# The X Toolkit: More Bricks for Building User-Interfaces
# –or–
# Widgets For Hire

*Ralph R. Swick*

Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
swick@ATHENA.MIT.EDU

*Mark S. Ackerman*

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
ackerman@ATHENA.MIT.EDU

*ABSTRACT*

Primitives for application-level user interface construction facilities currently under development at M.I.T. Project Athena are described. The design philosophy of the X Toolkit and associated widgets and some of the practical implications are discussed.

## Introduction

The X Window System† [1, 2] was developed at the Massachusetts Institute of Technology to satisfy the needs of a broad spectrum of users for a high-performance, high functionality, network-based window system that can be implemented on a wide variety of high-resolution raster graphics display devices. The widespread interest and unprecedented vendor support for the X Window System has assuaged one of the principal concerns of application developers: the cost of supporting multiple hardware platforms with different base technologies, including window systems.

† **The X Window System** and **X Windows** are trademarks of the Massachusetts Institute of Technology. Use of the latter is strongly discouraged. The developers prefer simply "**X**" when a shorter form is required.

The X Window System has been carefully designed to address two (sometimes conflicting) desires of application developers: to use hardware-level techniques for maximum performance and to maintain portability with a common programming interface across multiple vendor platforms. X has succeeded in gaining broad vendor support largely because its specification is intentionally restricted to the set of primitives needed to manipulate multiple independent window contexts on raster graphics displays without declaring (or restricting) the choice of particular user-interface semantics. It is explicitly intended that developers be able to choose a visual interface appropriate to their needs, their corporate philosophy, their research requirements, religious preferences, or whatever.

This restriction to low-level control and input primitives in the definition of the X communications protocol and in the corresponding application interface layer, Xlib[3] , is either a strength or a weakness in the X Standard, depending on the reviewer's point of view. Vendors whose installed base of products contains well-defined visual interface and human-computer interface researchers find this flexibility in the standard to be of major importance. However, many application developers consider user interaction and other higher-level graphics libraries to be a base system technology that should be provided by the hardware vendors and, of course, be standard across vendors.

To address the desires of such developers for common higher-level development tools, there are several projects under way at various locations covering different application needs and problem domains. One such project is the X Toolkit project, a collaborative effort of MIT/Project Athena, DEC/Western Software Laboratory, Hewlett-Packard Company/Corvallis Workstation Operation and others. The X Toolkit project is producing an applications interface layer above the Xlib layer specifically tailored to visual user interface construction.

**Toolkit Overview**

The X Toolkit (hereafter called simply "Xtk") recognizes that no single comprehensive set of user interface tools is likely to be acceptable for standardization in the near future. In order to maximize the utility and acceptability of the user interface library, Xtk has been divided into two separable pieces. These two layers will be described below.

The fundamental entity in Xtk for user interface construction is the *widget*.†

The core of Xtk, the "Intrinsics" (a term appropriated from a previous H-P user interface library for X), is a set of utility routines intended for use in developing widgets. The Intrinsics are a set of user interface primitives that are themselves free of visual and interaction style. The Intrinsics do not constrain the widget writer to make the widget look or operate in any particular way. These primitives may be used together or separately to produce higher layers which do incorporate specific policy and style. Such higher layers will further reduce applications development cost.

†We chose this term since all other common terms were overloaded with inappropriate connotations. We offer the observation to the skeptical, however, that the principal realization of a *widget* is its associated X *window* and the common initial letter is not un-useful.

Most applications will call only a few of the Intrinsic routines directly. These routines offer a uniform programming interface to the basic procedures (*methods*) of all widgets, regardless of the widget type.

The Xtk Intrinsics have been presented in an earlier paper[4] and, although the detailed design has evolved,[5] the philosophy and architecture of the X Toolkit remain the same. The principal additions are a class hierarchy for widget types; the separation of widget identifiers from the corresponding X window identifiers; and the ability, using the class hierarchy, for new widgets to inherit methods from an existing widget. These changes simplify significantly the task of widget development and make widgets more modular.

Widgets define input semantics and visual appearance. Some widgets are pliable; their input semantics (mouse buttons, pointer motion, keyboard input) are bound at run-time, while other widgets may have fixed (hard-coded) semantics. Likewise, visual appearance (highlighting, repositioning or other animation) may be fixed or may be adjusted at run-time.

The Intrinsics provide a uniform way for widget developers to handle the common chores of widget construction: initialization, input event dispatching (including enabling and disabling user input dispatch to sub-hierarchies of widgets), run-time configurability, uniform handling of common events (such as exposure and re-size), cleanup, and others. The Intrinsics also include the uniform application programming interfaces for creating, controlling, and destroying widgets. The Intrinsics currently consist of over 90 public procedures, of which half are intended solely for widget construction.

The goal of the Intrinsics is to make possible the quick development of widgets. Sets of widgets should adhere to a consistent application interface, user interaction policy, and visual appearance. It is viewed as desirable (or, by some, a necessary evil) that the construction of multiple such sets ("widget families") covering different philosophies be possible with the Xtk Intrinsics.

The second piece of the Xtk is a set of basic widgets. Most application developers require a minimum set of widgets as a component of any product-quality user interface library. The X Toolkit project also recognizes the need for concrete examples in a real widget family. To serve both these needs, the first author is leading the effort at Project Athena to produce a basic widget set that will be included with the version 1 release of the X Toolkit. To distinguish this set from others which we know of, or expect to be developed, we shall here call these the "Athena Widgets".

In addition to the goal of being a basic widget set, the Athena Widgets have another goal arising from code which had been written prior to X Version 11. Many of the components of Xtk had been prototyped in a toolkit for X Version 10 that was released by DEC Ultrix Engineering in the spring of 1987. The Athena Widgets borrow heavily from those prototypes in order to ease some of the porting burden for certain applications built on these prototypes.

The widgets described here are being developed in conjunction with a set of visual courseware projects at Project Athena. These projects vary considerably in their user dialogs and yet require a standard visual appearance. This has led to an emphasis in the Athena Widgets on handling text, graphics, and video in a variety of ways, and has extended the widget hierarchy to fulfill these needs.

The Athena Widgets are intended to fulfill 80% of application requirements. We have tried to select the critical widgets that will allow the easy solution of individual requirements. (See the section on Creating New Widgets for more on this.)

**Intrinsics**

One of the principles espoused in the design of the Xtk is the construction of widgets from primitives. We will describe two independent facilities available in the Intrinsics for such construction: subclassing and composition. From an application point of view, every widget is a single object. The actual semantics and appearance of the widget may, however, be very complex. For example, a "control panel" widget is likely to consist of simpler widgets with a "geometry manager" controlling the spatial relationship between the component widgets and possibly a "focus manager" controlling the dispatching of user input to those components. Depending upon the needs of the application, such a compound (or "composite") widget may be implemented independently and added to the widget library as a new widget class, may be constructed by the application at run-time with in-line calls to the Xtk Intrinsics, or may be constructed by the composite widget itself from a resource record retrieved through the Intrinsic resource management facilities.

Composition of widgets is most appropriate when there are distinct visual regions to a widget, each having separate input/display semantics, and especially when the same semantics may appear in a region of another widget class. In this case, the semantics common to both regions may be extracted into a more primitive widget class.

This is the principle of modularity of widgets: the application will still view a composite widget (a control panel, for example) as a single widget. The internals of this composite widget are built when the widget is instantiated. The composite widget may determine and instantiate all of its components, as for a custom application panel. The components may also be instantiated and assigned by the client of the composite widget. Some of the Athena Widgets exhibit this recursive construction behavior; e.g. Dialog, while others are ‘boxes’, or frames into which the client inserts independently instantiated widgets, e.g. Form and VPaned.

The second construction facility, subclassing, allows a widget class to semi-automatically inherit some or all of the characteristics of an existing widget class, and to share portions of the code that implement the parent (super-) class methods. The new widget may call upon the superclass methods to manipulate any part of the widget state defined by the superclass and needs only to implement the code to manage the state that is unique to the new class.

The subclassing facility is most appropriate for new widgets which need only to add additional semantics to an existing widget, or to constrain in some manner the full generality of an existing widget. Examples of both subclassing and composition in the Athena Widgets are described below.

Several widget classes have been defined solely for the purpose of being subclassed. The **Composite** class provides methods to maintain a list of child widgets, to manage the insertion and removal of children, to manage requests from the children for new geometries, and to manage the assignment of input events to specific children (input focus). The **Constraint** class has all the methods of Composite and in addition provides methods to automatically create and initialize an arbitrary data record attached to each child. The contents of this data record are defined by each subclass of Constraint and are intended to contain layout information used by the subclass geometry manager. Neither Composite nor Constraint are intended to be instantiated; only their subclasses are. Nothing in the implementation, however, will prevent an application from instantiating any class, should it prove useful.

All widgets are expected to be self-contained with respect to exposure, resize and input event handling. That is, the clients of the widget (i.e. the application program or a composite widget of which this widget is a component) are guaranteed that all exposure and input events sent by the X server for the window defining the widget will be processed completely by the widget. A client that creates an instance of the ScrolledAsciiText widget, for example, is not involved in any of the details of text re-painting, scrolling, selection, cut and paste, and so on. The client is also free to assign any shape to the widget and assume that the widget will adjust to the imposed size.

The principal mechanism the Athena Widgets use to communicate back to the client is the callback procedure. While a client has the option to query the widget state, it is usually more convenient for a command button, for example, to directly call a client-supplied procedure when ‘pressed’ by the user. Some widgets accept more than one callback procedure for alternate interactions that they implement.

**Runtime Configurability**

One of the major design principles followed by the Athena Widgets is to make as much of the user interface as possible customizable by the end-user of the application. Fierce debates (i.e. *wars*) break out every time someone proposes a single set of key or button bindings for all users, or that a fixed choice of colors or text fonts will be appropriate for all individuals. Even such characteristics as whether scrollbars go on the left or right (or top/bottom) of a window may be appropriate for individual customization.

The Xtk implements run-time configurability through the Xlib Resource Manager. The Resource Manager is a general-purpose repository for storage and retrieval of arbitrary data within a single process address space. During initialization, the Xtk pre-loads the resource database from the X server and from one or more files. When a new instance of a widget is created by the application, the widget resource list is examined and the widget instance is initialized with data from the resource database. Each widget declares an instance name and a class name for purposes of matching against resource names in the database. The Resource Manager defines rules for partial name matches so that a single resource entry may initialize many widget instances.

The implementor of a new widget has the choice of which widget characteristics to declare as resources and which to hard-wire into the widget. In general, any instance data for which the widget is willing to allow modification requests from the client should be declared in the resource list.

Widget characteristics such as text font and color are obvious resource choices. The Athena Widgets also declare the keyboard and mouse button bindings as resources. In this way, the user of any application linked against the widgets has the option to accept our default bindings or enter his/her own bindings. The Resource Manager naming mechanism allows the user to attach the new bindings to all instances of a widget class (e.g. Scrollbar) in any application, or to a specific widget in a specific application only, or any combination in-between. A client may, if it so chooses, override any entry in the resource database either by storing its own entry (preferred), or by passing an explicit value when instantiating the widget (discouraged).

The keyboard and button bindings are configurable in yet a second way. Each widget that accepts user input declares a list of action routines that may be invoked by input events. The Athena Widgets use the Translation Management facilities in the Intrinsics to bind keys and buttons to widget action routines. These bindings can specify parameters to the action routines to further configure their behavior. The Scrollbar widget, for example, declares three action routines, one of which is parameterized so that the range of values it returns may be established by the bindings.

Using these action routines and the default scrollbar bindings, the ScrolledAsciiText widget, for example, allows the user to scroll a block of text forward or backward by a variable amount and to drag the thumb (elevator) to a new position, displaying any portion of the text. A user may supply an alternate set of Scrollbar bindings that will cause the scrollbar to report full-length forward or backward scrolls, independent of the pointer position, possibly disabling the variable scrolling as desired.

Two of the Athena widget classes exist for the purpose of handling interprocess interactions with other X client processes. The **Shell** widget defines no user action routines, but maintains all the appropriate window properties established by convention for X window managers, including icon representation. Many of these parameters are controlled by command line options and parsed by the Xtk initialization routine. Most applications use a Shell widget as their outermost (top level) widget.

Additional semantics appropriate for temporary, or "pop-up", panels are added to a subclass of Shell, the **Popup** widget. From the client's point-of-view, both Shell and Popup are simple widgets; they manage exactly one child widget and have a trivial geometry manager. UIMS developers may find it desirable to extend Shell at some time in the future, even allowing site tailoring for specific choices of window manager. One such addition might be a Shell that, when made smaller by the window manager (as instructed by the user, of course); added scrollbars (or other interaction semantics); and provided a movable viewport on the application window which, from the application's point-of-view, retains its original size.

**Current Widgets**

The Athena Widgets are divided into two major classes. The first are simple widgets: various sorts of buttons, labels, edit buffers and the ubiquitous scrollbar. These form the elemental building blocks of a user interface. The second are composite widgets: scrolled text, dialog boxes, and various methods of putting together simple widgets in more complex arrangements.

All simple widgets have initialization, realization, display, and interaction methods. These methods may be fairly simple, as with the button widgets, or quite complex, as with the text widget.

All widgets use the Core widget as the root of their class hierarchy. The Core widget (whose class name, as a special case, is just **Widget**) has the minimal set of instance fields common to all widgets: width, height, border width, and so on. While it is not usually intended that an application ever create an instance of (Core) Widget, it is supported: an application that wants a simple window within a widget panel *may* instantiate Widget and use the resulting window.

The **Label** widget is just that; a widget that displays either a text string or (in the near future) a pixmap without any interaction semantics and therefore without any callback procedures. It can only center, right justify, or left justify its text in the client's choice of fonts within its assigned size. (The default size is a bounding box for the text or graphics.) A Label may be insensitive, or grayed-out, and the border, as for all widgets, need not be visible. Like all widgets, the Label widget processes exposure events on its window so the parent can be assured that all visible portions of the Label are properly refreshed on the screen. One will typically use the label to position and paint items intended for display only.

The **Command** button widget is a subclass of Label with interaction semantics and therefore a callback procedure. A Command button is a Label with *enter*, *leave*, *set*, *unset*, and *notify* actions. When insensitive, the Command button does not respond to user input events. On *enter* to a sensitive Command, the border will, by default, become visibly thicker. On *set*, the button is displayed in reverse video; on *unset*, the button is returned to normal. On *notify*, a single client-supplied callback procedure is activated. These actions are mapped, by default but not by necessity, to the pointer enter, button down, and button up events. One will typically use the Command button to create "clickable" icons or labels that start a program action. The Command button is an essential building block for point-and-click interfaces.

Figure 1 shows several Labels and Command buttons. The user has moved the pointer into one of the Command buttons, and the button has been highlighted. These Labels and Command buttons are enclosed in two layers of Boxes, which will be discussed below.

Figure 1

The Command widget can be extended into its subclass **TwoState** button. On *set*, the TwoState widget displays a second label or graphic. In all other respects, it is similar to Command. TwoState is useful for simultaneously displaying and changing a binary application state.

The **Toggle** widget is also a subclass of Command and also has simple interaction semantics and a callback procedure. A Toggle is a binary state button. Once *set*, e.g. by button down, it remains in that state until the user selects the Toggle again. Toggle is useful for selecting on-off options. The application may use the callback to be notified on each state change, or may query the Toggle for its current state as appropriate.

The **Grip** widget is a minimal Command button. Grip (a.k.a. Knob) has a single callback but no default interaction semantics. The interactions (event bindings) are provided by the client when Grip is instantiated. Grip simply uses the X window background as its display. This widget was built for clients who need to identify specific locations where, for example, the pointer may be used to drag an object.

The **Scrollbar** widget implements a vertical or horizontal scrollbar. There are three callbacks: one to move the scrollbar thumb (elevator), a second to execute a client callback passing a distance in pixels as data, and a third to execute an application callback passing as data the position of the pointer as a percentage of the scrollbar length. The inclusion of the *moveThumb* action routine in the actions table allows the client some control over whether or not the scrollbar widget automatically repositions the thumb on input.

The **AsciiFile** and **AsciiString** text widgets allow the entry, modification, and display of text. The widgets can be instantiated in read-only or read-write modes. They implement a subset of the ever popular Emacs, including mouse-driven cutting and pasting. Key bindings are, of course, completely settable through the Translation Manager.†

The AsciiString widget operates on a single in-core text string; AsciiFile on a disk file. They are both subclasses of **Text**, which implements

†It has been suggested that some users would prefer an AsciiFile widget that allowed them to specify their favorite text editor. While the full semantics of the Text widget may be difficult to support with an arbitrary external process, it may be an interesting exercise to implement a sub-process interface widget that, in carefully defined circumstances, could be substituted by the user for the AsciiFile widget.

most of the user interactions. The implementation of the Text widget follows a source/sink model, allowing for the development of additional text sources (other than string and file) and for additional display sinks.

These few simple widgets are an attempt to create a number of general widgets that can then be tailored for individual uses. A more complex button widget (for example, one that might display two lines of text) would require a different display method but keep the interaction method for Command.

However, few interfaces of any real complexity or use could be created using just these simple widgets. The simple widgets must be combined using Composite widgets. Composite widgets require geometry, child insertion and deletion, and input focus methods. The geometry manager for the composite may or may not be constraint based, and must handle resizing events by re-positioning the child widgets. Generally, the child insertion, child deletion, and input focus methods will be inherited from the superclass, Composite.

The **Box** widget arranges its children in the minimum bounding box. The children are automatically rearranged when one is deleted or added. One can create button areas or horizontal menus using Box.

The **RadioButtonBox** widget is a subclass of Box exclusively for Toggle Buttons. It allows only one Toggle button to be set at a time. If a second Toggle becomes set, the RadioButtonBox will check its list of children and unset any others. This is useful for mutually exclusive application options.

The **Form** widget can contain any number of simple or composite widgets. Form is a general-purpose constraint-based layout widget that can be instructed to maintain fixed separations between children, or to maintain fixed distances between children and the edges of the form, and so on. When Form is resized, it uses the constraint information to resize and reposition the children to maintain the assigned separations. Forms are useful for creating arbitarily complex interaction windows that exhibit ‘nice’ resizing behavior.

The **Dialog** widget is a subclass of Form. It arranges a specific combination of Label, AsciiString field, and Command buttons. The Label is displayed on the first line, followed by the text field, and the buttons are on the last line. Dialog is principally a convenience widget implemented to handle a common programming problem.

The **ScrolledAsciiText** widget contains a scrollbar and an AsciiString or AsciiFile widget. Using the default scrollbar bindings, the ScrolledAsciiText widget allows the user to scroll the contained text forward or backward by a variable amount. By dragging the thumb to a new position, s/he can display any portion of the text.

The **Paned** widget manages any number of simple or composite widgets in a tiled manner. The current Paned widget, **VPaned**, stacks its children vertically with the top and bottom edges of successive widgets touching. VPaned uses Grip to allow the user to re-position the boundaries between the tiled widgets.

Figure 2 illustrates the current Athena Widget hierarchy. This diagram should only be of interest to widget developers; application developers should be concerned only with the external characteristics of a widget. The fact that a Toggle button actually uses the Label code to display its text string should be of no consequence to the application. The widget class **Simple** contains only the procedure to change a widget’s borders to indicate sensitivity or insensitivity.

**Special-purpose Widgets**

The **Clock** widget displays an analog or digital time-of-day clock. The **Load** widget displays a continuous system load graph. Both of these were exercises in converting existing simple applications into widgets. Load allows the client to supply its own procedure to fetch data, or to use the built-in default *GetSystemLoadAverage* procedure.

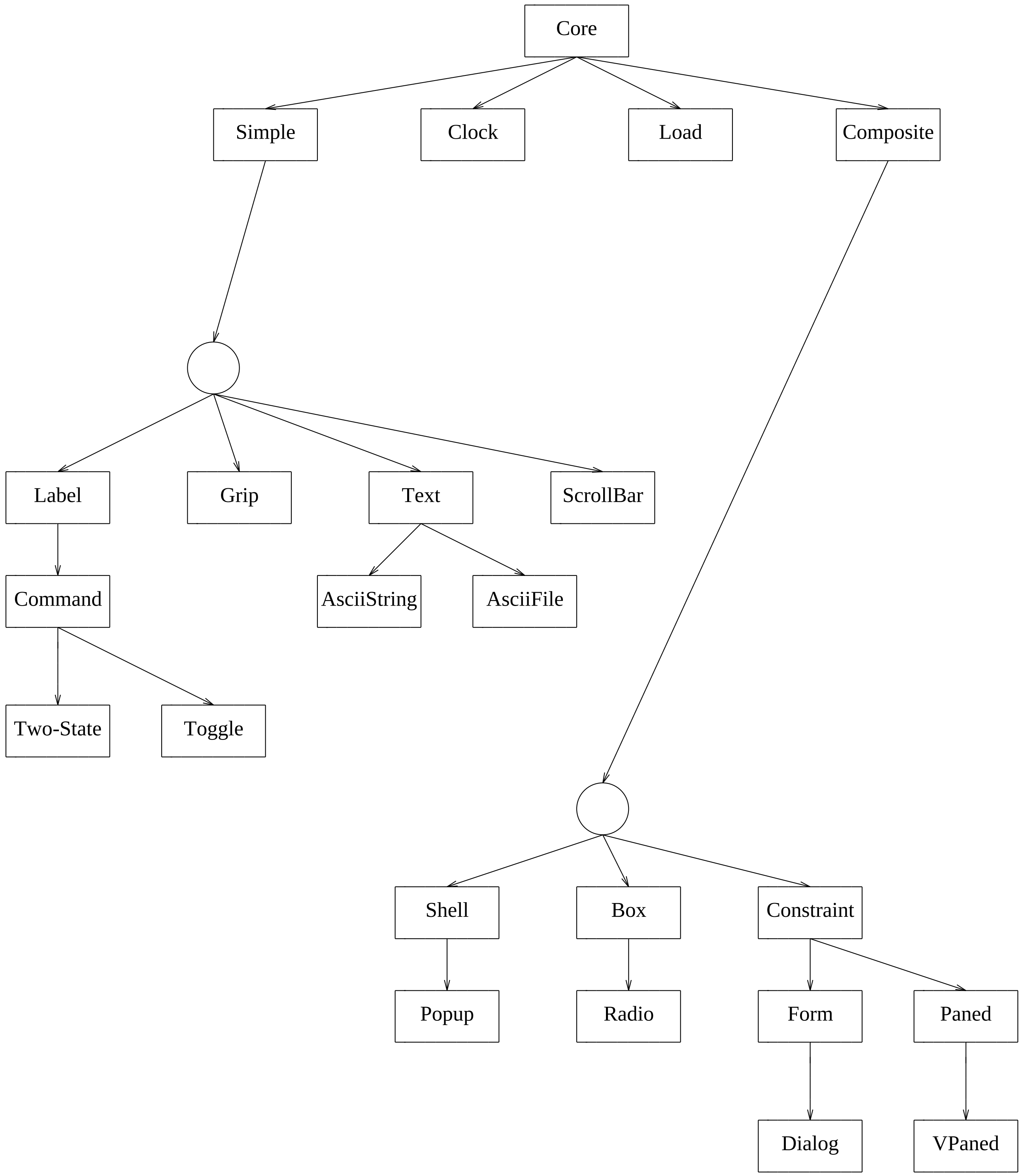
Core
Simple
Clock
Load
Composite
Label
Grip
Text
ScrollBar
Command
AsciiString
AsciiFile
Two-State
Toggle
Shell
Box
Constraint
Popup
Radio
Form
Paned
Dialog
VPaned

Figure 2

**Example**

Figure 3 shows a variety of composite and simple widgets instantiated in a single trivial application. The outermost widget is a VPaned (inside a Shell) which, in turn, contains several other composite and simple widgets. The small solid boxes are Grips which allow the user to move the attached pane boundary up or down, forcing the associated widgets to be resized.

The uppermost pane of the VPaned is a Command button marked "quit". Appropriately, when the user clicks on this button, the application exits. The pane below this is a Label.

The third pane is a Dialog containing a Label "I am a dialog form"; an AsciiString text field with an initial value; and a Command button marked "ok". The next pane is a Box containing a Label marked "label" and a Command button marked "command". The Clock and Load widgets form the next two panes.

The seventh pane is another Box with an AsciiString widget and a Scrollbar. The last pane in figure 3 is an AsciiFile widget displaying */etc/motd*.

The application that instantiated this entire hierarchy is less than 200 lines of C source, including four callback procedures.

Figure 3

**Creating New Widgets**

Small changes to existing widgets can be made through the general facilities of the resource and translation managers. With these managers, one can make changes to color, font, key bindings, and the such. Many changes that would require a new interface object in other toolkits can be done through resource changes to the existing widgets in the X Toolkit.

The simplest case of creating a new widget is to create a subclass of an existing widget. For example, if one wanted a command button with two lines of text, s/he might create a subclass of Command. Since all superclass structures are inherited, one would merely need to add an additional string field to the instance structure and nothing to the class structure. Since there is no change in user interaction semantics, only the display method would need to be changed. In addition, code to allow the modification of the second text string would be required. All other code could be inherited. This new widget could then be incorporated in any composite widget that allowed buttons.

Subclassing composite widgets is a little more work since new geometry managers may be desired. In general, the additional methods of a Composite widget tend to be complex and the new subclass will want to inherit as much as possible.

One potential ‘client’ of a widget that the implementor of the widget should keep in mind is the writer of the next widget. With a little thought, the designer of a new widget can decide which of the new methods added by the subclass should be declared as class variables, which as widget instance variables and which as in-line code.

Class variables are the best way to export methods to potential future subclasses. By installing a procedure pointer into a class variable, the new subclass has the option to easily inherit the old method or to define its own implementation.

The developer of a family of widgets may find circumstances in which a small modification to a parent class would make a new subclass much easier to implement. From the application point-of-view, widget instance and widget class data structures are opaque types. The effect of an addition to one of these structures can be isolated to subclasses of the changed widget without breaking existing application code.

A subclass is free to manipulate any of the widget instance variables defined by its superclasses. The subclass must be careful if it simultaneously manipulates superclass instance data and inherits superclass methods that use the same data. At present, the only way to determine such side-effects is by examination of the superclass implementation(s). We hope to establish conventions in the future for documenting all the public and semi-public interfaces of a widget.

## Widgets Of The Future

On our wish list are:

**read-only dial**: posts a position between 0 and 1 to a circular dial. Useful for indicating position or state.

**title bar**: the mandatory title, horizontal lines, and close button. A convenience widget, like Dialog.

**menus**: pull-down, pull-aside, and deck-of-cards menus.

**index pane**: allows the user to select from a scrollable list of text strings. Useful for selecting topics or filenames.

**video label**: handles the display of video from an external source inside a widget.

**video command button**: extends the command button to handle video windows.

**text sinks**: a variety of additional Text sinks is desirable. One such possibility is a sink that interprets bytes from the Text source as ANSI Terminal control sequences. Another desirable sink would interpret font change control sequences in the source and display multi-font text.

**viewport (frame)**: a movable (scrollable) window frame on a larger widget.

**data plotting**: a variety of data plotting widgets. Useful for engineering curriculum applications.

Many of these widgets are required by the visual courseware projects at Project Athena and will be implemented over the next several months.

## Issues

There are several open issues in the effective design and implementation of new widget classes:

- The class mechanism has only a simple inheritance structure. Therefore, a widget class gets both its display and interaction semantics from its single parent class. This leads to an awkwardness and duplicate code in composing widgets where multiple inheritance is really desired. One way around this is to export little procedures that may be used by other widgets to minimize duplicate code, but this is clearly undesirable.

  One example where simple inheritance influenced our widget implementation is the Text widget. Our preferred hierarchy would be a simple Text class, implementing only the editing and selection actions (without scrolling options) and to implement ScrolledText as a Form consisting of a Scrollbar, optionally some Command buttons, and the Text primitive.

  This would give the user interface designer the maximum flexibility: by customizing the

ScrolledTextForm, taking advantage of the constraint-based geometry manager of Form, scrolling could be controlled by a variety of interaction devices.

It is desirable, however, for ScrolledText to have the semantics of a Text widget from the programmer's viewpoint, and so ScrolledText cannot be implemented as a trivial Form instance without knowing the internals of Text.

- The Athena widgets have been implemented without the aid of a graphic artist. We hope that the implementations and interfaces are amenable to 'dropping in' new display methods should the opportunity arise. As presented now, the visual appearance can be reasonably characterized as stark.

- An infinite variety of convenience widgets that are fixed composites of other widgets (like Dialog and Titlebar) are possible. Before implementors go too wild defining such things and adding them to the standard library, we need to implement an example of a widget that determines its content entirely through the Resource Manager. Then, when there is a resource editor, new composite forms can be built using non-programmatic methods and the widget library can be trimmed back to just the primitives.

  Whether composite widgets are separately written as new classes or built from a resource record by a 'ConfiguredForm' widget, there is the issue of how much access a programmer should have to the individual component widgets. Good programming practice tells us that even composite widgets should always be opaque. Realism tells us that programmers will always want to look behind the curtain.

- As will no doubt have become clear to the reader by this point, the Xtk does not (yet) aspire to be a full UIMS, for reasons discussed at the opening. It is our desire to see the Xtk as the lower level of a UIMS.

**Summary**

The M.I.T. X Toolkit, comprised of both general-purpose Intrinsics for widget construction and implementations of widgets for direct application use, provides highly extensible mechanisms for application user interface construction.

The X Toolkit layer sits above Xlib but below a full UIMS layer. By providing a consistent application programming interface to widgets, the Xtk allows independent development of an application and its associated user interface.

Each of the widgets in the Athena widget set leaves as much as possible configurable by the end-user of an application, while still providing reasonable and useful defaults. We expect that, through use, this basic widget set will be extended and improved.

All of the work described here, including complete source code, will be released by M.I.T. with the next release of the X Window System. The Massachusetts Institute of Technology, Digital Equipment Corporation, and others maintain copyright protection over all materials released by Project Athena with explicit permission granted for unlimited use, modification and redistribution for any purpose and without fee.

**Acknowledgments**

Many individuals have contributed to the design and implementation of the Intrinsics and of the Athena Widgets. To name them all is impossible. Special mention must, however, go to Charles Haynes, Phil Karlton, Tom Benson, Mike Gancarz, Harry Hersh, Mike Chow, Loretta Guarino-Reid, Rich Hyde, Kathy Langone, Mary Larson, Joel McCormack, Leo Treggiari, Jake VanNoy, Terry Weissman, Smokey Wallace, Susan Angebranndt, Ram Rao, Chuck Price, Al Mento, Peter Smith, Jeanne Rich (all from DEC); Frank Hall, Tom Houser, Ed Lee, Rick McKay, Fred Taft, Ted Wilson, Phil Gust (all from H-P); Jack Palevich (formerly H-P); John Osterhaut (U.C. Berkeley); Steve Pitschke (Stellar), Richard Carling (Masscomp); Ron Newman, Dan Geer, Bill Cattey, Russ Sasnett, Steve Wertheim, Stewart Hou, Chris Peterson (M.I.T. Athena); and Matt Hodges (DEC and M.I.T. Athena). Each of

the above has made significant contribution to the X Toolkit distribution.

**References**

1. *X Window System Protocol*, M.I.T. Software Distribution Center (September, 1987).
2. R. W. Scheifler and J. Gettys, ‘‘The X Window System,’’ *ACM Transactions On Graphics* **5**(2), pp. 79-109 (April, 1987).
3. J. Gettys, R. Newman, and R. W. Scheifler, *Xlib - C Language Interface*.
4. R. Rao and S. Wallace, ‘‘The X Toolkit,’’ in *Usenix Conference Proceedings*, D.E.C. Western Software Laboratory (Summer, 1987).
5. *X Toolkit Intrinsics*, M.I.T. Software Distribution Center (September, 1987).

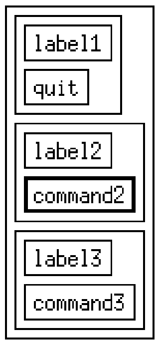


Figure 1

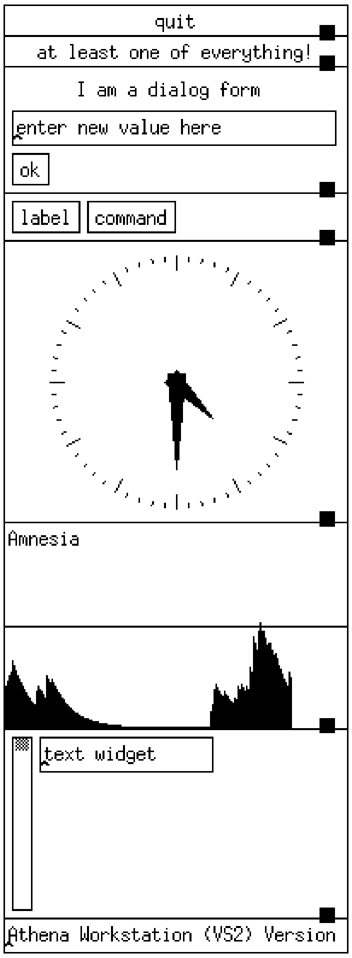
quit
at least one of everything!
I am a dialog form
enter new value here
ok
label
command
Amnesia
text widget
Athena Workstation (VS2) Version

Figure 3