IRIS ViewKit™ Technical Report

Doug Young

Silicon Graphics

6/14/94

Abstract

This paper describes IRIS ViewKit , a C++ application framework designed to simplify the task of developing applications based on the OSF/Motif™user interface toolkit[1]. IRIS ViewKit framework promotes consistency by providing a common architecture for applications and improves programmer productivity by providing high-level, and in many cases automatic, support for commonly-needed operations. In addition to providing facilities normally associated with a graphical user interface, the framework serves as a central integration platform for other facilities applications typically need. These include support for inter-application communication, integration with the Silicon Graphics Indigo Magic desktop, and automatic on-line context-sensitive help. The IRIS ViewKit framework is currently in wide use within Silicon Graphics and serves as the basis of all CASEVision™ software development products, the IRIS InSight™ on-line documentation product, the Indigo Magic™ Desktop, the InPerson™ desktop conferencing system, and many other products. IRIS ViewKit can also be used with the IRIS Inventor™ library, which is based in part on the ViewKit model.

Introduction

One of the problems faced by application developers is the lag time between the development of various software layers that all applications depend on and the ever-increasing needs of application software. Consider for example, the contemporary user interface. The basic paradigm of multiple windows displayed on a bitmap screen, along with a keyboard and mouse input devices, was developed at Xerox PARC in the late 70's and early 80's. Apple made the first successful attempt to commercialize this model with the Macintosh® computer, and also provided one of the first widely-used user interface toolkits to support development of applications to run in such an environment. The Macintosh toolbox provided a small set of user interface gadgets, such as buttons, scrollbars, and menus, from which an application could create a graphical interface. The needs of software development have changed considerably over the years, but the user interface toolkits that support current software development remain strongly rooted in the model provided by the original Macintosh toolbox.

In recent years, applications with interactive user interfaces have become more difficult to develop because the expectations placed on such applications have grown dramatically. An application was once considered "user friendly" if it provided a row of mouse-selectable buttons instead of, or in addition to, a command-line interface. Today, applications are expected to conform to much higher standards of excellence. Good contemporary user interfaces provide features such as context-sensitive on-line help, support error recoverability, provide the ability to undo the application's most recent actions, and so on. Contemporary interfaces may involve hundreds of user interface *widgets*, and support multiple real-time views of various types of underlying information. With new advances in hardware, modern applications routinely support audio, integrated video, and even speech input. Applications are generally integrated with a desktop manager, provide drag-and-drop interfaces, and support inter-application communication and data exchange. Combining these pieces to create a complete application stretches the model of most current user interface toolkits to their breaking points.

---

[1]. OSF/Motif is a trademark of the Open Software Foundation.

In addition to contending with advanced expectations from users, developers also have to deal with appearance and behavior guidelines. Most major vendors have begun to specify user interface guidelines that all applications are expected to follow. These guidelines generally take the form of a checklist of rules to which an application must conform. Often, no existing toolkit provides direct support for these rules, because the guidelines were developed independent of, and possibly later than, any available toolkits. For example, the OSF/Motif Style Guide forms the basis of many vendor's user interface standards.The guide specifies that applications should provide context-sensitive on-line help and specifies in some detail how help is supposed to work from a user's perspective. Yet the Motif toolkit provides only rudimentary support for such a system.

Developers who try to conform to the requirements of such style guides often find themselves facing an overwhelming task. Furthermore, asking each individual developer to implement these features from scratch defeats the very purpose of such guidelines. Programmers will inevitably introduce minor variations into each implementation, even if everyone agrees on the interpretation of the rules. Ideally, support for stylistic guidelines should be provided by a user interface toolkit as automatically as possible. Unfortunately, the commonly used piece-meal approach of providing a collection of low-level widgets cannot adequately address this need. Many style issues have broad implications on the architecture and design of applications that individual widgets cannot address.

The Motif User Interface Toolkit

Currently, the OSF/Motif toolkit is the most widely-accepted user interface toolkit available for UNIX®-based systems. This toolkit offers a rich supply of widgets, (buttons, scrollbars, menus, etc.) that can be used to construct an application's user interface. The Motif toolkit provides several key benefits to developers. Because the Motif toolkit is both available and recognized as a standard across nearly all UNIX platforms, it gives developers the ability to write software that runs on all vendor's versions of UNIX. The Motif toolkit also provides some features that have been lacking in earlier toolkits. For example, the Motif toolkit provides strong support for writing internationalized software. It also provides a distinctive appearance, support for various interaction styles, cut-and-paste, user-customization, and more. The widgets it provides are extremely flexible, allowing developers to mold the toolkit to suit their needs.

From a developer's perspective, however, the Motif toolkit offers a programmer's model that has changed little from the earliest user interface toolkits. Applications must construct interfaces from very low-level elements that provide only limited support for tasks applications need to perform. Buttons, scrollbars, and rendering areas represent a necessary and useful minimum feature set for an interactive application. However, the availability of such individual building-blocks falls short of enabling developers to create a complete application. The Motif toolkit provides the raw elements for constructing a user interface. Unfortunately, complete applications require much more than a collection of widgets.

Application Frameworks

Fortunately, there is a relatively recent development that has been more successful at meeting the real needs of developers. This new approach revolves around an architectural model known as an *application framework*. Application frameworks have been used successfully on Macintosh and personal computers, but have been less common in the UNIX environment. The concept behind an application framework is simple: instead of providing a library that contains small pieces from which an application can be built, an application framework implements an application that is fully functional, except for those parts that are application specific. Programmers who use such a framework can start their development with an application that already fully implements as many generic features as possible. Such features might include support for on-line help, support for audio input and output, automatic integration with the desktop or other applications in the environment, and so on.

Most application frameworks rely on object-oriented technology to provide as many basic services as possible without limiting the developer's ability to modify the behavior of the basic application model. Because of the way in which frameworks typically work, application frameworks have been described as "upside-down libraries." When using traditional user interface toolkits, applications implement the control flow of the application and call functions provided by the toolkit when various toolkit-level services are needed. In an application framework, a significant portion of the control flow and application-level logic is contained in the library. The library calls functions provided by the application when application-specific behavior is required. Generally, these hooks are supported through the use of inheritance. The framework provides a collection of classes that implement the default behavior expected of all applications. Applications modify these classes to support application-specific behavior by creating subclasses that override the default behavior of the framework, as needed.

One problem with the application framework model is its tendency to be less flexible than the traditional toolkit approach. Any given application framework generally provides strong support for a certain style of application, but provides only weak support for applications that fall outside the domain for which the framework was designed. In the worst case, an application framework's design may interfere with an application's desired behavior.

The IRIS ViewKit Application Framework

The *IRIS ViewKit* application framework makes it easier for programmers to develop applications that require a graphical user interface. IRIS ViewKit is not a stand-alone library, but instead leverages the Motif user interface toolkit to provide a complete solution that is more flexible than a typical application framework, but more powerful than using only the Motif toolkit. When developing with IRIS ViewKit, the framework provides a high-level application architecture along with pre-packaged solutions to common needs. At the same time, developers who require more flexibility always have full access to the Motif user interface library, with all its low-level widgets.

IRIS ViewKit is written in C++ and provides strong support for an object-oriented application architecture that works well with the Motif toolkit. The ViewKit library supports a strongly stylistic approach to using the Motif widget set, based on the usage patterns of experienced Motif programmers. Developers who are new to Motif programming will find themselves automatically guided to an architectural approach that is straight-forward and flexible, and that supports both small and large applications.

Because IRIS ViewKit emphasizes a particular application architecture, the ViewKit framework is best suited for new development. It may be difficult to move existing applications to IRIS ViewKit, but those beginning new development will find distinct advantages to using the IRIS ViewKit architecture.

Developers who are familiar with C++ will find that IRIS ViewKit offers a familiar and comfortable object-oriented architecture for writing applications while allowing them to take advantage of the standard user interface facilities provided by the Motif library. Programmers who have used the Motif widget set will find that the ViewKit framework performs tasks automatically that they have had to implement themselves in the past.

One of the strengths of the ViewKit library is its support for small details that are often neglected when using the Motif widget set. For example, many applications need to perform certain operations that prevent the application from interacting with the user while the operation is in progress. The accepted behavior in such situations calls for the application to display a busy cursor if the period of time is expected to be brief. For longer periods of time, a dialog may be posted, possibly providing occasional feedback about the progress of the task. In X applications, it is important to also be sure input is blocked during the period in which the application is too busy to handle events. If an application supports multiple windows, input needs to be blocked and a busy cursor displayed in all currently visible windows. Ideally, if the operation is lengthy, the application should allow the user to interrupt the task, as well.

From a user's perspective, such behavior is simply expected. However, when using the Motif toolkit, implementing these features for all situations in which an application may be busy performing some task is sufficiently difficult that many applications do not support this behavior. IRIS ViewKit provides extremely easy-to-use support for such situations. Programmers simply call a function to indicate that the application is entering a busy region and another to indicate when the application is no longer busy. Calls can be nested, and the same function can be used to display an optional status message. The ViewKit framework handles the details of creating busy cursors, displaying them in all windows, disabling input to all windows, creating and displaying dialogs if necessary, and so on. This simple feature alone can save a programmer hours, if not days of work. Allowing operations to be interrupted is trickier, and necessarily requires some cooperation from the developer. However, the ViewKit framework provides as much support as possible to make interruptible operations easy to implement.

The Silicon Graphics Development Environment

Silicon Graphics provides a complete collection of facilities to help programmers take advantage of the unique capabilities of the SGI platform. Some of these facilities take the form of assorted libraries that perform various tasks, while others are simply stylistic guidelines. For example, SGI supports an extension of the *Motif Style Guide*, and also builds on the ICCCM protocols for data exchange between applications. IRIS IM, SGI's implementation of the Motif user interface toolkit, provides a unique look and feel for SGI applications. The SGI platform also provides an extended set of Motif-style widgets that can be used to develop applications, and supports various high-level services, including a help system and network licensing.

IRIS ViewKit coordinates these separate pieces using an integrated set of classes that supports complete SGI-style applications with as little work as possible. For example, even the simplest IRIS ViewKit application is fully integrated with SGI's on-line help system. Programmers do not have to write a single line of code to provide full on-line help for a IRIS ViewKit application. Only the content of the help messages need to be provided.

Figure 1 and Figure 2 illustrate the difference between the architecture of applications developed using the low-level libraries supported on the Silicon Graphics platform and the architecture of an application based on the IRIS ViewKit framework.

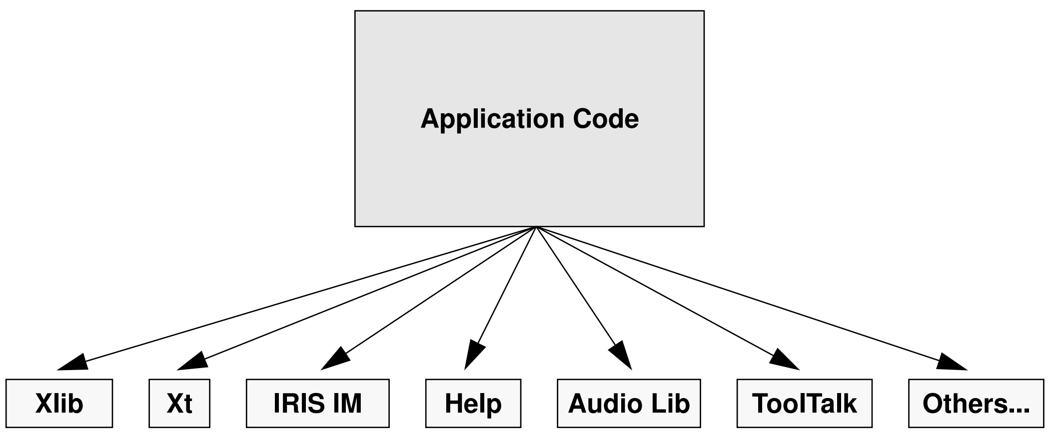

Figure 1. Traditional application architecture.

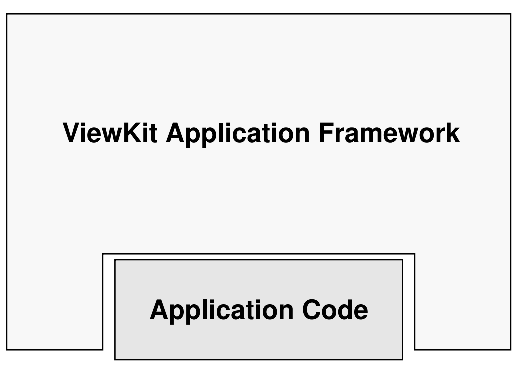

Figure 2. The IRIS ViewKit "upside down library" architecture.

Major IRIS ViewKit Elements

For purposes of discussion, the ViewKit framework can be divided into several logical groups of classes. The following sections discuss some of these logical groups.

Core Framework Classes

IRIS ViewKit provides a small set of classes that are essential for any application or that provide fundamental support for other classes in the framework. The most basic of these classes is the VkComponent class. This abstract base class supports the concept of a user interface *component*, as described in the book *Object-Oriented Programming with C++ and OSF/Motif*. (Prentice Hall, 1992). All user interface classes in the IRIS ViewKit library are derived from the VkComponent[2] class, which defines the minimum protocol for all components.

The core framework classes also include support for features needed by nearly all applications. These classes include VkApp, an application class that must be instantiated by every IRIS ViewKit application, classes that support top-level windows, a collection of classes that support menus, and several classes that support dialog management. All classes are designed to fully implement as many typical features as possible. For example, all top-level windows and dialogs handle the window manager quit/close protocol. Dialogs are cached to balance memory use and display speed. The menu system goes beyond simply constructing menus to support dynamically adding, removing, replacing items, and more.

The collection of classes that make up the core part of IRIS ViewKit are closely integrated and work together to support essential features required by most applications as automatically as possible. Among the basic services supported by the core ViewKit framework are single and multi-level undo, support for performing interruptible tasks, and support for an application-level callback mechanism that allows C++ classes to dynamically register member functions to be invoked by other C++ classes.

Components

In addition to the basic user interface elements supported as part of the core framework classes, IRIS ViewKit provides an assortment of useful interface components that may be useful in some situations. These include a graph viewer/editor, an input field that supports name expansion, an outliner component for displaying and manipulating hierarchical information, and others. Some ready-made components supported by the ViewKit library are directly useful in specific situations. However, developers are encouraged to use the architecture of the IRIS ViewKit framework to create new components, or extend existing components.

Figure 1 shows a typical IRIS ViewKit high-level component. This component is a self-contained graph editor/viewer that can be used to browse or create an abstract graph representation consisting of a collection of interconnected objects. Nodes in the graph can be moved interactively. The controls along the bottom of the component allow the graph to be zoomed, the orientation to be changed, and other operations to be performed. An overview of the graph can also be displayed for large graphs. Although the component described here represents a complex sub-system, the entire component can be included in an application simply by instantiating a single C++ class, and using the well-defined interface supported by that class.

2. The VkComponent class implements an extended superset of the component protocol described in *Object-Oriented Programming with C++ and OSF/Motif*, Prentice Hall, 1992. The underlying concepts are the same in each case, although the implementations differ.

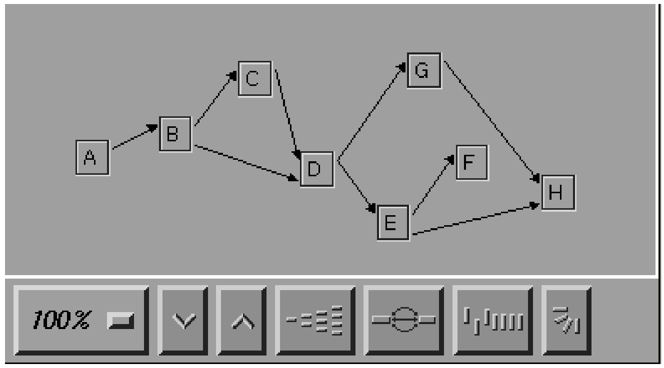

Figure 3. A typical high-level IRIS ViewKit component.

Creating high-level components such as the graph viewer shown in Figure 3 promotes consistency throughout a set of applications by providing elements that users can learn once and then easily recognize in multiple applications.

Convenience Utilities

IRIS ViewKit provides various utility functions and classes that are used throughout the framework. These utilities, which are available to developers as well, include simple functions that make it easier to load resources (including automatic type conversion), classes that support the use of icons, and other miscellaneous utilities.

One particularly interesting set of utility classes supports operations on groups of widgets. For example, VkRadioGroup allows an arbitrary collection of Motif toggle buttons to exhibit radio behavior, even if these widgets are in different locations in the interface or widget hierarchy. The VkAlignmentGroup class allows developers to specify a collection of widgets that should be aligned to the same width, height, x, or y position.

Process management

The UNIX system, as well as the Xt toolkit on which the Motif widget set is based, provides various low-level mechanisms for dealing with processes. A typical scenario involves starting an application from within an already running application and exchanging data between the two applications. IRIS ViewKit provides several C++ classes that make it easier to use UNIX processes in such situations. These classes handle the details of launching the sub-process, setting up pipes, getting information from the sub-process, and handling signals.

Help Support

IRIS ViewKit provides complete support for using the SGI on-line help system. Most of the time, a programmer who uses the ViewKit library does not have to do anything to support help, other than writing the content of the documentation. The ViewKit framework includes several classes that support on-line help within IRIS ViewKit applications, but these are seldom used directly by programmers.

Network Licensing

The NetLS licensing system supported on SGI and various other platforms provides low-level facilities to support license-protected applications. The IRIS ViewKit framework makes it easy for programmers to use license protection in a way that is consistent with the behavior of other SGI applications. To license-protect an IRIS ViewKit application, the application simply instantiates a VkNLS object, providing some simple information about the application to the VkNLS constructor. IRIS ViewKit automatically checks for license expiration at regular intervals, displays various dialogs when error situations occur, and closes down applications that are not properly licensed.

Preference Panels

Many applications support one or more preference panels that allow users to customize the behavior of the application. Such preference panels can be tedious to write because they may involve large numbers of text input fields, labels, toggle buttons, and so on. Users expect preference panels to work in a specific way, as well. Usually, users select a number of preferences and then select an “Apply” button or an “OK” button to apply all changes at once. Users also expect to be able to select “Cancel” and return all preferences to their previous state, regardless of how many changes the user may have made. Programming this behavior for every preference panel can be tedious at best.

IRIS ViewKit supports an easy-to-use collection of classes for building preference panels. Rather than dealing directly with Motif widgets, their placement, callbacks, and so on, programmers who use the ViewKit framework can simply create groups of *preference items*. These items maintain their own state, which allows an application to simply query each item to see if it has been changed. Layout is handled automatically, and the ViewKit library provides the ability to apply or revert all preferences to their previous state.

Inter-Application Communication

Few applications operate in a complete vacuum, and many applications need to be able to communicate with other applications to work effectively. IRIS ViewKit builds on the ToolTalk™ inter-application communications service[3] to support simple and effective inter-application communication. The ToolTalk system is based on a broadcast message model, in which applications send requests for services they need, as well as notices of actions they perform, to a central messaging service. (See Figure 4.) Applications also register with the message service by informing it of what types of notices and requests they are interested in. The message service routes all messages between interested parties without applications needing to be aware of what applications are running in the user’s environment.

IRIS ViewKit supports integration with the ToolTalk service in two ways. First, the ViewKit framework supplies a higher-level, easy-to-use interface to the ToolTalk library that supports a simple callback model, not unlike the Xt callback mechanism. Second, the IRIS ViewKit framework provides various abstract classes that handle many of the details of registering with the message service and handling messages automatically.

The ToolTalk system assumes that messages are handled on an application level (and in fact messages are always sent between processes). However, IRIS ViewKit also supports individual components that need to encapsulate messaging behavior. For example, imagine a component that allows the user to display and edit some data stored in a file. It is easy to imagine multiple applications using this component to browse or edit the same data simultaneously. Using the ToolTalk communication mechanism, the component can transparently coordinate access and display of the data between applications. For example, if the user edits the data in one view, that view can send notices to other instances of the component to keep them in sync. Because this behavior can be encapsulated in a component, applications that use the component can benefit with no effort on the part of the developer.

---

3. ToolTalk is a trademark of Sun Microsystems.

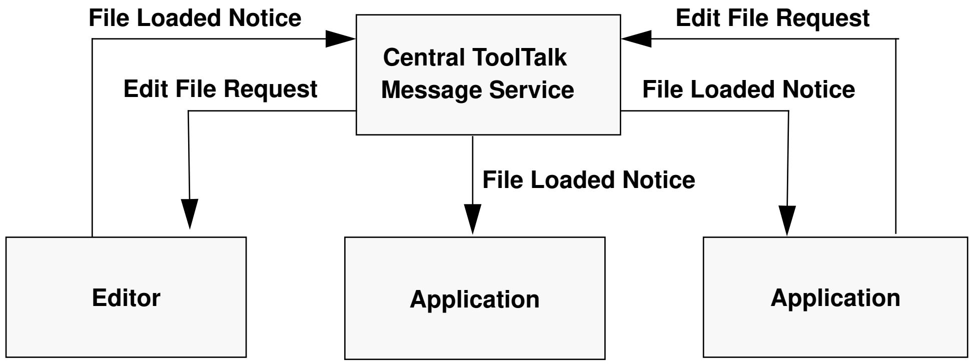

Figure 4. Using the ToolTalk mechanism to communicate between processes.

A IRIS ViewKit Generic Application

Like many application frameworks, IRIS ViewKit supports the notion of a *generic application*. A generic application is an application that fully conforms to the requirements of the environment in which it runs, and that lacks only the specific behavior of any given application. A generic application can be viewed as the common subset of all applications supported by a framework. As such, it provides the minimal behavior expected of all applications. It also provides a useful first look at what it is like to write a ViewKit application. The IRIS ViewKit generic application can be written as follows:

```
///////////////////////////////////////////////////
// Simplest possible ViewKit application
///////////////////////////////////////////////////
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>

void main ( int argc, char **argv )
{
    VkApp     *app = new VkApp("Generic", &argc, argv);
    VkWindow  *win = new VkWindow("generic");

    win->show();  // Display the window
    app->run();   // Run the application
}
```

The generic application instantiates an application object (VkApp), and a simple top-level window object (VkWindow). After displaying the window, the application runs. Figure 5 shows how the generic application appears on the screen.

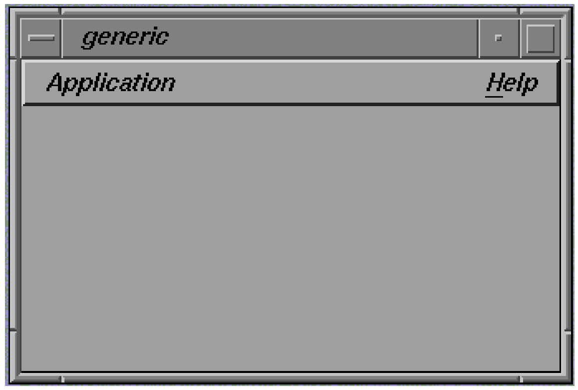

Figure 5. The IRIS ViewKit generic application.

The IRIS ViewKit generic application automatically supports on-line context-sensitive help, has ready-to-use services such as busy cursors and standard dialogs is integrated with the SGI desktop, and includes support for audio. The developer has only to add those elements that are application specific. IRIS ViewKit also makes it easy to extend the generic application to meet application-specific needs.

Adding Application-Specific Behavior

Developers can add application-specific functionality by extending the ViewKit library through subclassing. For example, the following simple program extends the generic application in two ways. First, instead of VkApp, this program creates an instance of VkMsgApp, a subclass of VkApp provided by the ViewKit framework. This class automatically handles the details of allowing applications to send and receive ToolTalk messages. Because this example instantiates a VkMsgApp object, the program can automatically receive and handle certain minimum ToolTalk messages, and it is easy to extend the application to allow it to send or receive additional messages.

This program defines some application-specific behavior by subclassing from the VkWindow class provided by the IRIS ViewKit framework. The MyWindow class inherits all the behavior of the VkWindow class, but adds a ViewKit component as a *view* to be displayed in the window. Subclasses of VkWindow can add any ViewKit component, any application-defined component, or any Motif widget as a view. This example adds an instance of the VkOutline class, which is a simple hierarchy browser included in the ViewKit library. The MyWindow class programmatically adds several items to the hierarchy, and then displays the entire outline.

The main program is similar to the generic application discussed earlier, but instantiates VkMsgApp and MyWindow objects to implement the desired behavior.

```
///////////////////////////////////////////////////////
// outline: exercise the ViewKit outline component
//////////////////////////////////////////////////////
#include <Vk/VkMsgApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkOutline.h>

class MyWindow : public VkWindow {

   public:
     MyWindow(const char *name);
};


MyWindow::MyWindow(const char *name) : VkWindow(name)
```

```
{
     // Create a VkOutliner component and add it as a view
     VkOutline *outliner = new VkOutline("outliner", (*this));
     addView(outliner);

    // Construct a hierarchy by adding parent/child pairs

    outliner->add("Heading 1",   "SubHeading 1");
    outliner->add("Heading 1",   "SubHeading 2");
    outliner->add("Heading 1",   "SubHeading 3");
    outliner->add("SubHeading 1", "Item 1");

    outliner->displayAll();  // Show the entire outline
}

void main ( int argc, char **argv )
{
    VkMsgApp  *app   = new VkMsgApp("Outline", &argc, argv);
    MyWindow  *win   = new MyWindow("outline");

    win->show();  // Display the window
    app->run();   // Run the application
}
```

Figure 6 shows this program as it appears on the screen. The various hierarchy levels can be collapsed or expanded by clicking on the arrows beside each heading.

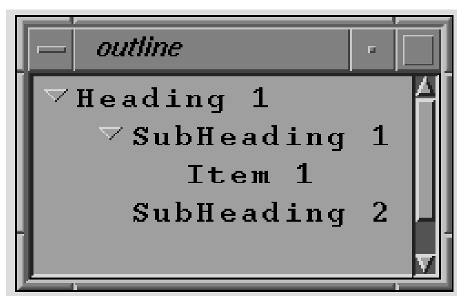

Figure 6. IRIS ViewKit application with application-specific behavior.

Improving Productivity with the IRIS ViewKit Framework

IRIS ViewKit provides many commonly-needed facilities as higher-level services. Without the IRIS ViewKit framework, these services would have to be implemented using the lower-level facilities of the Motif widget set, usually with much more difficulty, and a corresponding loss of productivity. Applications built using only a low-level library like the Motif widget set also tend to exhibit a lack of consistency, as well, because common features must be re-implemented for each application. For example, let's examine the steps necessary to add a quit command to a pane in the menu bar in the generic application described earlier. The quit command should first display a dialog that asks the user to confirm the command before actually exiting the application. Using the Motif toolkit, the following code segments would be required (note that only the code segments required to implement a menu and dialog are shown. A complete Motif application would require far more code.):

```
/////////////////////////////////////////////////////////////////
// Code required to add an Application pane containing
// a quit action to an existing menubar in an existing
// window, using the Motif toolkit.
//////////////////////////////////////////////////////////////////
```

```
  // Add an application pane to the menu

  pane = XmCreatePulldownMenu(menubar, "application", NULL, 0);

  // Add a cascade button to the menu pane and attach the pulldown menu pane

  cascade = XtVaCreateManagedWidget("application",
                                    xmCascadeButtonWidgetClass, menubar,
                                    XmNsubMenuId, pane,
                                    NULL );

  // Add a quit item to the pane

  quitB = XmCreatePushButton(pane, "quit", NULL, 0);
  XtManageWidget(quitB);

  // Add a callback to be invoked when the user issues the quit command

  XtAddCallback(quitB, XmNactivateCallback, safeQuitCallback, NULL);
```

The menu pane and quit entry have now been created, but we still need to implement the safeQuitCallback() function and related facilities. A “safe quit” mechanism could be implemented as follows, using raw Motif:

```
//////////////////////////////////////////////////
// Popup a dialog to ask the user to confirm
// a quit command before actually executing it.
//////////////////////////////////////////////////
void safeQuitCallback(Widget w, XtPointer, XtPointer)
{
     // Create a dialog

     Widget     dialog = XmCreateQuestionDialog(w, "question", NULL, 0);

     // Create the message to be displayed

     XmString xmstr  = XmStringCreateLocalized("Do you really want to quit?");

     // Set the dialog to display the message

     XtVaSetValues(dialog, XmNmessageString, xmstr,
                           XmNdialogStyle,   XmDIALOG_APPLICATION_MODAL,
                           NULL);

     // Free the compound string when it is no longer needed

     XmStringFree(xmstr);

     // Assign callbacks to be called when the user clicks on OK or Cancel

     XtAddCallback(dialog, XmNokCallback, reallyQuit, NULL);
     XtAddCallback(dialog, XmNcancelCallback, cancelCallback, NULL);

     // Handle the window manager close button

      Atom WM_DELETE_WINDOW = XInternAtom(XtDisplay(w),
                                          “WM_DELETE_WINDOW”, FALSE);

      XmAddWMProtocol(XtParent(dialog), WM_DELETE_WINDOW,
                      cancelCallback, NULL);
      XtVaSetValues(XtParent(dialog), XmNdeleteResponce, XmDO_NOTHING, NULL);

     // Display the dialog
```

```
     XtManageChild(dialog);
}

// This function is called if the user confirms the action

void reallyQuit(Widget, XtPointer, XtPointer)
{
    exit(0);
}

// If the user cancels the action, simply destroy the dialog widget

void cancelCallback(Widget w, XtPointer, XtPointer)
{
    XtDestroyWidget(w);
}
```

Programmers familiar with the Motif toolkit will recognize these code segments as typical of the level of detail they contend with every day. These code segments implement a single, trivial feature. Most applications support dozens, if not hundreds of such commands, and programmers who work with the Motif widget set or similar toolkits implement this type of feature over and over. The situation is actually worse than it appears, because the code segments shown here cut several corners that would not be acceptable for product-quality applications. For example, this code segment hard-codes the message "Do you really want to quit?" in English, preventing the application from being localized to some other language. The dialog positioning is not handled well, and there is also no provision for providing the same behavior for the quit command that the user can issue from the window manager menu. There is also no supported mechanism to allow other parts of the application to clean up before the program exits. The code simply exits.

IRIS ViewKit improves programmers' productivity dramatically by providing support for such common operations. For example, the following *complete* program adds the same functionality described above to the IRIS ViewKit generic application, but this time taking advantage of just a few of the IRIS ViewKit facilities.

```
//////////////////////////////////////////////////////////////
// Generic IRIS ViewKit application with a "safe quit" menu item
/////////////////////////////////////////////////////////////
#include <Vk/VkApp.h>
#include <Vk/VkWindow.h>
#include <Vk/VkSubMenu.h>

// Provide the function to be executed to really quit

void quitCallback(Widget, XtPointer, XtPointer)
{
    theApplication->terminate(0); // Clean shut down function
}
void main ( int argc, char **argv )
{
    VkApp      *app   = new VkApp("Generic", &argc, argv);
    VkWindow   *win   = new VkWindow("generic");

     // Add a menu pane to the window's menu

    VkSubMenu  *pane  = win->addMenuPane("Application");

    // Add an item to the pane. Action requires user confirmation first

    pane->addConfirmFirstAction("Quit", quitCallback, NULL);

    win->show();  // Display the window
    app->run();   // Run the application
}
```

This is the complete application. The ViewKit implementation not only provides all the capabilities of the substantially lengthier Motif code shown earlier, it does so in a comprehensive, robust, and extensible way. For example, the message displayed in the confirmation dialog can be customized easily. The application can also be easily localized. And of course, like the earlier example, this program supports context-sensitive on-line help and other features automatically.

Taking Advantage of an Object-Oriented Application Architecture

In addition to the various facilities provided by IRIS ViewKit, using the ViewKit library also has a less tangible, but even more valuable affect on application development. The ViewKit framework encourages an object-oriented approach to application development that results in cleaner architectural design and more easily maintainable code. The ViewKit architecture is based on the concept of a *component*. A component is simply a C++ class that encapsulates related elements of the user interface along with the semantics of those elements. Nearly everything in the ViewKit library is a component, including the application class, VkApp, the top-level window class, VkWindow, and the various menu classes such as the VkSubMenu class. Components can be nested, and a single logical user interface component often consists of collections of smaller user interface components. Programmers are encouraged to develop their own components and to base the architecture of their applications on collections of objects, which may include pre-defined ViewKit components, custom components, as well as other classes and objects.

The first step when developing an object-oriented application based on the ViewKit framework is to identify a set of objects that can be used to construct the program. There are many ways to design an object-oriented system, but one common approach is somewhat analogous to *functional decomposition*, a common design technique used with traditional programming styles. In functional decomposition, a programmer starts with a single function and breaks it down into multiple functions. The process is usually repeated until many small, easily understood functions have been identified. In *object-oriented decomposition*, the starting point is a large-scale object that can be broken down into smaller objects. For example, an automobile is an object that contains many smaller objects - an engine, a body, a chassis, and so on. These objects can be decomposed further. For example, an engine has valves, pistons, and so on. Programs can usually be organized in a similar way.

A mock-up of the user interface can be a useful place to start the design of an interactive application, because it is common for some parts of a program to correspond to logical groupings of elements of the user interface. For example, Figure 7 shows a proposed user interface for an overly-simplified address book program named *rolodex*.

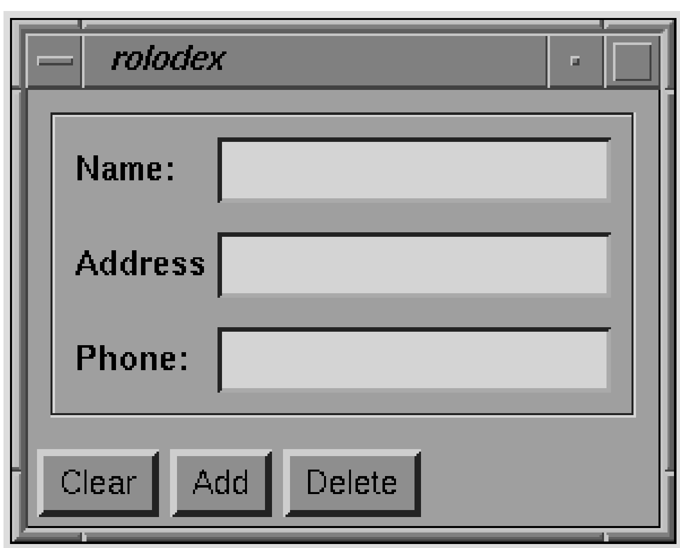

Figure 7. User interface for the rolodex program.

Looking at the proposed rolodex interface, one can observe several logical user interface components. First, it might be useful to encapsulate the entire panel in a class that can be instantiated whenever a rolodex is needed. (The interface in Figure 7 looks like a stand-alone program, but it is easy to imagine an application that might use a rolodex as just one of many useful facilities.) So, it seems reasonable to create a Rolodex class as a ViewKit component.

The Rolodex class can be decomposed further. The screen shows a logical address area, and a command panel along the bottom. These can be implemented as Address and Command classes. The Address object consists of individual labeled text areas, which could each be implemented as a still smaller component, a LabeledText class.

Besides the user interface, there must be some way to store the information associated with the rolodex, so we might create a Database class. It might also be useful to abstract the information stored in the database, so a Record class could be created to represent a single name and address. So, the list of classes that can be used to construct the rolodex program include the Command, Address, LabeledText, Database, and Record classes. All these classes work together to form a subsystem that forms a single logical user interface component, bound together by the Rolodex class.

The next step is to determine the interfaces between the various objects. For the best results, these interfaces should be based on the role the object plays in the system, and have little to do with internal implementation details. For example, the Address class might support member functions for clearing the current display, displaying a new record, and creating a record from the information typed in by a user. The Command object simply sends messages to other objects. For example, the Command object might send a "clear" message to the Address object, an "add" message to the Database object, and so on.

Figure 8 illustrates the architecture of the resulting system, by diagramming the messages that flow between objects in the system. This diagram shows all the classes used in the Rolodex example as boxes. The labeled vectors between the classes indicate member function calls (messages) that could occur between objects instantiated from these classes. Figure 8 clearly shows the relationships between the classes that implement the rolodex user interface component.

Once an initial design has been developed, the individual classes can be implemented. Many of these classes can be implemented as subclasses of the VkComponent class. The modular design makes it easy to extend or modify the system if needed. The new classes will fit smoothly with existing ViewKit classes to form a complete application.

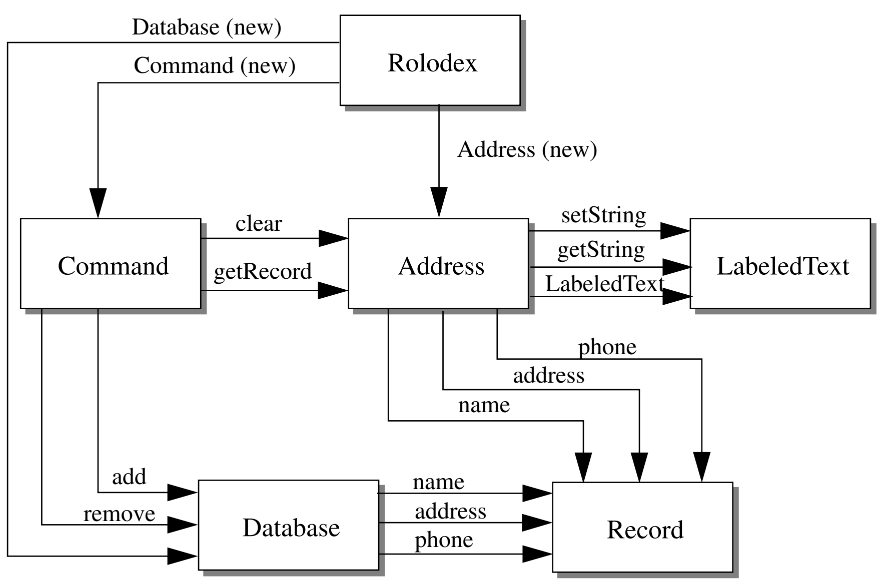

Figure 8. A message diagram of the rolodex subsystem.

Expanding the IRIS ViewKit Framework Through Reusable Classes

One of the primary reasons for the current interest in object-oriented programming techniques is the promise of greater productivity through reusable code. The IRIS ViewKit framework itself represents a large body of reusable code, and should provide a significant productivity gain to those who use it. However, an even more important goal of the ViewKit framework is to support developers who want to create new reusable classes. IRIS ViewKit defines a stylistic approach to developing applications, which, if followed carefully, can help produce reusable code.

The rolodex program described above provides one example. This simple program could be written in any number of ways. The object-oriented design described above may actually take longer to develop and require more code to implement than some other approaches, although the IRIS ViewKit library will provide a substantial amount of support, which tends to reduce the amount of code to be written. The additional work is a result of the emphasis on clean, abstract interfaces and a well-defined separation of tasks. In return for the additional time spent designing for an object-oriented model, the rolodex example has the potential to yield some new classes that may be reusable in some situations. The Rolodex class is rather specialized, but it could be useful in some applications. The LabeledText class is very simple, but is quite general purpose and could easily be useful elsewhere. The Address class is less task-specific than the Rolodex class, and might also be useful in other situations. Thus, the design of this simple application has resulted in three new, potentially reusable classes, which could be added to the pool of ViewKit-compatible components available to future applications.

IRIS ViewKit supports the creation of such components in several ways. First, the VkComponent class defines a common interface to be supported by all components. As long as user interface components are implemented as subclasses of VkComponent, the ViewKit library provides additional services that can operate on and work with these classes. The VkComponent class also supports C++ member function callbacks, which provides a way for classes to interact with one another without requiring hard-coded dependencies between classes. (Hard-coded dependencies decrease reusability.) Finally, the ViewKit framework provides a core set of services that all components can depend on. Classes can readily use dialogs, the undo facility, and so on, knowing that these core ViewKit facilities will always be available in any IRIS ViewKit application.

Using Interactive Development Tools with the IRIS ViewKit Framework

One of the most promising types of tools to be developed in recent years is the interactive user interface builder. These tools are known by different names depending on their exact capabilities. Common names include *builders*, *User Interface Management Systems* (UIMS), and *Interactive Development Tools* (IDT). This paper simply refers to all such tools as *builders*.

A user interface builder allows programmers to draw and test an interface interactively without writing code. In many cases, such tools can have an extremely beneficial effect on a programmers' productivity during the early development and design stages. Most such tools allow a programmer to create an application's interface and then go immediately to a "play" mode, in which the interface responds to user input. Users can press buttons, type in text and so on. By reducing the time a programmer spends compiling and debugging, a builder can greatly reduce the amount of time required to create a first prototype of a user interface. Using a builder allows a programmer to show a realistic prototype of an interface to potential users before committing the design to code. Most such tools can produce code that corresponds to the interactively-developed interface. Since generated code will normally be free of syntax errors, programmers can typically have a running program with a complete user interface after the first compilation.

Not everyone has been successful using user interface builders in the past, largely due to limitations in the technology. Programmers are often disappointed in the code structure produced by builders when generating complete programs. Particularly in C-based programs, it can be difficult to automatically generate code whose structure is appropriate for large systems. Most builders generate code that is incomplete, which also causes problems for programmers working in C. Typically, the code generated by the builder must be modified, which forces the programmer to understand the generated code, and may also prevent the programmer from using the builder for further development or modifications.

The component-based approach supported by the IRIS ViewKit framework is well-suited to an interactive user interface builder, and leads to solutions to some of these problems. First, a builder can be used to create individual user interface *components* rather than an entire application. A builder can allow IRIS ViewKit components to be created quickly and easily. Because these components tend to be small, functionally cohesive units, the problem of an architecture based on monolithic code segments is avoided. When used in this manner, a builder can actually encourage an object-oriented architecture based on collections of components.

The object-oriented technology used by IRIS ViewKit also provides one solution to the problem of modifying generated code. All classes created by a builder can be treated as abstract base classes, which are never modified. Programmers, of course inevitably need to write some code to connect such classes to the rest of an application. When using the object-oriented approach described here, application-specific code can be added easily by creating subclasses of the generated class and modifying only the derived class. So long as no changes are made to the base class generated by the builder, the class can be reloaded into the builder at any time and modified as needed. Of course, changes made to the base class must be carefully controlled to avoid removing any elements on which derived classes depend.

IRIS ViewKit provides an interactive "builder" solution by coordinating with BuilderXcessory, a Motif user interface builder from Integrated Computer Solutions (ICS). BuilderXcessory supports the concept of C++ components although it does not directly use IRIS ViewKit classes. SGI provides a IRIS ViewKit module that can be added to BuilderXcessory to allow BuilderXcessory to generate IRIS ViewKit-style programs.

Using ViewKit with IRIS Inventor

The IRIS Inventor library supports an object-oriented model for creating 3D graphics. Although Inventor does not require ViewKit, the two libraries are designed to be compatible, and Inventor shares the ViewKit concept of a component. Inventor can be used smoothly within a ViewKit program, as demonstrated by the following simple program[4]:

```
/////////////////////////////////////////////////////////////////
// Cube.C: Demonstrate a simple use of ViewKit and Inventor
/////////////////////////////////////////////////////////////////
#include <Vk/VkInventorApp.h>
#include <Vk/VkInventorWindow.h>
#include <Inventor/nodes/SoCube.h>

void quitCallback(Widget, XtPointer, XtPointer)
{
    theApplication->terminate(0); // Clean shut down function
}

// Main driver. Just instantiate a VkApp and a viewer, "show"
// the viewer and then "run" the application.

void main ( int argc, char **argv )
{
    // Init ViewKit
    VkApp              *app = new VkInventorApp("Cube", &argc, argv);
    VkInventorWindow   *win = new VkInventorWindow( "cube", new SoCube());

    // Add a menu pane with one menu item
    VkSubMenu   *pane  = win->addMenuPane("Application");
    pane->addConfirmFirstAction("Quit", quitCallback, NULL);

    // Show window and run
    win->show();
    app->run();
}
```

This program just creates an instance of VkInventorApp and an instance of window that creates an Inventor scene viewer an Inventor viewer. A new Inventor object (SoCube) is created and added the scene viewer window.

4. The classes used here are not part of current ViewKit or Inventor products.

Conclusion

IRIS ViewKit combines the extensive set of low-level facilities provided by the Motif toolkit, the ToolTalk library, the SGI help system, and other libraries, with the architectural support available within an object-oriented application framework. When developing with the ViewKit framework, the basic elements of the underlying libraries can be used whenever necessary to provide complete flexibility. Using the Motif toolkit provides users with a standard interface with a widely-accepted behavior and appearance. IRIS ViewKit complements the Motif widget set by providing those elements required by most applications that are not completely addressed by the Motif toolkit. The ViewKit library supports commonly needed services such as context-sensitive help and network licensing, and provides support for desirable features like busy cursors, interruptible tasks, and dialog management. IRIS ViewKit also makes it easy for developers to create significant application components such as preference dialogs and dynamic menu systems. Most importantly, IRIS ViewKit offers an environment in which developers are encouraged to design applications using an object-oriented approach that takes advantage of existing components and also produces an ever-growing number of reusable classes from which to build future applications.