Software Components - A Scalable Solution to Platform Independent Software Development for Commercial Applications

Douglas E. Donzelli and Dan R. Greening

Software Transformation, Inc.
1601 Saratoga-Sunnyvale Rd, Suite 100
Cupertino, CA 95014
(408) 973-8081

Abstract

The Universal Component System™ (UCS) is a platform-independent application development system targeted for large-scale, commercial applications. UCS runs on several platforms, including Microsoft Windows, Macintosh, Unix/Motif, and Open-Look. UCS addresses three major aspects of application portability: user interface, system services, and connectivity. UCS uniquely addresses traditional functionality-performance design tradeoffs by providing an architecture scalable to application developers' needs.

This paper describes the design philosophy of UCS, and contrasts it with other approaches. We highlight several unique aspects of the system, illustrating how our goals influenced the design of specific UCS components. Based on our experiences using UCS to develop and port software under contract, we believe the approach will be extremely useful for original software developers, as well as for those porting existing software.

Introduction

Several common software platforms provide graphical user interface facilities: Microsoft Windows, OS/2, Macintosh, Motif, Open-Look, Next, etc. Each platform has technical idiosyncrasies, strengths, and weaknesses, e.g., Windows and Macintosh provide no timesharing, Motif has oddly-constructed keyboard traversal mechanisms and provides no graphical printing facilities. Each platform has a different user profile, with different pricing expectations, graphical needs, and installed base.

Application designers have faced a choice: develop their application for a single platform, or use a portability toolkit to mask differences between the platforms. Unfortunately, most portability toolkits provide limited functionality or focus solely on user interface services. The Universal Component System™, a commercial product developed by Software Transformation, Inc. (STI), addresses three aspects of application development: operating system services, user interface management, and connectivity. As a result, UCS developers can use a broad range of features including platform-independent dynamic linking, printing, rich text editing, and object embedding. As opposed to other solutions based solely on abstraction and augmentation techniques, UCS's scalable approach resolves tradeoffs between functionality and performance by enabling developers to configure UCS to their needs.

This paper describes five design principles embodied in UCS and then examines how these principles influenced some particular design areas:

- Functionality-performance tradeoffs
- Messages
- Menubars
- Extensible visual objects
- Dynamic Linking
- Resources

Design Goals

UCS tries to maximize five qualities: efficiency, functionality, portability, modularity and extensibility.

Efficiency

Strive for maximum efficiency.

0-8186-2655-0/92 $3.00 © 1992 IEEE

Applications based on portable techniques must be indistinguishable from their natively-built competitors.

Functionality

Add capabilities to each platform where they lack functionality.

Constraining functionality to what is available on all platforms provides limited advantage to using a portability tool. Adding functions to individual platforms generated much design and implementation work for us—work the application writer avoids by using UCS.

Portability

Three portability goals drive UCS: Enable the user to write a complete application without using the native system, track attributes of new native releases, and retain native look-and-feel.

A complete solution to portability dramatically decreases the cost of developing and maintaining an application. The application writer need not understand and code separately for each platform. Development, testing, and repair all become easier.

Using higher-level native interfaces allows the application to grow naturally with new releases and variants of the underlying platform. Furthermore, an application written partially to UCS and partially to the native interface can use higher-level native objects.

Modularity

UCS has two modularity goals: Allow the developer to select only those components that the application needs, and to tailor *individual components* to the application's requirements.

Modularity distinguishes UCS from other solutions. While a single all-encompassing component can satisfy the requirements of most applications, unneeded functionality can degrade efficiency and increase program size. With different versions of a component, the developer can decide how to balance functionality and efficiency for herself.

Extensibility

Allow the application writer to freely mix native subroutine calls with portable system calls

Extensibility provides three advantages: An existing program can incorporate additional functionality supplied by UCS without rewriting the whole application. An application can take advantage of proprietary platform-specific technology, such as a special font engine. An implementation can provide platform-specific functionality or greater efficiency on platforms that support it.

Design Areas

In this section we describe some unique aspects of UCS, and point out how our design philosophy influenced the result.

Functionality/Performance Tradeoffs

The real challenge of any portable development system is to enable a developer to produce an application as efficient and feature-rich as one written natively. To accomplish this, the system must enable developers to use the same design process they follow when crafting an application natively:

1. A developer utilizes services provided by the native operating system when they meet the functional and performance needs of the target application.
2. When services are inadequate, the developer bypasses the system and implements missing capabilities using lower-level facilities of the system.

Traditional libraries, class systems, and portability abstractions cannot exactly match the performance and functionality required by a particular application, because libraries must provide a generic system usable by many applications. A single, generic library can't meet the specific needs of all applications. This is demonstrated by examining the performance/functionality tradeoff involved in text editing. Below are three example applications and the text-editing features they require:

Features	App 1	App 2	App 3
# of lines in view	multiple	multiple	multiple
scrollable view	yes	yes	yes
cut/copy/paste	yes	yes	yes
# of fonts in view	single	multiple	multiple
graphics	no	no	no
# of characters	< 32767	< 32767	> 32767

Figure 1: Sample text editing requirements

The developer must analyze the text-editing services provided by each of the native operating systems:

Features	Mac TextEdit	MSW Edit	X Widget
# of lines in view	multiple	multiple	multiple
scrollable view	yes	yes	yes
cut/copy/paste	yes	yes	yes
# of fonts in view	multiple	single	single
graphics	no	no	no
# of characters	< 32767	< 32767	> 32767

Figure 2: Platform text editing capabilities

Application One can use standard text-editing services on all platforms. Application Two can use native Macintosh TextEdit. Application Three cannot use standard services on any platform because of multiple fonts and memory limitations.

A traditional library or abstraction cannot satisfy all requirements. The developers of Application One can use the native text-editing services on all platforms. However, a library built on this basis would be unusable by Application Two and Three. To build a *single* text-edit library usable by Application Three, on the other hand, a library would have to replace native services on *every* platform. This diverges from the developer's desire to use native, minimalist facilities where appropriate.

Replacing native services to satisfy the functional requirements of a minority of applications leads to problems. First, the majority would have to be replaced, even native implementations of Push Buttons. More importantly, application performance would decline, due to unnecessary library complexity and inefficiency. Each replacement also results in decreased compatibility as native systems change in appearance and functionality .

UCS offers a more efficient implementation for applications because components have multiple implementations based on native operating system capabilities and application requirements. The developer can configure a UCS software component to the specific requirements of the application. In the case of text-editing, UCS offers more than six possible implementation choices varying in complexity. Configurability, often referred to as *scalability*, enables UCS to provide a diverse range of features, each at an optimal level of performance. This approach gives UCS a unique advantage over other traditional reusability, abstraction, and augmentation techniques.

UCS divides native services into software component *families*. Families consist of different components scaled to meet specific requirements. This core component of each family is the most efficient implementation of the most basic features. Each additional implementation of a family is called an *extension*. For example, the extension of Edit Text that supports multiple fonts in a view is called the 'Edit Text MultiFont Extension.' Some UCS families provide more than a dozen extensions, such as Graphics, and other require none at all, such as Keyboard.

No matter how many extensions a family may contain, all must support the same upwardly-compatible API. This is necessary for many reasons. First, a common family API enables the developer to easily change components as her product evolves. Second, developers will want to configure applications differently on different platforms, and in some cases, on different incarnations of the same platform. For example, developers may want to run with the Memory Swapping extension on platforms that do not support virtual memory natively, but without this extension on platforms that do. Third, a common family API enables developers to produce different application modules that share the same component configuration at run-time.

Messages

Due to the modeless nature of many system services, most development platforms, including UCS, provide a message system. Several issues motivate this approach:

1. Applications and interface objects must deal with events asynchronously; a message system ideally satisfies such requirements.
2. Multiple clients can send and receive messages from a single instantiation of a system service; this allows for service sharing between applications [Grec90].

3. Messages provide a straightforward method of interprocess communication.
4. Messages are easily mapped into methods defined by object-oriented classes.

UCS divides messages into two categories: *requests* and *notifications*. An application requests an operation by sending a message; UCS notifies an application when a component completes an operation on its behalf. Operations performed in response to messages are called *default behaviors*. For example, an application inserts a string into a list by sending a message. The list performs its default behavior by inserting the item and repainting as necessary. Upon completion, the list returns a message:

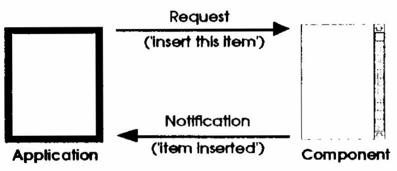

Figure 3: Requests and notifications

Components send notifications to developer-specified callback routines. UCS also provides an extension that enables developers to install multiple callback routines per object. This approach has been referred to as the *multiple listener model* [Grec90]. To provide further flexibility, developers can intercept messages before and after default processing. This mechanism, called *filtering*, enables developers to replace or modify a component's default behavior:

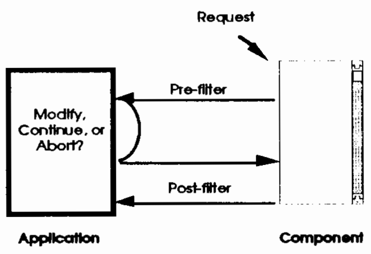

Figure 4: Pre- and post-filters

Unnecessary message traffic can cause severe performance problems. For example, some native systems notify the application of every keystroke in a text field, whether the application requires such information or not [WinRef91]. UCS eliminates unnecessary message flow by sending messages only when specifically authorized for a given message. The developer authorizes notifications by associating a filter bitmask with each interface object:

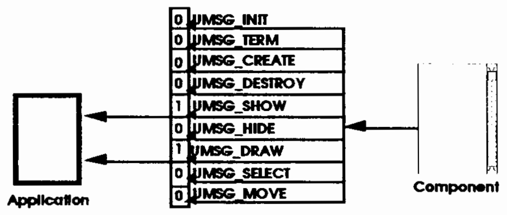

Figure 5: Notification Authorization

UCS always builds its messaging abstraction upon native message services, thus enabling developers to communicate with other executing native processes. Since UCS messages use the native system, developers can incrementally migrate their applications to UCS.

Comparisons

Most native systems limit notifications to a single user-installed callback [WinRef90], [MacRef90]. Most portability solutions limit access to the underlying message system and/or constrain notifications to a limited set of messages [Roch91].

Menubars

Application menubars in three common platforms, Windows, Macintosh, and X11/Motif, exhibit significant differences, as illustrated in Figure 6.

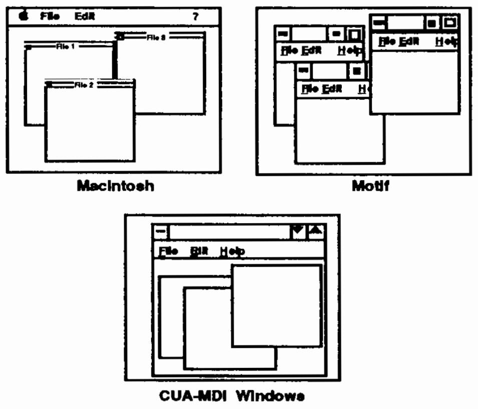

Figure 6: Menubars on three platforms.

There are only two user-interface menubar styles as far as the programmer is concerned: single menubar and multiple menubar. Microsoft

Windows allows either style, but typical Windows application use the single menubar style. Motif also allows both styles, but typical Motif applications use the multiple menubar style. Macintosh supports only the single menubar style.

The application writer wants to write a single, portable application which adopts the native look-and-feel. If there is a single menubar, she wants to modify it to correspond with the currently active top-level window. If there are multiple menubars, she wants to have each menubar look different. To accommodate the expectations of our users, we accommodate both styles in a uniform, application-transparent way.

A single "shared menubar" is associated with each UCS application. This shared menubar contains cascade-menu buttons to be included on all top-level windows. The shared menubar never physically appears on the screen. Each top-level window has a "local menubar," which can appear on the screen.

When a top-level window is activated, i.e., when one of its sub-windows acquires keyboard focus, the top-level window inherits the contents of the shared menubar.

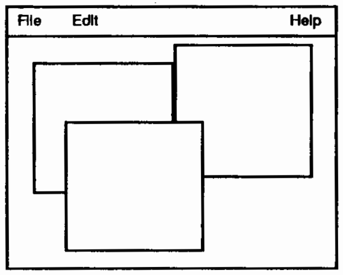

Figure 7: Single menubar style.

In the single-menubar model, shown in Figure 7, the application has access to only one visible menubar on the screen, regardless of the number of top-level windows. When a window gains control of the menubar—typically by becoming active, gaining keyboard focus or rising to the foreground—it can add window-specific cascade-menus to the menubar, or add items to individual cascades. When the window loses control of the menubar, it is required to remove any window-specific items that it added. UCS sends "menubar gained" and "menubar lost" messages to the window whenever its control of the menubar changes.

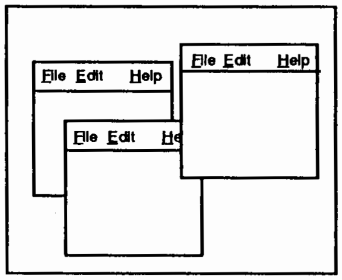

Figure 8: Multiple menubar style.

In the multiple-menubar style, shown in Figure 8, each window has an independent, visible menubar. When a window is created, UCS clones the window's local menubar from the shared menubar. UCS sends a "menubar gained" message to the window, then makes the local menubar visible. If the application never changes the shared menubar, the window instance will never receive a "menubar lost" message.

Uniform Menubars

A programmer with simple requirements should be able to write the program simply. If all menubars are identical, implementation is trivial; the application sets the contents of the shared menubar, and creates the windows (which can turn off menubar gained and lost messages).

Window-Specific Menubar Changes

The programmer can add or delete cascade-buttons on the shared menubar anytime, if the application follows a standard protocol; it first withdraws control of the menubar(s) from all window instances by calling UMenuGetMenubar. UCS sends a menubar lost message to all windows that have control of their local menubar. If the window has made local changes to the menubar, it removes them.

When UMenuGetMenubar returns, the program modifies the contents of the shared menubar. It returns control to the window(s) by calling UMenuReleaseMenubar. At this point, UCS sends a menubar gained message to each window that lost control when UMenuGetMenubar was called.

Some thoughts worth pondering: On a single-menubar platform, at most one window has gained control of its local menubar; on a multiple-menubar platform, either *all* windows have gained control over their local menubars or *none* have. The implementation can be made very efficient. Since a window must remove its changes whenever it loses control, on single menubar systems UCS creates only the shared menubar ID; it is also the local menubar ID! On multiple menubar systems, if the shared menubar is never changed (the common case), overhead for cloning the menubar only occurs when windows are created.

Cascade-Menu Changes

UCS supports shared cascade-menus, providing two principle advantages: the UCS system can efficiently clone menubars, and UCS applications can easily invoke the same cascade in different contexts.

Because application writers often want to modify only a portion of a pull-down menu (for example, a **File** cascade), UCS provides a convenient mechanism for modifying the contents of a menu-cascade immediately before it is invoked. Selecting a cascade on a menubar causes UCS to send a "menubar activated" message to the application. It may then add or delete individual cascades to correspond to the present context.

For even finer control, UCS provides an extension that notifies individual cascade menus immediately before they are invoked. Because this involves significant overhead on some platforms, we provide it as a UCS extension.

Comparisons

In summary, the UCS application writer can use any of three techniques to customize menubars for top-level windows:

1. Set the local menubar contents whenever a window receives a menubar gained message. Restore the local menubar contents whenever a window receives a menubar-gained message.

2. If each window's menubar looks exactly the same, but the contents of the cascade menus are different, modify the contents of the menu cascades whenever the application receives a menubar activate message.

3. If the Cascade Notify Extension is installed, modify individual cascades whenever they receive cascade-menu activate messages.

One internal portability toolkit adopted a fourth approach for menubars [Nich91]. The menubar has two components: a shared component and a local component. Under the single-menubar style, whenever a window is activated, the a new menubar is constructed by concatenating the shared component and the window's local component. Under the multiple-menubar style every window includes a menubar concatenating the shared component and its local menubar component.

UCS does not use that model because it was not a complete solution. It does not allow local menubar variations to be inserted, for example. The UCS approach, which appears to be unique, subsumes that approach as well as several others [Hayh91].

Extensible Visual Objects

A survey of commercial products demonstrates that developers require a wide variety of interface support facilities:

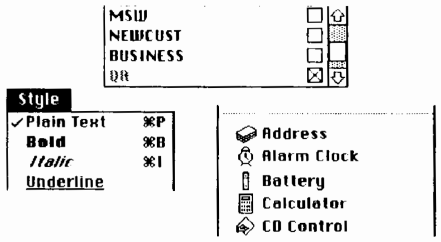

Figure 9: Common interface extensions

Commercial products combine text, pixmaps, colors, and multimedia images within standard interface objects. Other products enhance traditional interface objects with non-traditional input characteristics, keyboard equivalents, and macros. Few native systems support these techniques directly. Since competitive applications require powerful and unique user interfaces, development systems must meet a strict set of requirements:

1. To guarantee commercial acceptance by users and platform providers, user interfaces must conform to strict platform guidelines. Users should not be able to distinguish a native interface from one produced portably.

2. To ensure maximum performance and compatibility with platform revisions, portability solutions must use and extend native interface facilities.

3. To allow for product innovation, interface facilities must provide extensibility to new input mechanisms, display images, and other customizations.

UCS divides interface facilities into controls and items. Lists, buttons, tables, and menus are examples of controls. Text, bitmaps, and colors are examples of items. The developer is free to combine any type of item with any control. As a result, text lists, bitmap lists, and combinations thereof can be manipulated with equal ease. Likewise, all controls display pixmaps and text without requiring developer modifications. Based on the variety found in commercial applications, UCS provides components for all native controls and more than seven types of items.

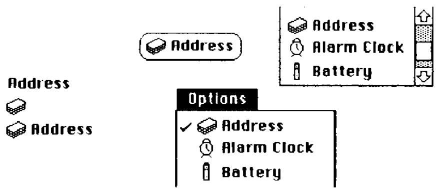

Figure 10: Items and controls

Controls define and manage the selection characteristics, layout, scrolling, and updating of a user interface object. Items manage the painting, storage, mnemonic selection, and other operations specific to an item's data type. Controls and items behave according to a master-slave relationship, where the control is always in charge and the item responds to requests on a control's behalf. Since the control-item API is externally defined, developers can freely create new item types and combine them with any existing control. Likewise, the developer can easily create a new control type for use with existing item types. This greatly reduces the complexity of enhancing existing interface management systems.

The developer specifies a control type and item type when an object is first created. UCS maps this request into the most efficient native implementation. For example, a text list request results in a standard Macintosh List or Windows ListBox, while a bitmap list request results in an LDEF-based list or 'owner-draw' ListBox. This technique avoids unnecessary messaging, sub-classing, or use of non-standard control definition procedures.

The developer manipulates a control independent of the item type it contains. For example, item selection, deletion, and addition use the same message types and message-specific data whether the control contains a pixmap or a text string. As a result, the developer can easily modify an interface object from a list to a menu or from a list of strings to a list of strings and bitmaps.

Because of the control-item separation, an item can optimize data storage based on a control's needs. For example, list items can be stored in heaps on one platform, zones on another, or linked lists on a third. In addition, developers can define application-specific storage methods, a feature which is commonly required by applications that store large quantities of data. This mechanism avoids the memory overhead of interface objects that store copies of data or data reference values. Items can also be used to define references to OLE or Edition Manager objects.

Dynamic and Run-Time Libraries

When one binds an application to subroutine libraries, one commonly expects code from the libraries to be merged with the application and written to the executable file. This operation is called "static linking." If the libraries can only be statically bound to applications, they are "static libraries."

In some operating systems, code from subroutine libraries can be omitted from the compiled program. When the system invokes an application, it finds and loads the library files, loads the executable file, resolves external references, and begins execution. This process is called "dynamic loading." Sometimes the operating system can satisfy several demands for a library with a single in-memory copy of the library. These libraries, which create a special case of dynamic loading, are called "shared libraries."

In some operating systems, an application can invoke a subroutine from a library file using a run-time generated string name. Neither the linker nor the loader binds the external reference; indeed, the string name could even be typed by the user. Such libraries are called "run-time libraries," since the binding of name to address occurs during program execution.

Dynamic libraries significantly reduce the size of executable files, particularly when graphics interface routines are in the shared library. On a Sparcstation, code from standard X11 libraries can easily consume 80 to 90% of a small Motif application's statically-linked executable file. Shared libraries reduce the memory requirements of co-resident programs that use the same libraries.

Run-time libraries further reduce memory requirements, particularly when an application user rarely exercises all the application's features. Infrequently used features can be relegated to a run-time library, and invoked only when required.

One might think shared and run-time libraries would be universal, given their usefulness. They are not. Only Microsoft Windows supports both capabilities natively. Even then, using native shared libraries on Windows is quite painful: static data is actually shared between different library invocations. Some variants of UNIX support only shared libraries, some support both shared and run-time libraries, and some support only statically-linked libraries. Macintosh supports neither shared nor run-time libraries.

UCS supplies shared and run-time subroutine libraries on all platforms. This was one of our most difficult implementation exercises. Particular difficulties included writing a dynamic loader for the Macintosh from scratch, and dealing with a plethora of executable formats on UNIX.

Nonetheless, we believe that our decision to support shared and run-time libraries has proved invaluable. Executable code produced under UCS is much smaller than code from other portability systems. We make frequent use of run-time libraries to reduce the memory requirements of our applications. This allows us to run applications on smaller, cheaper, and more common personal computers, giving UCS applications the widest possible audience.

Resources: Object Orientation

Resources are data records stored in an application's executable file, in library modules, or in separate resource files. Application designers typically store customization information about the application's appearance or behavior in resources. Typical resource values include colors, window sizes, bitmaps. UCS provides a facility for specifying an entire window system hierarchy in resource files.

Changing a value in a resource file does not require regenerating the executable. This provides a convenient means for an application designer to change the appearance and behavior of the program rapidly, and it allows the user to customize an application without having the source code. Using tools supplied by STI, an application designer can build the graphical user interface directly on a screen, and write the corresponding resource values to a resource file.

The program identifies a resource record by two numbers: its type-id and its key number. Unlike some platforms' native behavior, a resource record cannot be uniquely identified by its key alone. Programs can register a "type-handler" for any type-id. This handler becomes responsible for loading resources from files and for performing conversions to and from UCS "objects" (see below).

A number of standard UCS resource types are predefined, including a color type, a bitmap type, an icon type, etc. Each standard UCS resource type has a default type-handler pre-installed. The programmer may override this default type-handler if desired.

Although user-supplied type-handlers may support platform-independant data formats, most UCS standard resources have platform-dependant data formats, and the resource data must be converted when moving an application between platforms. STI supplies tools to convert resources from one platform to another. However, program designers typically change an application's visual appearance, through its resources, to match a new platform's interface style and display capabilities.

Resource type-handlers form part of an object-oriented system that works seamlessly with other UCS components. A type-handler controls access and external representation of objects that have the appropriate type-id. Code that has a resource object can convert it, through its type-handler, to the representation it needs. Other UCS components can define their own types and type-handlers: the resource component transparently handles them like its own.

A special object class, called a "packet", facilitates easy exchange between native and UCS objects. It also allows UCS to cache converted values. Packets are created from two things: a resource type-id, and either a native resource value, another UCS component value or a UCS resource component value. A 32-bit packet-id represents each packet.

Application code may request that a packet return an appropriate UCS object. If a resource packet contains only a native object value, UCS converts the native object to UCS form, through its type-handler, then saves the UCS object in the packet. UCS satisfies additional requests for the UCS object from the cached value. The converse, from UCS to native form, behaves similarly. Since native objects are often dependent on screen-type, in X11, there may be several cached native objects in a single packet.

UCS resources are stored natively as Macintosh resource forks, Windows resource files, and X11 resource databases.

Resource objects improve functionality and portability. The Xt toolkit allows resource conversions to be registered. UCS resource objects provide similar functionality.

Resource packets enhance efficiency. Without resource caching, resource type conversion can cause a noticeable slowdown. Furthermore, on X11 the caching of multiple representations is practically required, since UCS supports multiple screens and displays.

Resource packets improve extensibility. We have discovered that applications that use both native and UCS code perform frequent conversions. The ability to pass a single 32-bit id to represent both data representations has been extremely valuable.

Summary

UCS is a portable application development environment designed by and for commercial developers. We identified five important qualities for a portable system, based on problems encountered in porting and developing applications under contract: efficiency, functionality, portability, modularity and extensibility. Those qualities drove the implementation of UCS, and gave it characteristics not seen in other portability solutions.

To simultaneously maximize efficiency, functionality, and modularity, UCS provides multiple implementations of the same component.

It provides a rich message system for user-interface objects. Messages are sent only if the application requests them. The default behavior for most messages can be modified or avoided by filtering. Multiple callback routines can be installed for a given object, allowing greater modularity in the application program.

UCS has a unique interface to manage menubars, which portably hides whether the native system displays one or several menubars, while maintaining complete functional control over their contents.

UCS provides an extensible visual object system. It supplies simple text, rich text, pixmap and icon items. The application developer can supply new visual objects, and use them seamlessly with buttons, menus and lists.

UCS provides shared and run-time libraries on all platforms, including Macintosh and several UNIX variants.

Resource values in UCS are objects. Each resource type has a handler which controls loading, storing, and conversion of native resource values. The application can install its own type-handlers or use UCS-supplied defaults. A special resource object encapsulates native and UCS objects, allowing the developer to mix native and UCS code easily.

Under development since 1987, UCS has evolved into a very rich, portable application development environment. STI decided to incubate and validate UCS internally, rather than doom an immature product to upward compatibility. Under contract over the last few years, STI has ported and developed a number of commercial products with UCS. Based on its commercial success, STI is now making UCS available to customers. Our internal users and external customers continue to demand much from UCS — demands we intend to satisfy.

References

[Copl91] James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, New York, 1991.

[Grec90] Richard J. Greco, "Introduction to Window Management," *Course Notes of the 17th International Conference on Computer Graphics and Interactive Techniques* (August 6-10, 1991), pp. 19-35, SIGGRAPH, Dallas.

[Hayh91] Brett Hayhurst, "Issues in the Development of a Portable Toolkit," *Proceedings of the First Annual International Motif User's Meeting* (December 8–11, 1991), pp. 236–247, Open Systems, Bethesda.

[MacRef90] *Inside Macintosh: Volumes I-V*, Apple Computer Inc., Addison-Wesley, New York, 1991.

[Micr90] *Microsoft Windows Graphical Environment User's Guide, Version 3.0*, Microsoft Corporation, Redmond, Washington, 1990.

[Nich91] Robert T. Nicholson, "Designing a Portable GUI Toolkit," *Dr. Dobb's Journal*, (January 1991), pp. 68–75.

[Rose91] Larry Rosenstein and Joseph S. Terry, "OOPS Architectures: MacApp and THINK Class Library, *MacTutor*, 7 (1), (January 1991), pp. 14–23.

[Roch91] Marc J. Rochkind, "XVT—The Extensible Virtual Toolkit for Portable GUI Applications," *Proceedings of the First Annual International Motif User's Meeting* (December 8–11, 1991), pp. 229–235, Open Systems, Bethesda.

[WinRef90] *Microsoft Windows Software Development Kit, Version 3.0,* Microsoft Corporation, Redmond, Washington, 1990.