Chapter 6
Designing User Interfaces in Squeak

As mentioned in the last chapter, there are basically two challenges to builder interfaces for users:

- How do you create user interface software that you can maintain well, that is, it is easy to change pieces without impacting everything?
- How do you create user interfaces that people can actually use?

The last chapter addressed the first point. This chapter addresses the second. This chapter is *not* a replacement for a human-computer interface design class. User interface design is a challenging and complex task, perhaps even more an art than a science. The goal of this chapter is to provide some insights into process and issues. You now know how to build user interfaces. You should give some thought how to do it well, that is, how to avoid annoying your users. The most important point of this chapter is the heading of the very next section.

1 Know Thy Users for They Are Not You

The most important thing to learn about user interface design is that the user for whom you are designing is almost always *not* you. The real users may not even be *like* you. This means that, whenever there is a question about what to put in the user interface or what the users want, *you* are not the best authority on the subject. In fact, you may be the *worst.*

Most users are not computer programmers. They don't know anything about how computers work—nor do they want to. You, on the other hand, know a lot about how computers work and you know how to program them. Your expectations and desires are very different than most users.

On the other hand, users other than you know lots of things that you don't. Users know their jobs and the information needed to know their jobs. Let's take a concrete example. You wouldn't presume to know anything about how to do open heart surgery, and you wouldn't expect a medical doctor to know anything about object-oriented design. So, when a question arises on how best to organize or search for drugs in a prescription database to be used by medical doctors, who's better to answer the question: You who know databases, or the doctor's who know their jobs? Obviously, it's the doctor's.

At conferences on user interface design, you can find people wandering around with badges saying "Know thy users for they are not you." This is one of the most important maxims of user interface designers, and is probably the hardest lesson to learn in the field. You, as a prospective interface designer, know computers, and you've used lots of interfaces. When a question arises about how to do something, you feel

like you have lots of experience and knowledge to draw from. And you do—but it's probably wrong for the *users*. Your users are not you.

1.1 How do you decide between user interface alternatives?

The question of who the user is and what the user may want arises when there is more than one way to do something. When you have alternatives to choose between, the deciding factor is what's best for the users. 90% of the time, the users can tell you exactly what is best for them. Most users are adults who know their jobs and know what they want.

There are some times, however, when the user does not know what she wants. Maybe it's because the user has never used a computer tool for the given task. Maybe because the task, on the computer, will be completely new. In that case, you *will* have to make the choice yourself—but you do it from understanding of the users' *tasks*. What is it that users will have to do?

Again, the user is not you. You cannot know what the users' tasks are until you talk to users, watch them, and get to know them. Your perception of the task can be *very* different than what the users' perception may be.

Consider two interfaces for a clock. The first is the basic ClockMorph built into Squeak (Figure 1). The second is the Clock user interface that we built in the previous chapter (Figure 2. Which one is better for the user? Which one is better for the user's tasks with a clock? We'll argue before the end of the chapter that Figure 2 is very possibly the worst possible user interface for a clock.

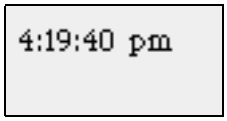

Figure 1: ClockMorph from Squeak

Figure 2: Clock User Interface from Previous Chapter

2 Understanding the User

The first question that you should always ask about a user interface that you are being asked to design is "Who is my user?" Who is it that will be using this tool?

There are lots of ways of modeling a user, including several formal methods of what to ask the user about. The key issue is to match the users' skills (and later, the task) to the user interface that you are going to design. You want to figure out what the users can and cannot do, then make sure that your interface allows the user to perform the task that he wants to perform, within the user's skill set.

Users' skills are pretty hard to pin down, and sometimes you have to ask questions that may not make much sense at first glance. You need to ask users about what they know how to do as it pertains to your program. Let's consider some sample questions.

How old is the typical user? A graduate student at Georgia Tech once developed a program to help first graders learn mathematics where the very first screen was full of text instructions—which first graders were incapable of reading. There's a reason why video games in arcades go into "demo mode" when not being used. Video games don't come with instruction manuals, and not all video game enthusiasts know how to read. A "demo mode" gets around this problem.

On the opposite end of this question are the senior citizens who may no longer be able to read small fonts nor may be able to manipulate a mouse to click on a small target. As people get older, vision acuity often fades, and hand-eye coordination may become more difficult. There are

technologies where this can come into play, such as the small fonts on ATM's or the fuzzy fonts on television-based Web browsers. If your user population is very broad (e.g., a kiosk in the mall that anyone should be able to walk up to and use), you must consider the limitations of each end of your user age range.

What do your users know about? There are some terrific programs for teaching physics which provide the user with all kinds of interesting worlds to explore, for example, worlds where Hooke's Law is invalid, or when a different gravitational constant applies. Unfortunately, newcomers to physics classes don't know Hooke's Law nor what a gravitational constant is. For these students, the wonderful program is useless because they can't figure out what to do with it.

Terminology on menus, buttons, and help screens are only one place where the users' past knowledge plays an important role. There are other programs where knowledge of a *process* is critical. For example, there are programs where data must be prepared in some way before it can be processed. If you don't know what the order of operations is, you cannot use the program.

A classic example along these lines is a spreadsheet. When you open a spreadsheet application, what should you do? You have a blank sheet of cells sitting in front of you. If the user doesn't have prior knowledge of spreadsheets, how could an interface help the user figure out what to do first?

What does the user want to do? Perhaps this is the most important question to ask a user, not just for the answer, but in the way that the answer is phrased. If a medical doctor says that she wants to "Find all the generics for suphedrine," that tells you that she wants to "find drugs," but also, that she's looking in terms of a specific medication that she already knows. If you had created the user interface in terms of "Medications for runny noses," your interface would not only get in the way of the doctor who *already knows* what medication she wants, but it would also be an insult to the doctor who has already matched symptom to drug.

2.1 Understanding the Task

Asking what the user wants to do is one part of understanding the users' task. The challenge to understanding the user's task is that not all of it may be explicit in what users' say. As people become expert at their tasks, their knowledge of the tasks becomes implicit. People say, "I don't know how I do it—my fingers seem to know." Some of the users' task you may have to discover from watching them, or even just looking around where they perform their task. Context of the task can tell you a lot.

Questions that the context might be able to answer include "*When* do you perform your task? *What* do you need to do your job?" Some studies

of jobs like airport tower controllers have found that users use surprising sources of information. In one study, airport tower controllers explained their tasks in terms of computer monitors and paper forms, but when observed, interface designers found that the tower controllers kept *looking out the window!* A quick glance out the window told them more about who was on the ground, how busy the terminals were, and where the most open terminals were—far more quickly than hunting through the forms and the monitors. This kind of observation pays dividends when it comes to creating a new interface that actually *does help* the users with their jobs, by providing them the knowledge that they really need.

Another important attribute of the task that the user may not be able to tell you explicitly is how often they try to achieve various goals. For example, users will often emphasize the time-critical and urgent aspects of their jobs. "When X happens, I have to do Y and Z, immediately!" But they may not tell you that X happens once a month. It *is* important to support the users in their time-critical and urgent tasks, but it's also important that the urgent tasks don't make the everyday, mundane tasks easier.

Imagine a fire alarm that was so sensitive that just running by it would fire off the alarm. That would certainly make calling the fire trucks as easy as possible. However, just walking down that hallway would become tricky, and you'd have to be careful not to go too fast, or to hold down the "not a real alarm" button as you walked by. This would be an awful interface since it would sacrifice the ease of everyday tasks for the ease of unusual but urgent tasks.

3 Matching Users to Interface: Avoiding User Error

There are *lots* of ways constructing an interface. Various interaction styles and widgets were introduced in the last chapter. A brief list includes:

- Buttons for clicking on, radio and checkbox buttons for selecting.
- Text areas
- Various kinds of direct manipulation, from drag-and-drop to resizing windows
- Menu selections
- Dialog boxes with buttons, text areas, and such.
- Natural language
- Command languages like UNIX shell

How do you pick between these? Obviously, you use your knowledge of the users and their tasks, but the match between users and interaction styles may not be obvious. It is definitely true that not all matches make sense.

For example, consider command languages as an interaction style. For expert users, command languages are great. Users of Microsoft Windows and Apple MacOS can't hope to do as much with as few keystrokes as a UNIX shell expert. The UNIX shell is just amazing for providing a succinct and programmable interface. But UNIX shell for new or casual (infrequent, logging on once every few days) users is a terrible idea. ("What did *rm* do again?")

On the other hand, expert users get frustrated when forced to use just an iconic and menu-driven interface. They want the speed and flexibility of shortcuts like command languages. Expert users use the system often enough that they won't forget obscure commands. Novice or casual users, on the other hand, need to see things rather than have to remember obscure details.

SideNote: Shortcut keys in laboratory tests, believe it or not, are *always* slower than mouse-driven menu actions! Bruce Tognazzini in his book *Tog on Interface* writes (p. 26): "We discovered, among other things, two pertinent facts: Test subjects consistently report that keyboarding is faster than mousing. The stopwatch consistently proves mousing is faster than keyboarding." While taking your hands off the keyboard does slow you down, using the mouse is cognitively easier than remembering the right shortcut key. There is a real time loss spent thinking up the shortcut key, but people are completely unaware of the time loss—they literally have a kind of "amnesia," Toganazzini claims. Note, however, that these tests are being conducted with users in novel applications. After years of use, some keyboard shortcuts may become automatic and not requiring remembering.

In general, computers are good at remembering things, but people are not. Command languages are great if users are with them often enough to remember the commands. For everyone else, provide icons, visible menus, and dialogs that make it clear what's to be done and when. User interface designers talk about "making knowledge visible." Make the state of the program, the options for what to do next, and how to go about those options visible in the interface.

You should make decisions about interaction mechanisms based on what people expect. For example, if you're building an on-line form that takes the place of an existing paper-based form that people know and have used for years, make the on-line form look like the paper-based form! That way, people will know what's expected and how to use it.

Where there are applicable guidelines for user interfaces on your platform, you should follow them. IBM, Apple, and others have developed notable guidelines for creating user interfaces. The reason to follow the guidelines is not to create a standard, corporate look to the interfaces, but to give people what they expect. If people expect dialog boxes to have the

OK and Cancel buttons in certain places, they will be confused if you decide to put them elsewhere. Do what people expect.

In general, your choice of interaction mechanisms should be made to *avoid user error*. You can give command languages to novices, but you can also expect to have lots of user error and frustration. However, if you map that command language to menus and dialog boxes, you'll have less error, but it may still be frustrating for the user if you basically provide all the same functionality but in the menu bar. If you figure out the users' tasks, then provide menus that correspond to users' operations in those tasks, and you bring up dialog boxes when necessary with options that relate to the task, then you will probably have even less error and frustration. Design your interface to reduce users' errors.

One way to reduce users' errors is to *avoid modes*. If you've ever used the UNIX *vi* editor, you experience modes all the time. In *vi*, you are either in "insert mode" where typing enters new characters into the file, or you are in "command mode" where typing controls the cursor, deletion, changes, and insertions. (There are actually some additional submodes that we'll skip.) For example, a "k" in command mode moves the cursor up a line, a "d" deletes a character, and a "w" writes the file. Serious *vi* users can tell you what damage typing their name in command mode will do. A mode means that users have to figure out what's valid when, and recall which mode they're currently in. Modes can lead to errors.

There's a general rule for interface designers to *put the knowledge in the world*. If there's something that a user needs to know, make it visible somewhere on the screen. If you *have* to have a mode, put a clearly visible indicator that tells the user the current mode. When there are choices for the user in the interface, use lists or menus to convey the choices, rather than require the user to invent them. Making things visible also invites exploration, since all the possibilities are available.

Finally, design *expecting* user errors. That's why *undo* is such an important user interface advancement. Everyone makes errors. Making recovery from errors a graceful and omnipresent option is an important goal for a good user interface.

4 A User Interface Design Process

For object-oriented design, we identified a process that made it more likely that we would produce a good (reusable and maintainable) product. For user interface design, there are again several kinds of process that make the claim that following the process will lead to a better design. In this section, we present two of these.

The first user interface design process is the *waterfall method*. The waterfall method sets up a series of steps that, if executed properly, lead to

a good design with a single pass through the process. The waterfall method typically has stages that look like this:

- **Requirements specification:** Elicit the user's needs, analyze the task, and define what the user interface must do.
- **Architectural design:** Figure out *how* the user interface provides the necessary functions.
- **Detailed design:** Refine the overall architecture into detailed descriptions that a programmer can code.
- **Coding and unit testing:** Build the user interface and test the low-level components as they are developed.
- **Integration and testing:** Integrate the low-level components and test them.
- **Operation and maintenance:** Actually use the system, and maintain it over time.

The problem with the waterfall method is that it assumes an accurate requirements specification. That may not be possible in all cases, especially for novel technology. User interface researcher John Carroll of Virginia Tech has pointed out how interfaces and systems impact users' activities and goals. Early requirements specification based on users' original goals may not be correct any more when they actually start using the system. For many interfaces, this may not be a problem, such as when replacing an existing system with a new one. But when technology is new, Carroll's point may be critical.

The second user interface design process that we'll discuss is *iterative design and prototyping*. The idea in this model is to *plan* on repeating the process until a usability goal is reached. There are lots of variations on this approach, some of which take the entire waterfall method as a subset of the process. The general structure can be understood as this:

- **Requirements gathering:** Do an analysis of the users and their tasks, similar to the first step above. One additional goal is deciding just how usable the system needs to be. Can users perform certain tasks within a certain amount of time? That's a measurable usability goal.
- **Build a prototype:** The process of building a prototype could be all the rest of the steps above.
- **Evaluate the prototype:** Trial the prototype. Actually test it with users and see if the usability goal is met. If you meet the goal, you're done.
- **Iterate:** If the prototype doesn't meet the goal, iterate on it. Maybe you have to go back and fix the goal and requirements. Maybe you have to rebuild the prototype from scratch. Maybe you only have to

tweak the prototype. Whatever level you decide to return to, you nave to evaluate the candidate interface before calling it done.

5 Critiquing Our Clock Interface

Given all of the above, let's consider Figure 2 and decide the quality of the interface that we invented in the last chapter for the clock. We are all users of clocks, so it's fair to use ourselves as the users for this interface. What are the tasks for which we use a clock? Your list will probably look something like this, from the most common to the least common:

- Look at the time.
- Perhaps look at the date.
- For an alarm clock, set the alarm time.
- Set the time (after a power outage or when Daylight Savings starts or stops).

Let's evaluate the clock interface in terms of this list. Can we look at the time? Yes, it's fairly big right on top. But the busiest part of the interface, the part that attracts our eye is the bottom. Those four buttons, the *most* visible part of the clock with all their margins and labels, are designed to enable us to *set* the time. That is, our interface draws attention to the least common activity for the clock.

If you still have the clock code available, try running the interface again. Try clicking on the buttons. Note that it's difficult to tell when you have clicked on the buttons: They don't highlight, and there isn't any other audible or visible sign that the button has been clicked. This makes it very easy to accidentally click the hour or minute change buttons and simply not notice.

These observations suggest that the clock interface in Figure 2 is *optimized* for *user error!* The interface draws the user's attention toward the task that is least common, and any use of the interface makes invisible that the least common (and least often desired) activity—changing the time— has even occurred. We could hardly have designed a worse interface for actual use if we tried.

How could we have had a better interface for setting the time, so that it needn't have been so obvious and dangerous? You have probably seen variations of interfaces like these suggestions.

- Maybe there's a small button next to the clock with the label "Set". Clicking this button might bring up a dialog box for setting the time. This can work for the computer clock, but if our model were used in a wristwatch, no dialog box is possible.
- Maybe there are up and down buttons for advancing or retreating the time. Holding them longer than a quick depress might advance or

retreat the time more quickly. This works well, and is used in many alarm clocks, but it can take a while to set the time to a particular time (after, say, a power outage, or when setting the time for recording the VCR).

- Maybe there's no way to set the time—the clock automatically sets the time to some externally accessed source. But then there's the problem of getting the right external source, moving the clock, and fixing it when you're pointing at the wrong external source.

Exercises: Consider Your Interfaces

1. Look around at the clocks that you use in your life. Which ones have interfaces that you can use easily? Why do you like them?

2. Consider a user interface that you use frequently, like your email program or your Web browser. Write down a list of your most common tasks. Now look at the user interface. Is it obvious (visible) how to perform your tasks? What tasks does your interface seem to be optimized for?

3. As a test of our Clock model, design an implementation of two of the interfaces from the above list (or some other interfaces that you invent). Do we have to change the Clock, or can we use the same model with multiple interfaces?

6 Evaluation of User Interfaces

Given a set of users needs, you are probably a creative person who can come up with a list of possible interfaces to meet these needs. Coming up with such a list is a good idea. Expert designs actively consider many alternatives in making design decisions. But how do you decide what to implement? Or if you can implement several of them easily, how do you decide which one to actually use in the final design?

Evaluating a user interface can be done before coding or at least before involving the users with some methods, or after implementation with other methods. Evaluating a user interface means trying to measure or get some general sense of the interface's usability by a user. The goals of an evaluation may differ dramatically between different studies. Some times you just want to know if a design *worked*. Other times, you may be trying to gather concrete evidence that one approach

6.1 Evaluation Before User Involvement

While it sounds strange, there's a lot that can be learned about an interface design even before coding it, simply by considering a careful analysis. A *heuristic evaluation* or *guidelines review* is about carefully analyzing an interface in terms of a standard set of questions or issues.

Evaluators review a user interface in terms of a set of standard heuristics (like "Is knowledge visible?") or standard user interface guidelines (e.g., "Is the Cancel button always in the right place?"). If you really need numbers, you can even score points for each question or issue to arrive at a quantitative result.

Some useful heuristics include:

- Can the user figure out their current state? Is everything that the user needs to know about visible on the screen?
- Is the language on the screen (in the menus, on the buttons, in labels) the language of the user, not the language of the programmer?
- Is help available?
- Are error messages adequate?

Another useful technique is a *cognitive walkthrough*. There are more formal methods of cognitive walkthroughs, but an informal description is to simply imagine being the user and walking through the interface to perform a task. The goal is to figure out if the system *makes sense*.

To perform a cognitive walkthrough, you start out with a description of the system (which may be the system itself, if it's already running in some form), a description of the users' goals, and a careful process description of the how to perform a normal, useful task. The evaluator then walks through the process description asking herself:

- Does this make sense for this user? Given what we know about users from their description, will they understand what it is that they are to do next? For example, if we're talking about a desktop publishing system for ten year old students, we can't expect them to understand something about *kerning* unless we've given them lots of help first.
- Will the users be able to figure out what to do next? Again, the issue here is about visual state.
- Will the users be able to understand the feedback that they get? If everything goes well, will the user be able to tell? If something goes wrong, will the user be able to tell what went wrong—and what to do next?

6.2 Evaluation With Users

You want to get your interface as correct as you possibly can *before* involving your users. Most users are not computer experts who are used to software crashing. Even if your users are experts, they will be using your software to complete some *task* that they *care* about. They want your software to work. For these reasons, you use all the analysis methods that you can before you involve users.

That said, you could not really know how your software will work with users until you actually involve the users. Remember: "Know they users for they are not you." Users will almost certainly surprise you with the way that they want to use the software, or the issues that you missed in your analyses of the users and their tasks.

What you evaluate when you involve the users depends on two factors: What you want to learn, and how much effort you want to spend.

- If you just want to learn if *someone* can use your software, ask a typical user to come in and use the software while you observe her.
- If you want to show that your software is much better than a competitors, you will want to involve enough users that you can claim statistical significance (typically, 25 or more per group) in two different groups—one using your software and one using your competitor's. You will want to ask each user to perform some set of standard tasks, then measure the time to completion and the accuracy.

For the user interfaces that you might create for this book, we want something toward the first example. You should be able to test your designs with users, to find out if your design decisions were the right ones. You want to know if your interfaces *work*, to convince yourself at least. For that goal, a single user may be too little, but you probably do not need a large study with careful analyses.

The first and perhaps most powerful technique to use in evaluation with users is to simply watch them. Observation alone may not get you all the information that you want. If you see a user do something that generates an error message over and over again, you don't want to just observe—you want to know *why* the user is doing that.

There are a couple of ways of doing observations that provide more data. One technique is the *think aloud* where you encourage your user to say out loud what she is thinking as she is working through her task. You expect to hear things like, "Okay, now I want to print the document. Where is print, anyway? Usually, it's in the File menu. There it is." A think aloud makes clear what it is that the user is doing and *why* she is doing it. The disadvantage of a think aloud is that it sometimes changes what the user is doing. It's particularly hard for experts to verbalize why they're doing what they're doing.

It's useful to conduct a think aloud as explicitly occurring for the sake of improving the software, as opposed to being perceived as some test for the user. By involving the user in the evaluation process, the user feels more comfortable, and may be more likely to explore the system and provide useful insights.

You can't always use observational studies. Sometimes you simply have too many users involved, or can't afford the cost of having someone

observing users, or you can't get to where the users are. For example, it's hard to use observational studies when the software is for communications that can occur day or night, even from the user's bedroom. You can request that your user use the software when you're observing, but then you're really changing the user's task to fit the evaluation, and your results may not reflect actual use.

A second technique is a questionnaire, perhaps followed up by an interview. In a questionnaire, you can ask the users their experiences and opinions about the software. You can get the user to answer exactly the questions that you have about the software.

However, there is no guarantee that you are asking the right questions. You may be dying to know if the users like the new whizbang scroll bar you invented, when the reality may be that they can't use the system because they can't figure out how to open that window. You have to phrase your questions carefully so that you get the answers that you really want, but stay open to the possibility that the most important information is the piece that you do not expect.

You should have at least a few open-ended questions, to gather the information that you would not have expected. HCI researcher John Stasko advocates two questions: *What would you have me change the next time that I revise the software?* and *What should I make sure that I leave the same next time I revise the software?* The problem with open-ended questions is that they are more difficult to analyze and summarize.

Multiple choice and scalar questions (e.g., "on a scale of 1 to 5, where 1 is Strongly Agree and 5 is Strongly Disagree…") are more useful for analysis. You can use a spreadsheet to compute the average (and even the variance) on each question. The best questionnaires explore an issue with one more than one multiple choice or scalar question. For example, if you want to know if the user could use the WhizBang ScrollBar, you might want to ask them if they agree or disagree with the statements "I found the WhizBang ScrollBar easy to use" and later in the questionnaire, "I was able to control the WhizBang ScrollBar to get where I wanted." If the users agree with both of these, you can be fairly confident that you have a useful widget. If the users disagree with both, or agree with only one of the two, you know that the issue isn't so clear cut.

You might want to ask for some volunteers to identify themselves as potential interviewees. In an interview, you can ask the users your favorite questions, but you can also follow up on questions you're concerned about. If the results came back from a survey mixed (e.g., users love the software, but claim that they can't always get what they want done), then it's useful to sit down with a user and find out what the response is so confusing.

If you do not have the time or budget to conduct either an observational or questionnaire, it's still possible to get feedback from the users. Provide an email address, or better yet, a mailing list or newsgroup where users can provide feedback. Users' questions to one another can provide very useful insight about what's confusing about your software.

6.3 Evaluating Groupware

Groupware is software meant to be used by groups of people, which is a class of software that is particularly hard to evaluate. Consider, for example, using a word processor that allows you to annotate versions of a draft for comments back to the author. Now the author has all these versions and annotations to deal with. Perhaps some of the comments use the annotation mechanism well, and the author can integrate the comments easily. But say that other users just INSERT THEIR COMMENTS IN ALL CAPS. Did the software just fail the author or the annotator? Maybe the software should help the author deal with all-caps annotations? Maybe the software should have made it easier for the annotator to use the supported annotation mechanism? (The answer might be *Both*, but that software might get big and complicated.)

What's more, groupware is more complicated for the design goals associated with it. The goals of usability are still there (e.g., prevent users' errors), but there is often a *social agenda* as well. For example, the annotation mechanism in the word processor is implicitly encouraging authors to collaborate with others who would comment on written work. Other agendas might be to encourage more discussion of issues, or to involve more people in a discussion. These are different goals than just usability goals.

Measuring groupware goals is more complicated, too. How do you measure if you achieved "more discussion of issues"? Observation would not be effective, especially of any single user. Questionnaires might make sense here. You might ask users if they feel that there is more discussion with use of the software.

Another way of measuring groupware (or any interface) is to modify the interface to *record* what the users do. Typically, this is done without recording any identifying characteristics of the users, in order to protect their privacy. By recording what users do, you can answer questions that you might not otherwise be able to. For example, you can easily address the question of how many people get involved in using a groupware application. You may also be able to identify features that get used frequently—or don't get used at all. There is a challenge to using interface recordings: You know nothing about the context of use. You don't know if a long pause indicates confusion about the interface, or a distracting phone call. You don't know why someone does something over-and-over again.

Interface recordings are a rich source of information, but are tricky to use properly.

Exercises: Evaluating Interfaces

4. Again, pick an interface that you use. Develop a questionnaire about the interface and its usability. Ask people you know (at least three) to complete the questionnaire. Any surprises? Note differences among the three—what would account the differences? Differences in individual users, or differences in the tasks, or differences in the *strategies* for using the software?

5. Ask someone you know to let them watch you use a piece of software that you use. Does he use it differently? How? Are there features that he uses that you do not use?

References

A recommended book that includes alot on user-interface evaluations is:

Dix, A., Finlay, J., Abowd, G., & Beale, R. (1998). *Human-Computer Interaction*. (Second ed.). London: Prentice-Hall Europe.

Tog's book is:

Tognazzini, B. (1992). *Tog on Interface*. Reading, MA: Addison-Wesley.