TASK: UR20
CDRL: 01030
9 July 1990

(2)

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Ada/Xt Toolkit
Interface Style Guide

STARS-RC-01030/001/00
Publication No. GR-7670-1150(NP)
9 July 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UR20
CDRL: 01030
9 July 1990

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Ada/Xt Toolkit
Interface Style Guide

STARS-RC-01030/001/00
Publication No. GR-7670-1150(NP)
9 July 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

Contents

List of Figures

List of Tables

1 Introduction

1.1 Scope

This Style Guide is an initial attempt to define conventions for the development of effective human-application interfaces (i.e., "User Interface"). Its purpose is to facilitate experimentation with Ada/Xt[1], and to foster the use of Ada/Xt for the development of friendly, effective application interfaces.

This report is targeted to application developers who will be using the Unisys Ada/Xt toolkit implementation. Because of the constraints on style specification imposed by any underlying user interface component system, this report should be considered a companion to the June, 1990 release of the Unisys STARS Ada/Xt implementation[2].

1.2 Organization of this Report

A definition of "style" in Webster's 7th Collegiate Dictionary provides motivation for the structure of this report:

> 1. style ['sti-(*)] n ... 2b: manner or tone assumed in *discourse* 2c: the custom or plan followed in spelling, capitalization, punctuation, and typographic arrangement and display 4a1: manner or method of acting or performing esp. as sanctioned by some *standard.*

This definition addresses the three domains of "style" targeted by this report. Section 2 addresses styles of *dialogue* (discourse). Section 3 addresses conventions useful in developing toolkit-level applications, and in extending the toolkit to support development of user interfaces in new application areas (conventions). Section 4 addresses the interplay of *style* specification standards, both formal and informal.

Additionally, within this report various style recommendations are made. These recommendations are highlighted in italicized text preceded by the keyword "Recommendation." For example:

Recommendation: Comply with all style recommendations.

The concluding section of this report will summarize all style recommendations made in this report.

[1] Ada/Xt is the Unisys/STARS Ada implementation of the MIT X Toolkit
[2] Ada/Xt version 2.0

1.3 Human Factors Issues

The concept of "style" is broad, and encompasses the full range of user interface development activities, from preliminary design to coding. One goal of a style guide could be specification of guidelines for the construction of user interfaces which support some well-defined notions of interface effectiveness (determined empirically from human factors studies).

However, this report claims no special expertise in human factors. Also, such a guidelines document would require in-place programming support for the specification. For example, both [6, 2] style guides refer to extensive, in-place software component support abstractions. At this stage of STARS User Interface component development, an insufficient number of toolkit abstractions ("widgets") are available to support a fully-defined, well-conceived, human factors-oriented style guide.

Recommendation: Adopt the MOTIF Style Specification

Rather than invent a new style specification, it is recommended that STARS draw upon evolving industry consensus style guides (if not adopt one outright). For example, the Open Software Foundation (OSF) Motif Style Guide [6] provides an excellent description of a mature look-and-feel specification. Such a specification can be supported by non-Motif widgets. The human-factors basis of well conceived style specifications such as [6] will provide a sound foundation even if STARS should find that domain-tailored environments require different interface guidelines.

1.4 Related STARS Efforts

The Boeing STARS effort ("BR21") has worked on defining a higher-level programming interface definition based upon the concept of *user interaction tasks* as opposed to *user interaction components*. User interface tasks refer to classes of specific interactions, e.g., "object selection", while user interface components refer to the concrete realization (look and feel) of such interactions. Without necessarily agreeing with all the results of the Boeing BR21 task, specifically the report [1], it appears that such a distinction is well motivated.

This style guide can be considered a companion to [1], being focused more at the component level than the interaction task level. This focus seems justified since the underlying components which support interaction tasks to a large extent constrain and define the *style* of the dialogue interaction.

2 Styles of Interaction: Dialogue Models

Although user interface construction has been a differentiated part of commercial software development practices for many years, the advent of affordable, bitmapped workstations has led to fundamental changes in end-user expectations concerning the quality and kind of the

human-application interface. The most fundamental change concerns the widespread use of so-called *modeless, direct manipulation* interfaces. These concepts (modality, directness) are closely related and somewhat overlapping. In fact, both concepts are part of the larger concept of *human-computer interaction*, or *dialogue*. An extended discussion of dialogue models and dialogue design tools and techniques can be found in [3].

Section 2.1 discusses the notion of *dialogue modality*. Section 2.2 discusses *dialogue sequentiality*. Section 2.3 discusses a specialized style of direct, modeless interaction known as *object-action selection*. Section 2.4 discusses Ada/Xt toolkit support for various models of human-computer dialogue.

2.1 Modal vs. Modeless

An application interface is considered *modal* if its model of interaction can be described as a state machine with more than one state, where each state describes a set of possible interactions which are differentiated from other application states. For example, the popular UNIX editor "vi" is a modal editor, consisting of at least three states: an insert state, a keyboard state, and a command-line state.

Modes are application states which constrain the permissible choices of human-computer interaction. The modern trend is away from modal interfaces, or at least to minimize the degree to which the human-computer interaction is modal. Modal interfaces should be discouraged because modes generally imply some context sensitivity, and greater reliance on modes imposes more requirements on the end-user to keep track of the current context of the application dialogue. Even worse, in some situations the end-user will need to anticipate and plan navigation through several application modes in order to enter into some desirable state. This sort of end-user navigation can be tedious in applications with many modes.

In practice, no interface is completely modeless. In some cases it is desirable or necessary to impose modality. Modal interactions are highly useful for focusing user attention on critical, narrow tasks, e.g., prompting the end-user whether data changes should be saved prior to application termination. Even within otherwise modeless interfaces, commonplace user interface abstractions impose modality. For example, pop-up menus constitute a kind of modal interaction, since some forms of pop-up menu focus user input to either the menu contents, or else require that the menu be "popped-down" before other human-application interactions can occur.

2.2 Direct vs. Sequential

Closely related to modality is sequentiality. Sequential dialogues are definable as those dialogues that require a strictly linear sequence of human-application interactions. One common example is found in automatic teller machines. In general, tools and applications which have a primarily linguistic interface (e.g., command-line processors, batch-oriented tools) tend to be sequential. In contrast, direct dialogues are those in which the application is a passive

entity, and the end-user engages in a potentially non-linear sequence of user-application interactions, freely interleaving interactions that are germane to potentially different tasks. A common example of direct dialogue can be found in many interactive drawing tools for bitmapped workstation environments. Direct dialogues imply a style of programming known as "event-driven" programming, which is a paradigm well supported by the Ada/Xt toolkit.

The modern trend is towards direct dialogues wherever possible. The goal of direct dialogue applications is to "empower the user" [6], to make the application subservient to the user's demands, and not the other way around. There are obvious similarities between the concept of modeless interface and direct dialogue, and in fact one generally implies the other. However, it is conceivable to have direct dialogue interfaces (also called direct manipulation interfaces) modified by application mode. For example, in a hypothetical graphical drawing tool, it is conceivable that various operations would be disabled if an existing drawing were being viewed in "read-only" mode.

2.3 Object-Action Selection

Both [6, 1] discuss a style of interaction known as "object-action selection" interaction. Essentially, this refers to a style of dialogue in which the application user *selects* an object which is to be the target or focus of some action, and then *selects* the appropriate action. For example, an iconic representation of a file could be *selected* (e.g., with the left mouse button), and then an action appropriate for this object type can be *selected* from a pull-down menu (e.g., print the file).

Note that toolkit support for object selection generally takes the form of default translation tables supplied with various "selectable" widgets. For example, the Athena command widget by default supports selection with a left mouse button-down event. By exporting these translation tables as widget resources, an application can allow end-users the luxury of tailoring the application to respond to different input events as selection events. On the other hand, such flexibility can result in widely dissimilar personal working environments.

Recommendation: Applications should implement object-action selection.

Figures 1 and 2 provide illustrations of two interactive direct manipulation drawing tools which support different styles of object-action selection. Idraw (figure 1) directly implements the object-action selection; in this figure, the upper rectangle has been selected by a left mouse button down event, and its selection is indicated by the grab-boxes at the rectangle corners.

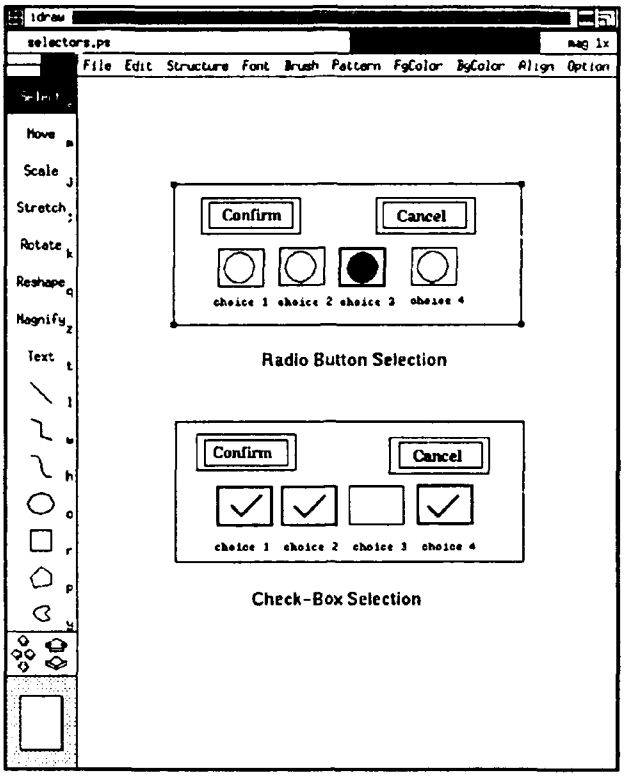

Figure 1: IDraw Drawing Tool

Recommendation: Provide visual feedback for all user interactions.

With this type of object selected, various operations located in the pull-down menus located across the top-most horizontal bar of the application can be executed. [3]

Xfig (figure 2) supports object selection differently. Instead of directly selecting objects, various operations introduce a mode which affects the display and semantics of mouse interaction with on-screen objects. For example, as depicted in the figure, the delete-object action has the effect of selecting all of the objects on the display (indicated by grab-boxes on all visible objects); the user then selects (with the left mouse button) individual objects for deletion. In effect, the delete operation puts the application into an "object selection" mode.

Another interesting distinction between these interfaces is their means of action-selection. Idraw chooses pull-down menus, which represents a compromise between screen real-estate and burden on end-user memory. Xfig chooses an iconic representation for all operations. This choice makes the Xfig application more unwieldy on the workstation display, but provides continuous cues to application functionality. On the other hand, it is not as natural to introduce keyboard accelerators in the Xfig model of action-selection as is the case with Idraw. Note however that it sometimes difficult to devise meaningful iconic representations for particular operations: do all the icons in figure 2 have an obvious meaning to the reader?

[3] Unfortunately, this application does not *dim* operations and menus not pertinent to the selected object.

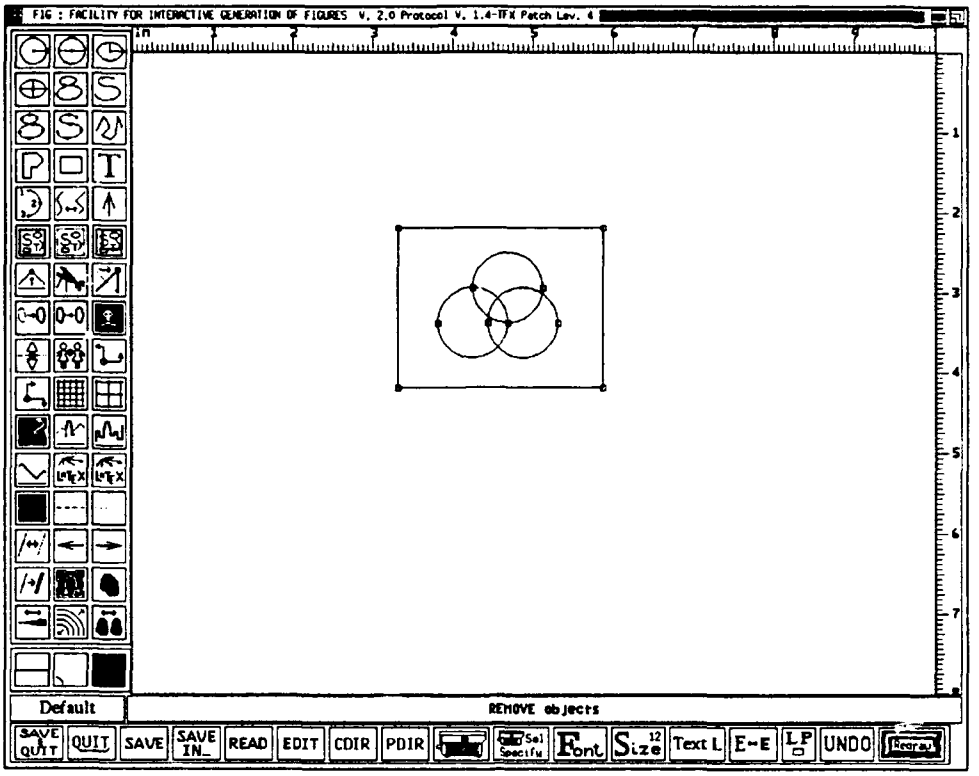

Figure 2: XFig Drawing Tool

It is impossible to say which application has a better interface, which is really the heart of the matter when considering specification of user interface style guides. In short, style guides must be rigorous enough to enforce some similarity among different applications, or different applications within the same class (e.g., drawing tools); style guides must also be flexible enough to support expression of new ideas in human-tool interaction.

2.4 Toolkit Component Support for Dialogue

As should be apparent, large, real-world applications tend not to be exclusively modeless, or exclusively direct-manipulation. In practice, some mixture of these dialogue models are present, and it is important to know what the consequences of different dialogue styles are in order to assess the impact of interface design decisions. For example, menus provide a commonly used technique for implementing action selection. Different forms of menus impose unique characteristics on the overall interface: pop-up menus conserve application "real-estate" since they take up no room on the display until some user event triggers the pop-up action, but provide no visual cues about their availability or meaning (in contrast to pull-down menus). Also, pop-ups tend to be modal, requiring either selection of an option from the pop-up menu, or else abandoning the menu-dialogue without selection; no interleaving of menu/non-menu activities is typically permitted.

The application developer needs to be aware of the toolkit component-level support provided

Menu Kind	Modeless	Modal	Direct	Sequential
Pop-Up	Stay-put, no focus	focus	all forms	cascading
Pull-Down	Tear-away, no focus	focus	all forms	cascading

Table 1: Menu Forms

for implementing various forms of user interface task in order to make reasonable tradeoff decisions. As a result, the quality and maturity of the underlying toolkit defines, to a large extent, the quality of dialogues supported by the application. At the time this report is being written, the Unisys Ada/Xt toolkit implementation requires an enhancement of its current set of user interface components ("widgets") for various important forms of dialogue. Two important classes of widget described in [6] are not yet implemented as Ada/Xt widgets: menu selectors, and radio-box and check-box controllers.

The following section shows how these forms of dialogue are implementable in the Ada/Xt toolkit for a narrow class of user interaction tasks.

2.4.1 Menu Selectors

The most commonly used action selection device is the menu. Menus come in a variety of forms, each tailored for a particular kind of dialogue style. The Motif style guide, for example, devotes twenty one pages to menu interactors [4].

For the puposes of this report, two basic kinds of menus are defined: pop-up and pull-down menus. Pop-up menus are invisible until some user action causes them to be "popped"-up into visibility; some other user action will cause the menu to be popped-down. Pull-down menus, in contrast, are always visible (or at least their titles are always visible). Thus, these menus provide visual cues to the end-user, at the cost of some application real-estate.

Ada/Xt Toolkit menus always implement a direct form of dialogue; various forms can in addition provide modal, modeless, and (partial) sequential dialogues. Table 1 provides a cross section of menu types and dialogue styles. In table 1 "focus" refers to a toolkit feature which allows the application to cause all input events occurring outside of the *focus* widget to be discarded. Thus, to create a pop-up, modeless menu, an application would need to select a stay-put, pop-up, non-focused variety of menu. The pull-down variety of modelness menu could be implemented as tear-away menus. In the table, we designate cascading menus as menus which support a mixed-mode of direct and sequential dialogue; cascading menus provide hierarchies of menus which refine selection choices as the mouse "walks" through the selection list.

Although the current release of Ada/Xt does not provide menu widgets, the effect can be achieved via composition of existing widgets, and use of various intrinsics calls, as illustrated

[4]Interactors are dialogue implementation devices, such as toolkit widgets.

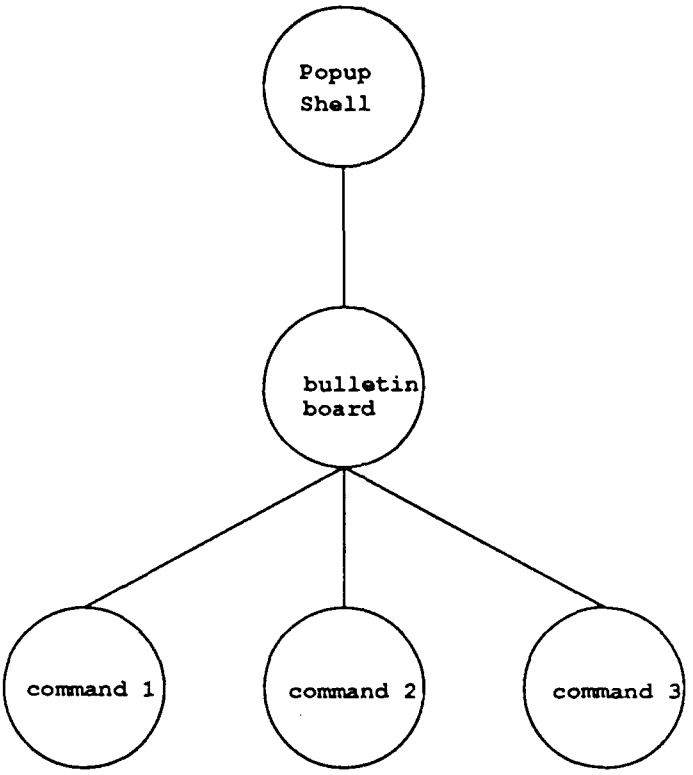

Figure 3: Widget Hierarchy for Menus

below in figure 3. Since the construction of menus is a straightforward process, it might be questioned whether widgets should be constructed, or whether convenience routines would be a better choice. In fact, Ada/Xt widgets should be constructed for STARS in order to enforce a common look-and-feel. Figure 4 provides a good indication of the complexity of even simple dialogue interactors, such as menus.

Figure 3 depicts a simple widget hierarchy which can be used to implement pop-up and pull-down menus. The predefined intrinsics widget popup_shell is the parent widget; the HP widget bulletin_board widget is used as an aggregation mechanism, and the Athena command widget is used as the selectable menu items. Since the command widget is a subclass of the Athena label widget, it can support both textual and iconic labels. The translations defined on command widgets are also resources, so the default interaction behaviors can also be tailored by either the application or end-user (if the application chooses to make such tailoring available).

To pop-up or pull-down menus, the application calls the intrinsics operation *xt_popup* with the parent popup shell widget as one argument, and the grab kind as the second argument. The grab kind initiates various forms of event focusing to support modal or modeless menu interactions.

Note that this widget hierarchy does not address issues such as menu-dimming, command grouping, keyboard equivalents, or cascading menus. It is these kinds of subtle features that

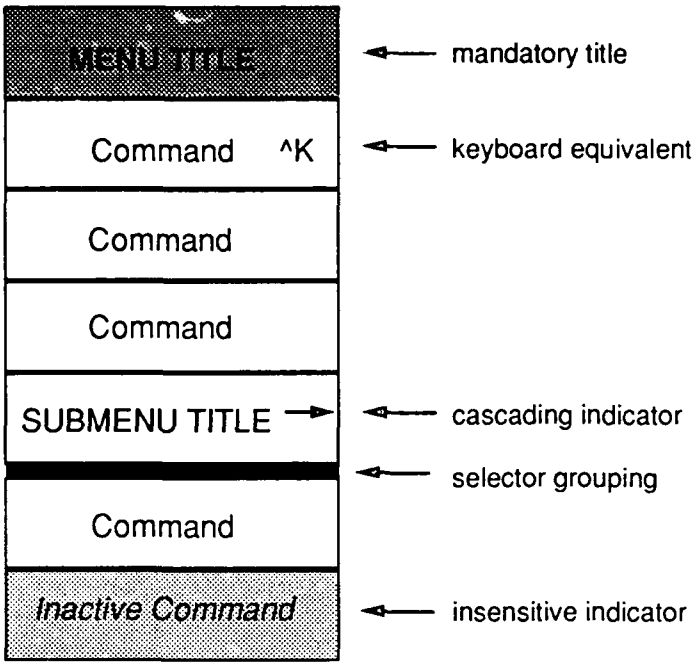

Figure 4: Menu Subparts

argue for the construction of a family of menu widgets.

Recommendation: Build a rich family of menu widgets.

Recommendation: In the interim, model menus after figure 4.

2.4.2 Radio-Button, Check-Box Selectors

While menus are sufficient for implementing dialogues for tasks requiring single item selection, they are not particularly useful for interactions requiring multiple selections. For example, selections requiring multiple choice, where the choices can be combined or are mutually exclusive, are not naturally expressed in menu form. For this purpose, a more elaborate aggregation of widgets is required. The current Ada/Xt release provides the Athena dialogue widget; however, this widget is really not sufficient for more than very primitive forms of multiple selection dialogue.

Radio-button dialogues[5] can be implemented in Ada/Xt similarly to menus: as a sequence of command buttons inserted into a bulletin board whose parent is a pop-up shell. The semantics of radio buttons is that selections are mutually exclusive. Check-box dialogues can also be implemented in Ada/Xt in a similar way, with the semantics that selections are not mutually exclusive. Figure 5 provides a sample depiction of radio-button, and figure 6 provides a sample depiction of a check-box dialogue. Note the visual cues provided for

[5]Radio and Checkboxes are called "Controls" in Motif.

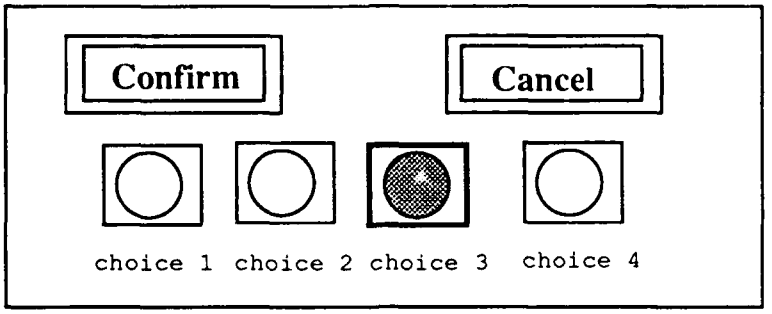

Figure 5: Radio Button Selection

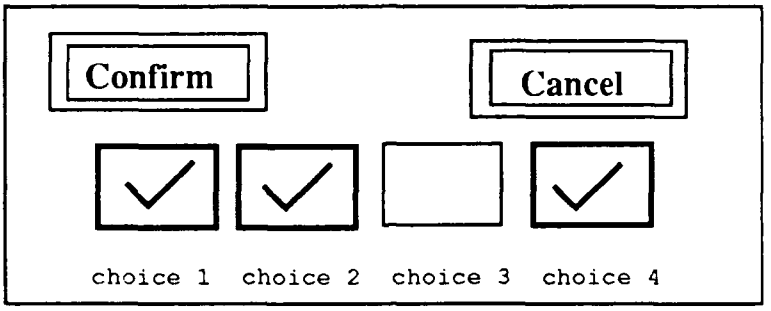

Figure 6: Check Box Selection

item selection. As is apparent, these interators are fairly complex, and require a more sophisticated geometry management capability than primitive menus do. For this reason, each of these kinds of interactors should also be implemented as special purpose widgets.

Recommendation: Build a rich family of controller widgets.

Recommendation: In the interim, model controller after figures 5 and 6.

2.4.3 Auxiliary Input Dialogues

Auxiliary input dialogues represent a grab-bag of interactors which support ad hoc communication between application and end-user. For example, warning messages, confirmer dialogues, and simple input dialogues fall into this category.

The Ada/Xt release provides support for these simple dialogues with the Athena dialogue widget. Figure 7 illustrates the general form of an Athena dialogue. In the near term, this widget should prove adequate. However, in the long term, a more sophisticated widget will be necessary, since the component geometry management provided by the Athena dialogue widget is rather constraining. That is, from a human factors perspective the grouping and ordering of the constituent parts of ad hoc dialogues (actually, for the earlier mentioned dialogue interactors as well, but the argument has already been made to construct new widgets for these) should be under application control; this is not supported by the Athena

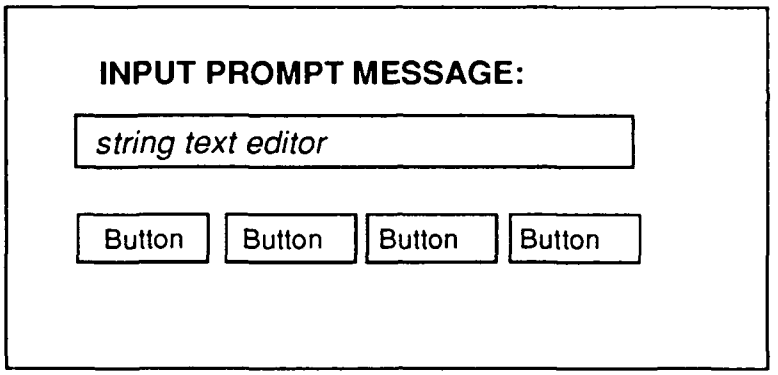

Figure 7: Athena Dialogue Widget

dialogue widget.

Recommendation: Build a rich family of dialogue widgets.

Recommendation: In the interim, model dialogues after Motif-style dialogues.

3 Programmatic Style

Considering the current level of maturity of Ada/Xt and the total lack of existing, real-world Ada/Xt client applications, it is appropriate to address issues of programming style. In this context, programming style applies both to Ada/Xt techniques which can foster the development of effective, flexible, reliable and maintainable application interfaces, and programming techniques to extend the toolkit itself. This section of the style guide describes useful conventions and techniques from the perspective of an application programmer, and from the perspective of a toolkit component designer. The former is concerned with developing effective application user interfaces; the latter is concerned with extending the Ada toolkit.

3.1 Ada Toolkit Applications

Two programming issues arise in developing Ada/Xt applications: how to structure the code to create internal application cohesion, and how to use Ada/Xt to create external application cohesion. Internal cohesion refers to the intrinsic quality of the Ada program; external cohesion refers to how well the application behaves in the enclosing software environment. This report is concerned only with external application cohesion.

3.1.1 Effective Use of Resources

From the application user's perspective, an Ada/Xt application provides a window-based interface and a modeless, direct manipulation style of interaction. The MIT X Toolkit itself

provides additional support for user tailoring of the look and feel of application interfaces via *resource management.* The toolkit resource manager supplies a rich variety of "hooks" which allow application writers, system administrators, project administrators, and end-users the opportunity to tailor the interface of an application to suit their own purposes. To a very large extent, constructing an externally cohesive toolkit application is an exercise in making effective use of the toolkit resource manager.

Resources: A quick overview

The toolkit consists of a set of user interface abstractions ("widgets"), and a set of generalized services ("intrinsics") which applications use to define their user interfaces. Each widget manages a collection of resources whose values are specific to each widget *instance*: these resources control the appearance and behavior of the widget. Resources can be thought of as *named data fields* within widget instances. That is, the toolkit provides, via toolkit intrinsics, operations to *get* and *set* widget resources by specifying the widget and resource name. This extra level of interpretation allows the toolkit to provide an elaborate system of widget resource initialization, at widget creation time, which includes a fairly complex search of environment-defined *resource files.* An externally cohesive Ada/Xt application fully exploits these environment-defined resource files.

Resource files consist of a sequence of *name:value* pairs which identify resource names and values. Within a resource file, resource names can refer to specific resources within specific widgets, or to entire *classes* of resources common to many widgets, or some combination of the two. Syntactically, resource names are specified with respect to the widget tree which represents the structure of an application interface. For pedagogical purposes, the simple iconic browser depicted in figure 8 will be used in the following examples. The application structure for this browser is depicted in figure 9.

Thus, to set the *borderWidth* resource of *button1*, the resource file entry could be specified as:

```
demo.viewport.bboard.button1.borderWidth : 0
```

Alternatively, a wildcard notation can be used as follows:

```
demo*button1.borderWidth : 0
```

The wildcard notation is preferred, stylistically, since it makes the resource file more or less insensitive to changes in the application interface structure (assuming buttons on a bulletin board are always leaf widgets).

If one wishes to set the border width resource to 0 for children of the bulletin board, one

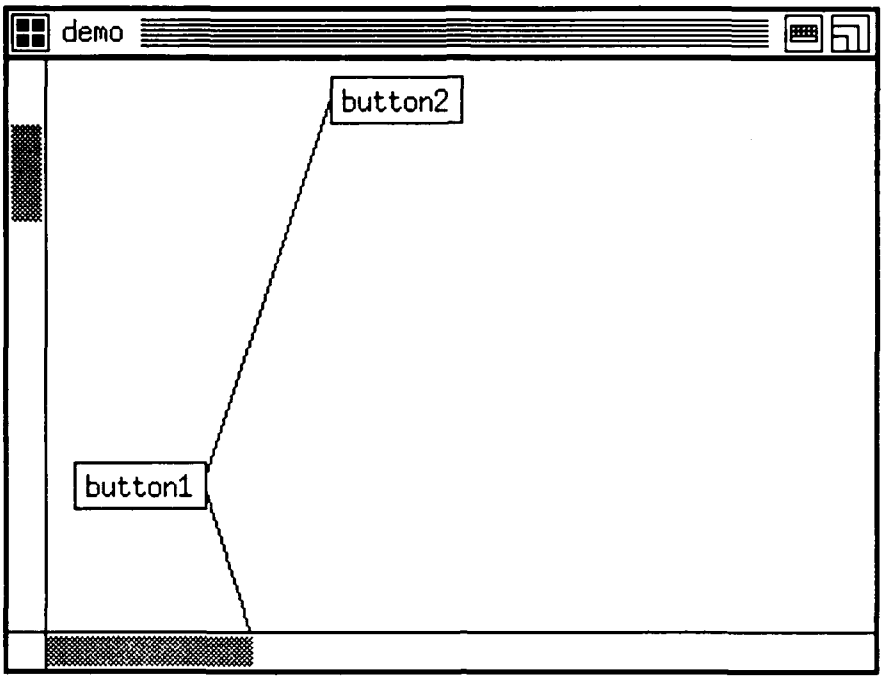

Figure 8: Screen Image of Sample Program

could use the class notation as follows[6]:

```
Demo*BulletinBoard*borderWidth : 0
```

A more complete discussion of resources can be found in [4], provided in the standard X distribution. Note that the class names used in the above resource file fragment are defined by the toolkit.

Resource Search Paths

The toolkit initializes widget resources by first searching the environment for resource files, then examining resource initialization specified on the command line for the tool invocation, and finally by examing the arguments to the programmatic call to create the widget. Specifically, the resource search path and logic is as follows:

1. Search the directory /usr/lib/X11/app-defaults for a resource file named *CLASS-NAME*, e.g., "Demo".

[6]The class name for this application is determined by the class name argument to the intrinsics routine xt_app_create_shell.

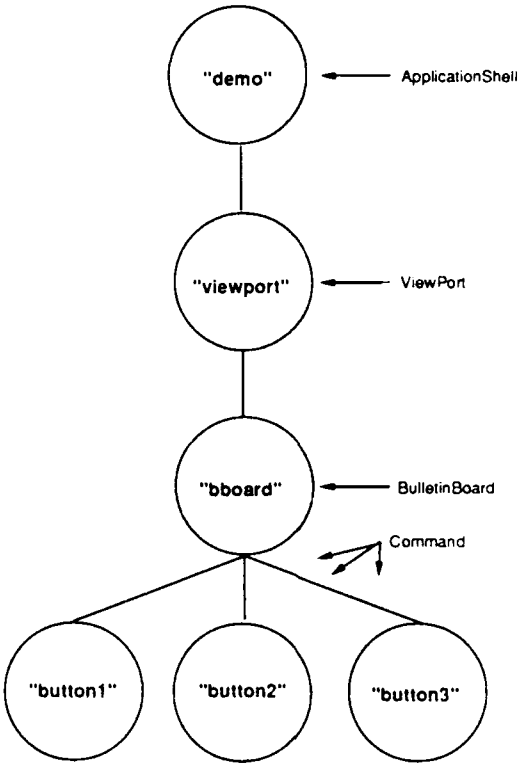

Figure 9: Widget Hierarchy for Sample Program

2. Search the directory specified in environment variable "$XAPPLRESDIR" if this variable is set; otherwise search in the "$HOME" directory for a resource file named *CLASS-NAME*.
3. Use display defaults, if any; otherwise search for the file "$HOME/.Xdefaults".
4. Use resources specified in environment variable "$XENVIRONMENT" if this variable is set; otherwise search for the file "$HOME/.Xdefaults-$HOSTNAME".
5. Search command line for resource specifications.
6. Use arguments to intrinsics widget creation routine.

The effect of this search list is that *name:value* pairs found later in the search overrides pairs found earlier in the search. Thus, providing resource values for widgets programmatically prevents any resource file or command line tailorability.

As a result, the following Ada/Xt application style recommendations are made:

Recommendation: Use programmatic resource specification sparingly.

A generalization of this recommendation is that resource values should be specified in the most flexible way possible; programmatic resource initialization is the least flexible of all.

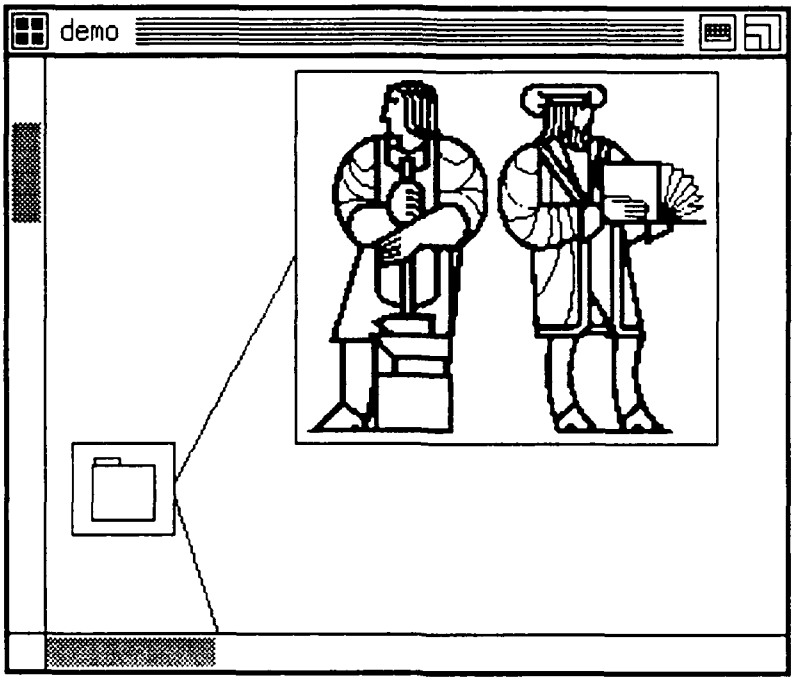

Figure 10: Resource-Modified Sample Program

There are reasons to set values programmatically, e.g., for setting the x and y coordinates of a tiled application (if specific component geometries are vital to the application).

Implementing generalized non-programmatic resource initialization in applications often requires considerable extra programming effort, but the benefits make the effort worthwhile. For example, the sample program above implements a node-layout mechanism which is based upon the values of widget resources, rather than some hard-coded coordinate set. Thus, it is possible to change the fonts or bitmaps of icons in the browser without "breaking" the layout appearance. See figure 10 for an example, where the visual presentations of button1 and button2 have been changed dramatically.

Recommendation: Choose widget names wisely.

The widget name specified programmatically to *xt_create_widget* (and its variants) provides the essential and most commonly used resource file hook for setting widget resource values. Well chosen widget names can enhance the flexibility of an application's resource file specifications. For example, in the demonstration program illustrated in figures 8 and 10, imagine that the command buttons lying on the bulletin board are nodes in a point-and-shoot iconic browser. If all nodes of a specific underlying node type (e.g., directory node) share the same name, then all instances of directory nodes can be given the same (differentiating) iconic representations with one line in a resource file:

```
*directoryNode.bitmap:folderIcon
```

Similarly, distinct translations could also be applied to different node types via resource specifications.

Recommendation: Define application classes.

To take advantage of the toolkit resource file search logic, applications should be grouped by classes, with common interface styles represented by the same class of application. For example, a generic, reusable iconic browser abstraction is being developed by the Unisys STARS program; all instantiations of this reusable browser should, programmatically, be specified as belonging to class "Browser" (or something similar). In this way, the resource file "/usr/lib/X11/app-defaults/Browser" can provide default behavior for all browser instances.

Recommendation: Use environment variables sparingly.

Installations *should not* require the use of environment variables for the location of resource files. However, the use of environment variables can be extremely useful in preparing demonstration versions of software. For example, using the hypothetical browser discussed above, for a demonstration using overhead projectors it would be reasonable to override default font sizes to a larger size. Use of the $XAPPLRESDIR environment variable for specifying special-purpose resource defaults is appropriate in such circumstances.

3.1.2 Translations

Translations are a specialized feature of the toolkit which provide limited end-user flexibility in the "feel" of applications. That is, translations allow the user specification of a mapping between sequences of events and application functionality. For example, the resource specification:

```
demo*bboard.translations : #override :<Key>Q:quit()\n
Demo*BulletinBoard*Command.translations : #override <Key>P:print()
```

specifies that uppercase 'Q', when typed onto the bulletin board widget, will cause the invokation of the application-defined function, "quit." Also specified is that 'p' and 'P' will cause browser nodes to be printed. Of course, this requires that the application has provided such functions, as well as a mapping from the string "quit()" and "print()" to these functions; a toolkit warning message will be generated otherwise. For further details consult [5].

Recommendation: Provide access to functions via the translation manager.

In order to make use of translation management, applications need to define primitive application functions in terms of translation management action procedures. In doing so, and by providing an external specification of the events to actions mapping, the application insulates itself from changes to *drivability* standards (i.e., which mouse button performs object selection tasks). A second benefit of translations is that it enables expert users to define keyboard equivalents to frequently used operations.

3.2 Extending the Ada Toolkit

Since the toolkit is extensible and since Ada/Xt is still quite new, it is anticipated that a large number of new toolkit widgets and resource types will be constructed in the near future. One of the most troublesome aspects of the toolkit implementation is the proliferation of types, and the sometimes baroque interactions between types. In order to manage the complexity of the system, strong guidelines are necessary in order to ensure that toolkit extensions are made in a uniform, controlled way. The following sections describe the programmatic style which is recommended for extending the toolkit with new resources and widgets.

3.2.1 Adding New Resource Types

This section defines a "cookbook" style for adding new resources. Each resource should be defined in a separate Ada package, whose name is *"resource name"_resource*. For example, the mask resource is defined in the package "mask_resource."

Resource Package Specification

The package specification for resources has five major components:

1. String definitions for resource name, class, and type.
2. Ada resource type definitions.
3. Constants for default resource values.
4. Instantiation of generic resource interfaces.
5. Useful Ada type converters.

Figure 11 provides the complete package specification for the mask example discussed, below.

String definitions for resource name, class, and type. This section defines string constants needed by the toolkit, application programmer, and end-user to access widget resources.

The resource manager is a kind of type interpreter which manages types as raw data; a substantial part of this interpretation is driven by type converter functions "registered" to convert from and to resources of various *types*. Resource *types* are specified as string values, as are resource *classes*. The package *xt_stringdefs* provides the definition of the pre-defined toolkit resources. New resources will need to introduce their own constants.

For example, the mask resource could be defined as:

```
xt_n_mask_32 : constant string := "mask32";
xt_c_mask_32 : constant string := "Mask32";
xt_r_mask_32 : constant string := "Mask32";
```

The *xt_n* prefix denotes a resource name constant; the *xt_c* prefix denotes a resource class name constant; *xt_r* denotes a resource type name constant. There is no restriction that the actual string names be identical, though in practice this is recommended. Note that class and type names begin with an uppercase letter, resource names begin with a lowercase letter. Also note that the class name value is the string users will specify in resource files (i.e., "Mask32").

Ada resource type definitions. This section defines the Ada types used by application and widget programmers.

These types define the concrete representation of types in widget instances. Because of the way Ada/Xt implements inheritance, it is *vital* that the actual resource type have a known size at compile-time. For example, in the current release of Ada/Xt an assumption is made that access types are all of some known, constant size. So the mask resource is defined as follows:

```
subtype mask_array is x_windows.boolean_array (0 .. 31);
type mask is access mask_array;
```

For other resource types, e.g., discrete integer types, the Ada LRM [8] chapter 13 type representation features must be used.

Constants for default resource values. This section defines values used by the resource manager to initialize widget resources when no default values are provided programmatically or from resource files.

For the mask resource example, the following is used:

```
null_mask_array : constant mask_array := (others => false);
null_mask       : constant mask    := new mask_array'(null_mask_array);
```

The constant null_mask_array, strictly speaking, is not necessary in the specification.

Instantiation of generic resource interfaces. This section defines the generic instantiations for programmatic manipulation of resources. This is straightforward, as illustrated:

```
mask_size       : cardinal        := 0;
package mask_interface is new resource_interface (mask, mask_size);
```

Useful Ada type converters. Finally, this section provides useful type converter functions for programmatic manipulation of resources. In particular, these functions are targeted for conversion among resource types and the base, raw data type, *caddr_t*:

```
function to_mask is new unchecked_conversion(caddr_t, mask);
function to_caddr_t is new unchecked_conversion(mask, caddr_t);
```

It is only an implementation artifact that unchecked conversion is used here. Actually, any arbitrarily complex type mapping can be performed instead, assuming source and target representation sizes are preserved.

Resource Package Body

Resource types require package bodies only if the resource type introduces new type converters which the toolkit will automatically call to convert between resource types. The most commonly required conversion is from string representation to Ada type representation, since this is the transformation required when reading resource values from resource files.

The actual conversion function need not be (and should not be) made visible in the package specification; rather, it should be implemented within the body of the package, and then installed via the intrinsics utility *xt_add_converter*. This installation should be done in the begin..end elaboration block of the resource package body[7]

3.2.2 Adding New Widgets

Adding new widgets to the toolkit is a specialized, intricate code development task. The details of implementing widgets is far beyond the scope of this report. However, the top-level packaging aspects of widget development require consistent treatment. In particular, it is necessary to define widgets in two packages: the public, and the private packages.

The details of widget subclassing and packaging can be found in the Ada/Xt [7] design report. Key aspects of these conventions are reprised, below.

[7]Note: this requires the resource package body to elaborates the intrinsics package.

```
with ...;
package mask_resource is

   -- resource class and name constants, if necessary (see xt_stringdefs_.a):
   -- xt_n_text_options : constant string := "textOptions";
   xt_c_options_32 : constant string := "Options32";
   xt_r_options_32 : constant string := "Options32";

   -- resource data representation:
   subtype mask_array is x_windows.boolean_array (0 .. 31);
   type mask is access mask_array;

   -- constants for default resource values:
   null_mask_array : constant mask_array := (others => false);
   null_mask       : constant mask   := new mask_array'(null_mask_array);

   -- resource interfaces from generic instantiation:
   mask_size       : cardinal        := 0;
   package mask_interface is new resource_interface (mask, mask_size);

   -- useful conversion operations
   function to_mask is new unchecked_conversion (source => caddr_t,
                                                 target => mask);

   function to_caddr_t is new unchecked_conversion (source => mask,
                                                    target => caddr_t);

   --  Resource converters
end mask_resource;
```

Figure 11: Sample Resource Type Specification

3.2.3 Widget Public Specification

The public specification is used primarily by toolkit clients, i.e., application programs. This interface defines:

1. Resource documentation: names and semantics.
2. Widget and widget class Ada type marks.
3. Widget class function.
4. Publically available, widget-specific operations (subprograms).

Resource documentation: names and semantics. This section of the specification provides a description of widget resources that are accessible programmatically or through resource files. For example, from the Athena Scrollbar widget:

```
-- Name          Class          RepType         Default Value
-- ----          -----          -------         -------------
-- background    Background     Pixel           White
-- jumpProc      Callback       Function        NULL
```

Note that the public specification should document the interfaces of all callbacks in terms of their usage from an Ada program. Such documentation is needed because of the raw-typed interface (i.e., *caddr_t*) for callback client and call data. Again, from the Athena Scrollbar widget:

```
-- jumpProc(w : widget; client_data : caddr_t; call_data : xt_float);
--     client_data is client defined;
--     call_data is the position of the thumb, in range 0.00 .. 2.00
```

This documentation is sufficient for the application program to perform necessary conversions between the caddr_t type and the actual parameter type.

Widget and widget class Ada type marks. This section of the specification provides Ada type marks. The current release of Ada/Xt employs a subtype hierarchy which parallels the widget subclass hierarchy. Later releases of Ada/Xt may incorporate use of a derived type hierarchy for widget types[8].

[8] The class hierarchy will still be subtype oriented, since parallel derived type hierarchies introduce intrinsics operator visibility problems.

Widget class function. The Ada/Xt implementation defines the subclass constant as a parameterless function returning the type of widget class defined above. This is an implementation decision.

Publically available, widget-specific operations. This section of the specification defines the interfaces to widget-specific operations. For example, the Athena TextEdit widget implements several operations for accessing the state of text editor widget instances. The only stylistic point to note is that the implementation of these operations should *all* verify that the actual widget parameter to the operation is indeed a subclass of the widget which introduced the operation. In this way toolkit dynamic typing can catch, albeit at run-time, simple type mismatch errors. The previously mentioned derived widget type hierarchy may make such testing unnecessary in future releases of Ada/Xt.

3.2.4 Widget Private Specification

The private widget package specification is used solely by the intrinsics: toolkit clients should never depend upon the contents of the definitions in the private package.

The purpose of the private specification is to define the concrete representation of Ada/Xt widgets. This concrete representation includes manually compiled inheritance relations which require considerable usage of Ada LRM chapter 13 features. The reader is advised to consult [7] and sample widgets in the Ada/Xt release before attempting to define a new widget.

4 User Interface Styles and Standards

With the move to open systems, the formal standards bodies (IEEE and ANSI in the U.S.) have recognized the need for program presentation and drivability standards. In IEEE Window Systems Application Portability Group Drivability subgroup (P1201.2) work has begun on drivability standards to address the following problems:

- error provoking inconsistencies
- misleading expectations about the results of user actions
- gross inconsistencies in the high level user model or metaphor
- incompatible motor control tendencies

Areas currently under examination by P1201.2 are

- keyboard

- feedback
- menus
- buttons
- pointing devices
- tasks and actions
- window

The result of the P1201.2 work will be a recommended practices standard. The intent of P1201.2 is not to produce a "look and feel" guideline, but to address problems in consistency of behavior among a diverse set of window system based applications on a variety of hardware platforms.

In addition to the P1201.2 work ANSI through X3V1.9 is working with humans factors organizations to develop "look and feel" standards for all aspects of the user interface. They have submitted work on icons to ISO, and are working on cursor control and keyboards.

Another group within ANSI, Human Computer Interaction (HCI), is taking a broader view of user interface standards beyond window system interface. This group is also feeding guidelines to ISO. Current work is on menu dialogues.

Several industry groups have developed standard "look and feels" for the open system environment. Open Software Foundation (OSF) developed its "look and feel" standard from the Presentation Manager Style Guide and requires Motif compliant applications to adhere to this style. Unix International, led by AT&T and Sun, have developed a competing "look and feel" standard, Open Look. Competition between Motif and Open Look is fierce as each tries to establish its style as the standard. Current estimates have Motif dominating with 70 to 80 percent of the market for windows system based applications.

5 Summary of Recommendations

Below is an itemized list of key recommendations made in this report:

- Adopt the Motif Style Specification.
- Applications should implement object-action selection.
- Provide visual feedback for all user interactions.
- Build a rich family of menu widgets.
- In the interim, model menus after figure 4.

- Build a rich family of controller widgets.
- In the interim, model controller after figures 6 and 5.
- Build a rich family of dialogue widgets.
- In the interim, model dialogues after Motif-style dialogues.
- Use programmatic resource specification sparingly.
- Choose widget names wisely.
- Define application classes.
- Use environment variables sparingly.
- Exploit the translation manager.

References

[1] The Boeing Company, Boeing Aerospace Division. *Programmer's Guild for [BOEING] STARS User Interface Toolkit*, January 1990.

[2] Apple Computer. *Inside Macintosh Volume 1.* Addison-Wesley, ISBN 0-201-17731-5, 1987.

[3] Hix Deborah H. Rex Hartson. *In ACM Computing Surveys, Human-Computer Interface Development: Concepts and Systems.* ACM Press, 1989.

[4] MIT X Consortium. *Using and Specifying X Resources*, April 1988.

[5] MIT X Consortium. *X Toolkit Intrinsics - C Language Interface*, July 1988.

[6] Open Software Foundation. *OSF/Motif Style Guide*, July 1989.

[7] Unisys Defense Systems. *Ada/Xt Architecture: Design Report*, December 1989.

[8] United States Department of Defense. *Reference Manual for the Ada Programming Language*, February 1983.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	2 November 1990	Final

4. TITLE AND SUBTITLE
Ada/X Interface Style Guide

5. FUNDING NUMBERS
STARS Contract
F19628-88-D-0031

6. AUTHOR(S)
Kurt C. Wallnau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER
GR-7670-1150(NP)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
Headquarters, Electronic Systems Division (AFSC)
Hanscom AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
01030

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This technical report defines conventions for development of effective application user interfaces. The guide is intended for use by application developers and user interface developers using the Unisys STARS/Ada/X User Interface Software. The intent is to produce a consistent style among Ada applications. The Style Guide addresses three areas of style:

- application/user interactions or dialogues,
- conventions for developing applications, and
- formal and informal user interface standardization efforts.

14. SUBJECT TERMS
Ada/Xt Toolkit Applications

15. NUMBER OF PAGES
22

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR