allocated dynamically and referenced through pointers. Thus in the class 'Bibliography', the pointer 'referenceList' is used to define a sequence. We shall refer to pointers to sequence-containing records as *sequence pointers*.

Figure 5(c) shows a display tree for a variable of type 'ReferenceList'. The tree consists of the singleton node of type 'ReferenceList', which is initialized with the

value of the variable. The user may 'dereference' a pointer by selecting its node and executing the 'expand' command. In response the dialogue manager creates a display tree for a variable of the type referenced by the pointer. In this example, the tree consists of display nodes for the record, its sequence field, elements of the sequence, and fields of the elements, which make the leaf nodes of the tree.

The nodes of the tree have default values (§4.10) if the pointer variable 'refList' stores the 'NIL' value. Otherwise they are initialized with values determined by the value of the referent of the pointer. Figure 5(d) shows a display tree for 'refList' after a user has executed the 'expand' command. We assume here that the value of 'refList' is not 'NIL' and there is one element in the sequence. The user may add new elements to the sequence by executing the 'next placeholder' and 'previous placeholder' commands, and delete them by executing the *delete* command. The user may also delete the complete subtree rooted by the pointer node.

A new display tree is created every time an object submits a variable for editing, or the user 'dereferences' an 'unexpanded' display node of a pointer variable. Thus, if an object submits the same variable twice for editing, the dialogue manager creates two trees with disjoint nodes. Similarly, if the user executes the 'expand' command on two different 'unexpanded' display nodes of the same pointer variable, the dialogue manager creates two different trees. It makes no effort to keep these trees consistent—it is the object's responsibility to do so. An object may use the 'broadcast' facility described in §4.4 to keep different nodes displaying the same variable consistent.

These semantics of creating display trees associate an *infinite* tree of display nodes with a recursive structure. The dialogue manager uses an algorithm of *lazy dereferencing* to prevent itself from creating such a tree. The initial display tree for

any node does not contain 'dereferenced' pointer nodes. These nodes are 'dereferenced' on *demand* from the user, each time the 'expand' command is executed on an 'unexpanded' pointer node. As a result an infinite display tree is never created.

The lazy dereferencing algorithm is illustrated by the 'Bibliography' example. If an object submits a non-NIL value of 'refList' for editing, the initial display tree is as shown in figure 5(c). A user has to explicitly 'dereference' the node to display the value to which the pointer refers.

4.5. Naming a Display Node

A display node may be named either by the object or the dialogue manager. An object names a display node to change the 'value' and other attributes of the node. A dialogue manager names it to inform the object about user changes to the 'value' and other attributes of a display node.

In this section we discuss our approach to naming display nodes. The section is organized into two parts. The first part discusses some of the factors we had to consider in the design of the approach. The second part describes the approach.

Issues

The following factors had to be considered:

- An object or a dialogue manager may name *any* display node, including internal nodes in a tree.
- An object receives node names in arguments of methods invoked by a dialogue manager. Therefore, it should be able to declare variables that store node names.
- A dialogue manager uses node names to inform an object about user changes to display nodes. In response, the object often updates the variables displayed by

the nodes. Therefore an object should be able to easily deduce from a node name the variable displayed by the node.

- Conversely, an object should be able to easily refer to the display nodes of a variable in response to changes in the variable.
- Display nodes can be updated with only those values that are allowed by their types. The naming scheme should allow errors in updating display nodes to be caught at compile time.

Approach

We now describe our approach to naming display nodes.

A node may be named by a a *node name* or a *path name*. The node name directly names the node, while the path name addresses the node by specifying the node name of one of its ancestors, and the 'path' from the ancestor to the node.

A node name may be a *variable name* or a *node number*. A variable name names a node through the variable, while a *node number* addresses a node through an 'index' that identifies the node to the dialogue manager.

Variable Name

The variable name of a node allows an object to think of the node in terms of the variable it displays. It is actually represented by the address of the variable it displays, and may therefore be stored in pointer variables. The following are valid variable names for nodes in a display tree for variable 'refList' of figure 4, assuming that the value of 'refList' is not NIL when it is submitted for editing, the sequence component has at least 1 element, and variable 'p' holds a pointer to 'refList':

@refList
p
@refList[0]
refList[0]
refList[0].author

The first and second names provide the address of the variable 'refList', and refer to the display node for the variable. The third and fourth names refers to the display node for the first element in the sequence field. In the fourth name, the object does not explicitly give the address of the variable. Since the variable is not a pointer variable, the system automatically 'references' it. Similarly, the last name refers to the display node for the 'author' field of the first element of the sequence.

An object can use a variable name only if the node it names is ***registered.*** The root nodes of display trees are always registered. A Boolean attribute ***registered*** determines if other nodes are registered or not. This attribute is defined to allow efficient translation in the dialogue manager from a variable name to an 'index' that points to the node in the data structures maintained by the dialogue manager. This translation process may be expensive if every display node is addressable by a variable name. The 'registered' attribute ensures that a dialogue manager searches only registered nodes. Typically, there will be only a few nodes that need to be be named by variable names. The others either do not need to be named, or can be specified by node numbers or path names.

A variable may be associated with more than one display node, each storing information about different views of the variable. In such a situation, the variable name addresses all registered nodes that display the node. Thus if an object executes the two messages to create two display nodes for 'refList':

Edit [refList]
Edit [refList]

and uses the name '@refList' in a message to the dialogue manager, then the variable name refers to both the nodes if they are both registered. Thus an object can *broadcast* changes to all nodes of a variable.

It is possible that a display node may not be associated with a variable in the object. This situation occurs when the object does not update its internal data structures on each change caused by a user. For instance, in the 'Bibliography' example, an object submits a 'NIL' value of 'refList' to the dialogue manager. The user may create a new sequence-containing record, and then add elements to the sequence. The object does not receive information about these incremental changes to the variable 'refList', and receives the new value only when the user selects the new list and executes the 'accept' command. At this point the display nodes added by the user are associated with storage in the object.

A display node not associated with storage in the object cannot be named by a variable name. In such a situation, an object has to use either a path name or a node number to refer to it.

Node Number

A node number is the index of a display node in the data structures of a dialogue manager. It is stored in *node variables* of the object, and may be used to name the node. A node variable 'n' that stores a node number of a display node of type 'T' is declared as follows:

n: **NODE OF** T

An object can receive the node numbers of all display nodes created by a dialogue manager. It receives node numbers of roots of display trees in in return values of 'Edit' messages. It can receive node numbers of other nodes in parameters of update methods, which are discussed in §4.7. The object may choose to ignore a node number, or store it in a node variable for later use.

Path Name

A node name (node number or variable name) may be qualified by a *tag list* and a *qualifier* to form a path name of a descendant of the node. The following is the syntax of a path name:

<path name> := [<tag list>] <variable name>$<qualifier>
<path name> := [<tag list>] <node number> <qualifier>

A qualifier indicates either a field of a record node, an element of an array or sequence node, or a referent of a pointer node. Its syntax is identical to the syntax of the suffix attached to a Mesa variable to indicate a field, element, or referent. Thus the qualifier '.author' attached to a node name of a display node of type 'Reference' accesses the display node for the 'author' field. Similarly, the qualifier '[0]' attached to a node name of variable of type 'ReferenceList' accesses the display node for the first sequence element.

(The separator '$' is used to disambiguate variable names from path names. For instance, consider a record variable 'r' that has a field 'f1'. Then the name 'r.f1' is a variable name while the name 'r$f1' is a path name. These two names may refer to *different* display nodes, as illustrated below. Assume that the object sends the following messages to a dialogue manager:

Edit[r];

Edit[r.f1];

The two messages display variable 'r.f1' twice: once as part of the display tree created in response to the first message, and once as the root of the display tree created in response to the second message. Assume that both nodes are registered. Then the path name 'r$f1' refers to the first node, while the variable name 'r.f1' refers to both nodes.)

A tag list is used to access display nodes of fields of variant records. It is required in Dost because fields in different variants of a Mesa record are not required to be distinct. This list indicates the values of the tags of the variant records, and is used to access the correct variants. The leftmost tag value specifies the innermost variant record, and the last specifies the outermost variant. Thus the tag list 'Journal' and the qualifier '.journal' may be attached to a node name for a variable of type 'Reference' to access the display node of the 'journal' field of 'Journal' variant of a display node of type 'Reference'.

Figure 6 shows how the display nodes in a display tree for the variable 'refList' may be named. We assume in this figure that the value of 'refList' is not 'NIL', the sequence component has exactly one element, and the root node and the display node of the author field of the sequence element are registered. The registered nodes in the figure are shaded. We also assume that 'n' is a node variable that points to the display node for the singleton element of the sequence.

Note the difference between the two names for the display node for the 'author' field. The first, '@refList$[0].author', is a path name derived from the variable name of the root node, while the second, 'refList[0].author' is the variable name of the node. The second name is applicable only if the node is registered.

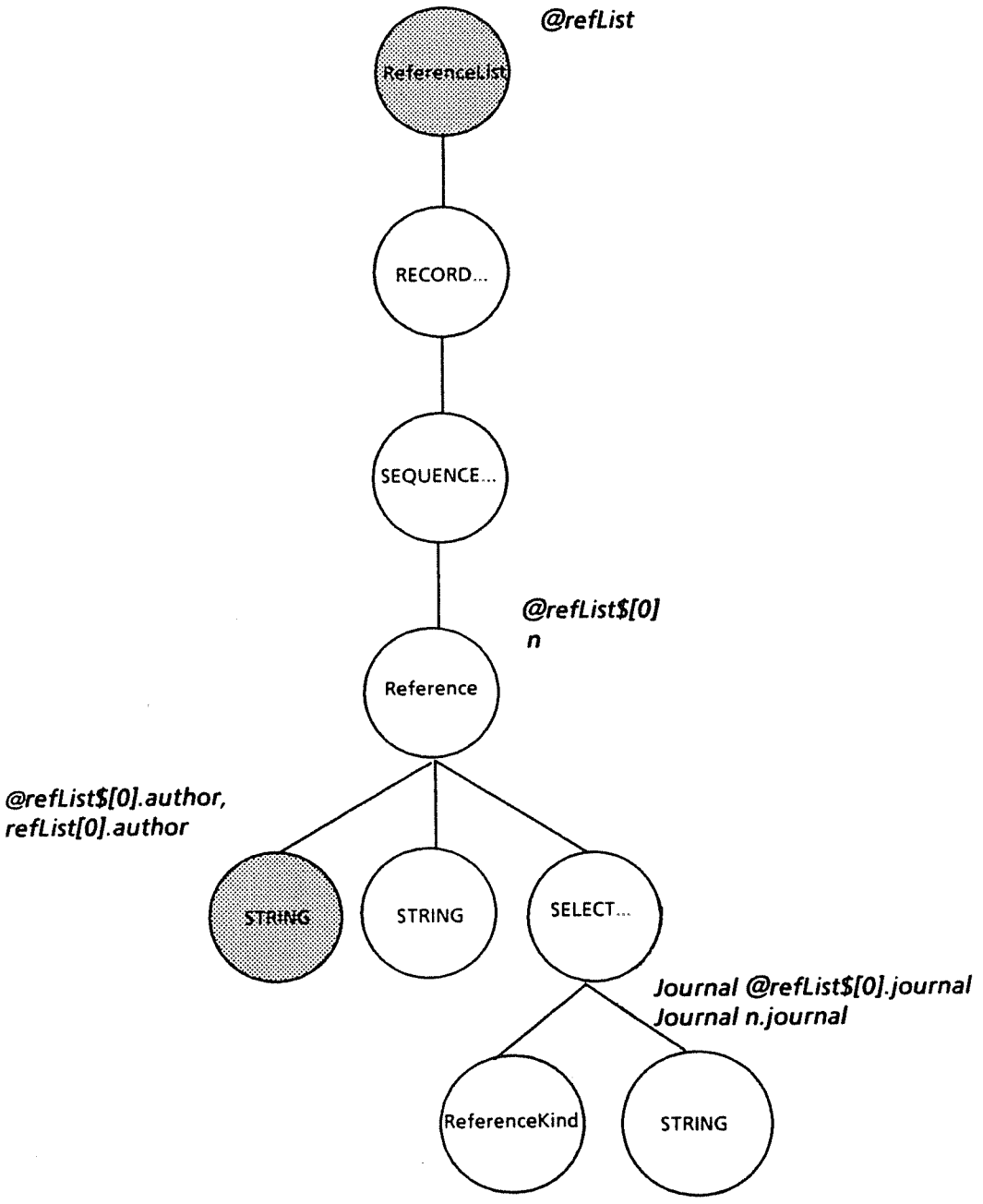


Figure 6: Naming Display Nodes in a Display Tree for 'refList'

Choosing a Node Name

Variable names, node numbers, and path names are three different ways to name a node. Each has its advantages, and is appropriate under different conditions, as discussed below. We first compare node numbers with variable names, and then path names with both node numbers and variable names.

The advantage of variable names over node numbers is that they allow an object to think of the nodes in terms of the variables they display. Thus, if an object wants to update the value of the display nodes of a variable with the current value of the variable, it does not have to compute the node numbers of the display nodes. It can simply name the display nodes with the address of the variable. Similarly, when a dialogue manager uses the variable name to inform the object about a change to a value of a display node, the object can simply dereference the variable name to update the variable. It does not have to translate from a node number to a variable address, which may require the object to search for the variable address. Finally, an object does not have to allocate storage for variable names, while it has to do so for node numbers.

The advantage of node numbers over variable names is that they directly name the node. As a result they uniquely identify a node whenever a variable name can name several nodes, as we saw. Moreover, their use by an object does not require a search by the dialogue manager. Finally, they can be used to name nodes that do not have storage allocated for them.

Node numbers are particularly appropriate when the object does not have to translate a node number to a variable name and vice versa. This situation often arises when it receives from the dialogue manager the node number and value of a node that has been changed by the user. The object can check the value for semantic consistency

and use the number to change those attributes of the node that announce and correct the semantic errors of the node. It should not use the variable name to do so, since it *knows* the node number, and thus, can save the dialogue manager from having to search for it.

Path names have certain advantages over both node names and node numbers. First, they limit the number of registered nodes, since descendants of such nodes can be named by qualifying the variable names of the registered nodes. Second, they limit the number of node numbers an object needs to store, since descendants of the nodes for which the object has allocated node variables can be addressed by qualifying these variables. Finally, they are useful if an object knows a node name for a parent, wants to address one of its descendants, and must *search* for a node name of the descendant. In this situation, the object may qualify the node name of the parent to form the appropriate path name for the descendant.

There are two disadvantages of path names. First, the dialogue manager has to follow the path from the ancestor node to the addressed node at execution time. Second, an object often knows the node name of a node, but has no information about its ancestors or their node names. In this situation, a path name cannot be used.

4.6. Presentations of Display Nodes

An object specifies how display nodes are to be presented by giving the dialogue manager information about *general* characteristics about the presentation. These characteristics are specified either as values of *formatting attributes* of display nodes, or as *display names* of enumeration literals. A dialogue manager uses these characteristics and its knowledge about the *types* of the nodes to construct appropriate presentations of the node.

Formatting Attributes

In this section we present examples of formatting attributes that determine

- the *alignment* of a presentation,
- whether a presentation *prompts* the user for an input value,
- whether the presentation of a display node of a simple type shows a *placeholder* or the *value* of the display node, and
- whether the presentation of a display node of a structure type *hides* or *shows* the values of its components.

As examples of display nodes, we consider those that can be defined by the types declared in figure 4.

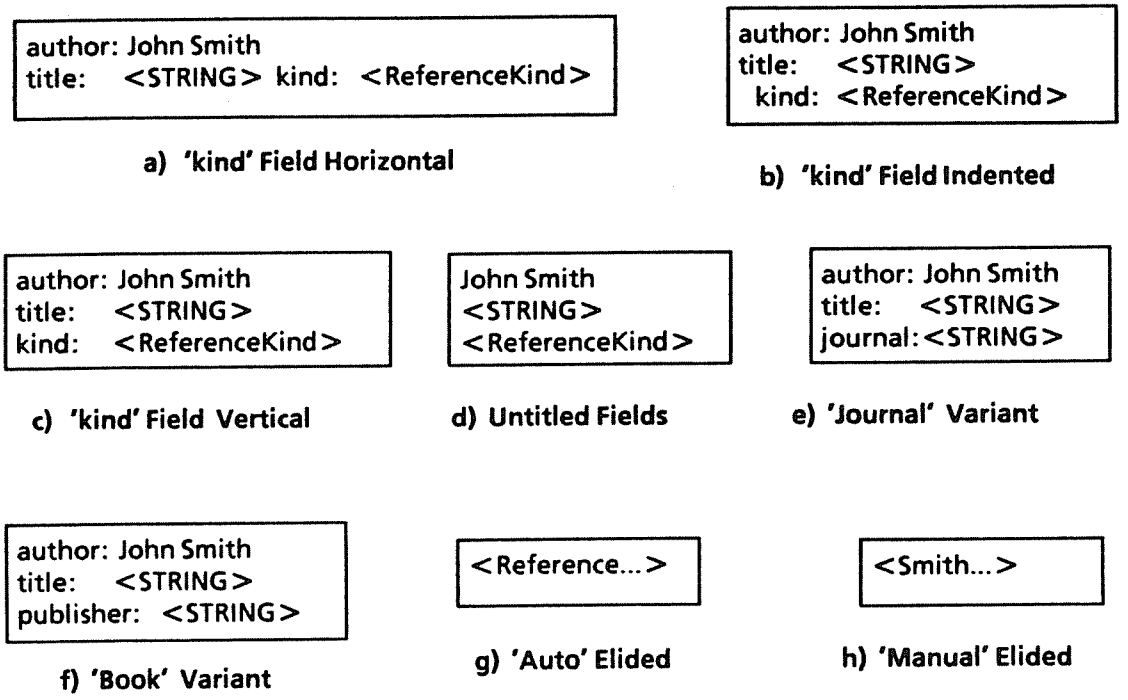
Figure 7 shows two presentations of a display node of type 'ReferenceKind'. The first is an 'initialized' presentation, and displays the 'value' attribute of the node. The second is an 'uninitialized' presentation, and shows a *placeholder* for a value of type 'ReferenceKind'. Such a presentation is appropriate for display nodes of variables that need to be initialized by the user. It serves two purposes: first, it tells the programmer that an input value is expected for the variable, second, it describes the set of legal input values. The 'initialized' attribute of a node determines if an 'initialized' or 'uninitialized' presentation is displayed for it.

Figure 8 shows several presentations of a display node of type 'Reference'. The presentations in figures 8 (a-f) are composed of the presentations of the fields of the display node, while the presentations in figure 8 (g-h) are 'elided', and hide the details of the node. A Boolean attribute *elided* determines if the presentation of a node is 'elided' or 'expanded'.

a) Initialized

b) Uninitialized

Figure 7: Presentations of a Display Node of Type 'ReferenceKind'



a) 'kind' Field Horizontal

b) 'kind' Field Indented

c) 'kind' Field Vertical

d) Untitled Fields

e) 'Journal' Variant

f) 'Book' Variant

g) 'Auto' Elided

h) 'Manual' Elided

Figure 8: Presentations of a Display Node of Type 'Reference'

a) NIL Pointer Value

b) Non-NIL Pointer Value

Figure 9: Presentations of a Display Node of Pointer Type 'ReferenceList'

In figure 8(a) the presentation of the 'kind' field is placed horizontally with respect to the presentation of the 'title' field, in figure 8(b) it is indented, and in figure 8(c) it is placed 'vertically'. These variations are caused by different values of the *alignment* attribute of the node.

Figures 8(c) and (d) show the difference between 'titled' and 'untitled' presentations of the fields of the record. The *titled* attribute determines which presentation is chosen. Figures 8(d), 8(e) and 8(f) show the difference between initialized and uninitialized presentations of the variant part of the record. In figure 8(d) the presentation of the variant field is the uninitialized presentation of the tag field, and in figures 8(e-f) it is the presentation of the fields of the variant selected by the initialized tag value.

Figures 8(g) and (h) show two elided presentations of a 'Reference' node. The first is the default presentation, while the second is a node-specific presentation determined by the object or the user. The *elideString* attribute determines the presentation displayed when the 'elided' attribute of a node is 'TRUE'.

Finally, consider a display node of the pointer type 'ReferenceList'. The presentation of the node is an uninitialized presentation if it contains the 'NIL' value, otherwise it is a presentation of the display node to which it points. Figure 9 shows presentations in the two cases— 9(a) shows a presentation corresponding to the 'NIL' value, and 9(b) shows a presentation corresponding to two elements in the sequence.

Display Names

Dost associates each enumeration literal in a class with a *display name*. A display name is an arbitrary string that displays the literal to the user. By default, it is the identifier that names the literal. Thus the display name of the enumeration literal 'ReferenceKind[Journal]' is the string 'Journal', as illustrated by figure 7(a).

An object may override the display name of an enumeration literal by sending a 'DisplayName' message. The following is an example of such a message:

DisplayName [enum: ReferenceKind[TechRept], val: "Technical Report"];

Display names allow the display of non-alphanumeric strings for the values of enumeration variables. Chapter 3 illustrates their use.

4.7. Optional Presentations

Several applications require that some parts of their presentations be *optional.* For instance, an editor of a Pascal program requires that the **else** part of an **if** statement be optional. Similarly, we can consider an extension of the class 'Bibliography', which allows the users to fill an optional 'otherAuthors' field in an entry.

Dost provides support for these applications by dividing the display nodes associated with the presentation of an object into *optional* and *required* nodes. If a node is required, then an initialized value of it is required in an initialized value of its parent. If, on the other hand, it is optional, the user does not have to initialize its presentation. Moreover, its presentation may be deleted and inserted from the presentation of an object.

Ideally, an object should have the choice of making any node optional. However, the following property of optional nodes restricts these nodes to pointers and elements of sequences. A node can be optional only if, when the node has not been initialized by the user, it is possible for a dialogue manager to either not send the object the value of the node in a legal value of its parent, or send a value that can be easily distinguished by the object as uninitialized. Otherwise, an object may mistake an optional uninitialized node for an initialized node.

Since Mesa does not provide support for uninitialized variables, it is not possible for the dialogue manager to meet either of the two conditions for most types. For instance, a non-pointer element of an array cannot be optional, since the value of the element is always included in a legal value of the array, and the value of the element does not indicate to the object if the node is 'initialized' or 'uninitialized'. Similarly, a non-pointer field of a record cannot be optional.

Pointer nodes, string nodes, and elements of sequences, however, can be optional. A dialogue manager sends the 'NIL' value for an uninitialized pointer or string, and does not include uninitialized sequence elements in a sequence sent to the object.

Thus the following criterion is used to decide if a display node is optional or required. Display nodes of elements of sequence nodes are optional. Display nodes of pointer and string variables are optional or required, depending on the value of the Boolean *optional* attribute. All other nodes are always required.

The elements of sequences are always optional, since there does not seem to be use for required sequence elements. Sequences store lists in which elements can be inserted and deleted, and thus are optional. Pointers and strings on the other hand can be optional or required. Examples in chapter 3 illustrate situations in which each is appropriate.

We illustrate our approach to supporting optional presentations through an example. Consider an extension of the class 'Bibliography' that includes an additional 'otherAuthors' field in a 'Reference'. Assume that the field is optional and is declared as follows:

```
Reference: TYPE = RECORD [
    ...
    otherAuthors: POINTER TO SEQUENCE size: CARDINAL OF STRING,
    ...];
```

Thus the user has the choice of entering a valid value for the field, or leaving it uninitialized.

Now assume that the user fills a few entries for 'refList' and executes the 'accept' command. The dialogue manager responds by sending the object a sequence containing only those entries initialized by the user. The 'otherAuthors' fields of these elements either point to lists of authors input by the user, or contain the 'NIL' value, depending on whether or not the user initialized their presentations.

4.8. Communicating Updates to an Object

In Dost, an object typically interacts with a user by first displaying several variables and then waiting for updates to them. The dialogue manager allows the user to enter values for these variables in *any order* and update them *several times*. In this section, we discuss our approach to communicating these updates to an object.

4.8.1. Update Methods

Simple Case

Assume that an object displays the variable 'A' declared as:

```
AIndex: TYPE = CARDINAL [1..10];
AType: TYPE = ARRAY AIndex OF INTEGER;
A: AType;
```

The object can bind the following method to the *selfUpdate* attribute of the display node of 'A' (§4.11 discusses how values are bound to attributes)

```
AUpdated: METHOD [newVal: AType] = {
    A <- newVal};
```

When the value of the display node for 'A' is changed by the user, the dialogue manager calls this method with the new value of the node. (There are several conditions that determine when a value of a display node is sent to the object. These are discussed later. Until then, we assume that the value of a display node is sent to the object when it is changed by the user.) The object uses this value to update variable 'A'.

The attribute 'selfUpdate' is an *update attribute* and the method 'AUpdated' is an *update method.* Update methods are used by an object to receive updates to its display nodes. Update attributes either hold the 'NIL' value or are assigned update methods.

In this example, the object receives the whole array when an element is updated. In several situations it is useful to receive *incremental* changes to a structure. It may be more *efficient* to transmit just the part that changed rather than the complete value. Moreover, an object may have to do less processing to find the part that changed.

We now discuss two ways to receive incremental updates to the array 'A'. The first can be used to receive incremental updates to any structure, while the second can be used to receive incremental updates to only arrays and sequences.

Incremental Updates to any Structure

The object can bind the following update method to *selfUpdate* attributes of display nodes of the elements of the array:

```
IncrementUpdated: METHOD [newVal: INTEGER,
  varName: POINTER TO INTEGER] = {
    varName^ <- newVal };
```

Whenever a display node for an element of the array is updated, the dialogue manager invokes the method with the value and variable name of the node. The body of the method uses the variable name to update the value of the element.

The update method 'IncrementUpdate' has two parameters while 'AUpdated' has only one. In general, an update method takes as arguments (1) a description of the change to the value of the node, (2) the node number and (3) variable name of the display node, (4) the number of the subwindow in which the node is displayed, and (5) a pointer to the dialogue manager that displays the node. Here we used only (1) and (3). The node numbers, the variable names, the subwindow number, and the pointer to the dialogue manager are received in *optional* parameters, and are useful only if the same method is associated with several display nodes. The object can use the input parameters to check the change for semantic consistency, locate the variable to be updated, update the variable if no semantic errors are found, transmit the new value to other objects, or perform some other computation on it.

Incremental Updates to Arrays and Sequences

We now present a second way to receive incremental updates to the elements of the array. It allows an object to receive the *index* of the changed element of the array.

The object can bind the following method to the *elementUpdate* attribute of the array:

```
AElementUpdated: METHOD [index: AIndex, newVal: INTEGER] = {
    A [index] <- newVal};
```

When a user changes the value of an element of the array, the dialogue manager calls the method with the index and new value of the element. The body of the method updates the element of the array.

The *elementUpdate* attributes are defined only for arrays and sequence pointers. Methods assigned to them can be used by an object to receive updates to elements of arrays and sequences.

Incremental Deletes and Inserts of Sequences

A sequence pointer is also associated with the *insertUpdate* and *deleteUpdate* attributes, which may be assigned methods that allow an object to receive information about addition and deletion of elements in the sequence. A method may be assigned to an 'insertUpdate' attribute, with parameters that identify the index of the new element, and its value. Similarly, a method may be assigned to a 'deleteUpdate' attribute, with parameters that identify the index of the deleted element. The following are examples of 'insertUpdate' and 'deleteUpdate' methods:

```
RFInserted: METHOD [index: CARDINAL] = {
    InsertSlotIntoList [index]};

RFDeleted: METHOD [index: CARDINAL] = {
    DeleteFromFromList [index]};
```

The procedures 'InsertSlotInList' creates a position in the sequence to store a new element, and the procedure 'DeleteSlotFromList' deletes an element from the sequence.

4.8.2. Invocation of Update Methods

We now discuss invocation of update methods. Our discussion consists of two parts. The first part discusses the conditions that trigger an update method. The second part discusses the order in which update methods are called when more than one update method is triggered.

Triggering Conditions

Our choice of the conditions that trigger update methods was influenced by the following goals:

- A user should have the freedom of controlling when to receive feedback from the object. For instance, a user who changes an element of the array 'A' should have the option of either receiving instant feedback or receiving feedback after modifying other related elements of the array.
- An object should be able to specify the *granularity* of the change transmitted to it. For instance, it should be able to receive changes to the array in increments of elements of the array or in increments of the array.
- An object should not receive uninitialized values of required nodes. For instance, the method 'AUpdated' should be activated only if the display nodes of all elements of the array have been initialized by the user. Otherwise, an object may mistake an uninitialized value for an initialized one.

We meet the last goal by sending the value of a required node in an update method only if it is initialized. §4.8 discusses the condition for optional nodes. We meet the other goals by associating a Boolean *incFeedback* attribute with each node and specifying the following additional condition for sending the value of a node in an update method:

- the node is a leaf node, and its 'incFeedback' attribute is 'TRUE', and the user changes its value, or
- the user executes the 'accept' command on the node, or

- the node is a structure node and the user either changes the value of a descendant of the node or executes the 'accept' command on a descendant.

The user is provided with a command to change the 'incFeedback' attribute of any attribute group (§4.11).

Invocation Order

A single user command may trigger the invocation of a *sequence* of methods. We illustrate the nature of this sequence through an example.

Figure 10 shows a display tree a variable of type 'ReferenceList', and values of the update attributes of the nodes of the tree. The 'NIL' values for these attributes are not shown. Assume that all nodes of the tree are initialized, except for the 'author' field of the singleton element in the sequence (marked by an arrow). Then the following sequence of methods is called when the user initializes the 'author' field and executes the 'accept' command:

STUpdated
ReferenceUpdated
RFInserted
ElUpdated
RFUpdated

That is, the order is bottom-up, and in one node an 'insertUpdate' or 'deleteUpdate' method is called before an 'elementUpdate' method, which is called before a 'selfUpdate' method.

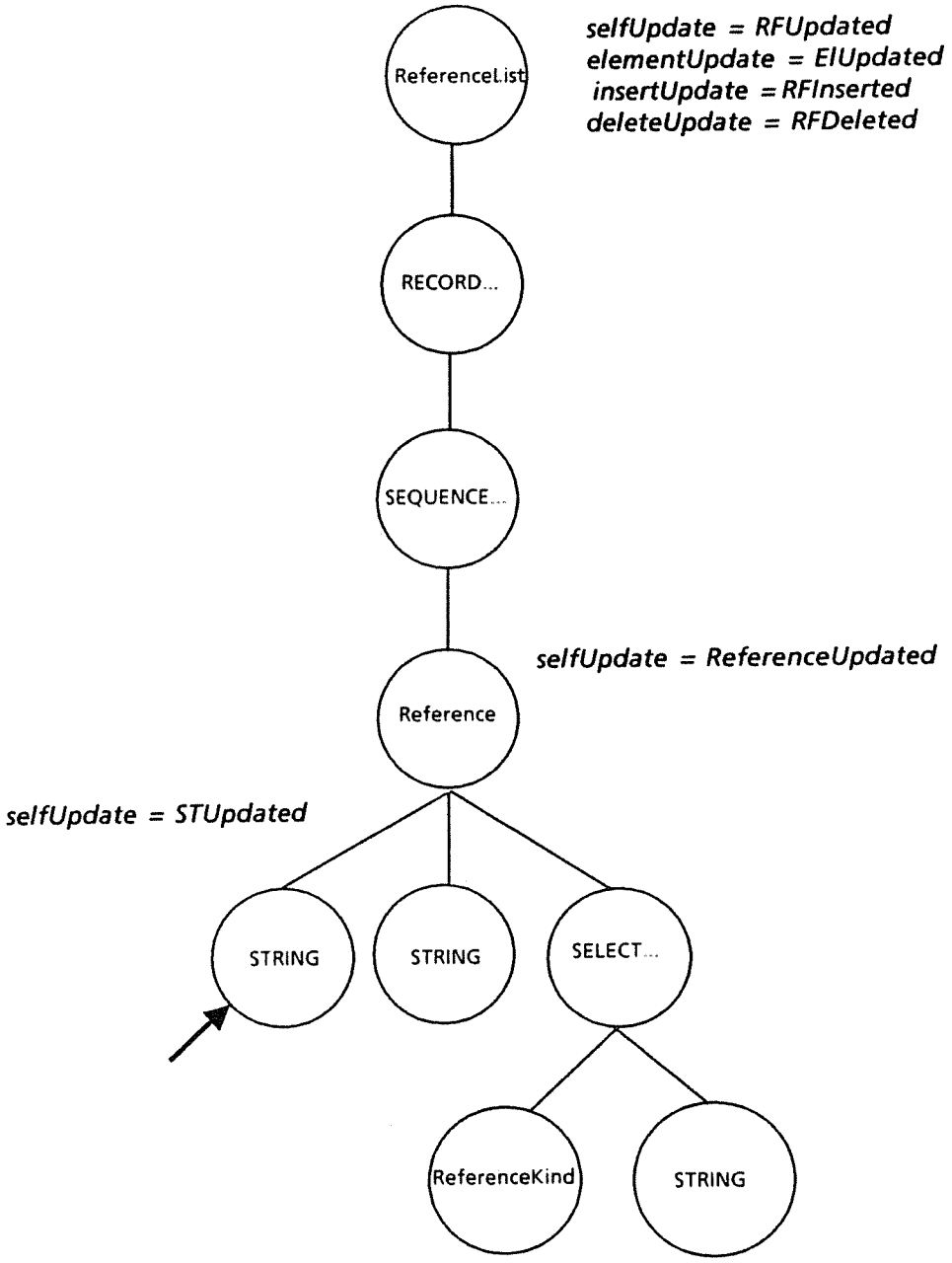


Figure 10: Update Methods of a Display Tree for a 'ReferenceList'

The method 'STUpdated' is called with the new 'STRING' value of the field. It can check the semantics of the change to the 'author' node. Then the method 'ReferenceUpdated' is called with the new value of the sequence element. It can check for any inconsistencies between the different fields of the reference element. Next the the method 'RFInserted' is called informing the object about a new initialized element in the sequence. The value of the element is then transmitted to the object as an argument of 'ElUpdated'.

The value of a node is communicated to the object only if an appropriate update method is defined. This feature allows an object to control the granularity of the change transmitted to it. Thus, if the object did not need to check the 'author' field for semantic consistency, it could set the 'selfUpdate' attribute of the node to 'NIL'.

4.9. Error Handling

A user can make an error in changing the value of a display node. The error is a *syntax error* if the value does not conform to the type of the variable. Otherwise it is a *semantic error.*

In Dost, syntax errors are detected and reported by the dialogue manager, which uses the information about the type of the display node to handle these errors. Semantic errors, on the other hand, have to be handled by the object. The object checks and reports these errors when it receives the values of display nodes in its update methods.

Semantic errors are reported in two ways. First, a message can be written in the 'message subwindow' (§5). Both the dialogue manager and the object can write into this window. Second, the Boolean *highlighted* attribute of the display node can be set to 'TRUE', which causes the presentation of the display node to be highlighted.

The following example illustrates how syntax and semantic errors are handled in Dost. Consider the following declaration of the variable 'date':

```
Day: TYPE = CARDINAL [1..31];
Month: TYPE = {January, February, ..., December};
date: RECORD [day: Day, month: Month, year: CARDINAL];
```

Now assume that the variable is submitted to a user for editing. If the user types a value of 'day' that is not in the range '1..31', or a value of month that does not indicate one of the months in the range 'January..December', or a non cardinal value for the year, then the dialogue manager immediately detects the *syntax error* and reports it to the user. If on the other hand, the user enters the date:

29 February 1986

the *semantic error* is not detected by the dialogue manager. It is the object's responsibility to report the error when it receives the new value of 'date'.

4.10. Tailoring Editor Commands

Section 1 illustrated some of the commands that can be applied to the presentation of a display node. A dialogue manager normally implements these commands, and decides which commands are available to edit a presentation of a display node. An object may, however, tailor its user interface by restricting the use of current commands, changing their implementation, or defining new commands.

For each command, each display node has a Boolean attribute that determines if the command may be applied on the presentation of the node. For instance, each node has a Boolean *expandEnable* attribute, which determines if the 'expand' command may be applied to it. An object may change the values of these attributes to restrict the set of commands that may be applied to a presentation of a node.

An object may override the implementation of a current command by providing an *override method* and assigning it to the *override attribute* of a node. When the user invokes the command, the dialogue manager checks the value of this attribute. If it is 'NIL', it uses the default implementation, otherwise it calls the override method for the command.

Similarly, an object may add a new command by providing a *command method* that implements the command. The object introduces the new command by sending a message to the dialogue manager that names the new command the group of nodes for which the command is applicable and the command method. This group is defined by an 'attribute group' (§4.11), and includes the display nodes corresponding to the leaf nodes of the inheritance tree rooted by the group.

The following is an example of a message that an instance of 'Bibliography' may send to a dialogue manager to add the *sort* command for a display node of type 'ReferenceList':

```
AddCommand [attrGrp: ReferenceList, name: "sort", cmdMethod: sort];
```

An object-defined implementation of an editor command is often most conveniently implemented using the default or current implementation of another command. Therefore Dost allows an object to send messages to the dialogue manager that ask for editor commands to be invoked. Each such message indicates the command to be invoked, the display node that forms the operand, and whether the default or current implementation is to be used. The following two examples illustrate the use of these messages.

Consider first the 'next placeholder' command when applied to an element of a sequence. The default implementation *inserts* an new element after the current node. Assume that an object wishes to change the implementation such that a new element is always *appended* to the list. The object can override the *default implementation* by supplying a command method that always invokes the default implementation on the last node of the sequence.

Now assume that an object wishes to add a new command called *append*, which, when applied to an element of a sequence, appends a new element to the list. The command method supplied for the new command can invoke the *current implementation* of 'next placeholder' command on the last node of the sequence.

4.11. Attribute Inheritance

Dost provides a mechanism for *inheriting* attributes, which relieves a programmer of the task of specifying all the attributes of all the display nodes in the presentation of an object. A presentation of an object is associated with a *tree* of **attribute groups**. The tree consists of four levels, and defines an inheritance relation among its nodes.

The nature of the attribute groups and the structure of the inheritance tree is illustrated by an example. Consider a presentation of an object that displays the variable 'refVar' described by the following declarations:

```
SimpRef: TYPE = RECORD [
    author, title: STRING];
refVar: SimpRef;
```

Figure 11 shows the inheritance tree associated with the presentation. The attribute groups of the tree can be divided into the *default* group, the *type groups*, the *field*

groups and the *display nodes*. The default group roots the tree and defines attributes common to all display nodes. A type group defines attributes common to all display nodes of a type. A field group is associated with a field of a record type, and defines attribute values common to display nodes corresponding to the field. A display node defines attributes special to the node.

In figure 11, the attribute groups 'STRING' and 'SimpRef' are the type groups, which define attribute values common to display nodes of types 'STRING' and 'SimpRef' respectively, The groups 'SimpRef.author' and 'SimpRef.title' are the field groups. These define attribute values common to display nodes corresponding to the 'author' and 'title' fields of any display node of type 'SimpRef'. Finally, the leaf nodes of the tree are the display nodes. which describe attribute values specific to the display nodes corresponding to 'refVar' and its two fields.

Each attribute group begins with an *initial* value of each attribute, which may be *explicitly* changed. The new value of is *propagated* to all children in which the value has not been *redefined.*

An attribute may be explicitly changed either by the object sending the dialogue manager a message (§4.11.1), or by a user executing an editor command (§5).

An attribute 'a' is considered redefined in an attribute group if either the initial value of 'a' in the attribute group is not the same as the initial value of 'a' in the parent of the attribute group, or it has been explicitly changed by the object or the user.

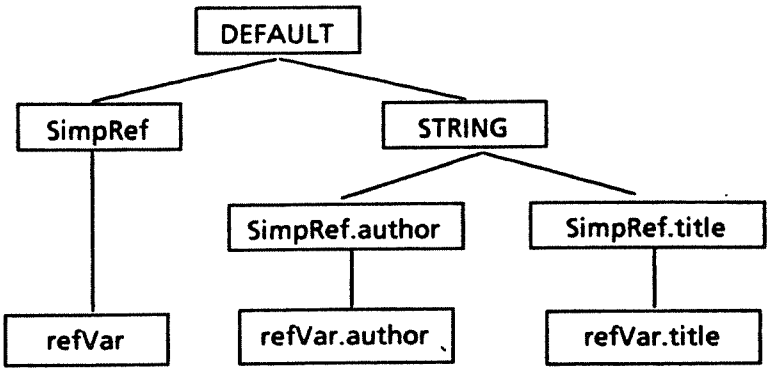


Figure 11: An Inheritance Tree

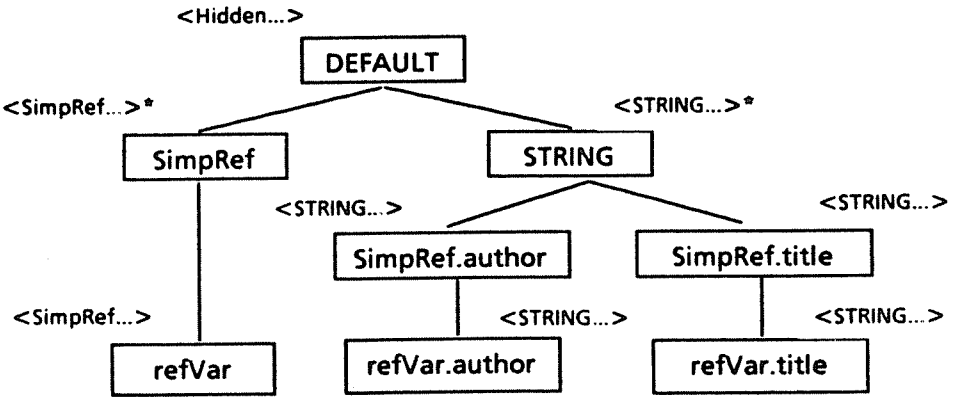


Figure 12: Initial Values of the 'elideString' Attribute

Figures 12-16 illustrates how explicit changes to the 'elideString' attribute are propagated in the inheritance tree of figure 11. The 'starred' values indicate redefined attributes.

Figure 12 shows initial values of the attributes. The attribute is redefined in 'SimpRef' and 'STRING', since the values of the attributes in the two nodes are not the same as the value of the attribute in the default group. Therefore, when the value of the attribute is changed in the default group, as shown in figure 13, the new value is not propagated to its children.

On the other hand, when the value is changed in 'SimpRef', as shown in figure 14, the new value is propagated to 'refVar'. Figure 15 shows redefinition of the attribute in 'refVar'. Now if the attribute is changed in 'SimpRef', as shown in figure 16, the value is not propagated to 'refVar'.

Attributes and attribute inheritance, together, provide a balance between *flexibility*, which allows the interface of an object to be tailored according to the specific needs of the object, and *automation*, which frees the programmer or the user from the task of implementing the interface of an object. Attributes provide the mechanism for flexibility, while attribute inheritance provide the mechanism for automation.

4.11.1. Setting Attributes from the Object

An object may send a dialogue manager messages to change the attributes of an attribute group. The message name indicates the attribute, and its parameters name the attribute group and the new value for the attribute.

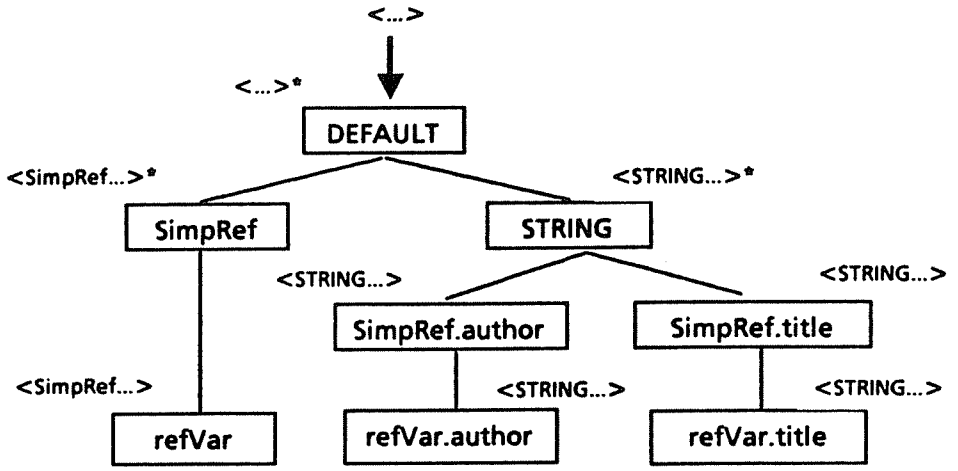


Figure 13: Explicit Change in 'DEFAULT'

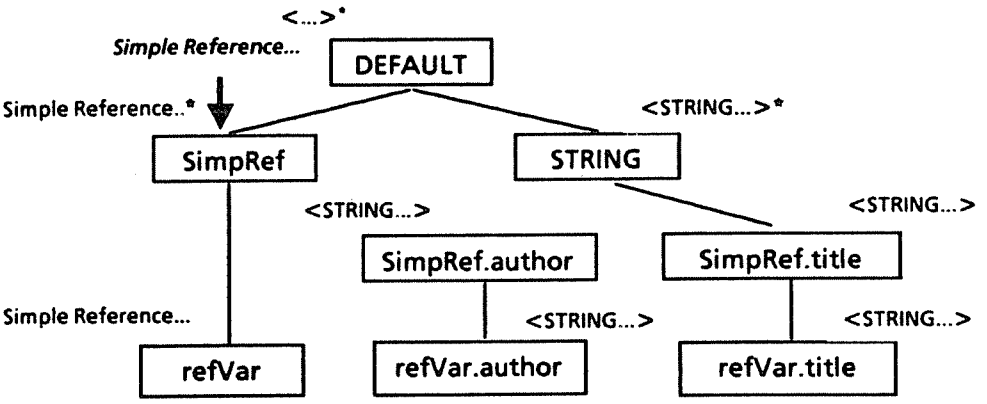


Figure 14: Explicit Change in 'SimpRef'

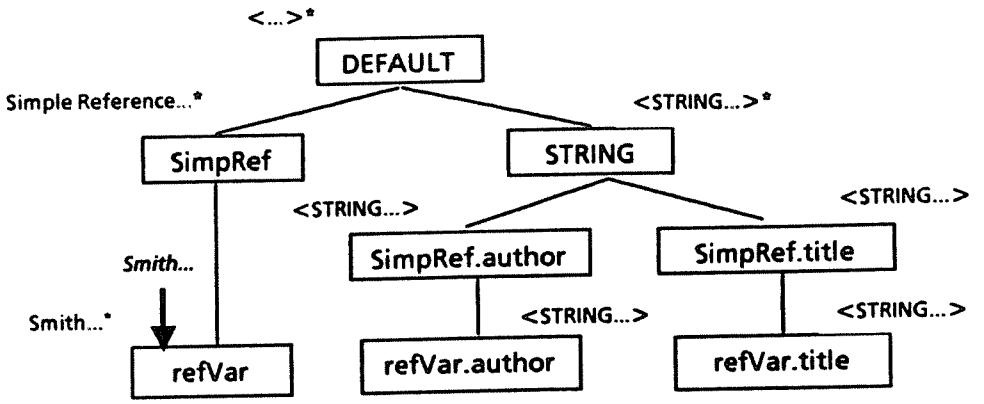


Figure 15: Explicit Change in 'refVar'

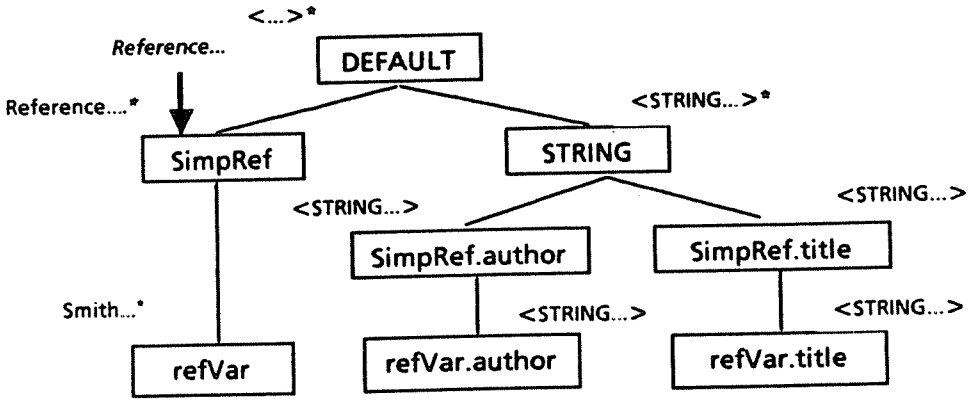


Figure 16: Explicit Change in 'SimpRef'

An attribute group may be the default group, a type group, a field group, or a display node. The default group is named by the symbol '*'. A type group, which is an attribute group that defines attribute values common to a type, is specified by naming the corresponding type.

A component group, which is an attribute group that defines attribute values common to fields of a record type, or elements of an array type, or elements of a sequence, is specified by a name that has the following syntax:

<component group> := <Tag List> <Record Type>.<Field>
<component group> := <Array Type> []
<component group> := <Tag List> <Sequence Pointer Type> []

The first alternative names component groups associated with fields of record types, the second names component groups associated with elements of an array type, while the third names component groups associated with elements of the sequence field of the referent of a sequence pointer type. The <Tag List> is similar to the tag list used to specify the path name of a display node, and indicates the tag values of the variants in which the field is defined.

A display node is specified by its node or path name.

The following are examples of messages that may be sent to change attributes of attribute groups associated with the 'Bibliography' example:

1: Value [attrGrp: refList[0].author, val: "John Smith"];
2: Titled [attrGrp: STRING, val: TRUE];
3: Alignment [attrGrp: *, val: vertical];
4: Registered [attrGrp: ReferenceList[], val: TRUE];
5: SelfUpdate [attrGrp: Journal Reference.journal, val: JournalUpdated];
6: Titled [attrGrp: refList[0], val: FALSE, range: children];
7: Titled [attrGrp: refList[0], val: FALSE, range: deep];

The first message sets the 'value' attribute of the display node for the 'author' field of the first reference element of 'refList' to "John Smith", the second sets the 'titled' attribute of the type group 'STRING' to 'TRUE', the third sets the 'alignment' attribute of the default group to 'vertical', the fourth sets the 'registered' attribute of the component group 'ReferenceList[]' to 'TRUE', and the fifth sets the 'selfUpdate' attribute of the component group corresponding to the 'journal' field of the 'Journal' variant of a 'Reference' to the method 'JournalUpdated'.

The last two messages show the function of the optional 'range' parameter of some attribute-setting messages. In the fist of these messages, the 'titled' attributes of the fields of the display node are set to 'FALSE', and in the second message, the 'titled' attributes of the display nodes and its fields are set to 'FALSE'.

4.12. Miscellaneous Primitives

Saving and Resetting an Object

Most conventional text editors provide commands to save and reset user changes to the file being edited. The changes are saved by writing the editor buffer into stable storage, and reset by loading the buffer with the saved version.

An object in Dost can provide a similar interface to save and reset changes to its presentations. It can send messages to a dialogue manager indicating its *save method* and *reset method.* The save method is expected to save appropriate data of the object in its data file. The reset method is expected to read the saved data, and reset the values of the display nodes with the saved ones. A user can invoke these methods by executing the 'save' and 'reset' commands respectively.

Insertion and Deletion of Sequence Elements

Dost allows an object to send messages to a dialogue manager asking it to insert or delete an element in the display tree for a sequence. For example, the message:

```
InsertAfter [seq: @refList, index: 4];
```

asks the dialogue manager to insert a new element after the fourth element of the sequence node in the display tree for the node '@refList'.

Readonly Presentations

Several applications require that some parts of their presentations be *readonly*. For instance a directory manager requires that the 'size' field of an entry be readonly. Dost provides support for the above applications by dividing the display nodes into *editable* and *readonly* nodes. The value of a *readonly* node cannot be changed by the user. A Boolean attribute determines if a node is required or optional.

Prompts and Suffixes

Dost allows the value of a display node to be preceded by a 'prompt string' and succeeded by a 'suffix string'. These strings are stored in the *prompt* and *suffix* attributes respectively of a display node. The 'prompt string' of a node is displayed only if its 'titled' attribute is 'FALSE'. Examples in chapter 3 illustrate the use of these attributes.

Hidden Presentations

A display node is associated with a Boolean attribute ***hidden***, which determines if the presentation of the node is displayed to the user or is hidden from him. An object can use this attribute to show different views of a structure variable by hiding or displaying the components of the structure. For instance, a directory manager can

show a 'long' listing of a directory by displaying all fields of a directory entry, and a 'short' listing by hiding some of these fields (Chapter 3, §6).

4.13. Summary of Attributes

The previous sections described the attributes defined in Dost. The following table gives a summary of these attributes together with the sections in which they were introduced:

value	§4.4
initialized	§4.4
registered	§4.5
elided	§4.6
alignment	§4.6
titled	§4.6
elideString	§4.6
optional	§4.7
selfUpdate	§4.8.1
elementUpdate	§4.8.1
insertUpdate	§4.8.1
deleteUpdate	§4.8.1
incFeedback	§4.8.2
highlighted	§4.9
command enable (per command)	§4.10
command override (per command)	§4.10
readonly	§4.12
prompt	§4.12
suffix	§4.12
hidden	§4.12

5. Model of Interaction

At any instant, the screen is composed of one or more Dost windows, as shown in figure 17. A Dost window is like an XDE text window except that it is managed by a dialogue manager instead of a text editor. As a result an object of an arbitrary class can be edited in such a window. A window is composed of the following subwindows: the *message subwindow*, the *command subwindow*, and one or more *presentation subwindows*.

The message subwindow displays error messages. The command subwindow is used to select commands and enter the name and class of the object to be edited. The presentation subwindows display presentations of the object.

The user interacts with an object by loading its presentations in the window, and editing them. The interface presented to edit presentations is an extension of the XDE text editor interface.

The editor commands available to the user are divided into the following categories:

(1) *Window editing* commands.

(2) *Object editing* commands.

(3) *Text editing* commands.

(4) *Structure editing* commands.

(5) *Attribute editing* commands.

(6) The *accept* command.

Figure 17: Sample Presentations of Instances of 'References' and 'ASPLE'

The window editing commands include commands to manipulate windows. The XDE window manager provides most of these. Dost provides the commands to create and delete windows, since they result in activation and deactivation of dialogue managers.

The object editing commands are used to load and empty the presentations of an object from the window, and to save and reset user changes to the presentations of the object. They replace similar commands provided by the XDE text editor to load and empty the contents of a text file into a window, and to save and reset user changes to the file.

The text editing commands are supported by the XDE text editor, and allow a user to manipulate a presentation of an object as text. They include commands for selecting text, modifying the presentations of leaf nodes, searching for patterns, scrolling, and so on.

The structure editing commands allow a user to view a presentation as structured text. They include commands for selecting nodes, 'dereferencing' pointer nodes, and inserting or deleting templates of optional nodes.

The attribute editing commands are used to change attributes of attribute groups. A user changes an attribute of a display node by selecting the corresponding display node and executing an appropriate command. The attributes of component groups, type groups, and the default group are set by selecting a display node that has the appropriate attributes, and executing the *component*, *type*, or *default* command. The first command checks if the display node is a field of a record or an element of an array or sequence, and sets the attributes of the corresponding component group. The second command sets the attributes of the group associated with the type of the display

node. The last command sets the attributes of the default group.

Figures 18 and 19 illustrate the use of the 'component' and 'type' commands. In figure 18 the user sets the alignment attribute of a 'kind' field of a 'Reference' record to 'indented', and the titled attribute to 'FALSE', and executes the component command. The appropriate values are propagated to all 'kind' fields of nodes of all 'Reference' records. Similarly, in figure 19 the user sets the 'titled' attribute of a display node of type 'STRING' to 'TRUE'. The value of the attribute is changed in all 'STRING' nodes.

The 'component', 'type', and 'default' commands are used, typically, to interactively define the format of display nodes. A user can define the format for one display node, and propagate the change to either all nodes, or all nodes that share its type or component group.

The 'accept' command is used to send the new value of a display node to an object. The dialogue manager reacts to the command by invoking appropriate update methods in the object, as discussed in §4.8.

In the design of the user interface of Dost, we have made a distinction between the *abstract command* and the *mechanism* provided to invoke it. For instance the abstract command 'replace', which changes the value of a leaf node, is invoked by selecting appropriate characters in the presentation of the node with the aid of the mouse, deleting them using the *delete* key, and inserting the new characters. The 'next placeholder' command is invoked by selecting a display node with the aid of the mouse, and then clicking the mouse at the *command item* 'NextPH' in the command window.

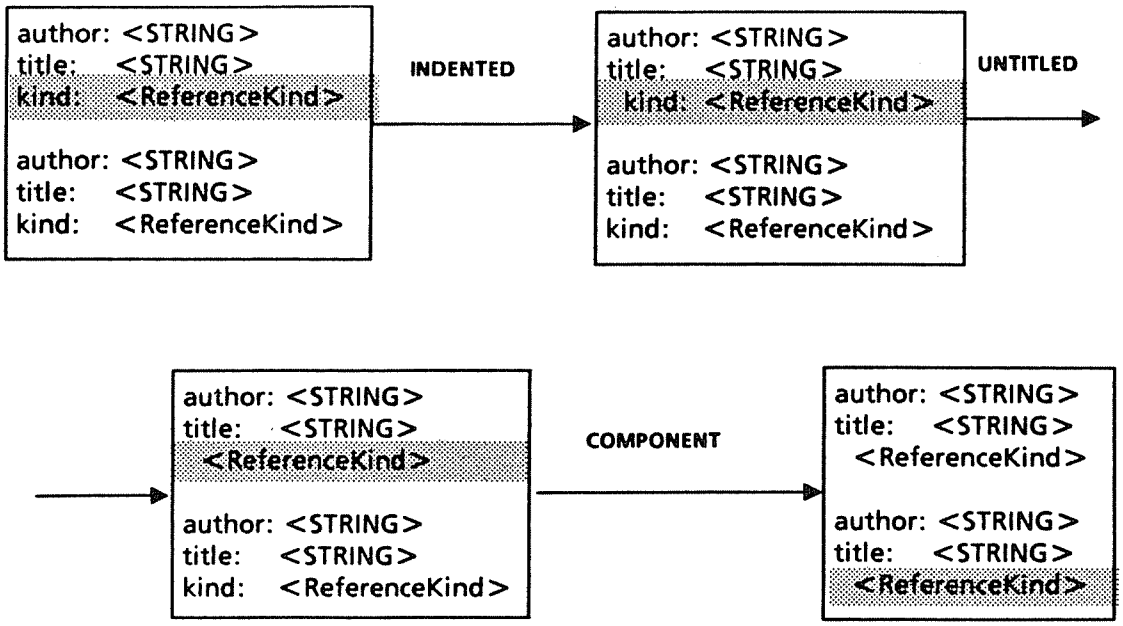


Figure 18: Changing Attributes of a Component Group

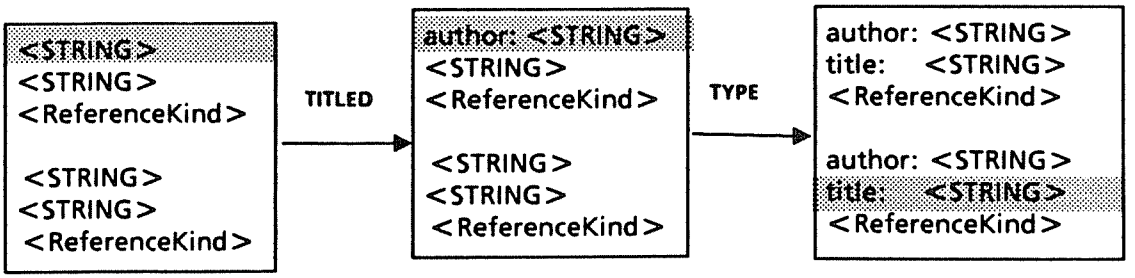


Figure 19: Changing Attributes of a Type Group

In general a user invokes a command with the aid of the mouse, keyboard, and menus.

6. Implementation

We have implemented and tested the major components of Dost. We have built a *dialogue manager program*, which can be instantiated to produce dialogue managers that support most of the input/output primitives discussed in section 4. Currently, it supports all the features described in this chapter except:

(1) 'elementUpdate', 'insertUpdate', 'deleteUpdate', 'highlighted', 'registered', and 'elideString' attributes (§4.13),

(2) 'insertAfter' message (§4.12),

(3) overriding of display names (§4.6), and

(4) tailoring of editor commands (§4.10).

We have also built an *object manager*, which supports objects. Finally, we have designed a *precompiler* which translates Dost classes into Mesa programs. It performs the translation by inserting procedures that allow an object to communicate with the dialogue and object managers. Since the precompiler has not been implemented, we have tested the system by hand-translating Dost classes into the equivalent Mesa programs.

The following subsections discuss the salient parts of the three components.

6.1. Precompiler

The precompiler takes a Dost class and translates it into a Mesa Program, which is called its *class program.* The class program contains code that handles the con-

structs in the class not available in Mesa programs.

The methods in the class are translated into procedures. The class is associated with a *method record type*, which defines fields that can point at these procedures. An instance of the class is associated with a variable of the type, called its *method record.* A pointer to the method record of an object is stored by the object manager. Other objects can query the object manager for this pointer by making the 'OpenObject' call described in section 2.2.

The declaration of an object pointer in a class is translated into the declaration of a pointer to the appropriate method record type. It can then be used to refer a method record of an object. A message that names the pointer is converted into a call to the appropriate field of the method record.

A message to a dialogue manager is handled differently. Typically, it requires special processing of its parameters, as illustrated by the following example. Consider the message:

SelfUpdate [attrGrp: ReferenceList, value: RefListUpdated]

sent by an instance of 'Bibliography' of figure 4. It asks the dialogue manager to call the procedure 'RefListUpdated' when a display node of type 'ReferenceList' is updated. This message cannot be simply converted to a Mesa procedure call in the dialogue manager, for two reasons. First, the type of the parameter 'attrGrp' cannot be described by a Mesa type. The precompiler needs to process the parameter to determine if it should send a variable address, a node number, or a string indicating a type or component group. Second, the dialogue manager calls the method 'RefListUpdated' with the value of a display node of type 'ReferenceList'. Therefore, the precompiler needs to check that the parameter 'newVal' of the method

'RefListUpdated' is of type 'ReferenceList',

6.2. Object Manager

The object manager performs several functions. It load and unloads objects and maintains the list of all the objects that are created. For each object, it maintains the class name, the data file, the name of the class program, a pointer to the method record, a pointer to the load method, and other information about the object. The information is used by it to define procedures that a precomplier inserts into a class to process the 'OpenObject', 'Passivate', and 'MakeEditable' calls.

The object manager is about 200 lines of Mesa code.

6.3. Dialogue Manager Program

The dialogue manager program is a Mesa program that can be executed to create a dialogue manager. It maintains two main data structures to allow editing of an object. The first is a table of the display nodes and other attribute groups associated with the presentation of an object. The second is the symbol table produced by the Mesa compiler after compiling the class program of the object. These two data tables are used by the dialogue manager to process editor commands executed by the user and messages sent by the object.

The dialogue manager uses the text editing and window management modules provided by XDE, and is about 7000 lines of Mesa code.

Chapter 3

Examples

In this chapter, we describe how Dost may be used, both to create classes, and to interact with their instances. We describe in detail six classes representing a diverse range of applications. For each class, we describe the user interface of instances of the class, and the code required to create the class.

We first describe two classes, 'Form' and 'ExtendedForm', whose instances present forms for the user to fill. These two classes are trivial but important applications of Dost. The class 'Form' includes the basic code needed to display a form on the screen and receive user input. The class 'ExtendedForm', includes, in addition, code required to save a form in stable storage and reset the screen with the saved version of the form.

Next, we consider a class called 'RoommateExpenses'. An instance of the class stores the common expenses of three roommates, and allows a user to insert, delete, or modify an expense entry. It displays each roommate's credit after each *incremental* change to an entry.

We then consider a more general application of Dost— a class whose instances store simple spreadsheets. A spreadsheet is displayed in two presentation subwindows: One shows the values of the spreadsheet, and the other shows the relationships among these values.

Next, we describe a class that defines lists of statements in a toy programming language. It illustrates how programming language constructs are defined and edited

in Dost.

Finally, we present a class that defines editable file directories. It illustrates how an object can show different views of a data structure.

The six examples are 'graded', each shows the use of new features of Dost. We discuss only those aspects an example that distinguish it from other examples.

1. Form

The class 'Form' defines a simple form that allows a user to enter the age, status (married, single, widowed, or divorced), permanent address, and mailing address of a person.

Figure 1 shows how a user may edit the presentation of an instance of the class. The topmost box displays the initial presentation, which names the fields in the form, and displays placeholders for them. The user may use the editor commands described in chapter 2 to replace the placeholders with appropriate values.

The last field of the form is optional, and needs to be filled only if the user's mailing address is not the same as the permanent address. Therefore it may be deleted from the form and reinserted, as shown in figure 1.

Once the user has filled all the fields, he can select the whole form by executing the 'up' command, and then hide its details by executing the 'elide' command. The effect of these two commands may be reversed by executing the 'expand' and 'down' commands in succession.

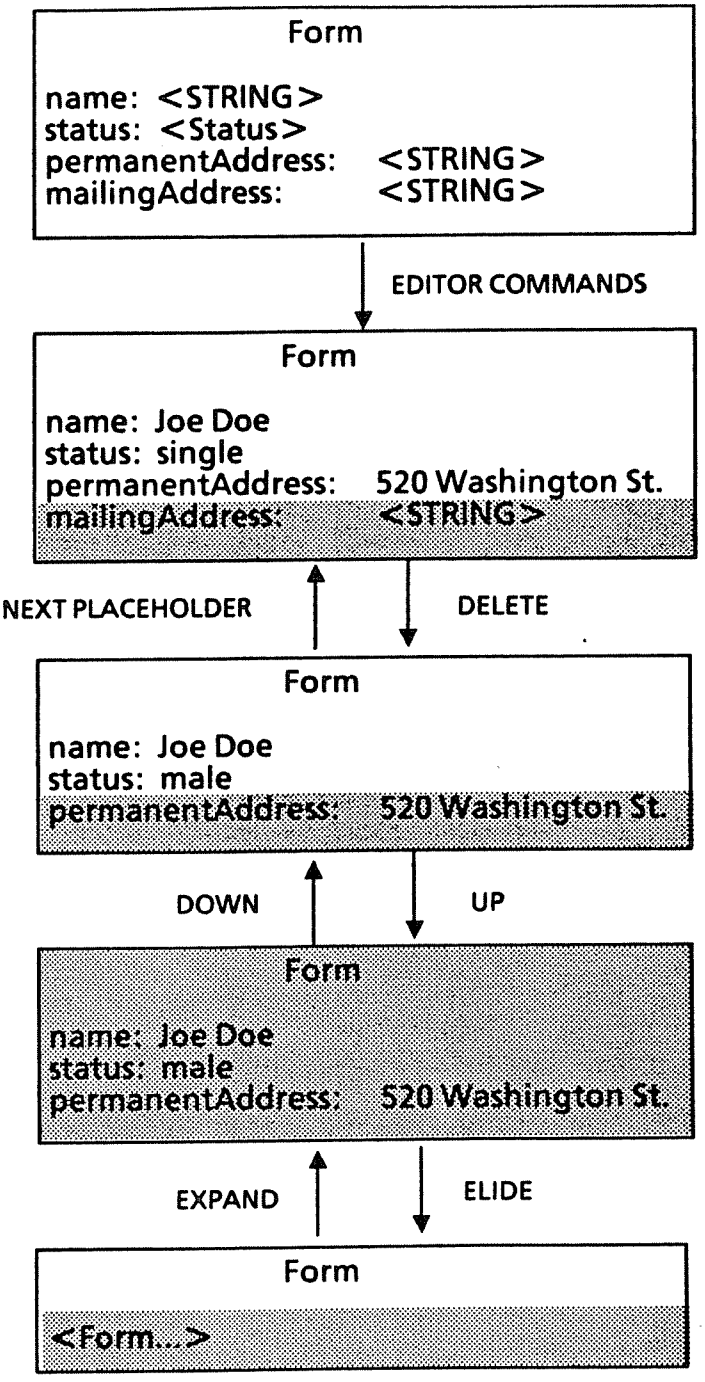


Figure 1: Editing a Form

Figure 2 shows the definition of the class 'Form'. The types 'Status' and 'Form' describe the contents of the form, and the variable 'form' stores an instance of it.

The method 'Load' is the load method of the class, and is invoked when a user asks a dialogue manager to load the presentation of the form in its window. The method sends initial messages to the dialogue manager, which submit the variable 'form' for editing, and specify various attributes of attribute groups. An example of these messages is the message:

```
Form: CLASS = {
  -- type declarations
  Status: TYPE = {married, single, widowed, divorced};
  Form: TYPE = RECORD [                                  -- contents of the form
    name: STRING,
    status: Status,
    permAddress: STRING,
    mailingAddress: STRING];

  -- variable declarations
  form: Form <- [NIL, single, NIL, NIL];                 -- variable that stores the form

  -- the load method
  Load: METHOD [dm: DM] = {
    OPEN dm;
    Edit [var: form];                                    -- submit 'form' for editing
    Prompt [attrGrp: form, val: "        Form"];         -- specify the prompt of 'form'
    SelfUpdate [attrGrp: form, val: FormUpdated];        -- specify the 'selfUpdate' method
    Initialized [attrGrp: form, val: FALSE, range: children]};  -- make all fields uninitialized
    Alignment [attrGrp: form, val: vertical, range: children]};  -- align each field vertically
    Titled [attrGrp: form, val: TRUE, range: children];  -- title each field
    Required [attrGrp: form.mailingAddress, val: FALSE]; -- make 'mailingAddress' optional

  -- a method that updates the value of 'form'
  FormUpdated: METHOD [newVal: Form] =
    {form <- newVal};

  -- main body
  MakeEditable [load: Load]};
```

Figure 2: The Class 'Form'

Initialized [attrGrp: form, val: FALSE, range: children]

which asks the dialogue manager to make the fields of the form uninitialized. As a result the dialogue manager displays placeholders for the fields of a new form. The comments in the figure explain the function of other messages.

The method 'FormUpdated' is the 'selfUpdate' method of 'form', and is called when a user modifies the form. It updates the variable with the current values of the fields of the form. The method is called when *any* field changes, and it receives the values of *all* its fields in the 'newVal' parameter. Later examples illustrate how an object can receive incremental changes.

In the above example, several dialogue managers may be used to interact with the object simultaneously. Each dialogue manager calls the load method, receives the messages from the object, displays a form, and sends the updated value in the parameter of the 'FormUpdated' method. The next example shows how an object can ensure that it talks to no more than one dialogue manger at a time.

The main body contains a call to 'MakeEditable', which registers the load method with the system.

2. Extended Form

We now describe the class 'ExtendedForm', which is an extension of the class 'Form'. It forces an instance to interact with only one dialogue manager at a time, and allows it to react to the 'save', 'reset', and 'empty' commands. Figure 3 shows how these commands are used.

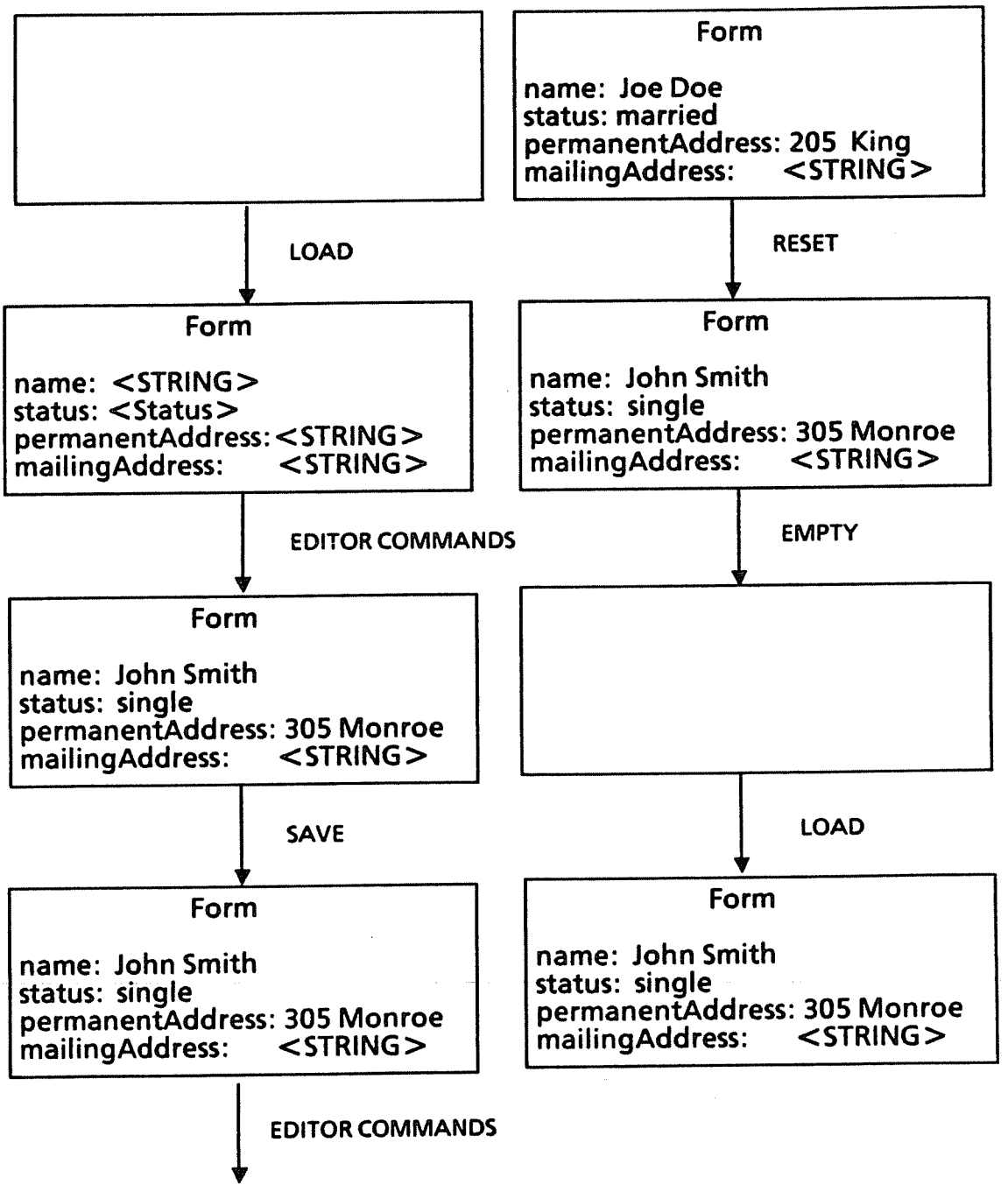


Figure 3: Editing an Extended Form

In the figure, the user loads an empty subwindow with an uninitialized presentation of a new instance of the class, fills the form, and executes the 'save' command to save the values in stable storage. He then changes the fields of the form, realizes it was a mistake to do so, and executes the 'reset' command to replace the current values to the saved ones. Next, he unloads the presentation from the window by executing the 'empty' command, and loads it again by executing the 'load' command. The fields of the form are initialized with the saved values.

Figure 4 gives an outline of the class. The header of the class declaration contains the the parameter 'dataFile'. As a result, instances of the class are associated with data files, and can read and write their variables into these files.

The variable 'curDm' stores a pointer to the current dialogue manager. The variable 'newForm' tells the instance if the value stored in the variable 'form' has ever been initialized by the user. This information is used by the load method.

The load method is similar to the corresponding method of 'Form'. There are three main differences. First, the method ensures that only one dialogue manager is connected to the object at a time. It checks the value of the variable 'curDM', which is initialized to 'NIL' when an object is activated. If the value is 'NIL', it initializes the variable with the value of the parameter 'dm', and sends appropriate messages to the dialogue manager. Otherwise, it sends an error message saying that object is currently being edited in another window. No variables are displayed in the new window.

Second, it submits methods to the dialogue manager to handle the 'reset', 'empty', and 'save' commands. Third, it initializes the fields of the form only if the form has not been initialized in current or past activations of the object. (A more

```
ExtendedForm: CLASS [dataFile: STRING] = {
  -- type declarations
  ...
  curDM: DM <- NIL;                          -- the current dialogue manager
  newForm: BOOL <- TRUE;                     -- tells the instance if form is uninitialized

  Load: METHOD [dm: DM] = {                  -- the load method
    OPEN dm;                                 -- send messages to 'dm'
    IF curDM = NIL THEN {                    -- if the object is not connected
      curDM <- dm;                           -- to any other dialogue manager
      ResetMethod [val: Reset];              -- submit the reset method
      SaveMethod [val: Save];                -- submit the save method
      EmptyMethod [val: Empty];              -- submit the empty method
      ...                                    -- other messages
      IF newForm THEN
        Initialized [attrGrp: form, val: FALSE,   -- uninitialize the fields of the form
          range: children]}
      ELSE
        Message ["..."]};                    -- send error message to user

  FormUpdated: METHOD [newVal: Form] = {
    newForm <- FALSE;
    ...};

  Save: METHOD [] = {                        -- the save method
    WriteDataStructures []};                 -- save data structures into data file

  Reset: METHOD [] = {                       -- the reset method
    ReadDataStructures [];                   -- read data structures from data file
    IF newForm THEN
      Initialized [var: form, val: FALSE,
        range: children]
    ELSE
      Value [attrGrp: form, val: form]};

  Empty: METHOD [] = {                       -- the empty method
    curDM <- NIL};                           -- reset curDM

  ReadDataStructures: PROCEDURE ...          -- reads data structures from data file

  WriteDataStructures: PROCEDURE ...         -- writes data structures into data file

  MakeEditable [load: Load]};                -- main body
```

Figure 4: The Class 'ExtendedForm'

advanced class would receive updates to the fields of the class *incrementally*, store for

each field a variable which tells an instance if the field is initialized, and uninitialize a field only if it is not initialized.)

The method 'FormUpdated' is similar to its counterpart in 'Form' except that it sets the variable 'newForm' to 'FALSE' (the method is called only if the form is initialized). The method 'Save' is called when the user wishes to save his changes in stable storage. It calls the procedure 'WriteDataStructures', which writes the value of 'form' and 'newForm' to the data file.

The method 'Reset' is called when the user wishes to reset the contents of a window with the saved data structures. It reads the values of the variable 'form' and 'newForm' from the data file, and either uninitializes all fields of the form, or updates them with the saved values.

The method 'Empty' is called when the user executes the 'empty' command, which unloads the presentation of the object from the window, and breaks the connection between the object and the dialogue manager. The method resets the value of 'curDM' to 'NIL'.

3. Roommate Expenses

We now describe the class 'RoommateExpenses'. An instance of the class allows three roommates, called 'A', 'B', and 'C', to enter their expenses incurred on behalf of the group, and displays, after each modification, deletion, and insertion of an expense, the money owed to each roommate. (We have considered here a *fixed* number of roommates to keep the example simple. Dost can easily handle a *variable* number, since an object can create display nodes *dynamically*.)

Figure 5 shows how a user may edit the presentation of an instance of the class. A presentation displays the money owed to the three roommates and the lists of expenses incurred by them. A user may insert, delete, or modify an element in a list. The object responds by updating the money owed to each roommate. The figure shows the insertion of a new element to the list of expenses incurred by 'B'.

Figures 6 contains an outline of the class. Like the classes 'Form' and 'ExtendedForm', and the the class 'Bibliography' described in chapter 2, it contains declarations that define the variables displayed in a window, a load method that sends initial messages to a dialogue manager, and methods to receive updates to a presentation displayed in a window.

Two important features of the class distinguish it from other classes defined so far. First, it *incrementally* updates each expense list submitted to the dialogue manager. This manner of updating a list can be contrasted to the one used by 'Bibliography', which updates a reference list in 'batch' when the user executes the 'accept' command.

Second, it keeps the contents of *all* windows displaying an object *consistent* by keeping track of all the dialogue managers that have invoked the load method, and announcing each change to all of them.

The type 'Expenses' describes lists of expenses. It is used to declare the variables 'AExpenses', 'BExpenses', and 'CExpenses', which store the expenses of 'A', 'B', and 'C' respectively.

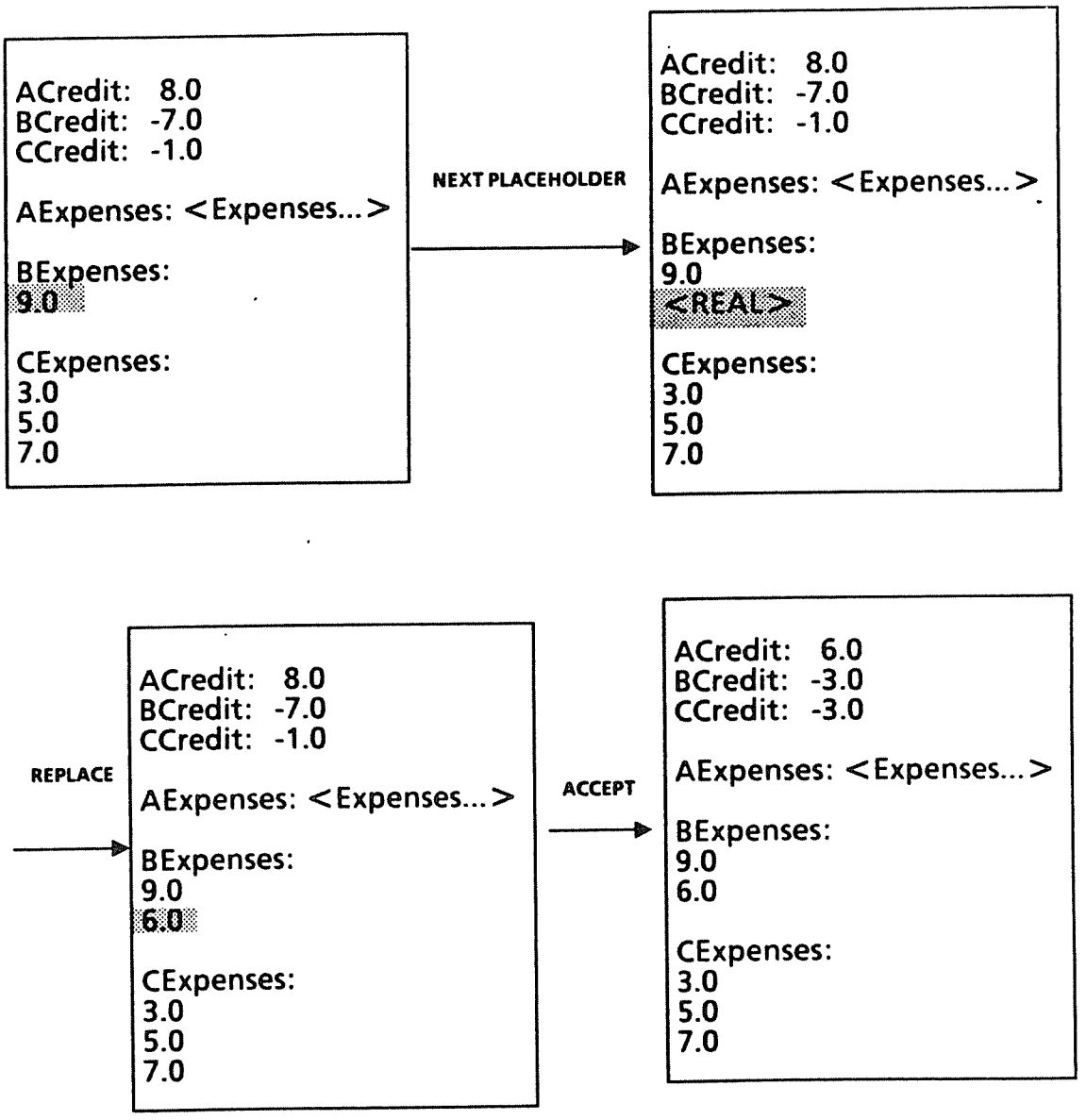


Figure 5: Editing Roommate Expenses

The variables 'ACredit', 'BCredit', and 'CCredit' store the money owed to each roommate. The variable 'dms' keeps track of all the dialogue managers interacting with the object.

The load method sends messages to a dialogue manager. It sets the attributes of different attribute groups, and submits the variables 'ACredit', 'BCredit', 'CCredit', 'AExpenses', 'BExpenses', and 'CExpenses' for editing.

The methods 'AInserted', 'ANewExpense', and 'ADeleted' are the 'insertUpdate', 'elementUpdate', and 'deleteUpdate' methods of 'AExpenses'. They are called when an element is inserted, modified, and deleted, respectively, in a presentation of

```
RoommateExpenses: CLASS = {
  Expenses: TYPE = POINTER TO SEQUENCE [
    content: SEQUENCE length: CARDINAL OF REAL];

  ACredit, BCredit, CCredit: REAL;                    -- credits of A, B, and C
  AExpenses, BExpenses, CExpenses: Expenses <- NIL;   -- expenses of A, B, and C
  ...                                                 -- other variables
  dms: POINTER TO RECORD [                            -- the dialogue managers
    contents: SEQUENCE length: CARDINAL OF DM];       -- connected to an object

  Load: METHOD [dm: DM] = {                           -- the load method
    OPEN dm;
    AddDM [dm];                                       -- add 'dm' to 'dms'
    InsertUpdate [attrGrp: AExpenses, val: AInserted];    -- set 'insertUpdate',
    ElementUpdate [attrGrp: AExpenses, val: ANewExpense]; -- 'elementUpdate',
    DeleteUpdate [attrGrp: AExpenses, val: ADeleted];     -- & 'deleteUpdate' attributes
    ReadOnly [attrGrp: ACredit, val: TRUE];           -- user cannot edit 'ACredit'
    ...                                               --set other attributes
    --submit variables for editing
    Edit [var: ACredit];
    Edit [var: BCredit];
    Edit [var: CCredit];
    Edit [var: AExpenses]...};
```

Figure 6(a): The Class 'RoommateExpenses'

```
-- 'insertUpdate' method of 'AExpenses'
AInserted: METHOD [index: CARDINAL, dm: DM] = {
  InsertSlotInAExpenses[index];                    -- insert a slot in 'AExpenses'
  -- announce insertion to other dialogue managers
  FOR i IN [0..dms.length) DO
    IF dm[i] # dm THEN
      dm[i].InsertAfter[seq: AExpenses, index: index]};
  ENDLOOP

-- 'elementUpdate' method of 'AExpenses'
ANewExpense: METHOD [index: CARDINAL,
  newVal: REAL, dm: DM] = {
  ...
  Value [attrGrp: ACredit, val: ACredit];          -- update 'ACredit'
  Value [attrGrp: BCredit, val: BCredit];          -- update 'BCredit'
  ...};

-- 'deleteUpdate' method of 'AExpenses'
ADeleted: METHOD [index: CARDINAL, dm: DM] = {
  ...};

-- similar methods for 'BExpenses' & 'CExpenses'
BInserted: METHOD...

InsertSlotInAExpenses: PROCEDURE...};              -- other code
```

Figure 6(b): The Class 'RoommateExpenses'(contd.)

'AExpenses'.

The object responds to an insertion in the presentation of 'AExpenses' in one window by creating a slot for the new element in the variable, and inserting a corresponding entry in all other windows displaying the object. It responds to a modification in one window by updating other windows, calculating the change in the credits of each roommate, and updating the new credits in all windows. It responds to a deletion by deleting the appropriate slot in the variable 'AExpenses', informing other dialogue managers about the deletion, recalculating the credits, and updating all windows with the new values.

4. Spreadsheet

We now describe a generalization of 'RoommateExpenses', a class whose instances store simple spreadsheets. Each instance of the class is associated with two presentations, which are displayed in different presentation subwindows. The top subwindow shows a 5 by 5 matrix that defines the spreadsheet, and the bottom one shows expressions that define an element of the matrix with respect to other elements of the matrix. Figure 7 illustrates how a user may edit the presentations of an instance of the class.

Figures 8(a) and (b) shows an outline of the class. There are two features of the class that distinguish it from other classes presented so far. First, an instance of the class has multiple presentations displayed in different subwindows. Second, the object uses path names and node numbers to name display nodes.

In the discussion of the class 'Spreadsheet' and the remaining classes in this chapter, we will assume that the an object is connected to at most one dialogue manager.

The type declarations define the variables 'values' and 'definitions'. The former stores the values of the matrix, and the latter stores the definitions.

The load method, like other load methods discussed so far, sends initial messages to a dialogue manager.

The methods 'ColUpdated' and 'RowUpdated' are the 'elementUpdate' methods for attribute groups 'ValRow' and 'Values' respectively, and are called in succession when a user changes an element of the spreadsheet. They use the values of their parameters to determine the row index, column index, and the value of the changed element, and call the 'PropogateValChange' method described below.

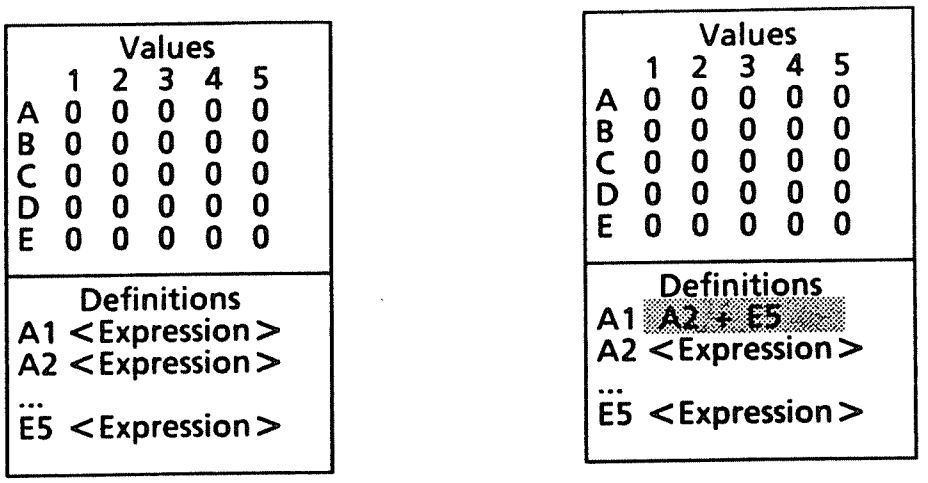


Figure 7: Editing a Spreadsheet

The procedure 'PropogateValChange' propagates the change in an element of the spreadsheet to the elements that depend on it. It uses path names to update nodes, as shown in the figure. The use of path names does not incur any runtime overhead, since elements of display nodes of arrays can be randomly accessed by a dialogue manager.

The method 'DefUpdated' is called when a user makes a new definition. It checks the definition for errors. If there is an error, it uses the node number to highlight the node. If there are no errors, it calls the 'PropogateDefChange' method to update appropriate nodes in the presentation of 'values'.

```
Spreadsheet: CLASS = {
  -- type definitions
  ...
  ValRow: ARRAY ColIndex OF CARDINAL;
  Values: ARRAY RowIndex OF ValRow;
  Definitions: ARRAY DefIndex OF Expression;

  -- variable declarations
  ...
  values: ARRAY RowIndex OF ValRow;                    -- the matrix of spreadsheet values
  definitions: ARRAY DefIndex OF Expression;           -- the definitions

  Load: METHOD [dm: DM] = {                            -- the load method
    OPEN dm;
    -- set 'elementUpdate' attributes
    ElementUpdate [attrGrp: ValRow, val: ColUpdated];
    ElementUpdate [attrGrp: Values, val: RowUpdated];
    ElementUpdate [attrGrp: Definitions, val: DefUpdated];
    ...                                                -- set other attributes
    Edit [var: values, pres: 0];                       -- display 'values' and 'definitions'
    Edit [var: definitions, pres: 1]};                 -- in appropriate subwindows
```

Figure 8(a): The Class 'Spreadsheet'

```
  -- method called when an element of a row of
  -- 'values' changes
  ColUpdated: METHOD [index: ColIndex,
     newVal: CARDINAL ] = {
     curCol <- index;                          -- save the column position
     colVal <- newVal};                        -- save the column value

  -- method called when a row of 'values' changes
  RowUpdated: METHOD [index: RowIndex,
    newVal: ValRow] = {
    values [index][curCol] <- colVal;
    PropogateValChange[index, curCol]};

  -- method called when a definition changes
  DefUpdated: METHOD [index: DefIndex,
     newVal: Expression,
     dn: NODE OF Definitions] = {
     IF NOT ExprError[newVal] THEN {
        definitions [index] <- newVal;
        PropagateDefChange[index]              -- if not error propagate changes
     ELSE Highlighted [attrGrp: dn, val: TRUE]}};   -- if error highlight node

  -- propagate the change to a  value
  PropagateValChange: PROCEDURE...
     [row: RowIndex, col: ColIndex] = {
     -- calculate 'tRow', 'tCol', 'tVal':
     -- the row, column, and new value
     -- of a dependent element
     ...
     Value [attrGrp: values$[tRow, tCol], val: tVal];   -- update dep.
element
     ...};

  -- propagate the change to a definition
  PropagateDefChange: PROCEDURE...

  MakeEditable [load: Load, numPres: 2]};          -- main body
```

Figure 8(b): The Class 'Spreadsheet'(contd)

5. Statement List

We now present a class called 'StatementList'. Instances of the class check the static semantics of statement lists of the programming language ASPLE, which has been used to compare several formal mechanisms for defining the semantics of

programming languages [23].

Figure 9 shows how a user may edit the presentation of an instance. The initial presentation shows a placeholder for a statement list. A user may 'expand' it to create a placeholder for a statement. A statement can be an **assignment**, **if**, **while**, **input**, or **output** statement. The user may execute the 'menu' command to select the kind of statement desired. In this example, he selects an **if** statement. The dialogue manager responds by showing a template for the statement. The user selects the else part of the statement, expands it, inserts a **while** statement, and elides it. He then inserts an **if** statment for the **then** part, deletes the **else** part of the new statement, and elides the statement. The **else** part is optional, and may be reinserted by executing the 'next placeholder' command.

Figures 10(a) and 10(b) describe the class. The types 'Expression' and 'ID' define expressions and identifiers of ASPLE. The type 'Statement' describes the structure of a statement and 'StatementList' defines the structure of a statement list. The variable 'stmtList', stores the list of statements defined in an instance.

The type 'Expression' may be declared as a structure or as text. Dost allows presentations to be edited as both structures and text. The presentations of simple display nodes such as strings and integers are edited as text. The presentations of structure nodes such as records, arrays and sequences are edited as structures. Therefore, if expressions are defined as strings, they may be edited as text. If they are defined using structure types, they can be edited as structures.

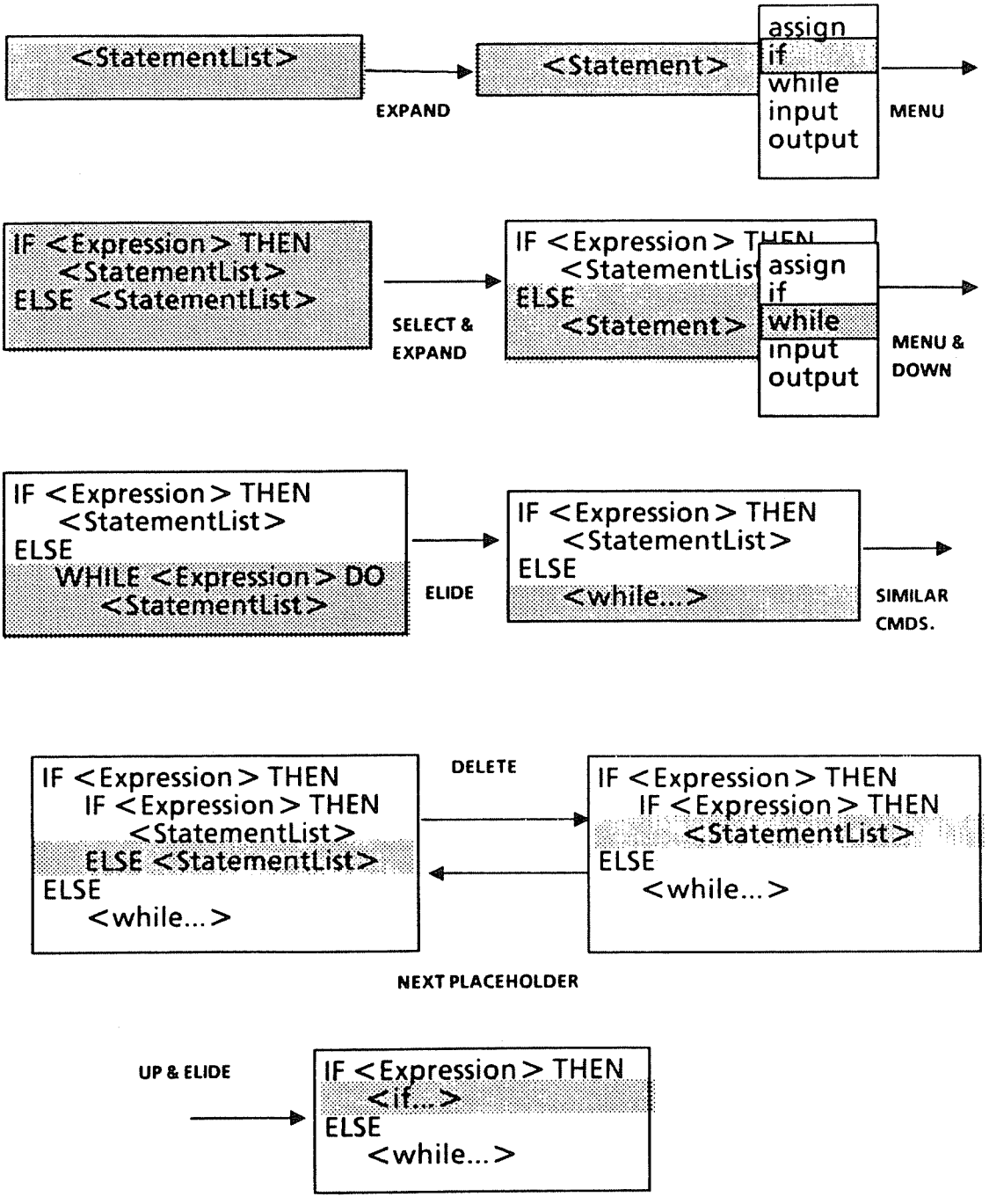


Figure 9: Editing a Statement List

The load method sends initial messages to a dialogue manager. These messages define formats, specify 'selfUpdate' methods, and submit the variable 'stmtList' for editing. We describe below the function of some of these messages.

The first message sets the alignment attribute of the 'default' group to 'horizontal'. All attribute groups inherit this value. The 'Identifier' message asks the dialogue manager to ensure that the presentations of display nodes of type 'ID' are restricted to alphanumeric strings starting with a character. The next two messages define the alignment of 'StatementList' and 'Statement' respectively. The 'Required' message

```
StatementList: CLASS = {
  -- type declarations
  StatementKind: TYPE =  {assign,  if, while, input, output};   -- the five kinds of statements
  ID: TYPE = STRING;                                           -- definition of identifiers
  Expression: TYPE = ...;                                      -- definition of expressions
  Statement: TYPE = RECORD [                                   -- declaration of a statement
    info: SELECT kind: StatementKind FROM
      assign => [                                              -- structure of an assign statement
        lhs: ID,
        rhs: Expression],
      if => [ expr: Expression,
        thenPart: StatementList,
        elsePart: StatementList],
      while => [
         expr: Expression,
         doStmts: StatementList],
      input => [
         id: Id],
      output => [
         expr: Expr];
      ENDCASE];
  StatementList: TYPE = POINTER TO RECORD [                    -- structure of a statement list
   contents: SEQUENCE size: CARDINAL OF Statement];

  -- variable declarations
  stmtList: StatementList <- NIL;                              -- variable to store the statement list
```

Figure 10 (a): The Class 'StatementList'

```
Load: METHOD [dm: DM] = {                                   -- the load method
  OPEN dm;
  Alignment [attrGrp: *, val: horizontal];                  -- default alignment
  Alignment [attrGrp: StatementList, val: indented];        -- indent statement lists
  Alignment [attrGrp: Statement, val: vertical];            -- align statements vertically
  Required [attrGrp: if Statement.thenPart, val: TRUE];     -- then parts are 'required'

  -- format input & output statements
  Prompt [attrGrp: input Statement.id, val: "INPUT "];
  Prompt [attrGrp: output Statement.id, val: "OUTPUT "];

  -- format assign statements
  Prompt [attrGrp: assign Statement.rhs, "<- "];

  -- format while statements
  Prompt [attrGrp: while Statement.expr, val: "WHILE "];
  Suffix [attrGrp: while Statement.doStmts, val: "DO"];

  -- format if statements
  Prompt [attrGrp: if Statement.expr, val: "IF "];
  Suffix [attrGrp: if Statement.expr, val: "THEN"];
  Prompt [attrGrp: if Statement.elsePart, val: "ELSE "];
  Alignment[attrGrp: if Statement.elsePart, val: vertical];     -- override 'StatementList'
  Alignment [attrGrp: if Statement.elsePart[], val: indented];  -- override 'Statement'

  -- specify 'selfUpdate' methods
  SelfUpdate [attrGrp: Expression, val: ExprUpdated];
  SelfUpdate [attrGrp: Statement, val: StatementUpdated];

  Edit [var: stmtList]};                                    -- display 'stmtList'

-- 'selfUpdate' method for nodes of type 'Statement'
StatementUpdated: METHOD [newVal: Statement,
  dn: NODE OF Statement] = {
  WITH newVal SELECT FROM
    assign => {...};
    if   => {};
    ...};

-- 'selfUpdate' method for nodes of type 'Expression'
ExpressionUpdated: METHOD [newVal: Expression,
  dn: NODE OF Expression] = { ... } ...};
```

Figure 10(b): The Class 'StatementList'(contd)

tells the dialogue manager that the 'thenPart' field of the 'if' variant of a 'Statement' is

required in an initialized presentation of the variant. A similar message is not sent for the 'elsePart' field, since else parts of if statements are optional.

The next group of messages defines the syntactic sugar of the different kinds of statements. For instance, the messages:

```
Prompt [attrGrp: while Statement.expr, val: "WHILE "];
Suffix [attrGrp: while Statement.expr, val: "DO"];
```

ensure that the display of the conditional expression in a **while** statement is preceded by the string "WHILE ", and succeeded by the string "DO". Thus, **while** statements are displayed as shown in figure 9. Since the 'alignment' attribute of 'StatementList' is 'indented', the beginning of the 'doStmts' field is indented with respect to the beginning of its parent. Moreover, since the 'alignment' attribute of 'Statement' is 'vertical', the beginning of the presentation of an element of 'doStmts' is aligned vertically with the beginning of the presentation of 'doStmts'.

The definition of the format of an **if** statement is slightly more complex. The component groups for 'elsePart' and its elements override the 'alignment' attribute of 'StatementList' and 'Statement' respectively. The alignment attribute of 'elsePart' is 'vertical'. As a result, the keyword 'ELSE' is aligned vertically with 'IF'. The alignment attribute of an element of 'elsePart' is 'indented'. As a result, elements of 'elsePart' are indented with respect to the keyword 'ELSE'.

The method 'StatementUpdated' checks the static semantics of statements. The role of 'ExpressionUpdated' depends on the declaration of 'Expression'. If expressions are declared as text, the method checks their static semantics. Otherwise, it first parses them and then checks their static semantics.

The example demonstrates the overhead of defining structures as text. An object is responsible for *parsing* these structures. However, we expect that in most cases, the task of parsing will be handled by standard library routines. (See Chapter 4 §2.4 for more discussion on this topic.)

In this example, we have considered only the static semantics of statement lists of ASPLE. A more advanced class would compile or interpret these statement lists.

6. Directory

Our last example is the class 'Directory', whose instances store editable Unix-like directories. An instance can present several views of the directory. It can show a 'long listing', which displays all the fields of an entry, or a 'short listing', which displays only the names of the entries. Moreover, it can sort the displayed entries by name or creation time.

Figure 11 shows how a user may edit the presentation of an instance of the class. Figure 11(a) displays a 'long listing' of the files in the directory sorted by name. The 'file' field is used to specify the operand of the 'copy' command, which is a command special to directories. It creates in an instance a copy of an existing file or directory.

The user may edit the presentation to manipulate the directory. Changes in the presentation cause corresponding changes in the object. For instance, if the user changes the name 'test.p' to 'prog.p', as shown if figure 11(b), the object registers the change, and resorts the directory if necessary.

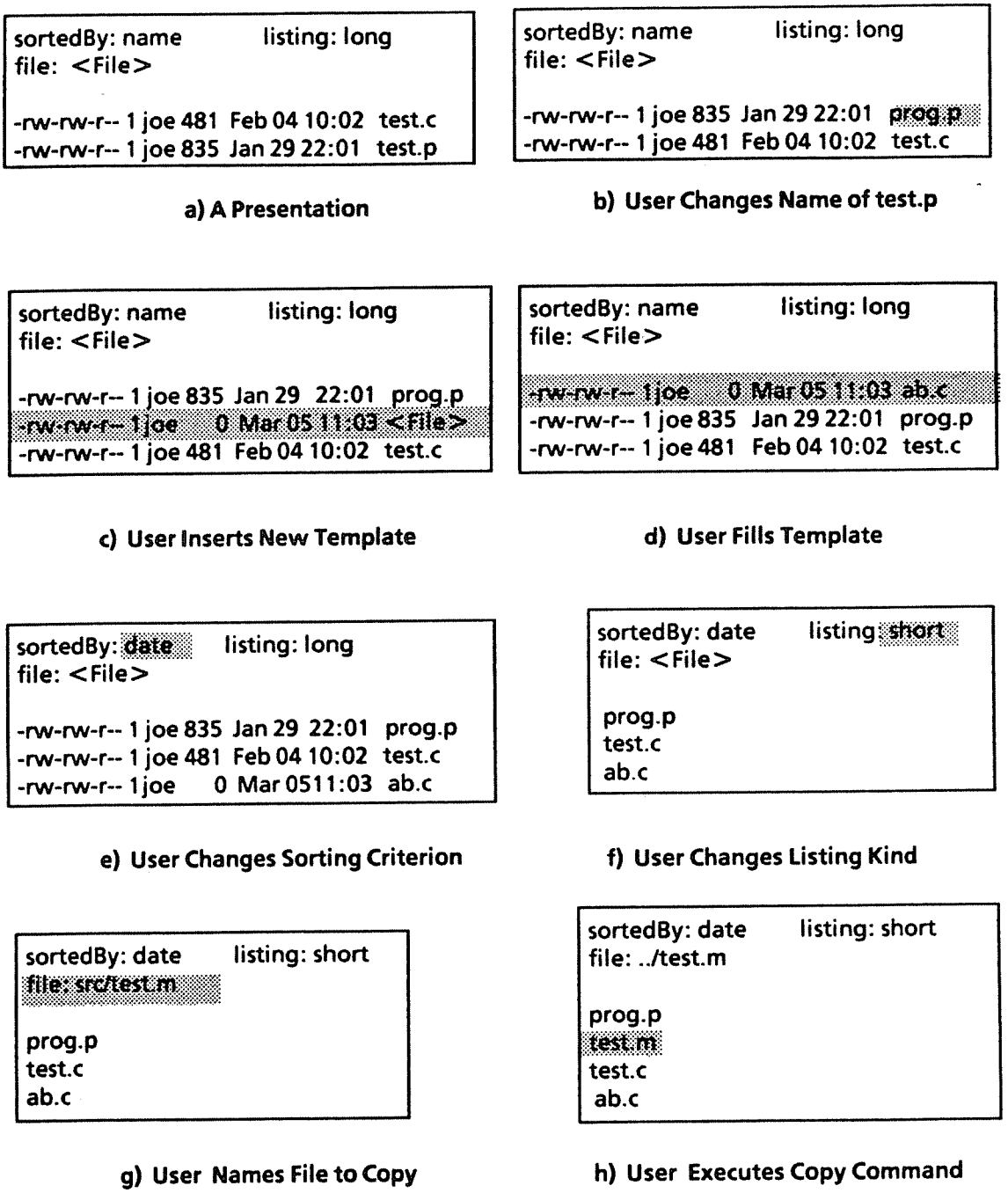


Figure 11: Editing a Directory

A user can create a template for a new entry by selecting an existing entry and executing either the 'next placeholder' or the 'previous placeholder' command. The template shows a placeholder for the name field, and initial values for the others. The user may initialize the placeholder, and modify the editable fields. The directory is resorted if necessary, as shown in figure 11(d).

The user may change the sorting criterion by editing the 'sortedBy' field, as shown in figure 11(e). Figure 11(f) shows how a user may change the 'long listing' to 'short listing'. Figure 11(g) and 11(h) show how a user may insert a copy of an existing file or directory. He fills the 'file' field and executes the 'copy' command. The new entry is inserted in the appropriate position. The user may change its name and other editable fields.

Figures 12(a) and 12(b) show how the class may be programmed. Four features of the directory make the class different from others presented so far. First, unlike the lists defined by previous classes, a list of directory entries can be sorted by different criteria. Second, certain fields of an entry may be included or excluded from the display, depending on the value of the 'listing' field in the presentation. Third, the class tailors the user interface of its instances by defining the 'copy' command. Finally, the class has to define display names for some of the enumeration literals it defines.

The type 'EntryRec' defines the contents of a directory entry, and the type 'Entry' defines a pointer to it. A directory list is stored in a variable of type 'Directory'. A list is composed of variables of type 'Entry', and not 'EntryRec'. Each instance stores two such lists. One list, stored in 'nameSorted', sorts the entries according to the 'name' field, while the other list, called the 'timeSorted', sorts it

```
Directory: CLASS = {
  -- type declarations
  SortedBy: TYPE = {name, date};
  ListingKind: TYPE = {short, long};
  ReadAccess: TYPE = {read , noRead};                    -- read access
   ...                                                   -- other accesses
  DirOrFile: TYPE = {dir, file};                         -- directory or file
  Access: RECORD [                                       -- different accesses
    selfR : ReadAccess,
    selfW: WriteAccess,
    selfX: ExecuteAccess,
    groupR: ReadAccess,
    ...
    othersX: Executeccess];
  ...                                                    -- other declarations
  Time: TYPE = RECORD [
    day: Day, month: Month, hour: Hour, min: Min];
  File: TYPE = STRING;
  EntryRec: TYPE = RECORD [                              -- definition of a directory entry
    dirOrFile: DirOrFile,
    access: Access,
    numLinks: CARDINAL,
    owner: STRING,
    size: CARDINAL,
    time: Time,
    name: File];
  Entry: TYPE = POINTER TO EntryRec;
  Directory: POINTER TO RECORD [
    content: SEQUENCE size: CARDINAL OF Entry];

  -- variable declarations
  sortedBy: SortedBy <- name;                            -- defines sorting criterion
  listing: ListingKind <- long;                          -- defines desired listing
  file: File;                                            -- name of file to be copied
  nameSorted, timeSorted: Directory;                     -- the two sorted versions of dir.

  Load: METHOD [dm: DM] = {                              -- the load method
    OPEN dm;
    ReadOnly [attrGrp: Entry.date, val: TRUE];
    Value [attrGrp: Access.selfR, val: read];
    AddCommand [attrGrp: Entry, name: "copy",
      comMethod: Copy];
    DisplayName [enum: ReadAccess[read], val: "r"];
    DisplayName [enum: ReadAccess[noRead], val: "–"];
    ...};
```

Figure 12(a): The Class 'Directory'

according to the 'time' field. Since both lists are composed of pointers, the information about an entry does not have to be duplicated. Each entry is associated with two pointers, one in the 'nameSorted' list, and the other in the 'timeSorted' list. The two pointers share a common referent.

The method 'Load' is the load method of the class. It sends initial messages to a dialogue manager, which, submit variables 'sortedBy', 'listing', 'file', and one of the variables 'nameSorted' or 'timeSorted' for editing, set attributes of attribute groups, submit the command method of the 'copy' command, and sets the display names of access fields. The variable 'nameSorted' is submitted if the current value of 'sortedBy' is 'name', otherwise 'timeSorted' is submitted.

Figure 12(b) outlines the other methods of the class. These are concerned with resorting the directory when a new entry is inserted, changing the directory list displayed when the user changes the sorting criterion, changing the listing when the user changes the 'listing' field, initializing a new entry with the current time, and implementing the command method of the 'sort' command.

The method 'EntryUpdated' is called when an entry is updated. It stores the new value in the variable 'newEntry', and updates it with the current time. This value is used by the method 'EntryInserted', which is called when a user enters initializes a new entry in a directory. Note that the order in which the two procedures are called is important. A call to 'EntryInserted' is always preceded by a call to 'EntryUpdated' (Chapter 2).

The method 'EntryInserted' performs the following tasks. First, it inserts a pointer for the new element in both the 'nameSorted' and the 'timeSorted' list. Next, it

```
-- 'selfUpdate' method for 'Entry'
EntryUpdated: METHOD [newVal: Entry] = {
  insertedEntry <- newVal; -- save the new entry for 'EntryInserted'
  insertedEntry.time <- curTime}; -- set the current time

-- 'insertUpdate' method for 'Directory'
EntryInserted: METHOD [index: CARDINAL, dn: NODE TO ENTRY, dm: DM];
  OPEN dm;
  ... insert element in 'nameSorted' and 'timeSorted'
  Delete [dn: dn];
  IF sortedBy = nameSorted THEN {
    -- find the new insertion position according to sorting criterion
    i <- FindNewPos [insertedEntry];
    InsertAfter [dn: nameSorted, index: i];
    Value [attrGrp: nameSorted[i], val: nameSorted[i]]}
  ELSE {
    ... -- insert in 'timeSorted'};

-- 'selfUpdate' method for 'sortedBy'
ChangeSorting: METHOD [newVal: SortedBy] = {
  OPEN dm;
  IF (newVal = name) AND (sortedBy # name) THEN {
     Delete [var: timeSorted];
     Edit [var: nameSorted]}
  ELSE {...};

-- 'selfUpdate' method for 'listing'
ChangeListing: METHOD [newVal: Listing] = {
   OPEN dm;
   IF newVal = long THEN {
     Hidden [attrGrp: Protection, val: FALSE];
     ....}
   ELSE {
     Hidden [attrGrp: Protection, val: TRUE];
     ...}};

-- 'selfUpdate' method of 'file'
FileUpdated: METHOD [newVal: File] = {
   IF ValidName [newVal] THEN file <- newVal
   ELSE Highlighted[attrGrp: file, val: TRUE]};

-- command method for the 'copy' command
Copy: METHOD [] = {...}; ...};
```

Figure 12(b): The Class 'Directory' (contd)

inserts the new element in the display according to the current sorting criterion.

Finally it deletes the old element from the display.

The method 'ChangeSorting' is called when the user changes the variable 'sortedBy'. It replaces the a presentation of 'timeSorted' with a presentation of 'nameSorted', if the new value of 'sortedBy' is 'name', and vice versa, if the value is 'time'.

The method 'ChangeListing' is called when the value of the variable 'listing' is changed by the user. If the new value is 'short' it sets the 'hide' attribute of the appropriate attribute groups to 'TRUE'. For instance, it sets the 'hide' attribute of 'Protection' to 'TRUE' to hide all protection fields of the directory. If the new value is 'long' it sets the 'hide' attribute of the above attribute groups back to 'FALSE'.

The method 'FileUpdated' is called when the value of the variable 'file' is changed. It updates the value of the variable if the new value is a legal file name. Otherwise it reports the error to the user.

The method 'Copy' is the command method for the 'copy' command. It uses the value of the variable 'file' to implement the command.

Chapter 4

Discussion

This chapter describes the role played by different components of our approach, discusses some useful features missing from it, describes our experience with Dost, compares our approach with related work, and presents conclusions and directions for future research.

1. Rationale

Our approach consists of four components: the editing model of interaction, objects, the dialogue manager, and the input/output primitives. Our design of these components was influenced by our goal of making *general* interactive programs both *easy to use*, and *easy to implement*. In this section we discuss the role played by each component in meeting this goal.

1.1. Editing Model of Interaction

The editing model can have several *variations* [25, 26, 14, 46, 13, 45, 5, 32, 44] of which the model supported by Dost is one example. In the rest of this dissertation, we shall use the Dost variation as a representative example of the editing model.

This model was supported by Dost for two reasons:

(1) User interfaces that follow the model are *easy to use*.

(2) It is a *general* model.

The following two subsections defend these claims.

1.1.1. Ease of Use

We discuss below the properties of the editing model that, we believe, contribute to the usability of user interfaces that follow it.

Point and Select Paradigm

Operands of editor commands are selected by the *point and select* paradigm. As a result a user does not have to provide a *description* of the operand. For instance, to delete an entry from a directory, the user can select the entry by directly pointing at it, instead of typing its name.

In Place Updates

Modifications to displayed values are made *in place*. A user changes a value by directly editing its presentation. As a result, the user can *reuse* parts of the old presentation. Similarly, an object displays a new value by directly updating its presentation in a window. It does not have to redisplay the *entire* contents of the window.

As an example, consider a modification to the name of a directory entry, both by a user and the object that manages the directory. A user can change the name by changing the desired characters in the presentation of the current name, instead of typing the *complete* name in a separate portion of the screen. Similarly, the object (possibly in response to some message from another object) can change the name by directly updating its presentation. It does not have to redisplay the entire directory.

Automation of Input

When appropriate, templates and menus are provided to *automate* the entry of input. Templates free the user from entering the 'syntactic sugar' around values of variables, while menus save the user from entering the values themselves.

Structure Commands

The *structure* of values is made available to the user. The interface provides commands to move up or down in a structure, travel a presentation in increments of the currently selected structure, and hide or show the details of a structure.

Aids User Input

Facilities are provided to *aid* the user in entering input. These include titled presentations, uninitialized presentations, and menus. Titled presentations provide *prompts* for user input. Uninitialized presentations provide *placeholders* that describe the set of values that can replace them. Menus allow a user to examine his *input choices.*

Interactive Formatting

The model supports *interactive formatting*, which allows a user to override the default formats defined by the object. While we do not expect this feature to be used very often, there are several situations in which it can be useful. For instance, consider the following default format for an **if** statement:

```
IF <Expr> THEN
  <StmtList>
ELSE
  <StmtList>
```

Now consider an application of the above format to the following instance of the statement:

```
IF b = 1 THEN
  a <- 0
```

The presentation uses two rows in the display, and may be undesirable in a situation where maximum use of the available screen space is important. In Dost, the user can

select the assignment statement and execute the 'horizontal' command to change the display to:

IF b = 1 THEN a <- 0

Miscellaneous

The user has control over the order in which input is specified. Moreover, he can decrease screen clutter by removing optional entries from the display. For instance a user interacting with an instance of the class 'Form' (Chapter 3) can fill the form items in any order. Moreover, he can delete the optional 'mailingAddress' field.

1.1.2. Generality

Chapter 2 and 3 described in detail how editing interfaces can be used to interact with a diverse range of applications. In this section, we describe, informally, some other applications that can fruitfully present such interfaces.

Sending Mail

A system could provide a special object responsible for sending messages. The object could display fields to specify the destination, subject, contents and other characteristics of the message to be sent. It could ensure that a user does not make syntax or semantic errors such as entry of an invalid user name in the destination field. ([1] and [28] contain related discussions on this topic.)

Debugger

We now illustrate how a debugger can present an editing interface. The debugger could have several presentations: one presentation to display the list of breakpoints, another to display variables of current interest (perhaps using some of the

ideas presented in [27]), and so on. A user could edit these presentations to debug a program. For instance, he could remove a breakpoint by deleting its entry from the list of breakpoints. Similarly, he could changes the value of a variable by editing its presentation.

Command Interpreter

An editing interface would also be useful for interacting with a command interpreter. A command interpreter could display the the history of commands specified by the user. A user would specify new commands by either entering a new command or selecting a previous command and executing the 'accept' editor command. The command interpreter could also allows deletion of selected commands to allow them to be undone.

An application such as a command interpreter may seem anomalous in an environment in which all applications offer editing interfaces. Command interpreters are traditionally used to send mail, manipulate directories, print files, and so on. However, we have argued earlier for editing interfaces to specify these operations. For instance, we have illustrated the advantages of using an editing interface to manipulate a directory.

Nonetheless, a command interpreter has an important role in an editing environment. It provides a central place to specify operations on a large number of objects. Moreover, it typically provides facilities to *program* a sequence of operations. We have shown in this example how a command interpreter could itself provide an editing interface.

Languages

The class 'StatementList' illustrated the usefulness of an interface that understands the structure and semantics of the constructs of ASPLE. Similar interfaces may be used to enter constructs of other languages including programming languages, configuration languages, and specification languages [5, 44, 11, 32, 50, 51, 29].

Databases

The class 'Bibliography' illustrated how a database of reference entries may by manipulated by editing its presentation. Other databases may be similarly manipulated.

1.2. Objects

Dost augments a traditional computing environment with extensions that allow applications to be created as *objects*. Objects have the following two properties that distinguish them from traditional applications (Chapter 2):

(1) They are part of the *permanent memory* of the system. They are associated with permanent names, and can be activated and passivated.

(2) They can *communicate* with each other and dialogue managers through messages.

This section discusses the rationale for supporting objects in Dost.

Object-Oriented Permanent Memory

The interaction model supported by Dost allows each application to act as an editor of *permanent* data that can saved between editing sessions. Therefore, it is important that the environment provide support for naming and accessing these data.

In traditional systems, permanent data are named and accessed through files. There is one drawback with this scenario: files are 'untyped'. There is no information in files about the programs that can manipulate them. Thus a user may ask a program to manipulate data it cannot process.

In Dost, files and the programs that manipulate them are kept together as objects. Therefore, a user is not responsible for connecting programs with files. As a result, there is no danger of an object being asked to manipulate data it cannot handle.

Application-Dialogue Manager Separation

The communication primitives in Dost allow an application and a dialogue manager to be two *separate* programs communicating with each other through messages. This separation offers several advantages:

- The different dialogue managers share a common code segment. In the absence of object-dialogue manager separation, a copy of the dialogue manager code neeeds to be linked to *every* class in the system. The dialogue manager code, in most cases, will be several times the size of a class. Therefore, it is important to have only one copy of it resident in memory.
- An object can be connected to several dialogue managers simultaneously. Thus it can be edited in several windows at the same time. In a multi-user environment, these windows can be on the screens of different users. As a result, several users can simultaneously examine and manipulate data of common interest. Examples of such data are directories, printer queues, and process lists, and databases.
- An object and a dialogue manager can reside on different computers, such as a workstation and a mainframe host.

Cooperation between Applications

Objects can communicate with each other. As a result, related objects can keep their data consistent. As an example, consider managers of Unix-like directories, which can share entries with other directories. In Dost, these managers can communicate with each other to keep the fields of a common entry consistent.

1.3. Dialogue Manager

The dialogue manager handles user interaction on behalf of the object. It presents a default user interface, which few objects need to change substantially. Thus an object is relieved from the task of converting user input into values of variables, specifying details of displaying data, and interpreting any of the default commands provided by the default interface. In our implementation, approximately thirty such commands are provided.

1.4. Input/Output Primitives

Our approach replaces traditional input/output primitives supported by conventional programming languages with an new set of primitives that support both *editing* and *automation*. In this section, we discuss the role played by the main components of the set of input/output primitives supported by Dost.

Display Nodes

Display nodes support the editing model of interaction. They provide an *external representation* of the data in an object that can be *edited* by the user and *updated* by an object.

Edit Messages and Update Methods

The 'Edit' messages and update methods replace traditional input and output procedures. They are necessary for an object to display variables and receive updates to them.

Input/Output of Programmer-Defined Types

The parameters of 'Edit' messages and update methods can be values of *programmer-defined* types such as variant records, sequences, and recursive structures referred by a pointer. As a result, an object is relieved from the substantial task of parsing input and displaying output.

Formatting Attributes

The formatting attributes allow an object to specify *general* properties about formatting information. As a result, an object is relieved from specifying the *details* of displaying data on the screen.

Attribute Inheritance

Attribute inheritance gives the object a powerful tool for specifying *default* attribute values. As a result, an object is responsible for specifying a small number of attributes.

Attribute-Changing Messages

Messages to a dialogue manager that change the attributes of display nodes allow an object to *dynamically* modify the values, formats, and other properties of display nodes.

2. What's Missing?

We now discuss some useful features missing from Dost and outline possible ways to include them.

2.1. More Attributes

Attributes are used by an object to specify various characteristics of a displayed variable: the format, the commands available to edit its presentation, the update methods, and so on. A dialogue manager uses the 'high level' description provided by these attributes to handle the 'low-level' details of user interaction.

One drawback with an approach using attributes is that a dialogue manager can support only a *limited* number of attributes. Therefore attributes can provide only limited flexibility in specifying user interfaces. For instance, currently the flexibility of our approach is limited by the lack of attributes to specify fonts and sizes of characters, spacing between lines on the screen, and *graphic presentations.*

However, our hope is that a finite but large set of attributes will be able to provide sufficient flexibility for most applications. Further research is needed to determine this set.

2.2. Displaying Objects

It would be useful if objects could display other objects in their presentations, as illustrated by the following example.

Consider the class 'Form', which allows a user to enter information about a person. Now assume that a programmer wishes to create a class that defines a variable declared as follows:

```
personDatabase: POINTER TO RECORD [
  contents: SEQUENCE size: CARDINAL OF CLASS Form]
```

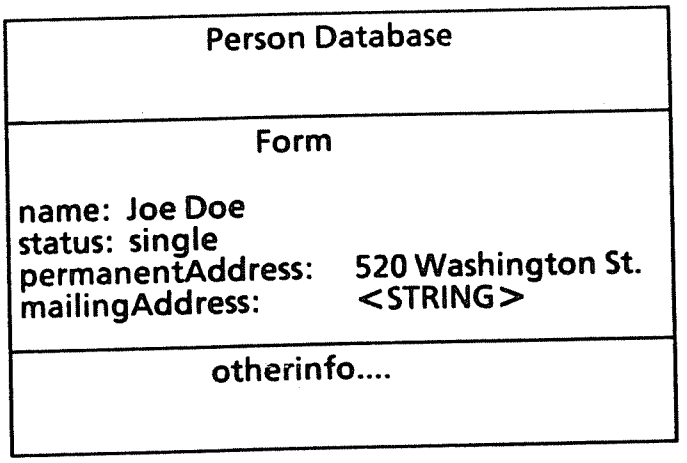
It would be useful if an instance of this class could display the elements of the sequence in its presentations. The class could reuse code that describes how instances of 'Form' are displayed and edited.

Currently, Dost does not allow an object to display the presentations of other objects in its presentations. Each object is displayed in a separate window, which is managed by a single dialogue manager. While a window may contain subwindows that show different presentations of an object, a window cannot be contained in another window.

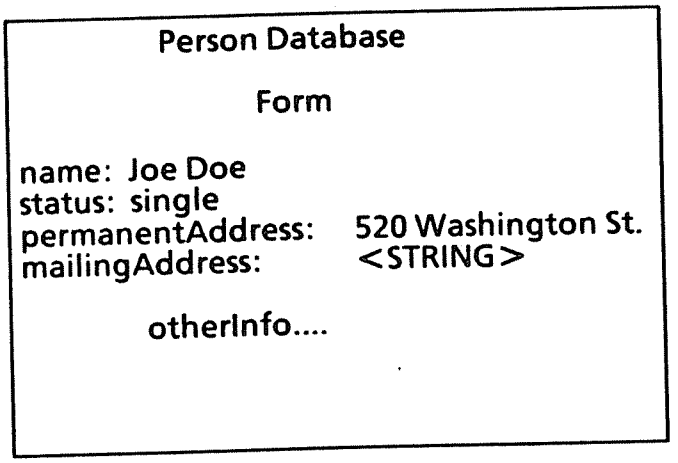
There are several possible extensions to Dost to allow objects to be displayed within other objects. The presentations of a subobject could be displayed in a separate window nested within the window of its parent, as illustrated by figure 1(a), or they could be displayed directly in the window of the parent, as shown in figure 1(b). Further research is needed to determine other extensions to support displaying of objects as parts of other objects.

2.3. More Commands

The generality of the Dost interaction model stems from the large number of editing commands provided by it. These include window editing, object editing, text editing, structure editing, and attribute editing commands. In the Dost implementation these commands number 30. Experience with Dost has shown that a typical object can use most of these commands. Moreover, it needs to provide very few object-specific commands.



a) Nested Windows



b) Single Window

Figure 1 : Displaying an Object in the Presentation of another Object

However, the commands provided by Dost are by no means an exhaustive set of commands. We outline below some of the commands currently missing in Dost.

Several editors provide a command to 'pick' an operand in a buffer, and insert it elsewhere [5]. Such a command would be useful in Dost to copy data from one display node to another. The two nodes could be in the presentations of the same or different objects. For instance a user could 'pick' a reference entry from the presentation of one instance of 'Bibliography' and 'put' it in the presentation of another instance.

The 'pick' and 'put' commands can be implemented in Dost by providing one or more central buffers to hold data 'picked' from presentations of objects. These buffers would be accessible to all the active dialogue managers and objects in the system. Thus objects can override these commands if they wish to.

Two other important commands missing in Dost are commands to *undo* and *redo* other operations. Currently, an object is responsible for implementing these commands. Further research, perhaps using the ideas presented in [47, 22, 5], is needed to define a general undo/redo facility in Dost.

Further work is also needed to determine other default commands that may be provided by a dialogue manager.

2.4. Text Editing of Structures

Dost allows presentations to be edited as both structures and text. The presentations of simple display nodes such as strings and integers are edited as text. The presentations of structure nodes such as records, arrays and sequences are edited as structures.

In some situations, it is useful to edit structures as text. For instance, consider the following declaration of a date:

```
Day: TYPE = CARDINAL [1..31];
Month: TYPE = {Jan, Feb, ..., Dec};
Year: TYPE = CARDINAL;
Date: TYPE = RECORD [
   day: Day,
   month: Month,
   year: CARDINAL];

day: Day
```

Assume that the user wishes to enter the date

1 Jan 1986

He may prefer replacing the placeholder

<Date>

with the string "1 Jan 1986" to replacing each of the placeholders

<Day> <Month> <Year>

with the values '1', 'Jan', and '1986' respectively.

Currently, an object may allow text editing of a structure by displaying it as a string and *parsing* the string to determine the values input (Chapter 3, example: 'StatementList'). Our hope is that that in most cases, the task of parsing input will be handled by standard library routines.

Nonetheless, it would be useful if a dialogue manager could relieve an object from the task of parsing input. For instance, it could create a grammar that describes the syntax of the presentation of a date, and use this grammar to parse input and return appropriate values to the object. It could use the help of a parser generator to generate

parsing and scanning tables.

A dialogue manager may not be able to parse *all* possible presentations (definable by different formatting attributes) due to limitations in the parser generator it uses. Moreover, it may have to disallow certain attribute changing commands on presentations it parses. If it builds all parsing and scanning tables at compile time, it cannot let a user or an object change attributes that change the syntax of the presentations it parses.

For similar reasons, a dialogue manager may have to disallow dynamic modification of display names of certain enumeration literals.

2.5. Automatic Saving of Attributes

A user may edit an object in several editing sessions, between which it important that user changes to the values and other attributes of display nodes be saved in permanent storage.

The 'ExtendedForm' example showed how the values of display nodes are saved. An object receives these values in parameters of update methods, and saves them in permanent storage when a user executes the 'save' method. It reloads these values in a presentation each time a user executes the 'reset' command.

Methods similar to the update methods could be defined in Dost to allow an object to save the values of other attributes of display nodes. For instance the attribute 'elided' could be associated with the 'elideUpdate' method which would be called with the value of the 'elided' attribute each time the user executes the 'elide' command. Thus the object could save and restore the values of all attributes of the display nodes edited by a user.

This approach of making an object responsible for saving its attributes is not very satisfactory. It burdens an object with the following tasks:

(1) *saving* attribute values into permanent storage each time the user executes the 'save' command

(2) *reading* attribute values from permanent storage each time the object is activated or the user executes the 'reset' command

(3) *resetting* the display with the saved attribute values each time the user executes the 'reset' command.

We outline below some some extensions that relieve an object from these tasks:

(1) The object could declare certain variables as *permanent*. The system could ensure that these data are automatically saved in permanent storage when an object is passivated and loaded into memory when the object in activated. Thus an object would not have to explicitly save and restore data from its data file.

(2) Each dialogue manager could save its state in permanent storage when a user executes the 'save' command. It could restores this state when the user executes the 'reset' command.

(3) A variable in an object would be allocated at the same virtual address each time the object is activated. Thus variable names of display nodes would remain constant over different activations of the object.

2.6. Specification of Attributes

Dost provides two ways to specify attributes: one for the *applications programmer* defining a class of objects, and another for a *user* interacting with a specific object.

The application programmer specifies attributes of attribute groups by writing *procedural* code to send attribute changing messages to a dialogue manager. The user specifies attributes of attribute groups *interactively* by using attribute editing commands.

It would be useful if an application programmer could also specify attributes of attribute groups interactively. The programmer could create a 'dummy' instance of a class and use the attribute editing commands to set default values of the formatting attributes of different attribute groups. He could experiment with different attribute values until the presentations are formatted to his satisfaction. He could then execute an accept command to 'freeze' these attribute values. The system would respond to this command by creating appropriate code that may be linked with the class to ensure that the attribute groups of all instances of the class are initialized with these default values.

It would also be useful if Dost allowed attributes to be specified declaratively. A declarative specification of attributes would allow definition of *initial* or *constant* values of attributes. The following is an example of a declarative specification to set the 'alignment' attribute of 'Reference.author' to the constant value 'horizontal':

```
Reference: RECORD [
  author: STRING ATTRIBUTES alignment = horizontal END,
  ...];
```

Further research is needed to determine how procedural, declarative, and interactive specification of attributes may be integrated with each other.

2.7. Updating different Presentations

In Dost, a variable in an object may be displayed by several display nodes managed by the same or different dialogue managers. It is the object's responsibility to ensure that the changes to one display node of a variable are reflected in other nodes displaying the variable. It would be useful if the dialogue managers of an object could communicate with each other to directly to keep related nodes constant.

3. Experience

We have used Dost to define the class 'Bibliography' described in chapter 2, the classes 'Form', 'ExtendedForm', and other classes described in chapter 3, and the classes 'References' and 'ASPLE' described below.

The class 'References' is an extension of 'Bibliography'. It defines more realistic fields for a reference entry. Moreover, its instances can send entries to other objects. Thus entries of common interest can be exchanged by the bibliography databases of different users.

The class 'ASPLE' checks the static semantics of the programming language ASPLE, which has been used to compare several formal mechanisms for defining the semantics of programming languages [23]. An ASPLE program is composed of a declaration list and a statement list. The data types of the language are integers, booleans, and pointers to booleans. The statements include **if**, **while**, **assignment**, **input**, and **output** statements, as described in chapter 3. Figure 17 in chapter 2 displays presentations of instances of 'References' and 'Bibliography'.

Figure 2 shows the sizes of the *interaction code* of these classes. The interaction code of a class, intuitively, is the code required to 'drive' the dialogue managers of an

Bibliography	6
Form	8
ExtendedForm	13
RoommateExpenses*	30
Spreadsheet*	14
StatementList	18
Directory*	39
References	16
ASPLE	31

Figure 2: Size (in source lines) of the Interaction Code in Different Classes

instance. It includes any code in the class that uses the input/output primitives described in chapter 2. Thus, it includes the code required to specify the attributes of attribute groups, submit variables for editing, specify update and command methods, specify 'save', 'reset', and 'empty' methods, and specify display names. It does not include the code required to define the data encapsulated by an instance, or the code required to manipulate them.

All classes except 'RoommateExpenses', 'Spreadsheet', and 'Directory', were actually implemented. These three classes were not implemented because of limitations of the current implementation, discussed in chapter 2 §6.

Two preliminary conclusions can be drawn from our experience:

(1) Dost objects are *easy to use*. While this is a highly subjective opinion, it appears to be the consensus of those who have seen demonstrations of Dost. Our implementation on a workstation also shows negligible response times to user commands, an important concern in determining the usability of an interface.

(2) Dost classes are *easy to implement*. The size of the interaction code a class is insignificant compared to the size of the dialogue manager code it drives. In our implementation, the latter is about 7000 Mesa lines. This number does not include the thousands of lines of code in the XDE libraries used in the implementation.

4. Related Work

The idea of generating user interfaces is not new. Previous approaches range in scope from form development systems to environments that support the editing model of interaction. In this section we compare our work with these approaches.

4.1. Form Development Systems

Several systems support a model of interaction based on *forms* [21, 35, 37, 36]. An application interacts with a user by displaying one or more forms to the user. Each form consists of one or more *items*, which a user fills with appropriate values.

There is an important similarity between the form model of interaction and editing model supported by Dost: A user can fill the items in any order. Thus interaction is not constrained to be sequential.

However, the form model is not as general as the editing model. It imposes three main restrictions:

(1) The number of items in a form is *static* and not a function of input.

(2) The values of items are restricted to simple types. Thus none of the structure editing commands are available to interact with forms.

(3) An application does not receive *incremental updates* to the items on the screen. It receives the values of all items together when the user executes the equivalent of an 'accept' command.

As a consequence of these restrictions, the model is unsuitable to interact with a large number of applications. For instance, it cannot be used to define any of the interfaces discussed in chapter 3. Typically, the use of forms is restricted to entering tuples in databases or setting initial parameters of an application.

4.2. EZ

Fraser and Hanson have built a software system called EZ [9, 10] based on a high-level string-processing language derived from SNOBOL4, SL5 [16], and Icon [15]. The language supports four basic types of values: numerics, strings, procedures, and SNOBOL4-like tables. The system has 'persistent' memory much like an APL workspace in which values exist until changed. As a result it integrates the traditionally distinct facilities of programming languages and operating systems into a single system. Files are represented as strings, and directories are provided as tables.

EZ provides a screen editor that edits all EZ values using the same interface. As a result it can be used to edit text files, directories, and relational data bases represented as tables. Moreover, procedure activations in EZ are just EZ tables. Therefore the editor is automatically a debugger as well.

There are two striking similarities between EZ and Dost. First, EZ's 'persistent' data corresponds to Dost's 'persistent' objects. Second, the EZ editor is similar to a Dost dialogue manager: Both are capable of editing instances of the types defined by a programming language.

There are, however, several important differences between Dost and EZ. First, the EZ editor and a Dost dialogue manager manage different data structures because of differences in the programming languages supported by them. The EZ editor manages only two types: strings, and tables. (Procedures and numerics are automatically converted to strings by the language.) A Dost dialogue manager, on the other hand, manages numerics, strings, enumerations, arrays, records, variant records and other data structures defined by Mesa types.

Second, the editing commands presented by the EZ editor and a dialogue manager are different. The EZ editor provides only *text editing* commands. Thus it does not provide equivalents of the structure editing and attribute editing commands provided by a dialogue manager. On the other hand, it provides an 'enter' command, which may be applied to a line displaying a key of a table. The command recursively invokes the editor on the value associated with the key. The value may be another table, so the editor can be used to 'walk' tables. A dialogue manager, currently, does not provide an equivalent of the 'enter' command.

Third, an EZ editor does not provide an application programmer facilities to format data. Fourth, it cannot be used to check user input for semantic consistency. Fifth, and lastly, the EZ editor can be used as a debugger since EZ stores procedure activations as tables. An equivalent facility is not provided in Dost.

4.3. Descartes

Descartes [39, 40] is a framework for building user interfaces having several characteristics in common with Dost. It allows an application to input and output values of programmer-defined types. Moreover, an application's interaction with the user is managed by an application-specific module called a *compositor*.

The compositors in Descartes corresponds to dialogue managers in Dost. There are two main differences between the two. First, a dialogue manager is provided *automatically* for each Dost object and is 'driven' by a small amount of application-specific code. On the other hand, a compositor has to be developed *manually* for each Descartes application using the *utility code* shared by all interfaces and provided by the system. Thus, Descartes provides less automation in generating a user interface, but gives an application programmer more flexibility in specifying an interface.

Second, Descartes supports only sequential interaction. An application asks for values it needs in a particular order and the user is constrained to supply each value as it is requested. Dost, on the other hand, allow the variables displayed in a presentation to be edited in any order.

Finally, Descartes supports a tight coupling between the variables in an application and their display: Every assignment to a variable causes its display to be updated. Dost provides a looser coupling which requires that an application explicitly update a display in response to changes in its variables.

4.4. Language-Oriented Editor Generators

Dost is closely related to the Synthesizer Generator [34], POE [5], ALOE [24, 29], sds [8], PECAN [32], PSG [2] and other language-oriented editor (LOE) generators. An LOE generator provides a *specification language*, which may be used to define the syntax and semantics of a *target language*. Traditionally, LOEs have been used to edit programs written in conventional programming languages. However, they have also been used to edit other structures such as documents, a desk calculator, and the specification language itself.

There are several similarities between Dost and LOE generators. The target language description used by an LOE generator corresponds to a Dost class, and the LOE generator corresponds to a dialogue manager. There are also several important differences between the approaches used by Dost and LOE generators. These are outlined below.

First, current LOE generators are *special programs* in a traditional software development environment. As a result, the environment consists of a set of *standard programs* that manipulate unstructured text, and a set of *editors* that manipulate syntax trees. Dost on the other hand is an *environment* in which *all* programs offer editing interfaces.

Second, the language for describing Dost classes is an extension of a *conventional general-purpose* programming language. On the other hand the specification language of a LOE generator is based on BNF grammar descriptions embellished with constructs for describing semantics such as action routines [24], attributes [34], attributes and action equations [18, 19], and denotational definitions [2, 42, 31]. These descriptions have been used mainly for describing *programming languages.*

Third, Dost allows an object and a dialogue manager to be two *separate* programs communicating with each other through messages. Thus, an object and a dialogue manager can reside on different machines, such as a workstation and a mainframe host. Moreover, an object can be connected simultaneously to several dialogue managers. As a result, several users can edit the object at the same time. LOE generators, on the other hand, do not provide an equivalent of the object-dialogue manager separation.

Fourth, since Dost classes are extensions of Mesa programs, they can use Mesa constructs for sharing declarations. The specification languages of current LOE generators do not provide similar constructs for sharing code.

Fifth, Dost objects can *share data* by exchanging messages. As a result, related objects can be kept consistent with respect to each other. An equivalent facility to share information is not provided by LOE generators.

Finally, LOE generators based on attribute grammars [34, 33, 5, 17] allow a programmer to specify the semantics of user interaction *declaratively*. This feature is useful for specifying the *static semantics* of a target language, since it relieves a programmer from the task of *explicitly* calling procedures that check related values for semantic consistency. Dost is based on a procedural language, and therefore, does not provide an equivalent facility to specify semantics declaratively.

4.5. AGAVE

Recently, Notkin [28, 30] has proposed an environment called AGAVE that replaces standard programs with editor modules written in a language based on the ALOE specification language. He has augmented the ALOE specification language with primitives that allow sharing of code between different modules. He has also proposed capability-based addressing to allow sharing of data between different syntax trees.

Dost and AGAVE are both environments in which all interaction is through structure editor interfaces. However, there are two significant differences between the two systems, which stem from the fact that AGAVE is derived from ALOE. First, AGAVE offers a grammar-based specification language for describing editors, while

Dost offers a Mesa-based programming language.

Second, AGAVE replaces a traditional operating system kernel with an editor kernel, which implements the editing interfaces. The kernel is (dynamically) linked to all the editor modules in the system and acts as the controlling module of a monolithic program. Dost, on the other hand distributes control over all the objects and dialogue managers in the system. As a result, AGAVE can edit only one syntax tree at a time, while Dost allows multiple objects to be edited simultaneously. Moreover, AGAVE cannot offer the advantages of object-dialogue manager separation described in the previous section. Finally, the AGAVE kernel is responsible for activating and passivating the syntax trees in the system. Dost, on the other hand, makes each object responsible for activating and passivating its data.

4.6. Voodoo

Voodoo [38] is a framework that supports the generation of editing interfaces in an object-oriented system. It divides the objects in the system into *emenands*, *images*, and *editors*. Each emenand is associated with with one or more images and one or more editors. An image consists of an abstract syntax tree, which describes the *external* structure of the emenand. The image is used by an editor to allow the user to interact with the emenand.

Both Dost and Voodoo (which were developed independently and contemporaneously) support editor-oriented interaction in an object-oriented system. A dialogue manager in Dost corresponds to an editor in Voodoo. There are two main differences between the two systems. First, a dialogue manager in Dost is created *automatically*, while an editor in Voodoo is created *manually* using the primitives for *inheritance* offered by the host object-oriented system. Second, an emenand in Voodoo is

associated with both an internal structure and an external structure. As a result, an emenand's internal structure can be changed without affecting the user's view of the object. However, the implementor of a new application has to be concerned with creating two structures, and keeping them consistent. In Dost, only one structure is created.

5. Conclusions

Our work presents an approach to automatic generation of user interfaces and demonstrates the usefulness of this approach through examples, discussion, and implementation of its major components.

In comparison to previous approaches, our approach

- supports the editing model of interaction,
- is based on a conventional procedural programming language,
- allows a user to interact with several applications at the same time,
- offers the advantages of object-dialogue manager separation, and
- allows an implementor to provide a single description of displayed information.

6. Future Work

To date, Dost has been tested by only one user, its author. More experience with programmers and users unfamiliar with its working is needed to prove that it may be used to create interfaces that are both easy to use and easy to implement.

Section 1.1.2 discussed several applications that can fruitfully present editing interfaces. These need to be implemented to show the generality of our approach. More creative ways of using editing interfaces also need to be explored.

Section 2 discussed several current limitations of Dost, and possible extensions to overcome them. These and other extensions need to be explored in depth. In particular it needs to be researched if attributes to display graphical presentations of variables can be included in our approach.

Our approach forces a programmer to specify semantics procedurally. Ways to support declarative specification of semantics need to be explored, perhaps using the ideas presented in [21, 34, 33, 5, 17].

Notwithstanding the current limitations of Dost, our work has presented an approach that can be used as a basis for future research in automatic generation of user interfaces.

REFERENCES

[1] G.T. Almes and C. Holman, "Edmas: An Object-Oriented, Locally Distributed Mail System," Technical Report 84-80-03, Department of Computer Science, University of Washington, August 1984.

[2] Rolf Bahlke and Gregor Snelting, "The PSG - Programming System Generator," *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Languages*, June 1985, pp. 28-33.

[3] Prasun Dewan and Marvin Solomon, "An Approach to Generalized Editing," *Proceedings of the IEEE Ist International Conference on Computer Workstations*, November 1985, pp. 52-60.

[4] Prasun Dewan and Marvin Solomon, "Dost: An Environment to Support Automatic Generation of User Interfaces," *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, to appear in December 1986.

[5] C. N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock, "The POE Language-Based Editor Project," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 21-29.

[6] Christopher W. Fraser, "A Generalized Text Editor," *CACM* 23:3 (March 1980), pp. 154-158.

[7] Christopher W. Fraser and A. A. Lopez, "Editing Data Structures," *ACM Transactions on Programming Languages and Systems* 3:2 (April 1981), pp. 115-125.

[8] Christopher W. Fraser, "Syntax Directed Editing of General Data Structures," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16:6 (June 1981).

[9] C.W. Fraser and D.R. Hanson, "A High-Level Programming and Command Language," *Sigplan Notices : Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems* 18:6 (June 1983), pp. 212-219.

[10] C.W. Fraser and D.R. Hanson, "High-Level Language Facilities for Low-Level Services," *Conference Record of POPL*, 1984, pp. 217-224.

[11] David B. Garlan and Philip L. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 65-72.

[12] Adele Goldberg and David Robinson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[13] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

[14] Michael Good, "Etude and the Folkore of User Interface Design," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16:6 (June 1981), pp. 34-43.

[15] R.E. Griswold and M.T. Griswold, *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[16] D.R. Hanson and R.E. Griswold, "The SL5 Procedure Mechanism," *Comm. ACM*, May 1978, pp. 392-400.

[17] Gregory F. Johnson, "An Approach to Incremental Semantics," Ph.D. Thesis, University of Wisconsin - Madison, August 1983.

[18] Gail E. Kaiser, "Semantics for Structure Editing Environments," PhD Thesis and Tech. Report, CMU-CS-85-131, Department of Computer Science, Carnegie-Mellon University, May 1985.

[19] Gail E. Kaiser, "Generation of Run-Time Environments," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986, pp. 51-57.

[20] G. D. Kimura, "A Structure Editor for Abstract Document Objects," *IEEE Transactions on Software Engineering* 12:3 (March 1986), pp. 417-436.

[21] J.M. Lafuente and D. Gries, "Language Facilities for Programming User-Computer Dialogues," *IBM J. Res. Develop.* 22:2 (March 1978), pp. 145-158.

[22] George B. Leeman, Jr., "A Formal Approach to Undo Operations in Programming Languages," *ACM Transactions on Programming Languages and Systems* 8:1 (January 1986), pp. 50-87.

[23] Michael Marcotty, Henry F. Ledgard, and Gregor V. Bochmann, "A Sampler of Formal Definitions," *Computing Surveys* 8:2 (June 1976), pp. 194-275.

[24] Raul Medina-Mora, "Syntax-Directed Editing: Towards Integrated Programming Environments," PhD Thesis, Department of Computer Science, Carnegie-Mellon University, March 1982.

[25] Norman Meyrowitz and Andries van Dam, "Interactive Editing Systems: Part 1," *Computing Surveys* 14:3 (September 1982), pp. 321-352.

[26] Norman Meyrowitz and Andries van Dam, "Interactive Editing Systems: Part 2," *Computing Surveys* 14:3 (September 1982), pp. 353-416.

[27] Brad A. Myers, "Displaying Data Structures for Interactive Debugging," Technical Report CSL-80-7, Xerox Corporation, Palo Alto Research Centers, June 1980.

[28] David Notkin, "Interactive Structure-Oriented Computing," PhD Thesis and Technical Report, CMU-CS-84-103, Department of Computer Science, Carnegie-Mellon University, February 1984.

[29] David Notkin, "The Gandalf Project," *The Journal of Systems and Software* 5:2 (April 1985).

[30] David Notkin, "Sharing and Modularization in Structure Editing Environments," *Proceedings of the 19th Hawaii International Conference on Systems Sciences*, January 1986.

[31] Anil Pal, "Generating Execution Facilities for Integrated Programming Languages," Ph.D. Thesis (in preparation), University of Wisconsin-Madison, December 1986.

[32] S. P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering* SE-11:3 (March 1985).

[33] Thomas Reps, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Transactions on Programming Languages and Systems* 5:3 (July 1983), pp. 440-477.

[34] Thomas Reps and Tim Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 42-48.

[35] Lawrence A. Rowe and Kurt A. Shoens, "A Form Application Development System," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1982, pp. 28-38.

[36] L. A. Rowe and K. A. Shoens, "Programming Language Constructs for Screen Definition," *IEEE Transactions on Software Engineering* SE-9:1 (January 1983), pp. 31-39.

[37] Lawrence A. Rowe, "'Fill-in-the-Form' Programming," *Proceedings of VLDB*, 1985, pp. 394-404.

[38] Jeffrey Scofield, "Editing as a Paradigm for User Interaction," Ph.D. Thesis and Technical Report No. 85-08-10, University of Washington, Department of Computer Science, August 1985.

[39] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A Programming-Language Approach to Interactive Display Interfaces," *Sigplan Notices : Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems* 18:6 (June 1983), pp. 100-111.

[40] M. Shaw, "An Input-Output Model for Interactive Systems," *CHI'86 Proceedings*, April 1986, pp. 261-273.

[41] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Eric Halsem, "Designing the Star User Interface," *BYTE* 7:4 (April 1982).

[42] Gregor Snelting, "Unification in Many-Sorted Algebras as a Device for Incremental Semantic Analysis," *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, January 1986, pp. 229-235.

[43] J. Sutton and R. Sprague, "A Study of Display Generation and Management in Interactive Business Applications," Tech. Rept. RJ2392(#31804), IBM San Jose Research Laboratory, November 1978.

[44] Tim Teitelbaum, Thomas Reps, and Susan Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer," *Sigplan Notices* 16:6 (August 1981).

[45] W. Teitelman, "A Tour through Cedar," *IEEE Transactions on Software Engineering* SE-11:3 (March 1985).

[46] Larry Tesler, "The Smalltalk Environment," *Byte* 6:8 (August 81), pp. 90-146.

[47] Jeffrey Scott Vitter, "USER: A New Framework for Redoing," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 168-176.

[48] Xerox Corporation, "Xerox Development Environment: Concepts and Principles," XDE3.0-1001, November 1984.

[49] Xerox Corporation, "Mesa Language Manual," XDE3.0-3001, November 1984.

[50] Marvin V. Zelkowitz, "A Small Contribution to Editing with a Syntax Directed Editor," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 1-6.

[51] Marvin V. Zelkowitz, Jennifer Elgot, David Itkin, Bonnie Kowalchack, and Michael Maggio, "The Engineering of an Environment on Small Machines," *Proceedings of the IEEE Ist International Conference on Computer Workstations*, November 1985, pp. 61-69.