Metaphor – A Pattern Language for User Centered Software Design

Todd Coram (todd@btg.com) and Jim Lee (jlee@btg.com)

July 25, 1996

> Software design is the act of determining the user's experience with a piece of software. It has nothing to do with how the code works inside, or how big or small the code is. The designer's task is to specify completely and unambiguously the user's whole experience. — *David Liddle, Bringing Design to Software* [Liddle96]

Here are the beginnings of a pattern language that can be used to generate software designs which are *user centered*. User Centered Software Designs place the user first and foremost. We will be concentrating our patterns at the point where a user interacts with the application. This is a critical area, often not given its due. Whether a software system works or not, the interface is the most important feature. This is true for all software. What use is a system call to read data off of a disk if it cannot return that data? Even if the read routine was poorly implemented or contains faults, our only way to judge it is through its interface. Software engineers and programmers expect well defined and consistent interfaces to their tools. Object oriented programming teaches us that the guts are replaceable, but the interface is forever. Ever notice how the Unix system's read() function interface seems frozen in time?

As tricky as a programmer's interface seems to be, it is even more difficult to define interfaces for less computer-capable individuals. We cannot always prevent errors from occurring in our software, but we need to convey these errors and failings in meaningful ways to our end-users. Perfectly working software systems can be rendered useless when fronted by unusable interfaces.

By using the patterns described here, you should be able to develop languages which help you build user interfaces and structure your software to be driven by user interactions. You won't find information here on how to use icons, pop-up menus, dialog boxes and other interface objects, but we will show you *when to use them*. Our primary focus is on the higher level patterns that can be found in all good user interfaces, but we will take you down to a low enough level to give you practical patterns you can use to select your tools.

In order to show how all the patterns fit together, we have drawn a map (see Figure 1).

Here is the summary of the language:

The following pattern leads us into the heart of the language:

PRIMARY INTERACTION METHOD(1)

Here are the types of mechanisms that are available to the designer. They make up the four major user interface paths to take in this language:

CONVERSATIONAL TEXT(??) <not included here...>

ENTRY FORM(??) <not included here...>

SELECTION MENU(??) <not included here...>

WIMP INTERFACE(2)

The different metaphors that drive our language are:

SINGLE SETTING(3)

MULTIPLE SETTINGS(4)

COMMAND CONTROL CENTER(5)

GARDEN OF WINDOWS(6)

ORGANIZED DESKTOP(??) <not included here...>

COOPERATING WINDOWS(??) <not included here...>

GOAL ORIENTED AREAS(7)

MODELESS FEEDBACK AREA(8)

Sound is an often ignored medium. Here are two types of sound patterns:

REWARDING SOUNDS(??) <not included here...>

WARNING SOUNDS(??) <not included here...>

Humans are visually oriented creatures. Here are some important patterns to help us understand what we see:

VISUAL SYMBOLS(9)

CLICKABLE SYMBOLS(10)

SYMBOL EXPLANATIONS(11)

CONTEXT SENSITIVE HELP(??) <not included here...>

Gadgets, widgets, components and gizmos. These are the concrete things interfaces are made of:

TOOLBAR(??) <not included here...>

MENUBAR(??) <not included here...>

LAUNCHPAD(??) <not included here...>

PALETTE(??) <not included here...>

The eleven included patterns are presented in a modified version of the form described by Christopher Alexander [ALEX77]. Whereas Alexander ended the body of his pattern with the solution, we place the solution after identifying our forces. We chose this form because we wanted to resist casting the patterns in a *recipe* form. We believe it is truly a *language* and as such, resistant to highly structured presentations.

This language, although not complete, contains many smaller languages that the user is encouraged to derive in order to suit the needs of the type of application. All languages will begin with PRIMARY INTERACTION METHOD(1). From there, you are offered four paths. Once you have chosen a path, your own language will evolve based on your application needs and desires.

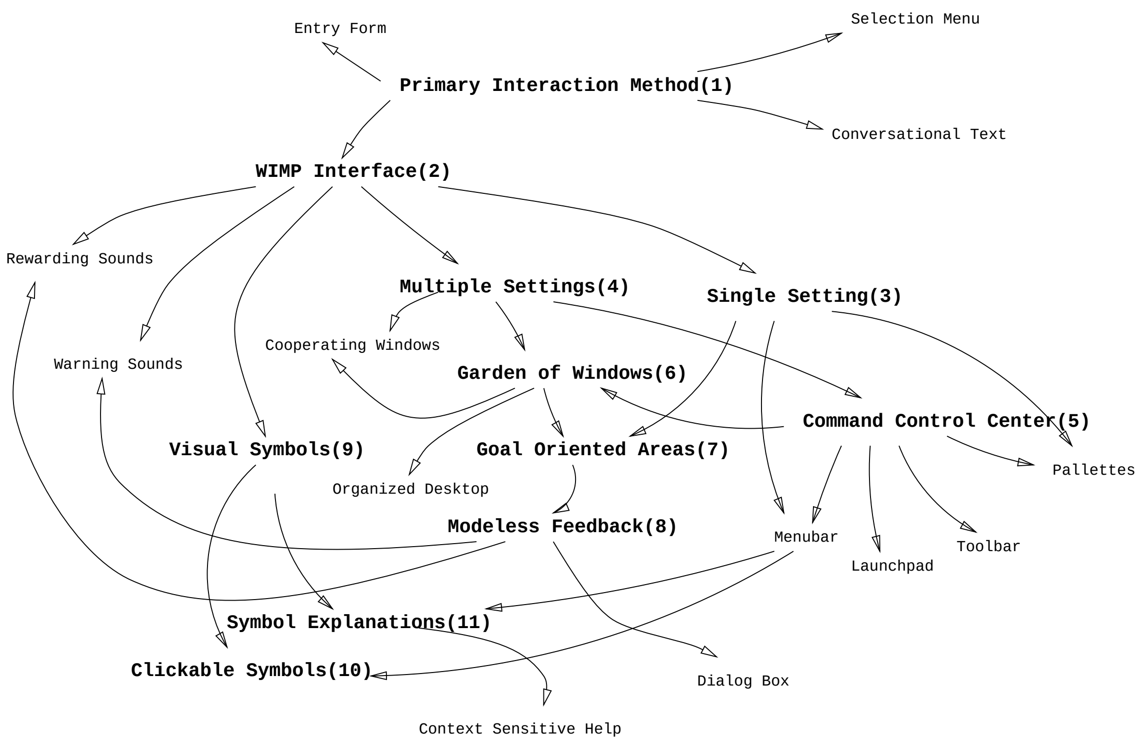

Figure 1

In this map, we have shown the patterns identified so far. The ones in bold are defined in this paper, the others are forthcoming. There are many more not even hinted at. We see this language growing to as many as a hundred related patterns.

---

1 Primary Interaction Method

...there are many ways the user may interact with your system. Now is the time to figure out how you will work with the user to attain his goals.

It is essential that the interaction style matches the needs of your audience.

You need to provide an interface for the user to interact with the system. But how much computer experience does your user have? How much training will your user receive? For instance, text editors such as vi or emacs are useful tools for most programmers, but inappropriate for occasional users.

As a designer, realize when users become frustrated with a program, it is because they know what they want to do, but cannot because the program has a "secret language" that the user does not understand.

The system you are creating is supposed to be a tool to help the user with his work.

Therefore:

Study the user and his environment. Work with the user to determine what interaction style is best. Keep things simple and consistent. There are several primary interaction styles: menu selection, form fill in, command language, and direct manipulation [Shneiderman92].

Use Menu Selection to structure the decision making path. Use Form fill in (Forms Entry) for data entry tasks. Use Command language (Conversational Text) when flexibility is an overriding factor. Use WIMP Interface to simplify the command syntax for highly interactive tasks.

The computer interface can be an unfamiliar and unnatural thing, but a well designed interface will minimize the gap between a user's goals and the knowledge required to use the interface. Edwin Hutchins, James Hollan and Donald Norman [Norman86] defines these as the Gulf of Execution, the effort required determining how to get a program to do what the user wants, and the Gulf of Evaluation, the effort required to interpret the feedback provided by the program.

Selecting an inappropriate interaction style could increase the Gulf of Execution. Consider a factory foreman with poor typing skills. Providing a command language interface would force this user to do a lot of typing, which in this case would exceed the user's capability, and would almost certainly result in a frustrated foreman.

This Gulf of Execution is lessened when the objects and commands of the software match the processes, goals, and ability of the user. The Gulf of Evaluation is lessened when the data is displayed in a format that is readily perceived, interpreted, and evaluated.

It might be better for our foreman if he were given a menu driven interface. He could be presented with a list of options, which could be selected with a single keystroke.

SELECTION MENU(??) is easy to learn, reduces keystrokes, structures the decision making path, and allows fairly easy support of handling errors, however may slow down frequent users when there are many submenus. FORMS ENTRY(??) is simple and requires minimal training, hovever it often consumes quite a bit of screen space, and the user must be capable of responding to error messages when invalid entries are made. CONVERSATIONAL TEXT(??) systems allow the user to express complex possibilities but often provide poor error handling capability and require substantial training and memorization in order to learn the command syntax. A WIMP INTERFACE(2) presents task concepts visually, making it easy for the user to learn and retain the required steps, but are a bit more difficult to design correctly, and require some sort of graphics display and pointing device....

2 WIMP Interface

...when you and the user agree that a direct manipulation interface is the most appropriate for the system (Primary Interaction Method(1)), it is still necessary to provide the user, who has knowledge in his task domain, a path to attain his goals even if he is not computer literate.

A user will be aprehensive of interfacing with the application when they view the computer as nemesis. An relationship needs to be formed between the user and the application. The interface must provide the means to do so.

For experienced users, prior bad experiences make them wary of using your application. That user wants an application interface that they don't feel timid around, an interface that doesn't feel dangerous to use.

If the user accidently deletes text, or bolds a selected paragraph instead of justifying it, he wants to undo that mistake. Applications which do not provide that functionality are mistrusted. The user will never venture beyond the safety of well known features and commands.

Both novices and experienced users will not only want a trustworthy interface, but will want the computer to help them complete their tasks in a manner that puts less of the burden on them, and more on the computer.

Complicated or boring interfaces (especially those that require repetitive actions) make the user aware that a computer is situated between them and the work. A person with low motivation will become even less motivated when confronted with a complicated interface. An ideal interface is *transparent* and lets the user concentrate on the problem at hand.

Computers are everywhere today. This widespread use creates a more diverse set of users. People with limited computer literacy and mediocre typing skills are now often required to use computers on a daily basis. There are also situations where a high turnover rate is normal, or there is a low frequency of use for the application. In these situations, extensive training on how to use the application is not cost effective. So an interface with a blinking cursor and a prompt that says "OK" is not okay. You must help maintain the productivity level of today's computer operator.

Therefore:

Engage the user, draw him in. Provide an interface which allows the user to experiment and explore the application, but don't let the interface get in the way. In order to keep the user engaged, you must build trust. Don't allow the result of manipulation to destroy the user's trust in the application. Any user initiated command should be immediately reversible or cancelable.

The point and click aspect of a direct manipulation interface presents the user with objects and actions she is familiar with. This helps minimize the amount of computer literacy and memorization required to use the interface. The low motivation types become engaged with the little gadgets they can click on. Ben Shneiderman [Shneiderman92] categorizes direct manipulation interfaces as having the following properties:

1. Continuous representation of the objects and actions of interest
2. Physical actions or presses of labeled buttons instead of complex syntax
3. Rapid incremental reversible operations whose effect on the object of interest is immediately visible

Engagement occurs when a user experiences direct interaction with the objects in a domain; the feeling of manipulating the actual object instead of computer representations. A similar type of feeling happens at the movie theater when a willing suspension of disbelief occurs. A good film will engross the viewer, draw the viewer into the plot, forget that he is at the movies. However, the movie must not mislead the viewer—

destroy the suspension of disbelief and you will cause the viewer to mistrust anything that happens in the rest of the movie.

Most if not all word processing software today use some sort of what-you-see-is-what-you-get (WYSIWYG) interface. When the user changes the font style, the display is updated to show how the page will look when the document is printed. If the user doesn't like what he sees, then he can choose another font or go back to the original. When the user prints the document, and suddenly realizes that the printer is not configured to use the font, he can cancel the print job before it wastes lots of paper. The user must feel that it is safe to play with the interface.

Now determine if the majority of the user interaction will occur in a Single Setting(3) or if Multiple Settings(4) will be required to support all of the user's activities. Provide appropriate Visual Symbols(9) to keep the user engaged and informed. In addition to visual feedback provide positive audio feedback (Rewarding Sounds(??)) or negative audio feedback (Warning Sounds(??))...

3 Single Setting

...one form of WIMP INTERFACE(2) involves applications or subsystems that support a *single conceptual focus*. These packages, such as simple word processors, text editors, paint programs, forms entry systems and terminal emulators, are focused on performing one basic task well. The focus is usually on the manipulation of a certain contextual type of task or data.

Single conceptual focus applications must not distract the user with unnecessary interface artifacts.

In a single concept application, lots of busy satellite windows can distract the user from the current task. These windows can draw the user's attention away from that basic task and break his train of thought.

Users want to focus on the problem at hand. Other windows should not physically get in the way of the data being focused on. Loss of focus can lead to catastrophic situations.

Therefore:

Choose a primary setting for the focus and don't let it be obscured by supporting agents such as toolbars, palettes and other windows. These agents are scenery for the primary thread of interaction. It is imperative that single concept applications always keep you in the context of the data being manipulated.

Consider a user whose primary task is to track enemy planes by observing and reacting to data displayed in a radar window. The user must maintain focus on this window. Any other windows displayed must support this task and not obscure the radar display. An example supporting window can be one that shows additional data on the capabilities of a plane being tracked. If this window obscures the radar display, the user cannot continue to track the location of the planes until it is dismissed.

In *Computers as Theatre* [LAUREL92], Brenda Laurel relates software to the field of Theatre.

Let's introduce the concept of a *single setting*. A single setting is the opposite of the theatrical term *multiple settings*. A single setting is focused on one particular thread of action occuring on the stage. It is here that all of the audience's attention is drawn. There is no additional action going on outside of the main dialogue. The scenery is supportive, not distracting.

Simple paint programs, such as the original MacPaint, are examples of a single setting. Your focus is on the painting at hand. All the other icons and the palette exist as scenery; the selected tool is an agent. All work together to present a single setting.

Even in more modern paint programs, such as Fractal Design Painter, although there are palette and tool windows that can overlap the painting canvas, the focus on the *painting at hand* remains. Additional canvas windows represent their own single settings. You focus on one canvas at a time. A canvas used as a *clipboard* is just another agent for the target canvas.

In most window based systems, this type of application almost always require a master MENUBAR(??) and one instance of GOAL ORIENTED AREAS(7). In addition, since the application will be the center of attention, compliment it with a rich assortment of TOOLBAR(??) and PALETTE(??) components (where applicable)...

4 Multiple Settings

...one form of WIMP Interface(2) involves applications or systems that support multiple tasks, often without a single focus. Packages such as desktop publishing systems, desktop operating environments, integrated development environments and animation systems, perform many different (but related) tasks to get the job done.

The system needs to perform multiple related tasks to achieve the user's desired goal.

When more than one task must be performed to get the job done, the tasks must work together seamlessly. This means multiple applications are required to cooperate.

Even though the applications are bound together by a single purpose, they may operate on different data or work in different settings.

Complex systems that perform many interelated tasks usually contain quite a few windows. Users can get lost in a myriad of windows.

Therefore:

Provide each application with its own setting and use navigational tools to tie them together. These settings must cooperate and work together, without requiring the user to halt or quit applications to reach each other.

Consider a desktop publishing system. What if you want to print a document before saving it? Does the print task cooperate with the editing task? Would you want to use the spell checker if it required you to save and exit the document you are working on? What about touching up photographs? Will that require you to shutdown editing mode and enter a new application? A decent publishing system will tie those tasks together seamlessly.

In Theatre terminology, multiple settings is defined as more than one setting on a stage at the same time. Jonnie Mobley provides the following example [Mobley92]:

> In this century, English playwright Alan Ayckbourn has used a multilevel setting for several of his plays. In *How the Other Half Loves*, two couples have their living rooms on the stage at the same time. The couples pass without seeing each other. At one point, two dinner parties, on different nights but involving the same guests, are enacted simultaneously with the table at center stage being in each couple's dining room. The timing by the actors must be impeccable to make it work, but it provides a delightful farce, due in no small part to the ingenuity of the multiple settings.

Just as we would shift our concentration to and from each dinner party, a user needs a mean of navigation to shift from one task or setting to another. While the *focusing* of tasks is mostly controlled through user initiated navigation, all tasks must cooperate in order to pull off our *farce*.

Contrast this pattern with Single Setting(3), but keep in mind that a multiple setting application can be seen as a collection of these. Regardless, you will need to define your Garden of Windows(6), Cooperating Windows(??) and a Command Control Center(5) to tie them together...

5 Command Control Center

...when your application has MULTIPLE SETTINGS(4), each application must be easily accessed and integrated in order to create the sense that everything is leading the user to reaching the established goal.

Users need a well defined mechanism to initiate tasks and navigate among them.

When there are many tools, windows, or areas of concentration that a user must reach, users will need a mechanism to get them there.

Users want a consistent interface that is always present and easy to reach, an interface they can use to instruct the application to do their bidding.

Separate windows in an application should not esxist without any well defined way to control them. This is especially true in a multitasking environment, where multiple distinct applications may be running at the same time. The user needs to know what visible or iconified window belongs to a particular application.

Therefore:

Provide a command control center that is always present, easily accessible and provides quick access to tools, tasks and windows. If a single control center window cannot be provided, present your control center as commands that are accessible from every window's menubar.

The command control center acts as the "home base" for coordinating applications within Multiple Settings, or for coordinating secondary windows within a Garden of Windows.

Consider an integrated development environment (IDE) that allows the users to bring up the debugger only from the compiler window's menu bar. What if the users wishe to debug a previously compiled program? How will they know that they need to bring up the compiler window to access the debugger?

In Symantec Cafe (a Java program development tool), the users know that they need to perform tasks such as edit, compile, run and debug. But, how do they navigate through these tasks? They use menubars, toolbars and other controls to reach each task. Once these tools are identified, they should be easy to locate and provide navigation among tasks as seamlessly as possible. In Cafe, each task is associated with a window. A command center model is adopted that makes it easy to select tasks by choosing windows.

The users remember where these controls are available and rely on them to *get around.*

When building your control center, you will want to use things such as a MENUBAR(??), or perhaps a LAUNCHPAD(??) to provide navigation among your GARDEN OF WINDOWS(6), while a TOOLBAR(??) or PALETTE(??) provides access to tools...

6 Garden of Windows

...for each application within MULTIPLE SETTINGS(4) the user may be performing more than one distinct task at a time or may require several views of information in order to get the job done.

You need a way to partition these tasks into assimilable pieces so that the user is not overwhelmed.

Placing too much functionality and data in a single window clutters that window and will overwhelm the user with information. Lots of graphs, scrolling text, blinking icons and multiple views of the same data can give the user a headache.

A huge single window is essentially a throwback to the days of screens and consoles. The applications tile the screen into areas of functionality. These areas may not even be related. In single task applications, such as a text editor, this may be fine. But in multiple task applications, this will only clutter the single window.

A user can only focus on one task at a time. A window nicely encapsulates a single task.

Therefore:

Use a separate window for each distinct task and each unique view of data that is related to the same application.

In Netscape Navigator, a huge single window is used as the primary canvas for all web page content. In order to prevent the user from having to do page loads for additional *windows*, web pages are often broken up into *poor man's* windows: frames. This is a poor partioner of tasks. They are still encapsulated by one window (the Navigator) and each begs for a user's focus.

In a typical integrated development environment (IDE), you have a collection of tools to help you get the job done. Each tool carries out a distinct task: composing build rules, editing source files, setting up resources, building GUIs, compiling source files, and running the resulting program.

It is the fact that these tasks can operate somewhat independently that makes an IDE useful. You can build the GUI in one window, and edit the code in another. At any time, you choose which window you wish to focus on, while keeping them all close at hand.

A user may want to see data visualized in different ways. Therefore, it may be useful to display data in both tabular and graphical form. The tabular form displays exact values while the graphical form allows the user to visualize trends and patterns. An example would be an EKG display which shows the exact pulse and blood pressure digitally, but also provides a display showing the heart rhythm.

Use ORGANIZED DESKTOP(??) to help determine how to arrange your windows.

Use a COMMAND CONTROL CENTER(5) to help manage the workspace, and allow for communication and organization between tasks through COOPERATING WINDOWS(??). Each of these windows may consist of one or more GOAL ORIENTED AREAS(7)...

7 Goal Oriented Areas

...whether you are working with a SINGLE SETTING(3) or a GARDEN OF WINDOWS(6) several functions may need to be performed in order for a user to achieve his goal.

Related functions, composing a task, need to be organized in a coherent manner.

Poorly organized layout is intimidating and difficult to use, so you must figure out how to group functions and tasks in ways that make sense.

Interfaces that require you to *jump around* your screen, selecting functions that affect your task at hand from nested windows, are inefficient and annoying, because they interrupt the natural flow of work.

While throwing all related functions into a window keeps them together, they must be grouped in such a way that the user sees a logic in the layout.

Therefore:

Group related functions, that work toward completing a singular goal, logically into areas on the same window. These functions should be organized into physically separate regions of that window based on closeness of relation. Floating palettes may be used to perform grouping, but new windows should be reserved for additional tasks that are not related.

The purpose of a kitchen is to prepare a meal. This involves getting food from the refrigerator and pantry, pots and pans from the cabinets, cooking the meal on the stove, and washing up afterwards. These are the related subtasks that are performed to achieve the goal of preparing a meal. The one tool one window approach to interface design is shortsighted. It would be like designing a house with a refrigerator in one room, the stove in another, the dishwasher and sink in yet another room. Consider grouping subtask functions together in the order they are typically performed.

If you are designing a drawing package, functions such as creating rectangles, creating circles, changing colors, erasing, and filling polygons should be easily accessible and not buried within several layers of sub-windows, nor should they be placed arbitrarily around the drawing canvas. They should be grouped logically so that functions can be found next to their closest kin. For example, keep all the *create* functions (rectangle, circles, etc) together in one area on the window.

Consider frequency of use as a way of grouping items. From the vast number of colors a painter has to choose from, she places the few that she needs for this particular painting on her paint pallette. Likewise, frequently used tools can be grouped in floating palettes that are not physically constrained by the drawing window, but are *logically* part of the drawing window.

Commands can be grouped in a MENUBAR(??), or in order to reduce window clutter, you can use PALETTES(??). When you need to group related data together, use a MODELESS FEEDBACK AREA(8)...

8 Modeless Feedback Area

...when using Goal Oriented Areas(7) a place should be provided to display system feedback. Use this pattern when you wish to provide non-critical information to the user.

There is a need to display information to the user without disrupting the natural flow of work.

For every action the user takes, the system should provide some feedback to assure her that the program has accepted the command. Or when some change in system state occurs, the user should be notified. However, the users should not be forced to interrupt their sequence of actions in order to respond to informational feedback.

You may be tempted to deliver application warnings in dialog boxes, but it can be inappropriate to obscure the main thread of interaction with a pop up dialog window.

Therefore:

Use a status area, rather than dialog boxes or popup windows , to display information about the current state of the application and to inform the user of errors that do not affect the current thread of interaction. Place it near to the main display area, so the user can view this information at the time she deems appropriate.

Just imagine an air traffic control system which has a pop up dialog informing the operator that her disk is 90% full. The intention is good, however, the execution could be detrimental. Since a pop up dialog box puts the program in a state that the user must respond to before returning to the task at hand, it should be reserved for when the normal thread of interaction must be altered.

Many word processors have a status bar at the bottom of the display indicating page number, line number, etc. Microsoft Word uses this area to display information about what a Toolbar button does.

The status area need not be restricted to text. A graph or a set of gauges can be just as effective. Netscape uses an animated icon to indicate when the Navigator is processing a request.

Be careful not to overload the purpose of the status area. Non-status information should not be displayed in this area because it will diminish the area's meaning. It will instead become a Noise Area which the user will learn to ignore. An example is the way some Web pages use the status area at the bottom of the Netscape Navigator as a ticker tape marquee.

When the environment is not required to be quiet (e.g. library), or not too noisy (e.g. factory floor) Rewarding Sounds(??) and Warning Sounds(??) will also be useful as a form of modeless feedback. If the feedback must change the user's thread of interaction, then use a Dialog Box(??)...

9 Visual Symbols

...seventy percent of a human's sensory input is based on vision and WIMP INTERFACE(2) interfaces are highly dependent on this visual component.

The visual display of data is the most efficient way to communicate information. However, you must not overwhelm the user with useless data or misleading metaphors.

The subtlety here is the ability to communicate. From the beginnings of human culture, visual awareness has been a key element to communication. From cave drawings to the printed word, the visual record has outlasted all other forms of communication.

Misleading or vague metaphors can cause the user to make critical mistakes, or cause the user to *second guess* what you had intended.

People are easily overwhelmed by visual stimuli. Overly complex symbols (and phrases) require much more time to recognize and understand.

Therefore:

Design the interface based on visual cues that are recognizable and understandable by the target audience. Determine if standard symbols (e.g. ISO or IEEE symbols) exist before creating your own. Resist being overly clever and keep your metaphors simple.

The tricky part is *thinking visually.* Paul Heckel [Heckel 91] states that

> Our instincts and training as engineers encourage us to think logically instead of visually, and this is counterproductive to friendly design.

An interface with a series of pushbuttons, which are labeled F1 through F8, is an example of a very literal translation of a hardware solution, but is not as helpful or friendly as it could be.

At the other end of the spectrum are overly clever icons: a piggy bank with a coin hovering above it representing a *save* function, or a badly cracked piggy bank spilling coins representing an *open* command. Here the programmer was thinking visually, but the the metaphor was too obscure.

Help the user associate symbols with the function through the use of CLICKABLE SYMBOLS(10). Providing SYMBOL EXPLANATIONS(11) will help the user understand the meaning of the symbol...

10 Clickable Symbols

...symbols should appear in your MENUBAR(??), LAUNCHPAD(??) or PALETTE(??), among other places. They are the cornerstone of graphical user interfaces.

Words must be read and converted to mental images to be understood. This can take time. Words can also take up quite a bit of screen real estate.

The purpose of a button can be portrayed clearly and precisely when labeled with text. However, to make the meaning clear and precise you may have to use phrases (which can clutter the screen).

The user must take time to focus and read text. When a drop down menu is selected, you end up reading at least half of the options before finding the one you were looking for.

Pictures are more engaging, and pleasing to the eye than words.

Therefore:

Use recognizable symbols instead of words. The user will learn to associate the symbol with a function. Pictures can save screen space and can be recognized much quicker than words.

People have a much easier time recognizing pictures. Once you have learned that the ABC with a check mark picture means run the spell checker on the current document, you no longer need to spend time reading and searching for that function.

You as a designer do not need to restrict yourself to the button metaphor either. The graphical button is an artifact carried over from the hardware world. Web-based applications allow the user to click on different areas of a picture to activate a function.

In addtion to this pattern you should also provide textual SYMBOL EXPLANATIONS(11) to aid the user learn the meaning behind the picture when it is not clear...

11 Symbol Explanations

...this pattern is often used in conjunction with VISUAL SYMBOLS(9) to make up a MENUBAR(??), LAUNCHPAD(??), PALETTE(??) or button more effective.

Symbols can be confusing, so a mechanism is needed to help the user associate a symbol with its purpose.

There are times when the picture alone is not enough to convey its purpose.

People are different and will not have the same pictorial dictionary that you do as a designer. So you need to teach them your picture language as a side effect of using the application, that is, as incidental learning.

Therefore:

Design simple text names or descriptions which can be displayed in conjunction with symbols. If these descriptions begin to take up too much screen real-estate, use popup text labels that appear when your cursor is over a symbol and disappear when you move the cursor away. Also make the display of descriptions optional, by user's choice.

The Netscape Navigator achieves this by allowing the user to display both text and icons, or icons alone. Once a user is comfortable with the meanings of the icons, they may want to reclaim screen real-estate by removing the text descriptions.

The Macintosh provides *help balloons* that give short help messages when the cursor is dragged over any item on the desktop.

When you don't have a lot of screen space to waste, use CONTEXT SENSITIVE HELP(??) to explain items in greater detail..

12 References

[Alex77] Alexander, Christopher. A Pattern Language. New York:Oxford University Press, 1977

[Heckel91] Heckel, Paul. The Elements of Friendly Software Design. San Francisco, CA: Sybex, 1991

[Laurel92] Laurel, Brenda. Computers as Theatre. Reading, MA: Addison Wesley, 1990

[Liddle96] Liddle, David. Bringing Design to Software, edited by Terry Winograd. New York, NY: ACM Press, 1996

[Mobley92] Mobley, Jonnie Patricia. NTC's Dictionary of Theatre and Drama Terms. Lincolnwood, Illinois: NTC Publishing Group, 1992.

[Norman86] Norman, Donald A., and Stephen W. Draper, eds. User Centered Design: New Perspectives on Human-Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum Associates, 1986

[Shneiderman92] Shneiderman, Ben. Designing the User Interface: Strategies for Effective Human-Computer Interaction, Reading, MA: Addison-Wesley, 1992 (2nd Ed.)