away each layer of skin reveals a layer beneath. There are various levels of "hiding": Here's a partial list of them in order from most exposed to least exposed:

- Toolbar (completely exposed)
- Menu item (exposed by trivial user gesture)
- Submenu item (exposed by somewhat more involved user gesture)
- Dialog box (exposed by explicit user command)
- Secondary dialog box (invoked by button in first dialog box)
- "Advanced user mode" controls -- exposed when user selects "advanced" option
- Scripted functions

The above notwithstanding, in no case should the primary interface of the application be a reflection of the *true complexity* of the underlying implementation. Instead, both the interface and the implementation should strive to match a simplified conceptual model (in other words, the *design*) of what the application does. For example, when an error occurs, the explanation of the error should be phrased in a way that relates to the current user-centered activity, and not in terms of the low-level fault that caused there error.

4. The principle of coherence

— The behavior of the program should be internally and externally consistent

There's been some argument over whether interfaces should strive to be "intuitive", or whether an intuitive interface is even possible. However, it is certainly arguable that an interface should be *coherent* — in other words logical, consistent, and easily followed. ("Coherent" literally means "stick together", and that's exactly what the parts of an interface design should do.)

Internal consistency means that the program's behaviors make "sense" with respect to other parts of the program. For example, if one attribute of an object (e.g. color) is modifiable using a pop-up menu, then it is to be expected that other attributes of the object would also be editable in a similar fashion. One should strive towards the principle of "least surprise".

External consistency means that the program is consistent with the environment in which it runs. This includes consistency with both the operating system and the typical suite of applications that run within that operating system. One of the most

widely recognized forms of external coherence is compliance with *user-interface standards*. There are many others, however, such as the use of standardized scripting languages, plug-in architectures or configuration methods.

5. The principle of state visualization

— Changes in behavior should be reflected in the appearance of the program

Each change in the behavior of the program should be accompanied by a corresponding change in the appearance of the interface. One of the big criticisms of "modes" in interfaces is that many of the classic "bad example" programs have modes that are visually indistinguishable from one another.

Similarly, when a program changes its appearance, it should be in response to a behavior change; A program that changes its appearance for no apparent reason will quickly teach the user not to depend on appearances for clues as to the program's state.

One of the most important kinds of state is the *current selection*, in other words the object or set of objects that will be affected by the next command. It is important that this internal state be visualized in a way that is consistent, clear, and unambiguous. For example, one common mistake seen in a number of multi-document applications is to forget to "dim" the selection when the window goes out of focus. The result of this is that a user, looking at several windows at once, each with a similar-looking selection, may be confused as to exactly which selection will be affected when they hit the "delete" key. This is especially true if the user has been focusing on the selection highlight, and not on the window frame, and consequently has failed to notice which window is the active one. (Selection rules are one of those areas that are covered poorly by most UI style guidelines, which tend to concentrate on "widgets", although the Mac and Amiga guidelines each have a chapter on this topic.)

6. The principle of shortcuts

— Provide both concrete and abstract ways of getting a task done

Once a user has become experienced with an application, she will start to build a mental model of that application. She will be able to predict with high accuracy what the results of any particular user gesture will be in any given context. At this point, the

program's attempts to make things "easy" by breaking up complex actions into simple steps may seem cumbersome. Additionally, as this mental model grows, there will be less and less need to look at the "in your face" exposure of the application's feature set. Instead, pre-memorized "shortcuts" should be available to allow rapid access to more powerful functions.

There are various levels of shortcuts, each one more abstract than its predecessor. For example, in the emacs editor commands can be invoked directly by name, by menu bar, by a modified keystroke combination, or by a single keystroke. Each of these is more "accelerated" than its predecessor.

There can also be alternate methods of invoking commands that are designed to increase power rather than to accelerate speed. A "recordable macro" facility is one of these, as is a regular-expression search and replace. The important thing about these more powerful (and more abstract) methods is that they should not be the most exposed methods of accomplishing the task. This is why emacs has the non-regexp version of search assigned to the easy-to-remember "C-s" key.

7. The principle of focus

— Some aspects of the UI attract attention more than others do

The human eye is a highly non-linear device. For example, it possesses edge-detection hardware, which is why we see Mach bands whenever two closely matched areas of color come into contact. It also has motion-detection hardware. As a consequence, our eyes are drawn to animated areas of the display more readily than static areas. Changes to these areas will be noticed readily.

The mouse cursor is probably the most intensely observed object on the screen — it's not only a moving object, but mouse users quickly acquire the habit of tracking it with their eyes in order to navigate. This is why global state changes are often signaled by changes to the appearance of the cursor, such as the well-known "hourglass cursor". It's nearly impossible to miss.

The text cursor is another example of a highly eye-attractive object. Changing its appearance can signal a number of different and useful state changes.

8. The principle of grammar

— A user interface is a kind of language — know what the rules are

Many of the operations within a user interface require both a *subject* (an object to be operated upon), and a *verb* (an operation to perform on the object). This naturally suggests that actions in the user interface form a kind of grammar. The grammatical metaphor can be extended quite a bit, and there are elements of some programs that can be clearly identified as adverbs, adjectives and such.

The two most common grammars are known as "Action->Object" and "Object->Action". In Action->Object, the operation (or tool) is selected first. When a subsequent object is chosen, the tool immediately operates upon the object. The selection of the tool persists from one operation to the next, so that many objects can be operated on one by one without having to re-select the tool. Action->Object is also known as "modality", because the tool selection is a "mode" which changes the operation of the program. An example of this style is a paint program — a tool such as a paintbrush or eraser is selected, which can then make many brush strokes before a new tool is selected.

In the Object->Action case, the object is selected first and persists from one operation to the next. Individual actions are then chosen which operate on the currently selected object or objects. This is the method seen in most word processors — first a range of text is selected, and then a text style such as bold, italic, or a font change can be selected. Object->Action has been called "non-modal" because all behaviors that can be applied to the object are always available. One powerful type of Object->Action is called "direct manipulation", where the object itself is a kind of tool — an example is dragging the object to a new position or resizing it.

Modality has been much criticized in user-interface literature because early programs were highly modal and had hideous interfaces. However, while non-modality is the clear winner in many situations, there are a large number of situations in life that are clearly modal. For example, in carpentry, its generally more efficient to hammer in a whole bunch of nails at once than to hammer in one nail, put down the hammer, pick up the measuring tape, mark the position of the next nail, pick up the drill, etc.

9. The principle of help

— Understand the different kinds of help a user needs

In an essay in [LAUR91] it states that are five basic types of help, corresponding to the five basic questions that users ask:

- 1. Goal-oriented: "What kinds of things can I do with this program?"
- 2. Descriptive: "What is this? What does this do?"
- 3. Procedural: "How do I do this?"
- 4. Interpretive: "Why did this happen?"
- 5. Navigational: "Where am I?"

The essay goes on to describe in detail the different strategies for answering these questions, and shows how each of these questions requires a different sort of help interface in order for the user to be able to adequately phrase the question to the application.

For example, "about boxes" are one way of addressing the needs of question of type 1. Questions of type 2 can be answered with a standard "help browser", "tool tips" or other kinds of context-sensitive help. A help browser can also be useful in responding to questions of the third type, but these can sometimes be more efficiently addressed using "cue cards", interactive "guides", or "wizards" which guide the user through the process step-by-step. The fourth type has not been well addressed in current applications, although well-written error messages can help. The fifth type can be answered by proper overall interface design, or by creating an application "roadmap". None of the solutions listed in this paragraph are final or ideal; they are simply the ones in common use by many applications today.

10. The principle of safety

— Let the user develop confidence by providing a safety net

Ted Nelson once said "Using DOS is like juggling with straight razors. Using a Mac is like shaving with a bowling pin."

Each human mind has an "envelope of risk", that is to say a minimum and maximum range of risk-levels which they find comfortable. A person who finds herself in a situation that is too risky for her comfort will generally take steps to reduce that risk. Conversely, when a person's life becomes too safe — in other words, when the risk level drops below the *minimum* threshold of the risk envelope — she will often engage in actions that *increase* their level of risk.

This comfort envelope varies for different people and in different situations. In the case of computer interfaces, a level of risk that is comfortable for a novice user might make a "power-user" feel uncomfortably swaddled in safety.

It's important for new users that they feel safe. They don't trust themselves or their skills to do the right thing. Many novice users think poorly not only of their technical skills, but of their intellectual capabilities in general (witness the popularity of the "...for Dummies" series of tutorial books.) In many cases these fears are groundless, but they need to be addressed. Novice users need to be assured that they will be protected from their own lack of skill. A program with no safety net will make this type of user feel uncomfortable or frustrated to the point that they may cease using the program. The "Are you sure?" dialog box and multi-level undo features are vital for this type of user.

At the same time, an expert user must be able to use the program as a *virtuoso*. She must not be hampered by guard rails or helmet laws. However, expert users are also smart enough to turn off the safety checks — if the application allows it. This is why "safety level" is one of the more important application configuration options.

Finally, it should be noted that many things in life are not meant to be easy. Physical exercise is one — "no pain, no gain". A concert performance in Carnegie Hall, a marathon, or the Guinness World Record would be far less impressive if anybody could do it. This is especially pertinent in the design of computer game interfaces, which operate under somewhat different principles than those listed here (although many of the principles in fact do apply).

11. The principle of context

— Limit user activity to one well-defined context unless there's a good reason not to

Each user action takes place within a given context — the current document, the current selection, the current dialog box. A set of operations that is valid in one context may not be valid in another. Even within a single document, there may be multiple levels — for example, in a structured drawing application, selecting a text object (which can be moved or resized) is generally considered a different state from selecting an individual character within that text object.

It's usually a good idea to avoid mixing these levels. For example, imagine an application that allows users to select a range of text characters within a document, and also allows them to select one or more whole documents (the latter being a distinct concept from selecting all of the characters in a document). In such a case, it's probably best if the program disallows selecting both characters and documents in the same selection. One unobtrusive way to do this is to "dim" the selection that is not applicable in the current context. In the example above, if the user had a range of text selected, and then selected a document, the range of selected characters could become dim, indicating that the selection was not currently pertinent. The exact solution chosen will of course depend on the nature of the application and the relationship between the contexts.

Another thing to keep in mind is the relationship between contexts. For example, it is often the case that the user is working in a particular task-space, when suddenly a dialog box will pop up asking the user for confirmation of an action. This sudden shift of context may leave the user wondering how the new context relates to the old. This confusion is exacerbated by the terseness of writing style that is common amongst application writers. Rather than the "Are you sure?" confirmation mentioned earlier, something like "There are two documents unsaved. Do you want to quit anyway?" would help to keep the user anchored in their current context.

12. The principle of aesthetics

— Create a program of beauty

It's not necessary that each program be a visual work of art. But it's important that it not be ugly. There are a number of simple principles of graphical design that can easily be learned, the most basic of which was coined by artist and science fiction writer William Rotsler: "Never do anything that looks to someone else like a mistake." The specific example Rotsler used was a painting of a Conan-esque barbarian warrior swinging a mighty broadsword. In this picture, the tip of the broadsword was just off the edge of the picture. "What that looks like", said Rotsler, "is a picture that's been badly cropped. They should have had the tip of the sword either clearly within the frame or clearly out of it."

An interface example can be seen in the placement of buttons — imagine five buttons, each with five different labels that are almost the same size. Because the buttons are

packed using an automated-layout algorithm, each button is almost but not exactly the same size. As a result, though the author has placed much care into his layout, it looks carelessly done. A solution would be to have the packing algorithm know that buttons that are almost the same size look better if they are exactly the same size — in other words, to encode some of the rules of graphical design into the layout algorithm. Similar arguments hold for manual widget layout.

Another area of aesthetics to consider is the temporal dimension. Users don't like using programs that feel sluggish or slow. There are many tricks that can be used to make a slow program "feel" snappy, such as the use of off-screen bitmaps for rendering, which can then be blitted forward in a single operation. (A pet peeve of this particular author is buttons that *flicker* when the button is being activated or the window is being resized. Multiply redundant refreshing of buttons when changing state is one common cause of this.)

13. The principle of user testing

— Recruit help in spotting the inevitable defects in your design

In many cases a good software designer can spot fundamental defects in a user interface. However, there are many kinds of defects which are not so easy to spot, and in fact an experienced software designer is often less capable of spotting them than the average person. In other cases, a bug can only be detected while watching someone else use the program.

User-interface testing, that is, the testing of user-interfaces using actual end-users, has been shown to be an extraordinarily effective technique for discovering design defects. However, there are specific techniques that can be used to maximize the effectiveness of end-user testing. These are outlined in both [TOG91] and [LAUR91] and can be summarized in the following steps:

- Set up the observation. Design realistic tasks for the users, and then recruit end-users that have the same experience level as users of your product (Avoid recruiting users who are familiar with your product however).
- Describe to the user the purpose of the observation. Let them know that you're testing the product, not them, and that they can quit at any time. Make sure that they understand if anything bad happens, it's not their fault, and that it's helping you to find problems.

- Talk about and demonstrate the equipment in the room.
- Explain how to "think aloud". Ask them to verbalize what they are thinking about as they use the product, and let them know you'll remind them to do so if they forget.
- Explain that you will not provide help.
- Describe the tasks and introduce the product.
- Ask if there are any questions before you start; then begin the observation.
- Conclude the observation. Tell them what you found out and answer any of their questions.
- Use the results.

User testing can occur at any time during the project, however, it's often more efficient to build a mock-up or prototype of the application and test that before building the real program. It's much easier to deal with a design defect before it's implemented than after. Tognazzini suggests that you need no more than three people per design iteration — any more than that and you are just confirming problems already found.

14. The principle of humility

— Listen to what ordinary people have to say

Some of the most valuable insights can be gained by simply watching other people attempt to use your program. Others can come from listening to their opinions about the product. Of course, you don't have to do exactly everything they say. It's important to realize that each of you, user and developer, has only part of the picture. The ideal is to take a lot of user opinions, plus your insights as a developer and reduce them into an elegant and seamless whole — a design which, though it may not satisfy everyone, will satisfy the greatest needs of the greatest number of people.

One must be true to one's vision. A product built *entirely* from customer feedback is doomed to mediocrity, because what users want most are the features that they cannot anticipate.

But a single designer's intuition about what is good and bad in an application is insufficient. Program creators are a small, and not terribly representative, subset of the general computing population.

Some things designers should keep in mind about their users:

- Most people have a biased idea as to the what the "average" person is like. This is because most of our interpersonal relationships are in some way self-selected. It's a rare person whose daily life brings them into contact with other people from a full range of personality types and backgrounds. As a result, we tend to think that others think "mostly like we do." Designers are no exception.
- Most people have some sort of core competancy, and can be expected to perform well within that domain.
- The skill of using a computer (also known as "computer literacy") is actually much harder than it appears.
- The lack of "computer literacy" is not an indication of a lack of basic intelligence. While native intelligence does contribute to one's ability to use a computer effectively, there are other factors which seem to be just as significant, such as a love of exploring complex systems, and an attitude of playful experimentation. Much of the fluency with computer interfaces derives from play -- and those who have dedicated themselves to "serious" tasks such as running a business, curing disease, or helping victims of tragedy may lack the time or patience to be able to devote effort to it.
- A high proportion of programmers are introverts, compared to the general population. This doesn't mean that they don't like people, but rather that there are specific social situations that make them uncomfortable. Many of them lack social skills, and retreat into the world of logic and programming as an escape; As a result, they are not experienced people-watchers.

The best way to avoid misconceptions about users is to spend some time with them, especially while they are actually using a computer. Do this long enough, and eventually you will get a "feel" for how the average non-technical person thinks. This will increase your ability to spot defects, although it will never make it 100%, and will never be a substitute for user-testing.

Bibliography

[TOG91] *Tog On Interface*, Bruce Tognazzini, Addison-Wesley, 1991, ISBN 0-201-60842-1

[LAUR91] *The Art of Human Computer Interface Design*, Brenda Laurel, Addison-Wesley, 1991, ISBN 0-201-51797-3

The Psychology of Everyday Things, Don Norman, Harper-Collins 1988, ISBN 0-465-06709-3

The Macintosh Human Interface Guidelines, Apple Computer Staff, Addison-Wesley 1993, ISBN 0-201-62216-5

The Amiga User Interface Style Guide, Commodore-Amiga, Addison-Wesley 1991, ISBN 0-201-57757-7

参考ウェブサイドリスト

[1] http://www.medis.or.jp/4_hyojyun/medis-master/index.html
[2] http://www.voelter.de/data/pub/ArchitecturePatterns.pdf
[3] http://www.nedo.go.jp/denshi/roadmap/2007/saku_usabiliry.pdf

別添５

研究成果の刊行に関する一覧表

書籍

著者氏名	論文タイトル名	書籍全体の編集者名	書　籍　名	出版社名	出版地	出版年	ページ
山野辺裕二 相良眞一 中里崇志	ベッドサイド端末による注射実施システム―国立成育医療センター病院の場合	松村泰志	注射に関するシステムの現状と課題	篠原出版新社	東京	2009	15

雑誌

発表者氏名	論文タイトル名	発表誌名	巻号	ページ	出版年
山野辺裕二 本多正幸 相澤志優 加藤五十六	電子カルテシステムの基礎的GUIガイドライン	医療情報学	28(Suppl.)	1135	2008
加藤五十六 寺下貴美 櫻井恒太郎	日本の病院情報システムでの診療科を色で表現するインターフェースの状況（アンケート調査）	医療情報学	27(Suppl.)	1167	2007