SANView USER INTERFACE DESIGN MANUAL

written by Mark F. Martino
September 15, 2002

CONTENTS

1. INTRODUCTION

1.1 PURPOSE

This is a guide for directing the development of the SANView user interface and its supporting code. It is a tool to keep the user interface and its code consistent and predictable.

For simplicity and clarity, the principles, guidelines, and rules are written in severe and declarative language. This may give the impression there is no flexibility on these issues. This is not the case. Rather, this guide prescribes the normal way of designing user interface components for SANView.

If you believe there is a situation where the function of the component dictates violating one or more rules, rethink the design of the component. If there really is no way to implement the component without violating the rules, decide whether this is an isolated exception or the beginning of a series of design changes. If it is the latter, revise this document to reflect the change.

1.2 USAGE

Design documents which describe modifying or creating user interface components should adhere to the design principles described within and include a reference to this document. When coding new user interface components or modifying existing user interface code, follow the coding practices described in this document.

Quality assurance defect reports should refer to specific parts of this document when reporting user interface defects. For example, if a button does not appear to be consistent with other buttons in the application in appearance or behavior, report the button rule or rules that have been violated. Do this instead of reporting only that "the button is inconsistent" so the defect specified enough to be fixed.

When this document is revised, add a revision item to the revision history and change the revision number on the cover.

1.3 CONTEXT

Much of user interface design is setting a context and providing information and operations relevant to that context. The context for the principles, guidelines, and rules in this document is the user interface design problem SANView presents. Although each individual rule may be applied to other situations, the entire set, and the relationships between them, only make sense in the context of SANView.

Understanding this problem is essential to understanding the reasoning behind the rules. Understanding the reasoning is essential to judging when and how to apply the rules.

Read the following two sections for an understanding of the problem these rules are intended to address.

2. BOUNDING THE SANView USER INTERFACE PROBLEM

The easy part of bounding SANView's user interface design is acknowledging that it is a corporate application. Unlike personal applications, corporate applications do not require a large amount of adaptation to the personal preferences set in the platform. For example, while it is useful for a SANView user to set attributes in a profile, it is not useful to make SANView adopt a green color theme because the user has set this in the operating environment.

In fact, spending time and code on adapting to personal preferences and platform appearances is detrimental. A corporate application needs to look and behave exactly the same across all platforms for all users. Any adjustments the user makes are limited to what needs to be accomplished for corporate goals.

The hard part of bounding the user interface for a storage management application are the devices it must manage. New types of devices and new ways of accessing those devices and observing their status are always being developed. Some general assumptions can be made about the nature of these devices, but each device has a myriad of details specific to it.

The user interface design rules presented later leverage the easy part to make the hard part manageable.

3. CHARACTERIZING THE USER INTERFACE PROBLEM

We know we want the user to be able to:
- observe the characteristics, status, and performance of devices
- manage devices
- adjust device performance
- observe the architecture and characteristics of a network and its subnetworks
- observe the status and performance of a network and its subnetworks
- manage the network and subnetworks

The details of each device and the details of how they connect and communicate prevent us from applying the same user interface to all devices and networks. These conditions suggest a design continuum with one end being an integrated user interface and the other being a collective user interface.

An integrated user interface is one in which the each feature is designed to fit with every other feature, or most other features in the application. A lot of information is shared among the features.

Advantages of an integrated user interface are:
- User can usually accomplish a lot with a few mouse clicks and key presses.
- There is a high degree of consistency across features making them easy to learn.
- The relationship of the features is fixed so navigation is more intuitive.
- Clear relationships among features make it easy to program and maintain.
- Assumptions can be made for automatic responses.
- Unchanging API aids in integration with other applications.

Some of these advantages are reduced or even eliminated if the problem domain changes too much or too quickly or both. This weakens the case for an integrated user interface. Even though we can set some wide bounds to get us started, our problem domain is always in flux. This is where a collective user interface does better.

A collective user interface is one in which all or most features exist independently of each other. Little or no information is shared among the features. If information is shared, it is in an explicit and controlled manner. In addition to this advantage, a collective user interface has these advantages:
- Can be tailored to specific needs without dragging along superfluous features.
- New features are easy to add. Old features are easier to remove.
- New device models are easy to add. Old device models are easier to replace.
- It can be easier to learn since there are no superfluous features to get in the way.

- Code can be more easily modularized.
- May be easier to integrate modules with other applications.
- May be easier to automate tasks within a module.

These last two advantages depend on how the other applications have modularized their features. If the features we want to integrate with another application are spread across more than one module or if the APIs don't match well, this advantage is reduced.

Because SANView is a storage management application, another aspect of the user interface we must consider is real time feedback. We can provide the user with a consistent picture of the network or a concurrent picture of entities in the network, but not both. We can collect data for a set of networked devices over a fixed period and present it to the user with the caveat that some of those devices may have changed after it was queried. We can monitor one device almost continuously and provide almost real time details about it. The more we can merge these two types of views, the better. The best thing we can do for our users is be clear about how well we are doing this at any given time. The user interface should make this clear even if the user demands more.

Similarly, the user interface cannot exactly represent the appearance of physical devices. The interface should make it clear to the user that what they are seeing is a schematic or a diagram of the device which, at best, can show roughly the spatial relationships among items on a device.

All of the previous aspects of the problem suggest that each user interface component be as independent of context as possible. To be useful, some components may be programmatically tied to others, but the user interface designer should limit these couplings as much as possible. This following sections are based on this principle.

4. USER INTERFACE DESIGN CHECKING MECHANISMS

A way to maintain and extend a user interface and its code is to copy existing code and modify it. This breaks down quickly if the code is not reviewed after it is copied and modified. If there are mistakes in the code, they are propagated. If a new feature cannot be implemented with existing code, it may accidentally be implemented in a manner inconsistent with existing code. Therefore, apply these checks when you are coding or reviewing code.

There are three levels of these checks: principles, guidelines, and rules. To describe and explain them, they are presented from general to specific. In practice, they are usually applied in reverse order. One tries to apply a specific rule, if one cannot be found or if their are conflicts, move up a level.

If you are still in doubt about what to do, it may be helpful to look at "Java Look And Feel Design Guidelines" at http://java.sun.com/products/jlf/ed2/book/index.html. Remember these guidelines were intended to apply to a wide variety of users, applications, platforms, and devices with almost no assumptions about the user's knowledge. Applying these guidelines blindly without considering the relatively limited nature of the SANView client can get cumbersome. For instance, it is probably counterproductive to apply a tool tip to everything on the screen.

4.1 PRINCIPLES

These principles apply to all visual design. Rely on them when it is not clear what rule or guideline to use.

Principle 1: When in doubt, leave it out.

If you are unsure about a visual element, leave it out. If you are unsure about a feature, leave it out until you know better. It is tempting to add something because it is easy to do or because out of context it looks cool. Don't do it. Some things are easy to implement. Everything is hard to support and maintain no matter how simple it seems at first.

Principle 2: Make like things look alike, different things different.
If two items serve the same purpose in the same way in the same context, they should look the same. If they differ in those aspects, make the difference clear in their appearance.

Principle 3: Make it look like it is supposed to be there.
This may seem arty and squishy because it is, but it is the final test. If all of the elements in a design fit together and the visual flow takes the user along the path of least resistance to task completion, you've done your job.

4.2 GUIDELINES

Use these guidelines if you cannot find or decide on a rule for your specific situation.

Guideline 1: Plan navigation as early as possible.
Even a small number of windows, dialogs, and controls can seem complex if navigation is not designed into the application at an early stage. Examine the information and control that must be presented to the user and select the simplest navigation model for it. Stick to the navigation model and make it apparent to the users so they know to expect.

Guideline 2: Design for one platform. Adjust the others if necessary.
Since a corporate application needs to look the same across all platforms, we only need to support one look. We base our look on the Metal look and feel. It's appearance remains the same across all platforms and across the many varieties of Windows user interfaces. Supporting one look and feel makes SANView easier to code, reduces the size of manuals, and makes it easier to train users.

Guideline 3: Use a fixed set of colors, fonts, and icons.
There are three class files containing immutable data that, when used exclusively throughout the application, replace written rules concerning color, font, and icon usage. Those files are AppColor, AppFont, and AppIcon. Whenever you need a color, font, or icon use only those available in these files. If you believe they do not contain what you need, consult the user interface designer. The designer will tell you what item to use or help you create a new item to add to the appropriate file.

Guideline 4: Prefer reuse over copy and paste.
To imitate the look of a particular visual component, reuse its class as opposed to copying code from the class and pasting it into a new one. This implies designing visual component classes so objects of a visual component class can be reused in other classes. Doing this helps ensure visual consistency for that set of objects. Since they are all objects of one class, it is easier to keep them looking the same. It also helps ensure code and behavioral consistency, reduces the amount of code, and reduces the chances of introducing defects.

Guideline 5: The length to height ratio of text should not exceed 20 to 1.
This guideline is almost as old as type itself and applies even more so to computer screens. For 12 point type the height, with leading, is about 20 pixels so the length can be 400 pixels.

Guideline 6: Strive for a width to height ratio of 5:3 for windows, dialogs, and blocks.
If a window, dialog, or block of visual elements does not lend itself to this ratio, try a vertical layout of 3:5. Make a window's size adjustable only if the number of elements or size of those elements can change unpredictably.

4.3 VISUAL LAYOUT RULES

These rules suggest how to select some of the more common attributes of user interface components and how to arrange them on the screen.

4.3.1 COLORS AND TEXTURES

Colors other than white, black, and gray should be used sparingly. This is because in the context of SANView, color is used to indicate the state of devices or in images of physical devices. Using color for purposes other than these will confuse the user and is likely to make screens hard to read. Colors representing state or health must be consistent from screen to screen. That is, the green for health must be the same green on all screens.

1) Use only the colors in AppColor. If you need a new color, add it to AppColor so it may be reused.
2) Unless otherwise noted, use untextured white background, untextured black foreground.
3) Use white for window and dialog backgrounds.
4) Use light gray or white for backgrounds in charts, maps, and diagrams.
5) Use consistent shades of gray. In most cases, three should suffice.
6) Use colors other than black, white, and gray only in charts, maps, and diagrams.
7) Use color only if it communicates more information than you could otherwise.
8) Use colors, especially saturated or bright ones, sparingly.
9) When colors are used together, they should differ in value as well as hue or saturation.

4.3.2 TEXT AND FONTS

Most of our application is made up of text in the form of labels, text fields, text areas, and text lists. They, and the rules that apply directly to them, are defined in their own section. This section applies to text in general.

Keeping these text items consistent with each other goes a long way to making our design clean and easy to use. Use font size and style to help group related lines and blocks of text. Normally one serifed font, one san serif font, and one monospaced font in three to five sizes each and in regular and bold styles provides enough variety to distinguish lines and blocks of text.

Do not use italics, strike-through, or any other font style because they are difficult to read. Underlining is reserved for hyperlinks as described below.

1) Use only the fonts in AppFont.
2) Use the 12 point sans serif regular font for everything unless otherwise noted.
3) Use the 12 point sans serif bold font for window and dialog titles.
5) Do not used underlined, strike-through, or italicized fonts.
6) Length of a line should not exceed 400 pixels.
7) Height of a line with leading is 20 pixels.
8) Height of a text field is 20 pixels.
9) Editable text should appear in a regular font style. Avoid bolding editable text.
10) If you must use a colored font or colored background, follow the color rules.
11) For all hyperlinks, use one standard blue color and the 12 point, serifed, underlined font.

4.3.3 NAMING

Take time to think about the names of menus, menu items, buttons, and so on. Think about the controls and visual elements around it and about the ones it works with. Make sure the name of the new item makes sense in that context. Use a dictionary and a thesaurus to find names or at least generate ideas for names.

1) Capitalize all the words unless it is a full sentence or more than five words.
2) If it has a subject and predicate, it's a sentence no matter how weird it may sound.

3) The titles of buttons and menu items must be verbs or verb phrases.

4.3.4 LAYOUT AND PLACEMENT

The only reliable way to determine the optimal shape, color, size, name, and location of a visual control is to make a good guess, have users test it, and try again. Use these rules to help determine your first guess. Adjust them with user feedback thereafter.

1) Place items in the order they are likely to be used, top to bottom and right to left.
2) If there is no clear functional order, alphabetize the items in the local language.
3) Place no more than seven, ideally only five, items in a single row.
4) If you have more than seven items, put them in one or more columns.

4.3.5 BORDERS

Borders can be easily misapplied. More than any other element, if you are not sure if you should use a border, leave it out.

1) Use borders to visually group items that are behaviorally related.
2) Do not use borders merely for decoration.
3) More than three bordered groups suggests the need for another window or dialog.
4) Use a black, one pixel border.
5) Put a four pixel margin between the border and the elements it contains.

4.4 COMPONENT USAGE RULES

These rules provide advice on creating, modifying, and deploying the standard types of user interface components available in SANView.

4.4.1 WINDOWS, DIALOGS, AND PANELS

To clearly discuss the rules for windows, dialogs, and panels, we need to define these terms for our purposes.

Window - an independent rectangular area on the screen with a frame and menu. Except for an identifying icon in the upper left corner of the frame, the frame's appearance is not under the control of Java. It is defined by the platform. By independent we meant that normally a window and its contents are not affected by the activities in other windows.

Dialog - similar in appearance to a window, but is owned by and is dependent on, a window. By dependent we mean dialogs may be affected by activities in the window or in other dialogs of the parent window. It also means a dialog inherits the icon of its parent window. Unlike a window, a dialog can be modal or non-modal. A modal dialog does not allow the user to access any other window or dialog in the application while it is open. These types of dialogs are reserved for short lived activities like informing the user or allowing the user to make simple selections. Non-modal dialogs are used for longer activities and may persist as long as the window itself.

Panels - a rectangular area in a window or a dialog. The boundaries of a panel are not normally visible or relevant to the user because a panel is a unit of organization for the designer. There can be many layers of panels within a window, dialog, or another panel.

SANView users seem to prefer to have one window with a large variety of information displayed as opposed to having to click through a number of windows to get that information. This must be balanced by what can fit on the average screen and what the average user can absorb from one view. Prefer a shallow hierarchy of windows and dialogs containing as much related information in each as possible.

Windows should be treated as small applications. That is, use windows only for sets of functions and features that can stand alone. If any data must persist from window to window or be used by more than one window, this data must be placed in a common repository accessible to all interested windows. The data should not be stored in any one window because there is no guarantee that window will be available when the data is needed.

Use dialogs for features and functions related to a given window. All dialogs must be associated with a parent window so they close when the window closes and so the dialog displays the corner logo.

Panels should be used as a means to construct viewable blocks of components. Panels should not be used as data repositories. Do not reuse a window or dialog for an entirely different set of tasks by replacing panels within it. Instead, create a new window or dialog for each set of tasks.

1) All windows and dialogs must display the company logo in the upper left corner.
2) Dialogs must get their data from or through the parent window.
3) If a dialog must use data that is also used by another window or dialog, the dialog must get the data from the common repository, not the window or dialog to ensure consistent data.
4) If a window or dialog is likely to become too large for the minimum screen size, be sure to insert code to adjust the window or dialog to fit into the screen. If possible, adjust to the smaller size by reducing the size and spacing of the contents as opposed to making the window scrollable.
5) Except for unavoidable screen size problems, do not make entire windows or dialogs scrollable. This invites the problem of hiding critical controls.

4.4.2 MENUS AND MENU ITEMS

Menus and menu items must have the same appearance throughout the entire application. Therefore, all menu and menu item fonts and colors should be set in the application's common look and feel. To ensure predictable behavior of menus and menu items, rely on the standard menu and menu item handling mechanisms in the language. There is no need to create external menu handling classes or methods.

1) Prefer standard menus over popup menus.
2) Commonly used menus and menu items should have their own classes.
3) Menus and menu items that can be used without change and whose resulting actions are always the same should be instantiated once and used throughout the application.
4) Menus and menu items whose titles are the same from window to window, but whose resulting actions vary must be instantiated within the window in which they are used.
5) Separate the action taken from the menu item. An action may need to be triggered from another menu, a button, or via some other user activity.

4.4.3 ICONS

1) Use the icons in AppIcon for application wide icons.
2) Do not add icons to AppIcon without first consulting a user interface designer.
3) Set the upper left corner icon in all frames.
4) Assign a dialog to a frame to insure that the corner icon appears in it.
5) If some or all of an icon is drawn in a paint() method, make it a stand alone class that can be instantiated in the AppIcon class.

4.4.4 PROGRESS BARS

The two types of standard progress bars are the percentage bar and the cycling bar. The first represents the amount of a task or the time for a task that has been completed. The second only indicates that a task is in progress and it is not clear when it will end. This is similar to the

function of the wait cursor, but each instance of the cycling progress bar is associated with a particular task.

1) If the time to execute a task can be predicted or if the task has a finite set of relatively short term steps, use an instance of the standard percentage progress bar class with its default properties. Otherwise, use the cycling progress bar with its defaults.
2) When using the percentage progress bar, display the percentage of task completion as a number along with the bar.
3) If you need to tailor either type of progress bar, adjust the attributes of the instance as opposed to creating a new class.
4) Do not allow the user to adjust the size of the bar or the dialog containing the bar.

4.4.5 SCROLLBARS

1) Show scrollbars only when the content exceeds the size of the container.
2) If possible, avoid using a horizontal scrollbar.
3) If possible, avoid using both a vertical and horizontal scrollbar.
4) Avoid using scrollable areas within a scrollable area.
5) When applying scrollbars to windows and dialogs, ensure that all controls are visible when the window or dialog first appears.

4.4.6 COMBOBOXS

1) If the list of items in a combo box exceeds seven items, make it a list.
2) Prefer an independent text field to an editable combo box.

4.4.7 BUTTONS

Screen elements that behave like buttons must look like buttons. If the user can click on an item to invoke an action, that item should be derived from a button class or be programmed to look and behave like button. That is, it should indicate when it is depressed and released.

4.4.8 LABELS, TEXT FIELDS, TEXT AREAS, AND LISTS

Text fields, text areas, and text lists are defined here as they appear and are used in SANView.

Label - A single word or short phrase whose background is the same as background of the panel on which it exists. It is not user-editable, but it may be programmatically changed.

Text Field - A bordered rectangle which can contain one line of user editable text.

Text Area - A bordered rectangle which can contain one or more lines of user editable text.

Text List - A bordered rectangle which can contain one or more lines of user selectable text. The text is usually not editable. Depending on the task, one or more than one line of text can be selected.

1) Unless otherwise noted, labels use the bold, san serif font.
2) Labels not associated with active elements must use the regular, san serif font.
3) Labels use black text and the same background color as the panel on which it is placed.
4) Labels for text fields must be right justified and place on the left edge of the field.
5) Labels for text areas and text lists must be left justified and placed above the area aligned with the left side of the area. This arrangement may also be used on text fields if there is not enough space to apply rule 4.
6) Use the 12 point serifed regular font in text fields and text areas.
7) Text fields and text areas use black text with a white background.
8) Set the background of disabled text fields and text areas to gray.
9) If column alignment is necessary within a text area, use a monospaced font.
10) Avoid using directly editable text lists.

11) Use single selection lists unless there is a good reason to allow the user to select more than one item.

4.4.9 TABLES AND TREES

All tables should have the same appearance throughout the application. Tables should be differentiated only by the number of their rows and columns, the titles of the columns, and the information in the columns. Trees may vary in appearance and behavior but should not do this arbitrarily. That is, variations in how a tree appears or behaves must convey information that could not otherwise be represented.

1) Tables should display at least ten rows even if some of the rows are blank.
2) If a table has a finite, maximum number of rows less than ten, display the maximum number of rows.
2) Avoid using horizontal scrollbars on tables if possible.

4.4.10 TOOLTIPS

The tooltip was originally conceived to allow the label beneath an icon to remain hidden unless the user needed to see it. Since then, the purpose of tooltips has expanded to include longer functional explanations, full help text blocks, and large blocks of informational text.

These are not bad ways to use tooltips; however, it has led some to believe that absolutely everything on the screen should have a tooltip associated with it. This not only increases the amount of programming to do and text to track and localize, but it confuses the user. Tooltips used in this way often make the user interface clumsier and harder to use. For instance, attaching tooltips to every field in a dialog causes a tooltip to pop up every time the user pauses to think about the next text entry.

In many cases, a tooltip is only helpful once or twice. Once the user understands the purpose of a screen element, they no longer need that particular tooltip. It is like being followed through a store by an overly helpful clerk asking "May I help you" every time you turn around.

Because SANView is a targeted, corporate application with training available for it, there is even less reason to attach tooltips to everything. Therefore, the following rules enforce selective use of tooltips.

1) All tooltips must use the same font, foreground color, and background color.
2) Tooltips should be no more than three lines long. Text that appears when a component is rolled over and is longer than three lines should have its own dialog or panel in the window.
3) Text fields, areas, and lists should not have tooltips.
4) Labels of text fields, areas, and lists may have a tooltip if the purpose of the field is not clear.
5) Do not use a tooltip in place of business rules. That is, if a text field requires a specific format or specific information, do not attempt to explain those rules in a tooltip. Enforce the rules as automatically as possible. If no dashes are allowed in a field, don't accept that key entry. Put up a dialog explaining the rules if the rules cannot be enforced automatically.
6) If a screen element may use a tooltip to either explain its usage or present blocks of information about whatever the element represents, but it can not and should not do both.
7) Do not use tooltips instead of consistent button functionality. The function of common buttons like "Close", "Exit", and "Delete" should be made clear through their consistent behavior.
8) Menus and menu items should not have tool tips. The text in these elements must make their purpose clear.
9) Avoid tooltips on often used icons. If an icon will be used often, its image and context should make its purpose and usage clear. Even if there is some initial ambiguity, the user will learn after using it a few times.

10) Scrollbars should not have tooltips.
11) When in doubt, leave a tooltip out unless testing and focus groups prove otherwise.