Astronomical Data Analysis Software and Systems IV
ASP Conference Series, Vol. 77, 1995
R. A. Shaw, H. E. Payne, and J. J. E. Hayes, eds.

A Portable GUI Development System—The IRAF Widget Server

D. Tody[1]

National Optical Astronomy Observatories, P.O. Box 26732, Tucson, AZ 85726

Abstract. We describe a new GUI (Graphics User Interface) development environment which extends the X window system and the X Toolkit (Xt) with a high-level, object-oriented, interpreted programming language. This approach will allow GUIs written using standard Xt-based widget sets to be constructed without requiring any window system programming on the part of the programmer. The architecture of the resulting program completely separates the GUI from the applications code, allowing the GUI to be developed separately and replaced at will without modifying the application. Despite the separation of the GUI from the application the two are tightly integrated using an asynchronous, event-driven messaging system based on requests and client events, with the remote client application appearing as just another class of object within the GUI. This approach maximizes window system and toolkit independence and is well suited to distributed applications, allowing the GUI and client application to be run easily on separate processors or computers.

1. Introduction

Most window system programming today is done via one of two approaches. The first approach is to code directly in C or C++ at the window system toolkit level, usually integrating the user interface code and applications code within the same program. The second approach, which is really just a variation on the first, is to use one of the many commercial GUI builders available. This simplifies the programmer's task by allowing the user interface to be interactively designed, relying on the GUI builder to generate the window system code required to implement the user interface specified by the programmer and requiring the programmer to code only the application functionality. As with the direct coding approach, the applications code is often integrated with the GUI in the same program.

Recently a third approach has been used. This employs a high-level, interpreted language in which the programmer codes all or part of the application, with at least the GUI being implemented in the high-level interpreted language.

---

[1]The National Optical Astronomy Observatories are operated by the Association of Universities for Research in Astronomy, Inc. (AURA) under cooperative agreement with the National Science Foundation.

Tcl/Tk is a recent example of this approach. Other examples of the high-level interpreted approach from outside the Unix/X world include Apple's HyperCard/HyperTalk, and the Visual Basic facility of Microsoft Windows.

This paper presents a new, high-level, interpreted window system toolkit which like Tcl/Tk is also based on the Tcl interpreter. Unlike Tcl/Tk, which is by definition Tk specific, our approach tries to maximize window system and window system toolkit independence to ease future upgrades to new window systems or toolkits. The initial implementation, which is for the X Window System, is based on the X Toolkit (a part of the X Consortium X11 release) and uses standard Xt-based widgets. Support for an asynchronous, event-driven messaging system is an integral part of the design, allowing the GUI to be isolated from the functional part of an application and making the facility well suited to distributed applications, e.g., where a GUI executing locally talks to application code executing remotely, with the application downloading the GUI to be run at startup time. The core facility is implemented as a simple C-callable library which can be used to implement new GUI-based programs without having to write any window system-level code.

An important aspect of the facility described in this paper is that it represents a general user interface management system (UIMS) tailored for astronomical GUIs. The intent is that by providing a high-level facility tailored for our applications, we can simplify the task of developing GUIs for astronomical applications while providing a greater degree of consistency between applications since they will share the same GUI components. This is particularly important in the area of 2D graphics and imaging, including presentation and user interaction with such data, since the standard toolkits do not address this area.

2. The Widget Server

2.1. Overview

"Widgets" (window objects), are the basic building blocks of graphical user interfaces. A window system toolkit provides a selection of widgets that the programmer can use to construct a user interface. Typical widgets include things like push buttons, scrollbars, pop-down menus, or scrolling text regions. The *widget server* serves up widgets to a remote client process in much the same way that the X display server serves up windows and other low-level display resources to a remote client application.

When a client application starts up it connects to the widget server and downloads its GUI. The GUI is a simple block of text which is interpreted and executed by the widget server. The GUI text contains a description of the widgets comprising the user interface, a number of interpreted action procedures to be called to process user interface or client events during execution, and any code needed to initialize the GUI. During execution the client application waits for and executes requests from the GUI, and sends messages to the GUI to inform it of any "client events," or changes in the state of the client. The conventional compiled client application implements all the application-specific functionality, but does not communicate directly with the user.

To the client application the GUI is merely a block of text to be passed on to the widget server; the client code knows nothing about the GUI other than

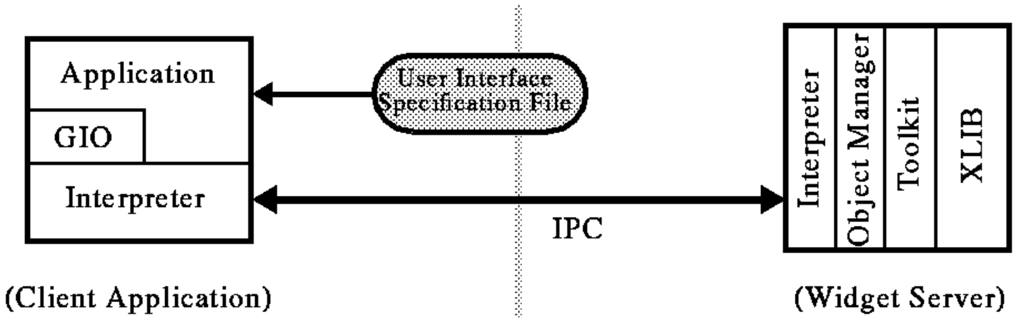

Figure 1. Widget Server Architecture.

the name of the GUI file to be sent to the widget server. The client knows only about the applications functionality which it implements, and the messages and requests used to communicate with the GUI. The GUI is completely isolated from the client application. While to the user the GUI appears to be an integral part of the application, the actual compiled client application has an interpreted command-line interface and can be executed stand-alone without any GUI.

2.2. Widget Server Architecture

The architecture of an application which uses the widget server is summarized in Figure 1. The typical widget server-based application consists of two processes, the process containing the client application, and the widget server process itself which executes the client's GUI. All user interface functionality resides in the GUI and all application-specific functionality resides in the client. During execution these processes communicate via an asynchronous object-based messaging system.

2.3. Advantages of the Widget Server

The widget server architecture separates the user interface from the application-specific code and provides a high-level interpreted language for developing GUIs. This approach has significant advantages, including the following:

1. The high-level, interpreted nature of the widget server makes it much easier to develop GUIs than is the case with toolkit-level programming or the conventional, compile-link approach.

2. The use of an interpreted runtime language (Tcl) to compose the GUI is more powerful than the visual programming approach used in GUI builders, since the latter only address the appearance of the user interface. User interface builders can still be used with the widget server to interactively layout the user interface, although this is less important than it might be given the interpreted nature of the GUI (i.e., changes can be made and the GUI redisplayed very quickly).

3. The high-level, integrated nature of the widget server-based GUI development environment makes it straightforward to customize the environment to support a particular class of application. This is important for large

systems where GUIs are developed by many people working independently, to reduce the overall effort and improve consistency.

4. The widget server isolates all window system and toolkit code into a single executable which can serve any number of applications. This greatly reduces disk space consumption in a large system that has many application GUIs.
5. Since the widget server is a single executable it is easy to have multiple versions, e.g., supporting different toolkits or incorporating proprietary software to optimize performance for a particular class of workstation.
6. Since all the window system code is isolated into a separate process the client application is completely window system independent, allowing the same client application to be used with a widget server GUI executing on any local or remote platform running any operating system.
7. Porting a whole system full of GUIs to a new platform can be done by a single individual since only the widget server itself need be ported.
8. No window system libraries, and indeed no compilation, is needed to develop GUIs. All that is needed is the widget server executable. This makes it much more feasible to use commercial or platform specific libraries should this be desired.
9. The GUI is completely isolated into a small text file separate from the application, allowing the GUI and the application to be developed separately, or several alternate GUIs to be used with a single application.
10. The widget server is well suited to distributed applications since the widget server can be run on the local workstation while the client application executes remotely. This allows the entire GUI to execute interactively on the local workstation, a much more efficient approach than, for example, running an X application over the network.

Possible disadvantages of the widget server approach are its relative complexity and the possibility of inefficiency when the application is distributed over two or more processes. For a small system where only a few GUI-based applications are needed many of our big-system concerns are unimportant and it might be simpler to program directly at the toolkit level, especially if a user interface builder tool is available. Efficiency can be a problem if the client code is required to respond in real time to user interface events, however this is rarely a problem in well designed applications since the more interactive portions of the program can be moved into the GUI, implemented as interpreted GUI procedures or as calls to the compiled functions in the widget server itself. The asynchronous nature of the messaging system ensures that the user interface will always be responsive even when the client is busy computing.

2.4. Platform Independence

The widget server automatically provides a high degree of platform and window system independence since the GUI is isolated from the client application; in the

```
reset-server

appInitialize hello Hello {
    Hello.objects:\
        toplevel        Form        helloForm\
        helloForm       Label       helloLabel\
        helloForm       Command     quitButton

    *helloForm*background:                bisque
    *helloForm*helloLabel.label:          Hello, world!
    *helloForm*quitButton.fromHoriz:      helloLabel
    *helloForm*quitButton.label:          Quit
}

createObjects
proc quit args { send client gkey q; deactivate unmap }
send quitButton addCallback quit
activate
```

Figure 2. A Simple GUI: The "Hello, world" Application

worst case only the GUI file has to be changed to use a GUI-based application on a new platform. The current implementation provides full platform independence for platforms which run the X Window System since the current widget server implementation is X-based. No changes to the GUI files are needed for these platforms.

The current widget server implementation does not, however, provide full window system toolkit independence for window systems other than X. Ideally the widget server should define a virtual set of widgets which can be implemented on a variety of window systems and window system toolkits; not only X but also Windows, Windows NT, Macintosh, etc. This problem has partially been solved in that the language used in widget server GUIs isolates the widget-dependent code into a portion of the GUI which describes the widget hierarchy, assigning widget classes to named GUI objects. The runtime part of the GUI, i.e., the interpreted action or callback procedures called at runtime as the GUI executes, is already almost completely widget- and toolkit-independent. Defining a fully toolkit-independent virtual widget set is a future problem which cannot be attempted until we have a better idea what widgets are needed for our applications. Several commercial window system toolkits or GUI development environments exist which have already attempted to address this problem, at least for the standard toolkits.

The portion of the widget server which interfaces to the underlying window system and window system toolkit (widget set) is the *Object Manager*. This is discussed in the next section.

3. The Object Manager Library

3.1. Overview

The widget server is actually just a shell around the Object Manager library (OBM). The widget server extends the Object Manager by providing a way for external clients to connect to the Object Manager to download and execute a GUI. All of the real work of creating and executing the GUI is done by the Object Manager library. The widget server adds a client-server communications method.

The Object Manager provides services for creating, deactivating, reactivating, or destroying a GUI, creating or destroying objects, and delivering messages and events to objects within the GUI. The OBM provides the framework within which GUIs execute, including the interpreter, automatic memory allocation, and a library of runtime services. The Object Manager defines four main classes of objects: *Server*, *Client*, *Parameter* (for client events), and *Widget*, for the graphical elements of the interface. Within the Widget class are many subclasses, one for each type of widget. All Object Manager execution is event driven and asynchronous and is based on messages (requests), callbacks, and events. For example, defining a new GUI is done by sending a message to the server object.

The set of widgets implemented by the Object Manager is not fixed, i.e., new widgets can be added or existing widgets removed to meet the requirements of the applications which will be using the widget server. The base Widget class provides a generic set of methods usable with all widget subclasses. Complex widgets subclass the base Widget class to add their own methods. The current Object Manager provides a mixture of Xt-based widgets which provide a Motif-like appearance but which are publically available and redistributable. Source for these widgets and for all code used in the widget server is included in the distribution. The current widget set includes the base Xt widgets, the 3D Athena widgets, selected FWF (Free Widget Foundation) widgets, plus a few others such as the Layout widget from MIT, the HTML hypertext markup language widget from NCSA *Mosaic*, and the gterm-image widget from the IRAF project. Additional Xt-based widgets (including Motif, OLIT, and commercial widgets) can easily be added.

3.2. The OBM Library

The main entry points of the OBM library are shown in Figure 3. The library is very simple since everything complicated is done by the interpreted GUI code. The main runtime function of the OBM library, from the point of view of the application which uses the library, is messaging. A window system application (such as the widget server) calls *ObmDeliverMsg* to deliver a message from the client application to a GUI object. A callback function is registered with the Object Manager to intercept client requests and pass them on to the client.

3.3. Using the OBM Library to Build Standalone Applications

Our discussion has thus far concentrated on the widget server and distributed applications, because the widget server provides the best architecture for adding GUIs to tasks in an existing, large data processing system. Another important

```
           obm = ObmOpen (app_context, argc, argv)
                ObmClose (obm)
           ObmInitialize (obm)
             ObmActivate (obm)
           ObmDeactivate (obm, unmap)
  status = ObmDeliverMsg (obm, object, message)

     id = ObmAddCallback (obm, callback_type, fcn, client_data)
       ObmRemoveCallback (obm, id)
```

Figure 3. Principal routines of the Object Manager library, *libobm.a*

class of applications are window system applications, where the focus is on the window system functionality implemented by the application. Most conventional X window system applications fall into this class.

An important use of the Object Manager library is to implement such applications. A stand-alone, single-process window system application can be built using the OBM library. In this case the "client" is not a separate process, but an application-specific interpreter within the same process as the OBM library. The program could be written as a conventional window system program making direct calls to the underlying window system toolkit, but by using the OBM library virtually all window system specific code is eliminated and the GUI is isolated to a high-level, interpreted GUI file. The only compiled code required is that which implements the functionality of the application itself. The resulting task is almost completely window system independent.

A good example of an existing stand-alone window system task built around the OBM library is *ximtool*, the IRAF image display server program. This is a stand-alone X window system application used for image display and image interaction. *Ximtool* contains only about one page of C code which has anything to do with X. All of the remaining C code handles window system-independent raster image processing and the fifo or socket-based binary protocol used to communicate with remote clients for image display. The *ximtool* GUI is an interpreted GUI text file, identical to what one might use with a widget server-based task. The widget server itself, of course, is another example of a stand-alone X application based on the OBM library.

3.4. The Object Manager Shell

Another stand-alone host application built around the Object Manager library is *obmsh*, the Object Manager shell. This is a Unix shell which executes OBM windowing scripts. It can be used to execute GUI files from the Unix command line, or be used in OBM-based scripts to write stand-alone Unix shell scripts that can be called as commands from the Unix environment. For example, the "hello, world" GUI shown in Figure 2 could be converted to a Unix command *hello* by changing the file name to "hello" and adding something like "`#!/usr/local/bin/obmsh`" to the file header.

4. Messaging

4.1. Messaging Fundamentals

The key to isolating the GUI from the client code of an application, while providing a tightly integrated, efficient application, lies with messaging. The messaging scheme used determines how objects within the application interact with each other during execution. This includes the interaction of the client code (client object) with the GUI. Our discussion here will concentrate on how messaging is used to link the client to the GUI, but it should be noted that the same messaging scheme is used for all object-to-object communications within the GUI as well.

Messaging as defined by the OBM consists of two parts: requests and client events. Requests are commands send to the client (or any other object). The recipient is free to modify or ignore the request as it wishes. Client events are messages sent to the GUI when the client state changes in any way. The same mechanism is also used to deliver other forms of client information to the GUI, e.g., in response to requests. The GUI is free to ignore client events. It is not unusual for a given GUI to be interested in only a portion of the client events generated by a client.

A client event is a message sent to an object of the OBM class *Parameter*. A parameter object is very simple, consisting of a name and a string value. The GUI registers callbacks with the user interface (UI) parameters, i.e., client events, that it wishes to know about. The string value of a UI parameter can be anything, for example a number, a structure, a list, or a large block of text. It is common for multiple callbacks to be registered on a single UI parameter by independent elements of the GUI.

Messaging is fully asynchronous. Both requests and client events are queued and buffered, and periodically flushed to the process on the other end. Synchronization occurs automatically when the client waits for input from the server (GUI). The GUI never waits for a request to complete, nor does it check to see that a request has been honored. Rather, when the client processes the request it sends client events back to the GUI to inform the GUI of any actions performed by the client. The same thing happens when the client performs actions for any other reason, hence the GUI always reflects the true state of the client.

Client events are an important abstraction mechanism. Client events and messages allow the client to provide the GUI with all the information it needs to function, without the client code having any knowledge of the nature of the user interface. Yet, since requests and client events are decoupled, the client will function even if the client events and messages it sends are discarded, as when running the client code without a GUI or with radically different GUIs.

4.2. Simple Messaging Example

Messaging is one of those things which is fundamentally simple, yet surprisingly hard to explain. As a simple example, consider what happens when the user selects a frame to be displayed in *ximtool*:

1. The user pushes the next frame button.

2. The callback procedure (in the GUI) for the nextFrame button sends the command “nextFrame” to the client.

3. The client receives and processes the request, changing the frame, sending the new frame number to the UI parameter “frame”.

4. A GUI callback procedure registered on the “frame” parameter is called, updating the GUI to indicate the new display frame number.

This example simplifies things considerably but is accurate so far as it goes. In the real program there are a number of different ways the frame can be changed, e.g., by the next frame or previous frame buttons, by menu selection, keyboard accelerators, the blink timer, IRAF running in another process, and so on. All of these end up sending a request to the *ximtool* client which directly or indirectly results in a frame change. When the display frame is changed a number of client events are generated to inform the GUI not only of the new frame number, but also the new frame title, zoom, pan, and frame flip values, type of enhancement used for the frame, and so on. Each of these items represents a separate client event. Although the action of the program may be arbitrarily complex in real world examples, the basic messaging mechanism on which this is all based remains very simple.

5. Software Products

5.1. The X11IRAF Package

All of the software described in this paper is packaged in a single distribution called *x11iraf*. This includes *xgterm*, *ximtool*, the Object Manager library, sources for all the third party widgets used in OBM, and assorted demo applications. Everything needed to build the package is included, including compatible versions of some publically available libraries, e.g., Tcl and Xpm. Despite the name the software is not tied to IRAF in any way, other than that it is a product of the IRAF project and is used for IRAF GUIs.

5.2. Xgterm

Xgterm is an upwards compatible version of the popular *xterm* with the *xterm* graphics ripped out and replaced with an OBM-based GUI which uses the gterm-image widget for graphics. The graphics supports a number of extensions, including full color support, an integrated imaging capability, dialog interaction, intelligent unconstrained resize, and a full crosshair cursor. Although *xgterm* is often used as a simple terminal emulator it is also a general widget server since it contains the full OBM library, and it can be used to execute arbitrarily complex OBM-based GUIs and manage the communications with the remote client. The current version of *xgterm* is based on the X11R6 version of *xterm*.

5.3. Ximtool

Ximtool is an image display server, used by remote client applications such as IRAF to display and interact with images. Several frames can be loaded and independently displayed in a full-frame or tiled configuration. Display frames can be any size. *Ximtool* is a good example of a conventional single process windowing application which uses the OBM library for the GUI and the window system interface.

5.4. The Object Manager Library

The Object Manager library (OBM) is a high-level, interpreted window system toolkit that is used to implement arbitrary graphics user interfaces. The OBM library uses Tcl as the interpreter. The current version of the OBM library is based on the X toolkit and can be used with any Xt-based widget or widget set.

5.5. The Gterm-Image Widget

The gterm-image widget is an X Toolkit-based widget for general 2D graphics and image display. This is a complex widget and a full description of its capabilities is beyond the scope of this paper. The Gterm widget provides a general GKS-like vector graphics and text display capability. An integrated image display capability allows any number and size of image rasters to be created within the widget or in the X server. *Mappings* can be defined to map one raster to another, permitting general graphics pipelines involving scaling and other geometric transforms to be set up. Colormap support is included for grayscale and pseudocolor rendering of raster data. An interactive *graphics marker* facility is provided for interaction with the displayed graphic. The Gterm widget is used for all graphics and imaging in *xgterm* and *ximtool*.

6. Adding GUIs to IRAF Applications

A major application of the widget server and the other software described in this paper is in adding GUIs to IRAF applications. In this case the IRAF task is the client: when the IRAF task starts up it downloads its GUI to the widget server, and during execution the IRAF task and the GUI communicate via the messaging facility described earlier. The changes required to add a GUI to an IRAF task are minor, ranging from changing a single line of code to cause the GUI file to be downloaded, to defining a set of client events and adding *gmsg* calls to allow more complete integration of the GUI. Adding a GUI to a task increases the system size by only the 10 Kb or so required for the GUI file.

7. Availability and Further Information

Further information on the software described in this paper, including more detailed documentation, full sources, and executables for a variety of platforms can be found in the X11IRAF Web page[1]. Further information on IRAF itself and other IRAF products can be found in the IRAF archives[2]. Documentation on Tcl, Xt/XLIB and the other standard software products used in X11IRAF is available from many sources.

---

[1] http://iraf.noao.edu/x11iraf

[2] http://iraf.noao.edu