Integumentary Customization

Implementing "Skins" in your Application

Toby W. Rush
toby@tobyrush.com

Abstract

One of the concepts which has helped make the Macintosh revolutionary is its ability to be customized to match the user's preferences. Several programs that have appeared in the last few years use the idea of "skins"—customizable interfaces—to extend this capability further. The most effective skins not only allow the user to change colors, background patterns and window styles, but also to determine the placement and presence of controls. While implementing skins in your program does require you to design the program from the ground up with this in mind, the actual implementation is not difficult. This paper discusses the advantages of using skins, how to reconcile skins with established interface design principles, and walks through the process of designing a "skinnable" application.

Control/display panels aboard the USS Enterprise are software-defined surfaces that are continually updated and reconfigured for maximum operator efficiency and ease of use.…Cruise mode operating rules allow each crew member to define a customized operating configuration for his/her work station. [1]

Since software programming began, the primary goal of interface design has always been to improve ease of use. The creation of the Macintosh OS in the early eighties was a defining event in interface design because it provided programmers with the capability, through the Toolbox, to quickly create easy-to-use interfaces. In fact, it was easier to use the predefined, familiar and well-designed interface elements than it was to create one's own. This scheme enforced consistency across all applications; a checkbox control in one application looked and behaved like a checkbox control in any other. This allowed the user to learn new applications more quickly.

In the past five years, processor speed has been rapidly increasing and interface designers have had the computing power to pursue a secondary goal: flexibility. Many of the recent advances in interface design have been efforts to allow more user customization of the interface.

The current culmination of this design strategy is the popular concept of "skins": sets of instructions for displaying the interface of a program. These instructions can exist as a removable part of the application program (a resource, for example), but more often they are separate documents that the program can read and use. Users can change the interface of a program by installing a different skin, and new interfaces can be developed without rewriting the program. Because each program is different and requires a different method of implementing skins, skin files are generally specific to a

[1] Rick Sternbach and Michael Okuda: *Star Trek: The Next Generation Technical Manual.* New York: Pocket Books, 1991: 33.

particular program and cannot be used in other programs. As a result, skin file formats are not standardized.[2]

Unfortunately, by giving users the power to create their own interface, the consistency and predictability of pre-skin interface design is no longer ensured. Skin designers can create their own interface elements that, while perhaps fitting well with the rest of the skin, do not match interface elements in other programs. This paper discusses the process of reconciling the skin concept with fundamentals of good interface design, as well as what the programmer can do to make the skin development process approachable for designers. The process of designing a skinnable application will be outlined, as well as examples of implementing such a design.

In this paper, I use the following terms in ways that haven't quite made it into Webster's: a *skinnable* application is one that makes use of skin technology; to *skin* an application is to change the interface by applying a skin. Note that *skinning an application* and *designing a skinnable application* are two different things; the former is usually done by the user, and the latter by the developer.[3]

Why Make an Application Skinnable?

Let's first address the various strengths of skinning.

Aesthetics. The most common use of skins in currently available software is to provide the user with different appearances for any given application. While this aspect of interface is of little functional importance, it fits well with the Macintosh design philosophy. Using skins, users may make the application harmonize with their desktop pattern or picture, their hardware, or even their mood.[4]

Adaptation to User Workflow. The Human Interface Guidelines published by Apple are meant to ensure consistency based on average user expectation. A skin implementation, however, allows the application to address specific user expectation. For example, a sound synthesis program might have skins that mimic the control layout of different rack-mounted synthesizers, so users of a specific synthesizer could use a skin that is familiar to them.

Ability to Serve Various Needs. Skins can be particularly useful for programs that combine several functions, since skins allow the user to choose only the functions needed for a particular task and combine them into a single console. An all-purpose audio utility like Apple's

[2] This begs the question: "Could a standardized skin file format work?" The format would need to be extremely flexible, but the concept is within the realm of possibility. A single file could contain appearance information for standard design elements, much like a Kaleidoscope or Appearance Manager theme, but the file could also contain specific layout information for various skinnable programs. Since each different program would require a planned skin, however, the benefits of such a standard are debatable.

[3] This terminology is admittedly a little confusing since *skinning an application* involves putting "skin" *on* something, *while skinning a cat* involves taking skin *off* something. (Sources indicate there is more than one way to do the latter.)

[4] A related concept in this vein is the ability to create branded versions of the application. This level of commercialism will often make the true developer shudder, but if creating an Eddie Bauer edition of one's FTP client brings the programmer a little more income, it may be worth considering!

iTunes would be an excellent candidate for a skin implementation: one skin can present the basic playback controls, and another skin can provide a large window for database manipulation. Elements of each of these can be combined according to the user's preference.

Localization. While most localization needs can be met through the use of resources, skins take the idea further. With a skin, language-based text and culturally specific images are still stored separately from the program code, but the need for having separate downloads for different versions of the application is eliminated. Instead, a single executable can be provided, with skins that contain localized information for each country or market.

Accessibility. Just as software localization is made simpler with skins, a skin implementation can ensure access to an application by users who have disabilities or special needs. Users may make modifications such as increasing text size, increasing contrast, changing colors, replacing elements that cannot be rendered by a speech synthesizer, and so on. Again, if a skin implementation is included with the program, skins can be created without needing to recompile the program. This reduces development time but increases the user base.

Skinning vs. Good Interface Design

Many of the skins available for currently popular programs are decidedly un-Macintosh. They contain obvious inconsistencies, such as customized replacements for standard controls like buttons, edit fields and popup menus as well as non-standard window shapes and color usage. But some of the more problematic difference are subtleties: controls that activate on mouseDown instead of mouseUp, changes to the enabled/disabled paradigm for windows and controls, and inconsistent methods of dragging and resizing windows. This has larger problems than just being an affront to obsessive interface designers; it steals from the inter-application consistency that makes the Macintosh interface great.

However, the benefits of making an application skinnable cannot be denied. How, then, can the developer provide a method to skin an application while still maintaining a well-designed interface? To answer that, we should take a look at the different degrees of skin implementation. These categories are, of course, mine, but I have tried to make them as non-arbitrary as I can by basing them on real-world usage. A higher level number indicates greater degree of flexibility.

Level I Implementation. A level I implementation of skin capability allows the skin to make changes to only characteristics of the window and the superficial aspects of the controls inside it. Controls may be neither redesigned, moved, replaced, added nor removed. Applications with a level I implementation generally allow customization of the window color (or placement of an image as the background), the text font, size, style and color in controls, and other superficial changes. Programs with this implementation may also technically allow customization of the window style (*i.e.,* the procID), but in practice this is relatively rare.

Level II Implementation. The second level of skin implementation allows the same functionality as Level I, but adds the

capability of redesigning controls. Application windows which use this implementation have the same layout of controls regardless of the skin used, but allow for complete customization within the layout. Controls are generally constrained to a maximum size to avoid overlapping.

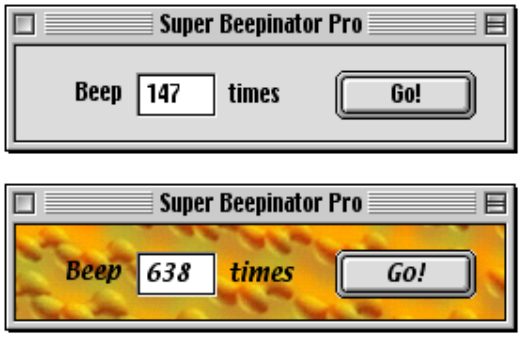

Figure 1. In a Level I skin implementation, colors, fonts, and other superficial properties can be changed, but fundamental aspects of the interface such as layout or control type cannot be altered.

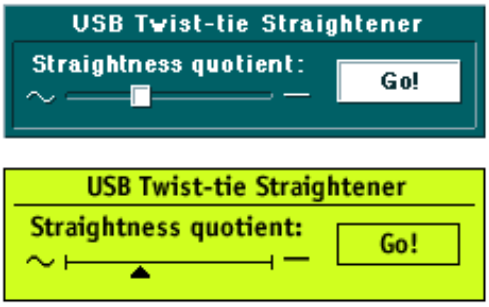

Figure 2. Level II skin implementations allow the redesigning of controls, but not a change in layout. Because controls can be redesigned, toolbox controls are generally not used.

Often the implementation of this type of skin involves the use of custom controls whose mouseDown, mouseUp and mouseOver appearances are specified by graphics in the skin file, so that changing these graphics has the appearance of changing the control itself. As a result, these implementations do not use standard toolbox controls, since they would need to be replaced by the custom control.

Level III Implementation. This degree of implementation embraces the functionality of both Level II and III, but also allows the controls to be moved, resized or hidden. In this situation, a specific number of controls are defined for the window in question, but all characteristics of the controls may be changed by the skin.

Level IV Implementation. The highest degree of skin implementation allows for the creation of controls as directed by the skin. In this implementation, the number of controls created by the skin file is limited only by memory (and practicality). A set of available controls is defined, and a skin file may use any or all of them to construct the window. This provides the most flexibility but is also the most complex from a design standpoint. The steps in realizing this implementation are described later in this paper.

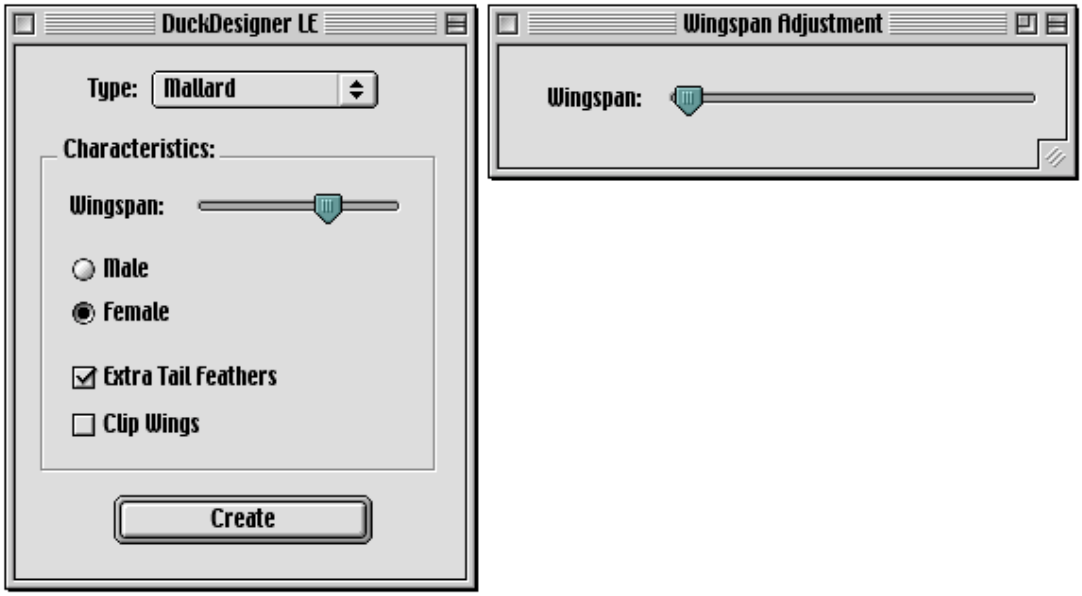

Figure 3. In a Level III implementation, controls may be moved, resized or hidden, but new ones may not be created. The window to the right displays all the controls for this program, whereas the window below shows only a static text item and a slider.

Given these levels of implementation—and the understanding that a program of this type necessarily involves handing some control of interface design to the skin designer—we can see that different types of implementations have different ways of reconciling the "skinning vs. good interface design" issue. A level I implementation is fairly static, and while the skin designer can impact the aesthetics of the program, the remainder of the interface (spacing, layout, type of controls) are unchanged. The other three levels are far more dangerous from an interface design perspective, since the skin designer is given free rein to replace well-designed controls with those that may be more aesthetically pleasing but less efficient or predictable for the user.

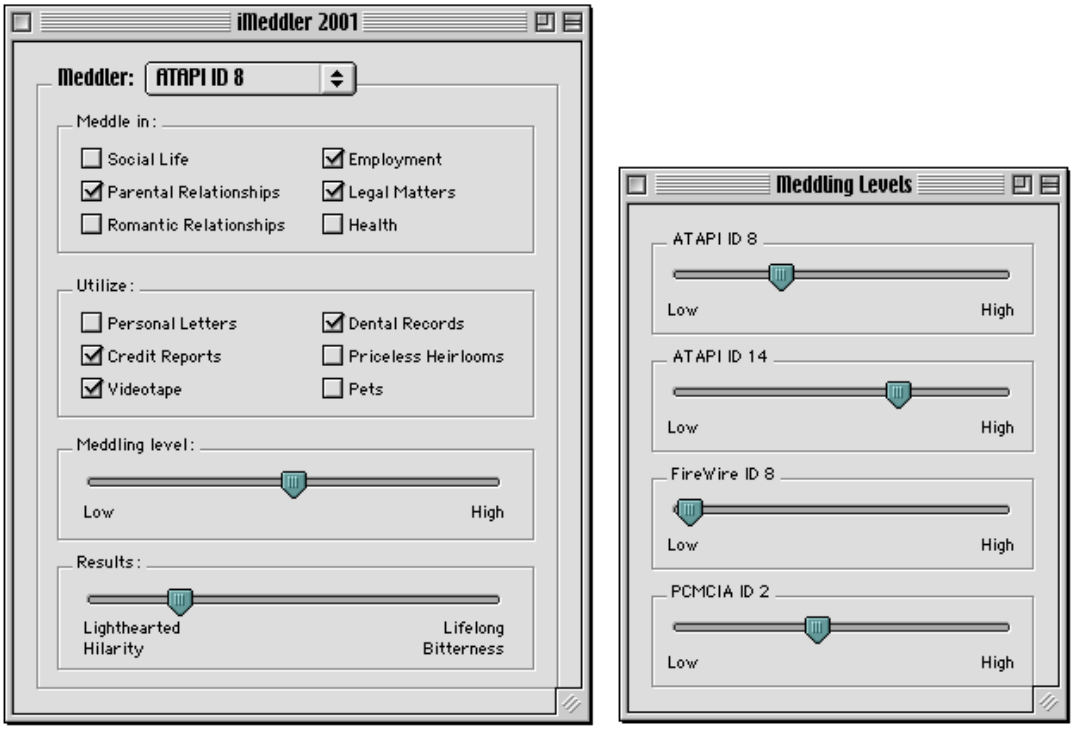

Figure 4. A Level IV combines the capabilities of the other three levels with the ability to dynamically create controls. The window on the left displays all possible controls for the program. The window on the right contains four instances of the same control, each set to control a different physical device.

The flexibility of a level IV implementation seems like it would be the most problematic in this regard, but in fact it has an interesting benefit. Since the skin designer may choose from a number of controls to construct the interface, the programmer can provide instances of the standard toolbox controls as well as custom controls that accomplish the same task but provide more design flexibility. If the standard toolbox controls are present as options, the undiscriminating skin designer will use these since they require less effort to implement. Should the designer wish to take the time to design a custom control, the capability to do so is provided.

Careful designing of these custom controls can also enforce good interface design. If one of the options for the skin designer is a custom button which allows for different mouseDown, mouseUp and mouseOver pictures, the control template can be designed to enforce the "action on mouseUp" paradigm which pervades the Macintosh interface. Interface consistency is thus built in for the skin designer, as it is for the programmer who chooses toolbox controls over custom solutions.

Making your Implementation Approachable for Skin Designers

When implementing skins in an application, a programmer must decide on what level of programming or design will be required to design skins for the

program.[5] Generally, a low level of required expertise is best, as it allows more people to create skins and increases the number of skins available for users who do not design skins. Unfortunately, a simple method of creating skins is not always possible, especially for higher-numbered levels of implementation. In this case, the programmer must strike a balance between simplicity and flexibility: an overly simplistic method will not provide for flexibility, but a method that requires high-level programming concepts will not be accessible to most people.

The programmer can reduce the complexity of the skin design process by requiring only the essential information of interface design be included in the file, and by allowing it to be presented in a simple way. It is best to avoid knowledge or concepts that are foreign to non-programmers. Additionally, a good design strategy for this process is to ignore how the program itself is structured and determine how the user perceives it to be structured. For instance, a complex network chat program might make use of several different concurrent connections to coordinate a single chat session. If the user perceives the session as one continuous connection, any customizable interface elements should maintain this illusion.

[5] Users of a skinnable application can be grouped in three categories: those who design skins for the application, those who do not design skins but make use of skins designed by others, and those who ignore the skinning capabilities altogether. Programmers should keep each of these types of users in mind when designing and distributing the program. The last type of user is generally addressed by including a default skin with the program.

Perhaps the most important limiting factor in determining who will be able to design skins for an application is which tools are required for the job. Requiring specific programming or compiling applications will certainly restrict skin development for the application; it is better instead to require only basic, freely available tools such as resource and text editors. The best solution is to create an easy-to-use utility for designing skins and include it with the application.

In all levels of skin implementation, the skin file must contain a list of characteristics for controls. In the first three levels, the list of controls is constant and can be specified in a set order. For instance, a skin file for a basic clock application might be a text-based file in the following form:

<header information for file> <CR>
<characteristics for time display> <CR>
<characteristics for date display> <CR>
<characteristics for alarm button>

If this program has a Level III skin implementation, each line might contain data for position and size of the individual controls, but the skin file could be required to contain the information in the order specified (so the information for the date display is always found on line 3).

With a Level IV implementation, however, the skin file must accommodate an unrestricted list of controls. Since there is no guarantee about which controls will be include in any given window, the program cannot depend on the order or even presence of controls. In this case, an XML-based file works well, since it reflects the object-oriented nature of the skin. Below is a

slightly more specific example for our hypothetical clock program:

```
<WINDOW STYLE="plainBox" WIDTH="200" HEIGHT="100">
<TIMEDISPLAY LEFT="45" TOP="13" WIDTH="150" HEIGHT="20"
FONT="Courier">
<ALARMBUTTON LEFT="45" TOP="80" WIDTH="80" HEIGHT="20"
FONT="Charcoal">
```

Because of the popularity of HTML and other XML-based languages, computer-literate users are likely to find a format like this familiar and simple.

The XML-based file allows the skin designer to specify characteristics of controls, but it does not allow for media elements such as graphics and sound. These media types could conceivably be encoded and placed in the data fork alongside the XML description file, but this is needlessly complex. The obvious solution is to include these elements in the resource fork. They can then be referred to in the XML file by their resource number or name, as shown here:

```
<CUSTOMBUTTON LEFT="20" TOP="20" WIDTH="40" HEIGHT="20"
   DEFAULTIMAGE="4000" MOUSEOVERIMAGE="4100" MOUSEDOWNIMAGE="4200">
```

The above example describes a button whose appearance is based on graphics included in the resource fork. The normal image for this button is stored as 'PICT' resource ID 4000. The image displayed when the mouse is over the button is found as 'PICT' ID 4100, and the image displayed when the mouse is held down over the button is 'PICT' ID 4200. Storing the graphics in the resource fork allows them to be edited and replaced easily and makes initial skin development easier, since it takes advantage of the Mac's innate ability to handle media stored in files. Sounds, video and other media can be stored in a similar fashion.

Designing a Skinnable Application: Concepts

The remainder of this paper discusses the process of designing a level IV skin implementation in a program. For this level of implementation, it is recommended that the program be designed from the start with the implementation in mind. The concept of a skinnable application fits very well into the structure of object-oriented programming, so the idea of classes and subclassing will be used in the following sections. For an example we will continue to use the idea of a simple clock program.

The first step to designing this type of implementation is to create a list of all possible controls to make available to the skin designer. It is often tempting to create very vague, open-ended controls in this step, but doing so will require that the skin file be more complex and difficult to design. Rather than having a small number of generalized controls, it may be better to create a larger number of function-specific controls. This decision must be made with reference to the way the user and skin designer perceive the structure of the application.

For instance, a CD player application may require buttons for many different functions (play, pause, eject, scan

forward/backward, *etc.*). The skin implementation may also want to allow many different type of buttons (standard pushbutton, bevel button, customized graphic button). In this example, the programmer must decide between the following:

1. Create a single control template, allowing the skin designer to set characteristics of button type and function;

2. Create a control template for each button type, allowing the skin designer to set the function of each;

3. Create a control template for each function, allowing the skin designer to set the type of button of each; or

4. Create a control template for each combination of button type and function.

Of these options, number 1 might be the best from the skin designer's perspective, followed by number 3 (which emphasizes function over appearance). Number 2 is generally the easiest to implement in an object-oriented format, since it involves a simple subclassing of the control types. Number 4 is needlessly complex for both skin designer and programmer.

In our simple clock example, we might have the following list of controls be available to the skin designer:

- A text-based time display
- A text-based date display
- A graphic-based analog time display
- A button to turn the alarm on and off

Since both the time and date displays are going to be text items, it might be tempting to combine them into a generic text item. This might be worth doing in a more complicated program, but in this simple example the application structure is better portrayed by creating separate control templates.

The next step is to determine what characteristics each control has that can be changed through the skin file. To continue with our clock:

Characteristics for all controls: position and size

Time display control: time format, text color, font, size and style

Date display control: date format, text color, font, size and style

Analog Time Display: hand, background and numbers colors, presence of second hand, type of numbers (Arabic, roman, or none), etc.

Alarm Button: button type (checkbox or sticky bevel), font, size, style, text color

Note that Human Interface Guidelines can be enforced through some restrictions built in here. If the Alarm Button is supposed to be a two-state button, the skin designer should only be able to use controls that would work in this way: a checkbox or a bevel button in "sticky" mode. A standard pushbutton, which does not change state, is not appropriate here; nor is a radio button, since the control is not among a group.

Figure 5. By allowing only certain controls to be used for a particular function, the Human Interface Guidelines can be enforced in a skinnable application. In this case, a checkbox and a "sticky" bevel button are appropriate, but a radio button, a popup menu, a pushbutton and a tab panel (!) are not.

Each of these control templates should then be created as control subclasses. In designing these subclasses, the programmer must have the goal of making the controls completely self-sufficient. Controls that rely on other controls will not work in this situation, since the presence of other controls is entirely up to the skin designer. Controls should handle all user interaction with themselves, and should be written to handle the fact that there may be two or more identical controls in the same window.

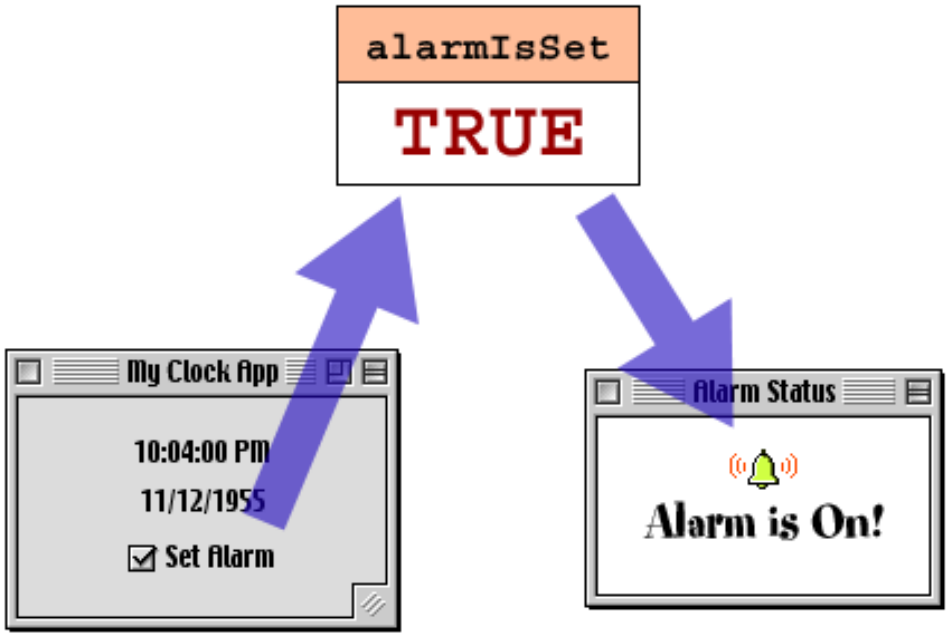

Figure 6. In a Level IV implementation, controls may not modify each other directly; they must set and read information in the application itself. The checkbox in the window on the left cannot send a message to the graphic in the window on the right; it must instead set a flag in the application. This flag is checked periodically by the graphic.

Because no controls can assume the presence of other controls in the window, controls that normally influence the appearance or function of other controls must be redesigned. For example, our program may have a control that displays a particular graphic when the alarm is set. In a non-skinnable application, the "set alarm" control could be programmed to signal the control to display its graphic. In a skinnable application, however, there may be no such graphic, or there may be several such graphic controls; the set alarm control has no way of knowing. In this case, the set alarm control must set a flag somewhere in the application that the alarm is set, and the graphic control must periodically check the status of this flag and change its appearance accordingly.

Once each of these subclasses have been created, the next step is to create a window that can construct itself, creating instances of these subclasses, according to a given skin file. Once the window setup is complete, the subclasses take care of themselves. However, in order to ensure that the classes are kept updated, one of the few duties of the window is to cycle through all the controls and send update events to each on a periodic basis.

If the window itself is a subclass, it is then easy to allow multiple windows to exist at once, each one created from a different skin file.

Designing a Skinnable Application: Implementation

To illustrate the concepts above, I use the REALbasic development environment; however, the concepts can be easily ported to another object-oriented environment, such as C++. The use of REALbasic in this paper is primarily based on my familiarity with the environment, but also stands as proof that creating a skinnable application does not require low-level programming.

Having devised a list of controls we want to make available to the skin designer, the first step is to implement these as subclasses. For example, the time display control described above would be a subclass of a staticText object. Create a new class, set the super to staticText, and name it timeDisplayClass. Since this object inherits the properties of staticText, we do not need to add properties for text color, font, size or style; they are already built in to the control. We do need to add a timeFormat property, however. To keep the example simple, we will allow the skin designer to choose between a short time format (6:10 PM) or a long time format (6:10:34 PM). To do this, create a new property for the subclass and define it as "useLongTime as boolean." A value of true in this property indicates that the control should display the long time; otherwise the control will display the short time.

We could leave the creation of the control to the window, but object-oriented programming techniques dictate that we have the control responsible for its own creation. By setting up a constructor method that takes a line of XML as a parameter, we can do just that. Create a new method, name it "construct," and enter "dataLine as string" as the parameter. Enter the following code into this newly created method:

```
Sub construct(data as string)
   // this method sets the characteristics of the control
   // according to the XML tag passed in data
   me.left=val(getXMLParameter(data,"LEFT"))
   me.top=val(getXMLParameter(data,"TOP"))
   me.width=val(getXMLParameter(data,"WIDTH"))
   me.height=val(getXMLParameter(data,"HEIGHT"))
   me.textFont=getXMLParameter(data,"FONT")
   me.textSize=val(getXMLParameter(data,"SIZE"))
   me.bold=(instr(getXMLParameter(data,"STYLE"),"BOLD")<>0)
   me.italic=(instr(getXMLParameter(data,"STYLE"),"ITALIC")<>0)
   me.underline=(instr(getXMLParameter(data,"STYLE"),"UNDERLINE")<>0)
   me.textColor=decodeColor(getXMLParameter(data,"COLOR"))
   me.useLongTime=(getXMLParameter(dataLine,"LONGTIME")="TRUE")
End Sub
```

This bit of code requires the following support methods, which can be placed in a separate module in the project (perhaps titled "Utilities"):

```
Function getXMLParameter(XMLTag as string, parameter as string) as
string
   // Given an XML tag, this function returns the value
   // for a named parameter in that tag.

   dim newXMLTag,returnString as string
   dim i as integer
```

```
   // if it's an XML tag, it will be surrounded by angle brackets
   if left(XMLTag,1)="<" and right(XMLTag,1)=">" then
      newXMLTag=mid(XMLTag,2,len(XMLTag)-2) // remove the brackets
      // go through the tag and find the parameter;
      // then assign that parameter's value to returnString
      for i=1 to countFields(newXMLTag," ")
         if nthField(nthField(newXMLTag," ",i),"=",1)=parameter then
            returnString=nthField(nthField(newXMLTag," ",i),"=",2)
         end
      next
      // if the value had quotes around it, strip the quotes
      if left(returnString,1)=chr(34) then
         returnString=right(returnString,len(returnString)-1)
      end
      if right(returnString,1)=chr(34) then
         returnString=left(returnString,len(returnString)-1)
      end
      // this tag requires that the following characters be encoded
      // as entities, so that parsing can be made easier.
      // here we decode the entities
      returnString=replaceAll(returnString,"&sp;"," ")
      returnString=replaceAll(returnString,"&amp;","&")
      returnString=replaceAll(returnString,"&cr;",chr(13))
      return returnString
   else // if it's not an XML tag (not surrounded by angle brackets)
      return ""
   end

End Function

Function decodeColor(data as string) as color
   // Given a color description string (I use the RGB hexadecimal
   // color triplet found in HTML), this function returns a REALbasic
      // color object.

   dim newString as string
   if left(data,1)="#" then
      newString=right(data,len(data)-1)
   else
      newString=data
   end

   return

   RGB(val("&h"+left(newString,2)),val("&h"+mid(newString,3,2)),¬
      val("&h"+right(newString,2)))

End Function[6]
```

The control is now capable of constructing and positioning itself, given a description in XML. Our window setup method will need only to instantiate the object and send it the XML tag from the skin file. If this control were designed to respond to user interaction, we would define this in the subclass as well; for instance, the alarmButtonClass would need code in the Action event to handle turning the alarm on and off.

[6] In the last line of this example, I use the logical not character ("¬") as AppleScript does to show line continuation. Note that REALbasic does not support this notation, and this line must be entered without a line break.

The only other issue we need to address for the subclass is the capability to keep the control updated. In our example of the timeDisplayClass, we obviously need the control to constantly show the correct time. Writing a perpetually cycling loop into the class is a possibility, but it would be horribly processor-intensive and inefficient. Instead, we should add an update event:

```
Sub update()

   dim d as date

   // set the text of me to the current time
   d=new date
   if useLongTime then
   me.text=d.longTime
else
   me.text=d.shortTime
end

End Sub
```

When this method is called, the control will automatically display the correct time.[7] We will worry about how to periodically call this method in a moment.

Once a subclass is created for each different type of control, the next step is to create the window subclass. Every window in REALbasic is automatically a subclass of the window object anyway, and REALbasic places a window subclass (Window1) in every new project automatically. We'll use this one for our skin implementation.

Figure 7. The completed console window, with one of each type of control. Since these template controls are hidden, they can be placed anywhere in the window; the skin will clone them and place the cloned controls according to the skin file.

Since REALbasic can only instantiate new controls when they already exist in the window, we must place one of each type of control in the window. So that these control templates do not appear, set their "visible" property to false. To be able to create new instances of these controls, we must also set the index of each one to "0." Name each of these something easy to work with, like "timeDisplayTemplate" and "alarmButtonTemplate."

[7] Due to the way that REALbasic handles screen refreshes for staticText controls, this method will actually cause flickering to occur. It is better to use a canvas and rely on the drawString method instead, but this is outside the scope of our simple example.

In this example, we should rename the window we've created from "Window1" to "consoleWindow," since we'll need to refer to it that way later. We could even create several versions of this template window, each with a different procID, so our skin designers would have access to different styles of windows and different combinations of window controls.

Windows have something of a constructor method, the Open event, but it does not allow parameters to be passed to it; therefore, we will create our own constructor method and call it after the window is instantiated. Create a new method, "construct," and give it a parameter of "data as string" so that we can send the XML file as a parameter. Enter the following code:

```
Sub construct(data as string)

   dim i,numTags as integer
   dim tag as string
   dim cTimeDisplay as timeDisplayClass
   dim cDateDisplay as dateDisplayClass
   dim cAnalogTimeDisplay as analogTimeDisplayClass
   dim cAlarmButton as alarmButtonClass

   numTags=getNumberOfXMLTags(data) // number of tags in the file
   for i=1 to numTags // for each tag,
      tag=getIndexedXMLTag(data,i) // get the title
      select case getXMLTagName(tag) // create the appropriate control
      case "TIMEDISPLAY"
         cTimeDisplay=new timeDisplayTemplate
         cTimeDisplay.construct(tag)
      case "DATEDISPLAY"
         cDateDisplay=new dateDisplayTemplate
         cDateDisplay.construct(tag)
      case "ANALOGTIMEDISPLAY"
         cAnalogTimeDisplay=new analogTimeDisplayTemplate
         cAnalogTimeDisplay.construct(tag)
      case "ALARMBUTTON"
         cAlarmButton=new alarmButtonTemplate
         cAlarmButton.construct(tag)
      end
   next

End Sub
```

This code works through an XML file and instantiates a control for each control-defining tag. The code relies on a few other XML-related methods that must be included:

```
Function getNumberOfXMLTags(XMLdata as string) as integer

   // Given an XML file, this method returns the
   // number of tags in that file.
   return countFields(XMLdata,">")-1

End Function


Function getIndexedXMLTag(data as string, n as integer) as string
```

```
   // This method returns the nth tag in a given XML file.
   return nthField(data,">",n)+">"

End Function


Function getXMLTagName(tagData as string) as string
   // This method returns the name of the given tag
   // (so <A HREF="xyz.html"> returns "A").
   return right(nthField(command," ",1),len(nthField(command," ",1))-1)
End Function
```

The window can now instantiate all of its controls according to the XML file, and the controls will then initialize themselves. The only other duty we've outlined for the window is to periodically update the controls, and we can do this with a timer. Create a timer in the window and set it's action event like so:

```
Sub Action()

   dim i as integer

   i=1

   // we use while/wend here instead of for/next because
   // there is no easy way to determine the number of controls
   // in the window... so we cycle through the control array
   // until we run out

   while self.control(i)<>nil
      if self.control(i).index<>0 then // if it's not a template

         // we need to determine the class of the object
         // before we can update it, because the control
         // parent object doesn't have an "update" method

         if self.control(i) isA timeDisplayClass then
            timeDisplayClass(self.control(i)).update
         elseif self.control(i) isA dateDisplayClass then
            dateDisplayClass(self.control(i)).update
         elseif self.control(i) isA analogTimeDisplayClass then
            analogTimeDisplayClass(self.control(i)).update
         elseif self.control(i) isA alarmButtonClass then
            alarmButtonClass(self.control(i)).update
         end
      end
      i=i+1
   wend

End Sub
```

This code cycles through all the controls present in the window, checks to make sure they aren't the control templates we have hidden, and, if so, calls their "update" handler. The timer's period should be set to a time period that keeps the controls updated without bogging down the CPU; two or three times a second should work fine for this program.

The last step is to determine a method for loading the skin files (scanning a "Skins" folder on start-up seems to work pretty well) and instantiate one window per skin file. Since this code should occur when the program starts (but before any windows open), we should create a new class, set it's parent to "Application" and name it "App." This class now represents the application itself. In the Open event of this newly created class, we can place the following code:

```
Sub Open()

   dim skinsFolder as folderItem
   dim i as integer
   dim XMLdata as string
   dim t as textInputStream
   dim w as consoleWindow

   // first we find the folder named "skins", located in
   // the same directory as our application
   skinsFolder=getFolderItem("").child("skins")
   if skinsFolder<>nil then            // if the skins folder exists...
      for i=1 to skinsFolder.count     // open each file inside it

         // read the file into the XMLdata variable
         t=skinsFolder.item(i).openAsTextFile
         XMLdata=t.readAll
         t.close

         // create a new consoleWindow and run it's constructor method
         // with the XMLdata we got from the file
         w=new consoleWindow
         w.open
         w.construct XMLdata

      next
   end

End Sub
```

Final Thoughts

The example shown above is very simple, but the concept should work with a wide variety of programs. With more complex programs, programmers may of course run into situations not covered here that necessitate individual variation.

Most users consider skin technology a way to aesthetically customize their workspace, but the customization can be functional as well. A well-designed skin implementation can even simplify software localization and allow for disability access. While there are certainly types of programs that cannot implement skins for various reasons, the capability can add function and value to many more programs than currently take advantage of it.