11

Interfaces: User-Interface Design Patterns in the Unix Environment

> All our knowledge has its origins in our perceptions.
>
> —Leonardo Da Vinci

The interface of a program is the sum of all the ways that it communicates with human users and other programs. In Chapter 10, we discussed the use of environment variables, switches, run-control files and other parts of start-up-time interfaces. In this chapter, we'll untangle the history and explain the pragmatics of Unix interfaces after startup time. Because user-interface code normally consumes 40% or more of development time, knowing good design patterns is especially important here in order to avoid a lot of false starts and time-intensive rewrites.

In the Unix tradition of interface design, we encounter two themes over and over again. One is anticipatory design for communication with other programs; the other is the Rule of Least Surprise.

Unix programs can give you extra power from being used in synergistic combinations; we discussed various methods for hooking together such combinations in Chapter 7. The 'other programs' part of Unix interface design is not an afterthought or a marginal case as it is under many other operating systems. Rather, it is a central challenge that has to be balanced and integrated carefully with the demands of interface design for human users.

Much of Unix-community tradition about program interface design may seem odd and arbitrary—or even, in the age of the GUI, downright regressive—when you encounter that tradition for the first time. But in spite of various blemishes and irregularities, that tradition has an inner logic to it which is worth learning and understanding. It reflects heuristics accumulated over Unix's long history about ways to do effective communication both with human beings and with other programs. And it includes a set of conventions which create commonalities between programs—it defines 'least surprising' alternatives for a wide range of common interface-design problems.

After startup, programs normally get input or commands from the following sources:

- Data and commands presented on the program's standard input.
- Inputs passed through IPC, such as X server events and network messages.
- Files and devices in known locations (such as a data file name passed to or computed by the program).

Programs can emit results in all the same ways (with output going to standard output).

Some Unix programs are graphical, some have screen-oriented character interfaces, and some use a starkly simple text-filter design unchanged from the days of mechanical teletypes. To the uninitiated, it is often far from obvious why any given program uses the style it does—or, indeed, why Unix supports such a plethora of interface styles at all.

Unix has several competing interface styles. All are still alive for a reason; they're optimized for different situations. By understanding the fit between task and interface style, you will learn how to choose the right styles for the jobs you need to do.

11.1 Applying the Rule of Least Surprise

The Rule of Least Surprise is a general principle in the design of all kinds of interfaces, not just software: "Do the least surprising thing". It's a consequence of the fact that human beings can only pay attention to one thing at one time (see *The Humane Interface* [Raskin]). Surprises in the interface focus that single locus of attention on the interface, rather than on the task where it belongs.

Thus, to design usable interfaces, it's best when possible not to design an entire new interface model. Novelty is a barrier to entry; it puts a learning burden on the user, so minimize it. Instead, think carefully about the experience and knowledge of your user base. Try to find functional similarities between your program and programs

they are likely to already know about. Then mimic the relevant parts of the existing interfaces.

The Rule of Least Surprise should not be interpreted as a call for mechanical conservatism in design. Novelty raises the cost of a user's first few interactions with an interface, but poor design will make the interface needlessly painful forever. As in other sorts of design, rules are not a substitute for good taste and engineering judgment. Consider your tradeoffs carefully—and consider them from the *user's* point of view. The bias implied by the Rule of Least Surprise is a good one to hold consciously, mainly because interface designers (like other programmers) have an unconscious tendency to be too clever for the user's good.

One implication of the Rule of Least Surprise is this: Wherever possible, allow the user to delegate interface functions to a familiar program. We already observed in Chapter 7 that, if your program requires the user to edit significant amounts of text, you should write it to call an editor (specifiable by the user) rather than building in your own integrated editor. This will enable the *users*, who know their preferences better than you, to choose the least surprising alternative.

Elsewhere in this book we have advocated symbiosis and delegation as tactics for promoting code reuse and minimizing complexity. The point here is that when users can intercept the delegation, and direct it to an agent of their own choice, these techniques become not merely economical for the developer but actively empowering to users.

Further: When you can't delegate, emulate. The purpose of the Rule of Least Surprise is to reduce the amount of complexity a user must absorb to use an interface. Continuing the editor example, this means that if you must implement an embedded editor, it's best if the editor commands are a subset of those for a well-known general-purpose editor. (Or more than one. Both `bash` and `ksh` have command-line editors that allow the user to choose between *vi* and *Emacs* editing styles.)

Under the Unix versions of the Netscape and Mozilla Web browsers, for example, fill-in fields in forms recognize a subset of the default bindings for the *Emacs* editor. Control-A goes to start of line, Control-D deletes the next character, and so forth. This choice helps people who know *Emacs*, and leaves others no worse off than an arbitrary, idiosyncratic command set would have. The only way it could have been bettered was by choosing key bindings associated with some editor significantly more widely used than *Emacs*; and among Netscape's original user population there was no such animal.

These principles can be applied in many other areas of interface design. They suggest, for example, that it is deeply foolish to create novel document formats for an on-line help system when users are comfortable with an HTML Web browser. Or even that if you are designing an arcade-style game, it is wise to look at the gesture sets of previous games to see if you can give new users a feeling of comfort by allowing them to transfer joystick skills learned in other games.

11.2 History of Interface Design on Unix

Unix predates the modern graphics-intensive style of software interface design. For over a decade after the first Unix in 1969, command-line interfaces (CLIs) on teletypes and dumb text-mode terminals were the norm. Most of the basic Unix toolset (programs like *ls*(1), *cat*(1), and *grep*(1)) still reflect this heritage.

Gradually, after 1980, Unix evolved support for screen-painting on character-cell terminals. Programs began to mix command-line and visual interfaces, with common commands often bound to keystrokes that would not be echoed to the screen. Some of the early programs written in this style (often called 'curses' programs, after the screen-painting cursor-control library normally used to implement them, or 'roguelike' after the first application to use curses) are still used today; notable examples include the dungeon-crawling game *rogue*(1), the *vi*(1) text editor, and (from a few years later) the *elm*(1) mailer and its modern descendant *mutt*(1).

A few years later in the mid-1980s, the computing world as a whole began to assimilate the results of the pioneering work on graphical user interfaces (GUIs) that had been going on at Xerox's Palo Alto Research Center since the early 1970s. On personal computers, the Xerox PARC work inspired the Apple Macintosh interface and through that the design of Microsoft Windows. Unix's adaptation of these ideas took a rather more complicated path.

Around 1987 the X windowing system outcompeted several early contenders and prototype efforts to become the standard graphical-interface facility for Unix. Whether this was a good or a bad thing has remained a topic of debate ever since; some of the other contenders (notably Sun's Network Window System or NeWS) were arguably rather more powerful and elegant. X, however, had one overriding virtue; it was open source. The code had been developed at MIT by a research group more interested in exploring the problem space than in creating a product, and it remained freely redistributable and modifiable. It was thus able to attract support from a wide range of developers and sponsoring corporations who would have been reluctant to line up behind a single vendor's closed product. (This, of course, prefigured an important theme in the breakout of the Linux operating system ten years later.)

The designers of X decided early on that X would support "mechanism, not policy". Their objective was to make X as flexible and portable across platforms as possible, while putting as few constraints on the look and feel of X programs as they could manage. Look and feel, they decided, would be handled by 'toolkits'—libraries calling X services linked to user programs. X would also be designed to support multiple

window managers,[1] and would not require a window manager to have any special privileges or uniquely close integration with X's machinery.

This approach was the polar opposite of that taken by the Macintosh and Windows commercial products, which enforced particular look-and-feel policies by designing them right into the system. The difference in approach ensured that X would have a long-run evolutionary advantage by remaining adaptable as new discoveries were made about the human factors in interface design—but it also ensured that the X world would be divided by multiple toolkits, a profusion of window managers, and many experiments in look and feel.

Since the mid-1990s X has become ubiquitous even on the lowest-end personal Unix machines. Use of Unix from text-mode terminals, as opposed to graphics-capable computer consoles, has sharply declined and seems headed for extinction. Accordingly, the use of curses-style interfaces for new applications is also in decline; most new applications that would formerly have been designed in that style now use an X toolkit. It is instructive to note that Unix's older CLI design tradition is still quite vigorous and successfully competes with X in many areas.

It is also instructive to note that there are a few specific application areas in which curses-style (or 'roguelike') character-cell interfaces remain the norm—especially text editors and interactive communications programs such as mailers, newsreaders, and chat clients.

For historical reasons, then, there is a wide range of interface styles in Unix programs. Line-oriented, character-cell screen-oriented, and X-based—with the X-based world somewhat balkanized by the competition between multiple X toolkits and window managers (though this is less an issue in 2003 that was the case five or even three years ago).

11.3 Evaluating Interface Designs

All these interface styles survive because they are adapted for different jobs. When making design decisions about a project, it's important to know how to pick a style (or combine styles) that will be appropriate to your application and your user population.

We will use five basic metrics to categorize interface styles: *concision*, *expressiveness*, *ease*, *transparency*, and *scriptability*. We've already used some of these terms earlier in this book in ways that were preparation for defining them here. They are

1. A window manager handles associations between windows on the screen and running tasks. Window managers handle behaviors like title bars, placement, minimizing, maximizing, moving, resizing, and shading windows.

comparatives, not absolutes; they have to be evaluated with respect to a particular problem domain and with some knowledge of the users' skill base. Nevertheless, they will help organize our thinking in useful ways.

A program interface is 'concise' when the length and complexity of actions required to do a transaction with it has a low upper bound (the measurement might be in keystrokes, gestures, or seconds of attention required). Concise interfaces pack a lot of leverage into a relatively few bits or state changes.

Interfaces are 'expressive' when they can readily be used to command a wide variety of actions. The *most* expressive interfaces can command combinations of actions not anticipated by the designer of the program, but which nevertheless give the user useful and consistent results.

The difference between concision and expressiveness is an important one. Consider two different ways of entering text: from a keyboard, or by picking characters from a screen display with mouse clicks. These have equal expressiveness, but the keyboard is more concise (as we can easily verify by comparing average text-entry speeds). On the other hand, consider two dialects of the same programming language, one with a complex-number type and one not. Within the problem domain they have in common, their concision will be identical; but for a mathematician or electrical engineer, the dialect with complex numbers will be much more expressive.

The 'ease' of an interface is inversely proportional to the mnemonic load it puts on the user—how many things (commands, gestures, primitive concepts) the user has to remember specifically to support using that interface. Programming languages have a high mnemonic load and low ease; menus and well-labeled on-screen buttons are simpler.

Recall that we devoted an entire earlier chapter to 'transparency'. In that chapter we touched on the idea of interface transparency, and gave the Audacity audio editor as one superb example of it. But we were then much more interested in transparency of a different kind, one that relates to the structure of code rather than of user interfaces. We therefore described UI transparency in terms of its effect (nothing obtrudes between the user and the problem domain) rather than the specific features of design that produce it. Now it's time to zero in on these.

The 'transparency' of an interface is how few things the user has to remember about the state of his problem, his data, or his program while *using* the interface. An interface has high transparency when it naturally presents intermediate results, useful feedback, and error notifications on the effects of a user's actions. So-called WYSIWYG (What You See Is What You Get) interfaces are intended to maximize transparency, but sometimes backfire—especially by presenting an over-simplified view of the domain.

The related concept of discoverability applies to interface design, as well. A discoverable interface provides the user with assistance in learning it, such as a greeting message pointing to context-sensitive help, or explanatory balloon popups. Though

discoverability has to be implemented in rather different ways for each of the interface styles we shall consider, the degree to which it is achievable is largely independent of interface style. Thus, we shall not use it as a metric in this discussion.

Note that transparency of code and design does not automatically imply transparency of interface, or vice-versa! It is all too easy to point to code that has one but not the other.

The 'scriptability' of an interface is the ease with which it can be manipulated by other programs (e.g. through the IPC mechanisms discussed in Chapter 7). Scriptable programs are readily usable as components by other programs, reducing the need for costly custom coding and making it relatively easy to automate repetitive tasks.

That last point—automating repetitive tasks—deserves more attention than it usually gets. Unix programmers, administrators, and users develop a habit of thinking through the routine procedures they use, then packaging them so they no longer have to manually execute or even think about them any more. This habit depends on scriptable interfaces. It is a quiet but tremendous productivity booster not available in most other software environments.

It will be useful to bear in mind that humans and computer programs have very different cost functions with respect to these metrics. So do novice and expert human users in a particular problem domain. We'll explore how the tradeoffs between them change for different user populations.

11.4 Tradeoffs between CLI and Visual Interfaces

The CLI style of early Unix has retained its utility long after the demise of teletypes for two reasons. One is that command-line and command-language interfaces are more expressive than visual interfaces, especially for complex tasks. The other is that CLI interfaces are highly scriptable—they readily support the combining of programs, as we discussed in detail in Chapter 7. Usually (though not always) CLIs have an advantage in concision as well.

The disadvantage of the CLI style, of course, is that it almost always has high mnemonic load (low ease), and usually has low transparency. Most people (especially nontechnical end users) find such interfaces relatively cryptic and difficult to learn.

> On the other hand, the 'user-friendly' GUIs of other operating systems have their own problems. Finding the right buttons to push is like playing Adventure: the interfaces are just as burdensome as any Unix command line interface, save that one can in theory find the treasure by sufficient exploration. In Unix, one needs the manual.
>
> —Brian Kernighan

Database queries are a good example of the kind of interface for which pushing buttons is not just burdensome but extremely limiting. Neither keystroke commands to a full-screen character interface nor GUI gestures on a graphic display can express typical actions in the problem domain as expressively or concisely as typing SQL direct to a server. And it is certainly easier to make a client program utter SQL queries than it would be to have it simulate a user clicking a GUI!

On the other hand, many non-technical database users are so resistant to having to remember SQL syntax that they prefer a less concise and less expressive full-screen or GUI interface.

SQL is a good example for illustrating another point. The most powerful CLIs are not ad-hoc collections of commands, but imperative minilanguages designed along the lines we described in Chapter 8. These minilanguages are the highest-power, highest-complexity end of the CLI spectrum; they maximize expressiveness, but minimize ease. They are difficult to use and generally need to be discreetly veiled from ordinary end-users, but unbeatable when the capability and flexibility of the interface is the most important thing. When properly designed, they also score high on scriptability.

Some applications, unlike database queries, are naturally visual. Paint programs, Web browsers, and presentation software make three excellent examples. What these application domains have in common is that (a) transparency is extremely valuable, and (b) the primitive actions in the problem domain are themselves visual: “draw this”, “show me what I’m pointing at”, “put this here”.

The flip side of paint programs is that it is difficult to capture relationships within the pictures they are manipulating. It takes careful, thoughtful design to give the user any handle on the structure of images with repeated elements, for example. This is a general design problem with visual interfaces.

In Chapter 6 we looked at the Audacity sound file editor. Its interface design succeeds because it does a particularly clean job of mapping its audio application domain onto a simple set of visual representations (borrowed from equalizer displays on stereos). It does this by thoroughly following through the consequences of a single translation: sounds to waveform images. The visual operations are not a mere grab-bag of low-level tweaks; they are all tied to that translation.

In applications that are *not* naturally visual, however, visual interfaces are most appropriate for simple one-off or infrequent tasks performed by novice users (a point the database example illustrates).

Resistance to CLI interfaces tends to decrease as users become more expert. In many problem domains, users (especially *frequent* users) reach a crossover point at which the concision and expressiveness of CLI becomes more valuable than avoiding its mnemonic load. Thus, for example, computing novices prefer the ease of GUI desktops, but experienced users often gradually discover that they prefer typing commands to a shell.

CLIs also tend to gain utility as problems scale up and involve more in the way of canned, procedural and repetitive actions. Thus, for example, a WYSIWYG desktop-publishing program is usually the easiest route to composing relatively small and unstructured documents such as business letters. But for complex book-sized documents that are assembled from sections and may require many global format changes or structural manipulation during composition, a minilanguage formatter such as *troff*, *Tex*, or some XML-markup processor is usually a more effective choice (see Chapter 18 for more discussion of this tradeoff).

Even in domains that are naturally visual, scaling up the problem size tends to tilt the tradeoff toward a CLI. If you need to fetch and save one Web page from a given URL, point and click (or type and click) is fine. But for Web forms, you're going to use a keyboard. And if you need to fetch and save the pages corresponding to a given list of fifty URLs, a CLI client that can read URLs from standard input or the command line can save you a lot of unnecessary motion.

As another example, consider modifying the color table in a graphic image. If you want to change one color (say, to lighten it by an amount you will only know is right when you see it) a visual dialogue with a color-picker widget is almost mandatory. But suppose you need to replace the entire table with a set of specified RGB values, or to create and index large numbers of thumbnails. These are operations that GUIs usually lack the expressive power to specify. Even when they do, invoking a properly designed CLI or filter program will do the job far more concisely.

Finally (as we observed earlier on) CLIs are important in facilitating using programs from other programs. A GUI graphics editor that *can* handle making a batch of thumbnails for a list of files probably does it with a plugin written in a scripting language, calling an internal CLI of the graphics editor (as in the GIMP's script-fu facility). Unix environments bring the value of CLIs into sharper relief precisely because their IPC facilities are rich, have low overhead, and are easily accessible from user programs.

The explosion of interest in GUIs since 1984 has had the unfortunate effect of obscuring the virtues of CLIs. The design of consumer software, in particular, has become heavily skewed toward GUIs. While this is a good choice for the novice and casual users that constitute most of the consumer market, it also exacts hidden costs on more expert users as they run up against the expressiveness limits of GUIs—costs which steadily increase as the users take on more demanding problems. Most of these costs derive from the fact that GUIs are simply not scriptable at all—*every* interaction with them has to be human-driven.

Gentner & Nielsen sum up the tradeoff very well in *The Anti-Mac Interface* [Gentner-Nielsen]: "[Visual interfaces] work well for simple actions with a small number of objects, but as the number of actions or objects increases, direct manipulation quickly becomes repetitive drudgery. The dark side of a direct manipulation interface is that you have to manipulate everything. Instead of an executive who gives

high-level instructions, the user is reduced to an assembly-line worker who must carry out the same task over and over." Noted science-fiction writer Neal Stephenson made the same point, less directly but more entertainingly, in his brilliant and discursive essay *In the Beginning Was the Command Line* [Stephenson].

A typical Unix old hand's take on this problem is rather less theoretical:

> The commercial world generally goes for the novice mode because (a) purchase decisions are often made on the basis of 30 seconds trial, and (b) it minimizes the demands on customer support to have only a dumbed-down GUI. I find many non-Unix systems very frustrating because, for example, they will provide no way to do something on a hundred or a thousand files; I want to write a script, and there's no support for it. The basic problem is that they've assumed all users are novices all the time, and then they bash Unix because it doesn't cater to that model.
>
> —Mike Lesk

For the long haul, then—for serving both casual and expert users, for cooperating with other computer programs, and whether the problem domain is naturally visual or not—support for *both* CLI and visual interfaces is important. Unix's history positions it well to meet both sets of needs. After presenting an indicative case study, we will examine the characteristic design patterns that the Unix tradition has evolved to meet them.

11.4.1 Case Study: Two Ways to Write a Calculator Program

To be more concrete, let us contrast how the GUI and CLI styles can be usefully applied to the design of a simple interactive program: a desk calculator. Our examples for contrast are *dc*(1)/*bc*(1) and *xcalc*(1).

The original Unix desk calculator program, first distributed with Version 7, was *dc*(1)—a reverse-Polish-notation calculator that could handle unlimited-precision arithmetic. Later, an algebraic (infix notation) calculator language, *bc*(1), was implemented on top of dc (we used the relationship between these programs as a case study in Chapter 7, and again in Chapter 8). Both of these programs use a CLI. You type an expression on standard input, you press enter, and the value of the expression is printed on standard output.

The *xcalc*(1) program, on the other hand, visually simulates a simple calculator, with clickable buttons and a calculator-style display.

The *xcalc*(1) approach is simpler to describe because it mimics an interface with which novice users will be familiar; the man page says, in fact, "The numbered keys,

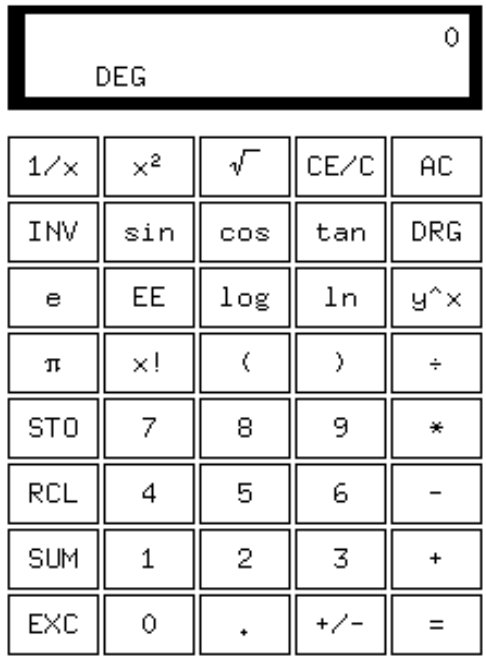

Figure 11.1: The xcalc GUI.

the +/– key, and the +, –, *, /, and = keys all do exactly what you would expect them to." All the capabilities of the program are conveyed by the visible button labels. This is the Rule of Least Surprise in its strongest form, and a real advantage for infrequent and novice users who will never have to read a man page to use the program.

However, *xcalc* also inherits the almost complete non-transparency of a calculator; when evaluating a complex expression, you don't get to see and sanity-check your keystrokes—which can be a problem if, say, you misplace a decimal point in an expression like (2.51 + 4.6) * 0.3. There's no history, so you can't check. You'll get a result, but it won't be the result of the calculation you intended.

With the *dc*(1) and *bc*(1) programs, on the other hand, you can edit mistakes out of the expression as you build it. Their interface is more transparent, because you can see the calculation that is being performed at every stage. It is more expressive because the *dc*/*bc* interpreter, not being limited to what fits on a reasonably-sized visual mockup of a calculator, can include a much larger repertoire of functions (and facilities such as if/then/else, stored variables, and iteration). It also incurs, of course, a higher mnemonic load.

Concision is more of a toss-up; good typists will find the CLI more concise, while poor ones may find it faster to point and click. Scriptability is not a toss-up; *dc*/*bc* can easily be used as a filter, but xcalc can't be scripted at all.

The tradeoff between ease for novices and utility for expert users is very clear here. For casual use in situations where a mental-arithmetic error check is not hard, *xcalc* wins. For more complex calculations where the steps must not only be correct but must be *seen* to be correct, or in which they are most conveniently generated by another program, *dc*/*bc* wins.

11.5 Transparency, Expressiveness, and Configurability

Unix programmers inherit a strong bias toward making interfaces expressive and configurable. Like programmers from other traditions, they think about how to match their interfaces to the target audience—but they differ in how they deal with uncertainty about that target audience. Software developers whose experience is primarily with client operating systems default toward making interfaces simple; they are willing to sacrifice expressiveness to gain ease. Unix programmers default toward making interfaces expressive and transparent, and are more willing to sacrifice ease to get these qualities.

The results of this attitude have often been described as interfaces written "by programmers, for programmers". But this oversimplifies the matter in an important way. When a Unix programmer opts for configurability and expressiveness over ease, he is not necessarily thinking of his audience as consisting solely of other programmers; rather, he is often acting on a gut-level instinct that in the absence of knowledge about end-users' intentions it is best not to patronize or second-guess them.

> The downside of this attitude (which is a close cousin to "mechanism, not policy") is a tendency to assume that when the highly configurable and expressive interface is done, the job is finished... even if the result is almost impossible for anyone else to use without lengthy study. The flip side of configurability is an urgent need for good defaults and an easy way to set everything to the default. The flip side of expressivity is a need for guidance—be it in the program or the documentation—on where to get started and how to achieve the most commonly-desired results.
>
> —Henry Spencer

The Rule of Transparency also has an influence. When a Unix programmer is writing to meet an RFC or other standard that defines a set of control options, he tends to assume that his job is to provide a complete and transparent interface to all of those

options; whether or not he thinks any given one will actually be used is secondary. His job is mechanism; policy belongs to the user.

This mind-set leads to a much stricter attitude about what constitutes standards conformance, one in which incomplete support is much less tolerable. In cases where a Macintosh or Windows developer would say "We don't need to support that feature of the standard; most users won't care, and it's too complicated for them," a Unix developer is likely to say "We don't know that nobody will ever want this feature or option, therefore we must support it."

These attitudes can lead to clashes when a Unix programmer is working with others, who are likely to interpret his design choices as a blithe willingness to burden users with technical details that are obscure, pointless, and even frightening. Mac or Windows programmers fear scaring away the many to serve the advanced needs of the few.

The Unix programmer, on the other hand, is likely to see defaulting away from expressiveness as a sort of cop-out or even betrayal of future users, who will know their own requirements better than the present implementer. Ironically, though the Unix attitude is often construed as a sort of programmer arrogance, it is actually a form of humility—one often acquired along with years of battle scars.

The extent to which the Unix attitude is appropriate varies. Whichever side of this divide you the reader are on, it is wise to learn to listen to the other, and wise to understand the premises behind the opposing point of view. Rather than falling into the trap of either intimidating users or condescending to them, it may be possible to build transparent interfaces in which the advanced features are present but inconspicuous. The *audacity* and *kmail* case studies in Chapter 6 are good examples to follow.

Finally, a note about user-interface design for nontechnical end-users. This is a demanding art, and Unix programmers don't have a tradition of being very good at it. But with the ideas we've developed from examining the Unix tradition, it is possible to make one strong and useful statement about it. That is: when people say a user interface is *intuitive*, what they means is that it (a) is discoverable, (b) is transparent in use, and (c) obeys the Rule of Least Surprise.[2] Of these three rules, Least Surprise is the least binding; initial surprises can be coped with if discoverability and transparency make longer-term use rewarding.

The user interfaces of today's cellphones (for example) have relatively high mnemonic load in that you have to maintain at least a rough mental map of the interface menus to use them rapidly without constantly having to spend attention on checking where you are in the hierarchy. But the better-designed ones rapidly become 'intuitive' for their users anyway, because they have these three qualities.

2. This insight comes to us from a nontechnical end-user who just happens to be the author's wife Catherine Raymond.

Intuitiveness is not quite the same quality as ease, because (as the cellphone example shows) people can develop what they think of as ‘intuitions‘ about transparent interfaces that have fairly high mnemonic load, as long as simple operations are easy and there is a discovery path that allows them to assimilate the interface's more difficult corners one step at a time.

11.6 Unix Interface Design Patterns

In the Unix tradition, the tradeoffs we described above are met by well-established interface design patterns. Here is a bestiary of these patterns, with analyses and examples. We'll follow it with a discussion of how to apply them.

Note that this bestiary does not include GUI design patterns (though it includes a design pattern that can use a GUI as a component). There are no design patterns in graphical user interfaces themselves that are specifically native to Unix. A promising beginning of a discussion of GUI design patterns in general can be found at *Experiences—A Pattern Language for User Interface Design* [Coram-Lee].

Also note that programs may have modes that fit more than one interface pattern. A program that has a compiler-like interface, for example, may behave as a filter when no file arguments are specified on the command line (many format converters behave like this).

11.6.1 The Filter Pattern

The interface-design pattern most classically associated with Unix is the *filter*. A filter program takes data on standard input, transforms it in some fashion, and sends the result to standard output. Filters are not interactive; they may query their startup environment, and are typically controlled by command-line options, but they do not require feedback or commands from the user in their input stream.

Two classic examples of filters are *tr*(1) and *grep*(1). The *tr*(1) program is a utility that translates data on standard input to results on standard output using a translation specification given on the command line. The *grep*(1) program selects lines from standard input according to a match expression specified on the command line; the resulting selected lines go to standard output. A third is the *sort*(1) utility, which sorts lines in input according to criteria specified on the command line and issues the sorted result to standard output.

Both *grep*(1) and *sort*(1) (but not *tr*(1)) can alternatively take data input from a file (or files) named on the command line, in which case they do not read standard input but act instead as though that input were the catenation of the named files read in the order they appear. (In this case it is also expected that specifying "-" as a

filename on the command line will direct the program explicitly to read from standard input.) The archetype of such 'catlike' filters is *cat*(1), and filters are expected to behave this way unless there are application-specific reasons to treat files named on the command line differently.

When designing filters, it is well to bear in mind some additional rules, partly developed in Chapter 1

1. *Remember Postel's Prescription: Be generous in what you accept, rigorous in what you emit.* That is, try to accept as loose and sloppy an input format as you can and emit as well-structured and tight an output format as you can. Doing the former reduces the odds that the filter will be brittle in the face of unexpected inputs, and break in someone's hand (or in the middle of someone's toolchain). Doing the latter increases the odds that your filter will someday be useful as an input to other programs.

2. *When filtering, never throw away information you don't need to.* This, too, increases the odds that your filter will someday be useful as an input to other programs. Information you discard is information that no later stage in a pipeline can use.

3. *When filtering, never add noise.* Avoid adding non-essential information, and avoid reformatting in ways that might make the output more difficult for downstream programs to parse. The most common offenders are cosmetic touches like headers, footers, blank/ruler lines, summaries and conversions like adding aligned columns, or writing a factor of "1.5" as "150%". Times and dates are a particular bother because they're hard for downstream programs to parse. Any such additions should be optional and controlled by switches. If your program emits dates, it's good practice to have a switch that can force them into ISO8601 YYYY-MM-DD and hh:mm:ss formats—or, better yet, use those by default.

The term "filter" for this pattern is long-established Unix jargon.

> "Filter" is indeed long-established. It came into use on day one of pipes. The term was a natural transferral from electrical-engineering usage: data flowed from source through filters to sink. Source or sink could be either process or file. The collective EE term, "circuit", was never considered, since the plumbing metaphor for data flow was already well established.
>
> —Doug McIlroy

Some programs have interface design patterns like the filter, but even simpler (and, importantly, even easier to script). They are cantrips, sources, and sinks.

11.6.2 The Cantrip Pattern

The cantrip interface design pattern is the simplest of all. No input, no output, just an invocation and a numeric exit status. A cantrip's behavior is controlled only by startup conditions. Programs don't get any more scriptable than this.

Thus, the cantrip design pattern is an excellent default when the program doesn't require any run-time interaction with the user other than fairly simple setup of initial conditions or control information.

Indeed, because scriptability is important, Unix designers learn to resist the temptation to write more interactive programs when cantrips will do. A collection of cantrips can always be driven from an interactive wrapper or shell program, but interactive programs are harder to script. Good style therefore demands that you try to find a cantrip design for your tool before giving in to the temptation to write an interactive interface that will be harder to script. And when interactivity seems necessary, remember the characteristic Unix design pattern of separating the engine from the interface; often, the right thing is an interactive wrapper written in some scripting language that calls a cantrip to do the real work.

The console utility *clear*(1), which simply clears your screen, is the purest possible cantrip; it doesn't even take command-line options. Other classic simple examples are *rm*(1) and *touch*(1). The *startx*(1) program used to launch X is a complex example, typical of a whole class of daemon-summoning cantrips.

This interface design pattern, though fairly common, has not traditionally been named; the term "cantrip" is my invention. (In origin, it's a Scots-dialect word for a magic spell, which has been picked up by a popular fantasy-role-playing game to tag a spell that can be cast instantly, with minimal or no preparation.)

11.6.3 The Source Pattern

A *source* is a filter-like program that requires no input; its output is controlled only by startup conditions. The paradigmatic example would be *ls*(1), the Unix directory lister. Other classic examples include *who*(1) and *ps*(1).

Under Unix, report generators like *ls*(1), *ps*(1), and *who*(1) tend strongly to obey the source pattern, so their output can be filtered with standard tools.

The term 'source' is, as Doug McIlroy noted, very traditional. It is less common than it might be because 'source' has other important meanings.

11.6.4 The Sink Pattern

A *sink* is a filter-like program that consumes standard input but emits nothing to standard output. Again, its actions on the input data are controlled only by startup conditions.

This interface pattern is unusual, and there are few well-known examples. One is *lpr*(1), the Unix print spooler. It will queue text passed to it on standard input for printing. Like many sink programs, it will also process files named to it on the command line. Another example is *mail*(1) in its mail-sending mode.

Many programs that might appear at first glance to be sinks take control information as well as data on standard input and are actually instances of something like the *ed* pattern (see below).

The term *sponge* is sometimes applied specifically to sink programs like *sort*(1) that must read their entire input before they can process any of it.

The term 'sink' is traditional and common.

11.6.5 The Compiler Pattern

Compiler-like programs use neither standard output nor standard input; they may issue error messages to standard error, however. Instead, a compiler-like program takes file or resource names from the command line, transforms the names of those resources in some way, and emits output under the transformed names. Like cantrips, compiler-like programs do not require user interaction after startup time.

This pattern is so named because its paradigm is the C compiler, *cc*(1) (or, under Linux and many other modern Unixes, *gcc*(1)). But it is also widely used for programs that do (for example) graphics file conversions or compression/decompression.

A good example of the former is the *gif2png*(1) program used to convert GIF (Graphic Interchange Format) to PNG (Portable Network Graphics).[3] Good examples of the latter are the *gzip*(1) and *gunzip*(1) GNU compression utilities, almost certainly shipped with your Unix system.

In general, the compiler interface design pattern is a good model when your program often needs to operate on multiple named resources and can be written to have low interactivity (with its control information supplied at startup time). Compiler-like programs are readily scriptable.

The term "compiler-like interface" for this pattern is well-understood in the Unix community.

3. Sources for this program, and other converters with similar interfaces, are available at the PNG website (`http://www.cdrom.com/pub/png/`).

11.6.6 The *ed* pattern

All the previous patterns have very low interactivity; they use only control information passed in at startup time, and separate from the data. Many programs, of course, need to be driven by a continuing dialog with the user after startup time.

In the Unix tradition, the simplest interactive design pattern is exemplified by *ed*(1), the Unix line editor. Other classic examples of this pattern include *ftp*(1) and *sh*(1), the Unix shell. The *ed*(1) program takes a filename argument; it modifies that file. On its input, it accepts command lines. Some of the commands result in output to standard output, which is intended to be seen immediately by the user as part of the dialog with the program.

An actual sample *ed*(1) session will be included in Chapter 13.

Many browser- and editor-like programs under Unix obey this pattern, even when the named resource they edit is something other than a text file. Consider *gdb*(1), the GNU symbolic debugger, as an example.

Programs obeying the *ed* interface design pattern are not quite so scriptable as would be the simpler interface types resembling filters. You can feed them commands on standard input, but it is trickier to generate sequences of commands (and interpret any output they might ship back) than it is to just set environment variables and command-line options. If the action of the commands is not so predictable that they can be run blind (e.g. with a here-document as input and ignoring output), driving *ed*-like programs requires a protocol, and a corresponding state machine in the calling process. This raises the problems we noted in Chapter 7 during the discussion of slave process control.

Nevertheless, this is the simplest and most scriptable pattern that supports fully interactive programs. Accordingly, it is still quite useful as a component of the "separated engine and interface" pattern we'll describe below.

11.6.7 The Roguelike Pattern

The roguelike pattern is so named because its first example was the dungeon-crawling game *rogue*(1) (see Figure 11.2) under BSD; the adjective "roguelike" for this pattern is widely recognized in Unix tradition. Roguelike programs are designed to be run on a system console, an X terminal emulator, or a video display terminal. They use the full screen and support a visual interface style, but with character-cell display rather than graphics and a mouse.

Commands are typically single keystrokes not echoed to the user (as opposed to the command lines of the *ed* pattern), though some will open a command window (often, though not always, the last line of the screen) on which more elaborate invocations can be typed. The command architecture often makes heavy use of the arrow keys to select screen locations or lines on which to operate.

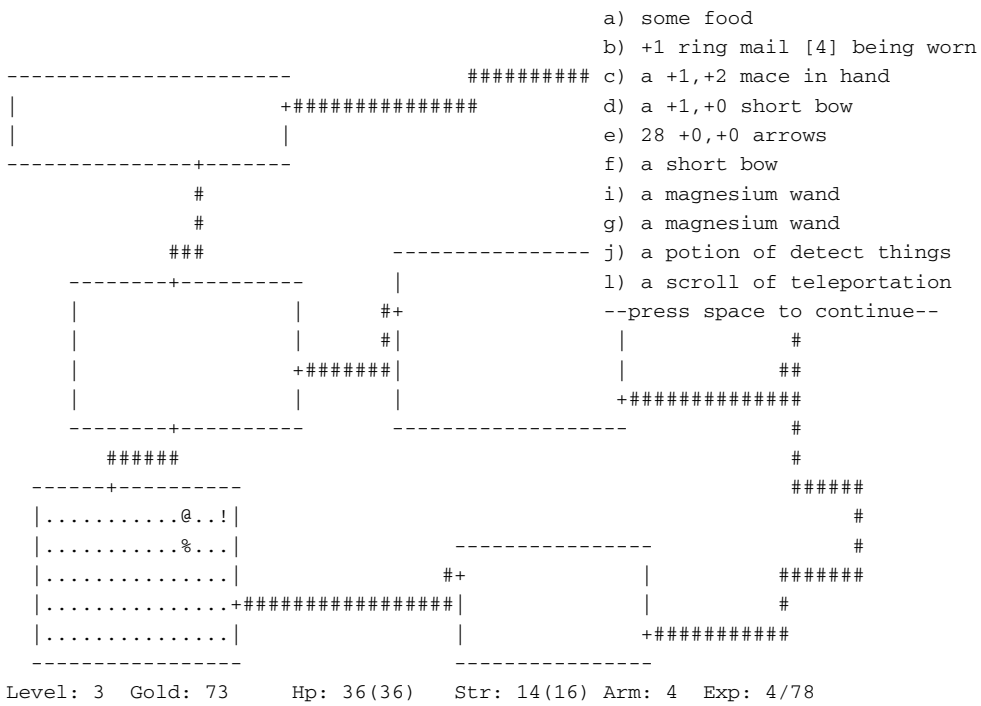

Figure 11.2: Screen shot of the original Rogue game.

Programs written in this pattern tend to model themselves on either *vi*(1) or *emacs*(1) and (obeying the Rule of Least Surprise) use their command sequences for common operations such as getting help or terminating the program. Thus, for example, one can expect one of the commands ‘x’, ‘q’, or ‘C-x C-c’ to terminate a program written to this pattern.

Some other interface tropes associated with this pattern include: (a) the use of one-item-per-line menus, with the currently-selected item indicated by bold or reverse-video highlighting, and (b) ‘mode lines’—program status summaries carried on a highlighted screen line, often near the bottom or at the top of the screen.

The roguelike pattern evolved in a world of video display terminals; many of these didn’t have arrow or function keys. In a world of graphics-capable personal computers, with character-cell terminals a fading memory, it’s easy to forget what an influence this pattern exerted on design; but the early exemplars of the roguelike pattern were designed a few years before IBM standardized the PC keyboard in 1981. As a result, a traditional but now archaic part of the roguelike pattern is the use of the h, j, k, and l as cursor keys whenever they are not being interpreted as self-inserting characters

in an edit window; invariably k is up, j is down, h is left, and l is right. This history also explains why older Unix programs tend not to use the ALT keys and to use function keys in a limited way if at all.

Programs obeying this pattern are legion: The *vi*(1) text editor in all its variants, and the *emacs*(1) editor; *elm*(1), *pine*(1), *mutt*(1), and most other Unix mail readers; *tin*(1), *slrn*(1), and other Usenet newsreaders; the *lynx*(1) Web browser; and many others. Most Unix programmers spend most of their time driving programs with interfaces like these.

The roguelike pattern is hard to script; indeed scripting it is seldom even attempted. Among other things, this pattern uses raw-mode character-by-character input, which is inconvenient for scripting. It's also quite hard to interpret the output programmatically, because it usually consists of sequences of incremental screen-painting actions.

Nor does this pattern have the visual slickness of a mouse-driven full GUI. While the point of using the full screen interface is to support simple kinds of direct-manipulation and menu interfaces, roguelike programs still require users to learn a command repertoire. Indeed, interfaces built on the roguelike pattern show a tendency to degenerate into a sort of cluttered wilderness of modes and meta-shift-cokebottle commands that only hard-core hackers can love. It would seem that this pattern has the worst of both worlds, being neither scriptable nor conforming to recent fashions in design for end-users.

But there must be some value in this pattern. Roguelike mailers, newsreaders, editors, and other programs remain extremely popular even among people who invariably run them through terminal emulators on an X display that supports GUI competitors. Moreover, the roguelike pattern is so pervasive that under Unix even GUI programs often emulate it, adding mouse and graphics support to a command and display interface that still looks rather roguelike. The X mode of *emacs*(1), and the *xchat*(1) client are good examples of such adaptation. What accounts for the pattern's continuing popularity?

Efficiency, and perceived efficiency, seem to be important factors. Roguelike programs tend to be fast and lightweight relative to their nearest GUI competitors. For startup and runtime speed, running a roguelike program in an Xterm may be preferable to invoking a GUI that will chew up substantial resources setting up its displays and respond more slowly afterwards. Also, programs with a roguelike design pattern can be used over telnet links or low-speed dialup lines for which X is not an option.

Touch-typists often prefer roguelike programs because they can avoid taking their hands off the keyboard to move a mouse. Given a choice, touch-typists will prefer interfaces that minimize keystrokes far off the home row; this may account for a significant percentage of *vi*(1)'s popularity.

Perhaps more importantly, roguelike interfaces are predictable and sparing in their use of screen real estate on an X display; they do not clutter the display with multiple windows, frame widgets, dialog boxes, or other GUI impedimenta. This makes the

pattern well suited for use in programs that must frequently share the user's attention with other programs (as is especially the case with editors, mailers, newsreaders, chat clients, and other communication programs).

Finally (and probably most importantly) the roguelike pattern tends to appeal more than GUIs to people who value the concision and expressiveness of a command set enough to tolerate the added mnemonic load. We saw above that there are good reasons for this preference to become more common as task complexity, use frequency, and user experience rise. The roguelike pattern meets this preference while also supporting GUI-like elements of direct manipulation as an *ed*-pattern program cannot. Thus, far from having only the worst of both worlds, the roguelike interface design pattern can capture some of the best.

11.6.8 The 'Separated Engine and Interface' Pattern

In Chapter 7 we argued against building monster single-process monoliths, and that it is often possible to lower the global complexity of programs by splitting them into communicating pieces. In the Unix world, this tactic is frequently applied by separating the 'engine' part of the program (core algorithms and logic specific to its application domain) from the 'interface' part (which accepts user commands, displays results, and may provide services such as interactive help or command history). In fact, this separated-engine-and-interface pattern is probably the one most characteristic interface design pattern of Unix.

(The other, more obvious candidate for that distinction would be filters. But filters are more often found in non-Unix environments than engine/interface pairs with bidirectional traffic between them. Simulating pipelines is easy; the more sophisticated IPC mechanisms required for engine/interface pairs are hard.)

Owen Taylor, maintainer of the GTK+ library widely used for writing user interfaces under X, beautifully brings out the engineering benefits of this kind of partitioning at the end of his note Why GTK_MODULES is not a security hole (`http://www.gtk.org/setuid.html`); he finishes by writing "[T]he secure setuid program is a 500 line program that does only what it needs to, rather than a 500,000 line library whose essential task is user interfaces."

This is not a new idea. Xerox PARC's early research into graphical user interfaces led them to propose the "model-view-controller" pattern as an archetype for GUIs.

- The "model" is what in the Unix world is usually called an "engine". The model contains the domain-specific data structures and logic for your application. Database servers are archetypal examples of models.

- The "view" part is what renders your domain objects into a visible form. In a really well-separated model/view/controller application, the view component is

notified of updates to the model and responds on its own, rather than being driven synchronously by the controller or by explicit requests for a refresh.

- The "controller" processes user requests and passes them as commands to the model.

In practice, the view and controller parts tend to be more closely bound together than either is to the model. Most GUIs, for example, combine view and controller behavior. They tend to be separated only when the application demands multiple views of the model.

Under Unix, application of the model/view/controller pattern is far more common than elsewhere precisely because there is a strong "do one thing well" tradition, and IPC methods are both easy and flexible.

An especially powerful form of this technique couples a policy interface (often a GUI combining view and controller functions) with an engine (model) that contains an interpreter for a domain-specific minilanguage. We examined this pattern in Chapter 8, focusing on minilanguage design; now it's time to look at the different ways that such engines can form components of larger systems of code.

There are several major variants of this pattern.

11.6.8.1 Configurator/Actor Pair

In a configurator/actor pair, the interface part controls the startup environment of a filter or daemon-like program which then runs without requiring user commands.

The programs *fetchmail*(1) and *fetchmailconf*(1) (which we've already used as case studies in discoverability and data-driven programming and will encounter again as language case studies in Chapter 14) are a good example of a configurator/actor pair. *fetchmailconf* is the interactive dotfile configurator that ships with fetchmail. *fetchmailconf* can also serve as a GUI wrapper that runs fetchmail in either foreground or background mode.

This design pattern enables both *fetchmail* and *fetchmailconf* to specialize in what they do well, and indeed to be written in different languages appropriate to their task domains. Fetchmail, which usually runs in background as a daemon, need not be bloated with GUI code. Conversely, *fetchmailconf* can specialize in elaborate GUIness without exacting size and complexity costs from fetchmail. Finally, because the information channels between them are narrow and well-defined, it remains possible to drive *fetchmail* from the command line and from scripts other than *fetchmailconf*.

The term "configurator/actor" is my invention.

11.6.8.2 Spooler/Daemon Pair

A slight variant of the configurator/actor pair can be useful in situations that require serialized access to a shared resource in a batch mode; that is, when a well-defined job stream or sequence of requests requires some shared resource, but no individual job requires user interaction.

In this spooler/daemon pattern, the spooler or front end simply drops job requests and data in a spool area. The job requests and data are simply files; the spool area is typically just a directory. The location of the directory and the format of the job requests are agreed on by the spooler and daemon.

The daemon runs forever in background, polling the spool directory, looking there for work to do. When it finds a job request, it tries to process the associated data. If it succeeds, the job request and data are deleted out of the spool area.

The classic example of this pattern is the Unix print spooler system, *lpr*(1)/*lpd*(1). The front end is *lpr*(1); it simply drops files to be printed in a spool area periodically scanned by *lpd*. *lpd*'s job is simply to serialize access to the printer devices.

Another classic example is the pair *at*(1)/*atd*(1), which schedules commands for execution at specified times. A third example, historically important though no longer in wide use, was UUCP—the Unix-to-Unix Copy Program commonly used as a mail transport over dial-up lines before the Internet explosion of the early 1990s.

The spooler/daemon pattern remains important in mail-transport programs (which are batchy by nature). The front ends of mail transports such as *sendmail*(1) and *qmail*(1) usually make one try at delivering mail immediately, through SMTP over an outbound Internet connection. If that attempt fails, the mail will fall into a spool area; a daemon version or mode of the mail transport will retry the delivery later.

Typically, a spooler/daemon system has four parts: a job launcher, a queue lister, a job-cancellation utility, and a spooling daemon, In fact, the presence of the first three parts is a sure clue that there is a spooler daemon behind them somewhere.

The terms "spooler" and "daemon" are well-established Unix jargon. ('Spooler' actually dates back to early mainframe days.)

11.6.8.3 Driver/Engine pair

In this pattern, unlike a configurator/actor or spooler/server pair, the interface part supplies commands to and interprets output from an engine after startup; the engine has a simpler interface pattern. The IPC method used is an implementation detail; the engine may be a slave process of the driver (in the sense we discussed in Chapter 7) or the engine and driver may communicate through sockets, or shared memory, or any other IPC method. The key points are (a) the interactivity of the pair, and (b) the ability of the engine to run standalone with its own interface.

Such pairs are trickier to write than configurator/actor pairs because they are more tightly and intricately coupled; the driver must have knowledge not merely about the engine's expected startup environment but about its command set and response formats as well.

When the engine has been designed for scriptability, however, it is not uncommon for the driver part to be written by someone other than the engine author, or for more than one driver to front-end a given engine. An excellent example of both is provided by the programs *gv*(1) and *ghostview*(1), which are drivers for *gs*(1), the Ghostscript interpreter. GhostScript renders PostScript to various graphics formats and lower-level printer-control languages. The gv and ghostview programs provide GUI wrappers for GhostScript's rather idiosyncratic invocation switches and command syntax.

Another excellent example of this pattern is the `xcdroast/cdrtools` combination. The cdrtools distribution provides a program *cdrecord*(1) with a command-line interface. The *cdrecord* code specializes in knowing everything about talking to CD-ROM hardware. *xcdroast* is a GUI; it specializes in providing a pleasant user experience. The *xcdroast*(1) program calls *cdrecord*(1) to do most of its work.

xcdroast also calls other CLI tools: *cdda2wav*(1) (a sound file converter) and *mkisofs*(1) (a tool for creating ISO-9660 CD-ROM file system images from a list of files). The details of how these tools are invoked are hidden from the user, who can think in terms centered on the task of making CDs rather than having to know directly about the arcana of sound-file conversion or filesystem structure. Equally importantly, the implementers of each of these tools can concentrate on their domain-specific expertise without having to be user-interface experts.

> A key pitfall of driver/engine organization is that frequently the driver must understand the state of the engine in order to reflect it to the user. If the engine action is practically instantaneous, it's not a problem, but if the engine can take a long time (e.g., when accessing many URLs) the lack of feedback can be a significant issue. A similar problem is responding to errors. For example, the traditional (although not very Unix-like) confirmation question about whether it's OK to overwrite a file that already exists is kind of painful to write in the driver/engine world; the engine, which detects the problem, has to ask the driver to do the confirmation prompting.
>
> —Steve Johnson

It's important to design the engine so that it not only does the right thing, but also notifies the driver about what it's doing so the driver can present a graceful interface with appropriate feedback.

The terms "driver" and "engine" are uncommon but established in the Unix community.

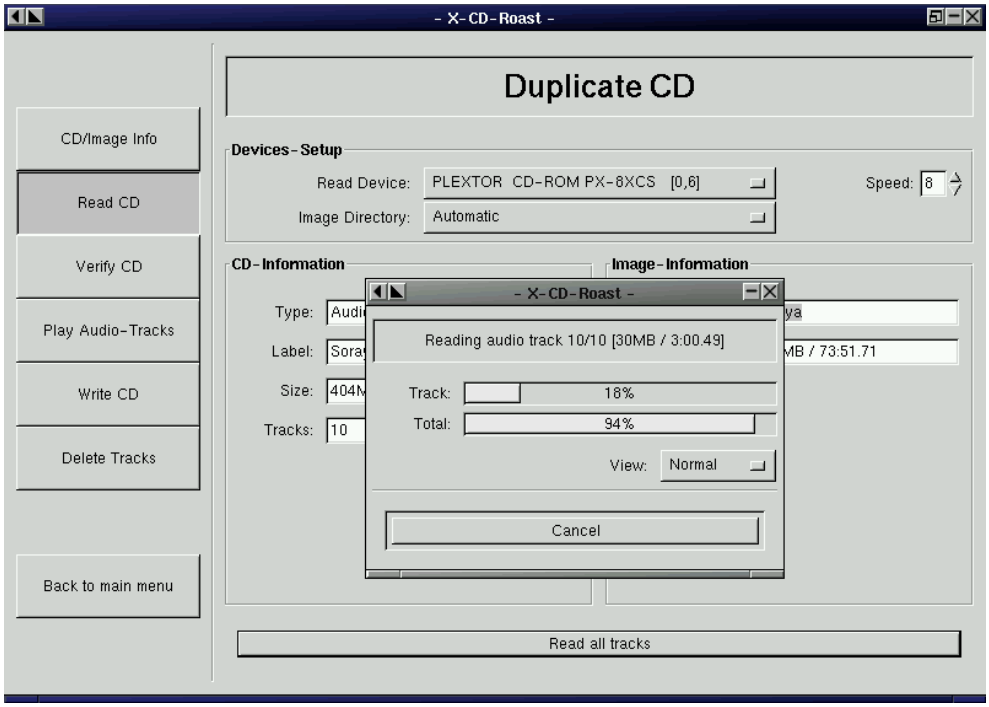

Figure 11.3: The Xcdroast GUI.

11.6.8.4 Client/Server Pair

A client/server pair is like a driver/engine pair, except that the engine part is a daemon running in background which is not expected to be run interactively, and does not have its own user interface. Usually, the daemon is designed to mediate access to some sort of shared resource—a database, or a transaction stream, or specialized shared hardware such as a sound device. Another reason for such a daemon may be to avoid performing expensive startup actions each time the program is invoked.

Yesterday's paradigmatic example was the *ftp*(1)/*ftpd*(1) pair that implements FTP, the File Transfer Protocol; or perhaps two instances of *sendmail*(1), sender in foreground and listener in background, passing Internet email. Today's would have to be any browser/webserver pair.

However, this pattern is not limited to communication programs; another important case is in databases, such as the *psql*(1)/*postmaster*(1) pair. In this one, *psql* serializes access to a shared database managed by the postgres daemon, passing it SQL requests and presenting data sent back as responses.

These examples illustrate an important property of such pairs, which is that the cleanliness of the protocol that serializes communication between them is all-important. If it is well-defined and described by an open standard, it can become a tremendous opportunity for leverage by insulating client programs from the details of how the server's resource is managed, and allowing clients and servers to evolve semi-independently. All separated-engine-and-interface programs potentially get this kind of benefit from clean separation of function, but in the client/server case the payoffs for getting it right tend to be particularly high exactly because managing shared resources is intrinsically difficult.

Message queues and pairs of named pipes can be and have been used for front-end/back-end communication, but the benefits of being able to run the server on a different machine from the client are so great that nowadays almost all modern client-server pairs use TCP/IP sockets.

11.6.9 The CLI Server Pattern

It's normal in the Unix world for server processes to be invoked by harness programs[4] such as *inetd*(8) in such a way that the server sees commands on standard input and ships responses to standard output; the harness program then takes care of ensuring that the server's stdin and stdout are connected to a specified TCP/IP service port. One benefit of this division of labor is that the harness program can act as a single security gatekeeper for all of the servers it launches.

One of the classic interface patterns is therefore a CLI server. This is a program which, when invoked in a foreground mode, has a simple CLI interface reading from standard input and writing to standard output. When backgrounded, the server detects this and connects its standard input and standard output to a specified TCP/IP service port.

In some variants of this pattern, the server backgrounds itself by default, and has to be told with a command-line switch when it should stay in foreground. This is a detail; the essential point is that most of the code neither knows nor cares whether it is running in foreground or a TCP/IP harness.

POP, IMAP, SMTP, and HTTP servers normally obey this pattern. It can be combined with any of the server/client patterns described earlier in this chapter. An HTTP server can also act as a harness program; the CGI scripts that supply most live content on the Web run in a special environment provided by the server where they can take

4. A harness program is a wrapper whose job it is to make some special sort of resource available to the program(s) it calls. The term is most often used for test harnesses, which make available test loads and (often) examples of correct output for the actual output to be checked against.

input (form arguments) from standard input, and write the generated HTML that is their result to standard output.

Though this pattern is quite traditional, the term "CLI server" is my invention.

11.6.10 Language-Based Interface Patterns

In Chapter 8 we examined domain-specific minilanguages as a means of pushing program specification up a level, gaining flexibility, and minimizing bugs. These virtues make the language-based CLI an important style of Unix interface—one exemplified by the Unix shell itself.

The strengths of this pattern are well illustrated by the case study earlier in the chapter comparing *dc*(1)/*bc*(1) with *xcalc*(1). The advantages that we observed earlier (the gain in expressiveness and scriptability) are typical of minilanguages; they generalize to other situations in which you routinely have to sequence complex operations in a specialized problem domain. Often, unlike the calculator case, minilanguages also have a clear advantage in concision.

One of the most potent Unix design patterns is the combination of a GUI front end with a CLI minilanguage back end. Well-designed examples of this type are necessarily rather complex, but often a great deal simpler and more flexible than the amount of ad-hoc code that would be necessary to cover even a fraction of what the minilanguage can do.

This general pattern is not, of course, unique to Unix. Modern database suites everywhere normally consist of one or more GUI front ends and report generators, all of which talk to a common back-end using a query language such as SQL. But this pattern mainly evolved under Unix and is still much better understood and more widely applied there than elsewhere.

When the front and back ends of a system fulfilling this design pattern are combined in a single program, that program is often said to have an 'embedded scripting language'. In the Unix world, *Emacs* is one of the best-known exemplars of this pattern; refer to our discussion of it in Chapter 8 for some advantages.

The script-fu facility of GIMP is another good example. GIMP is a powerful open-source graphics editor. It has a GUI resembling that of Adobe Photoshop. Script-fu allows GIMP to be scripted using Scheme (a dialect of Lisp); scripting through Tcl, or Perl or Python is also available. Programs written in any of these languages can call GIMP internals through its plugin interface. The demonstration application for this facility is a Web page[5] which allows people to construct simple logos and

5. Script-Fu page (`http://www.xcf.berkeley.edu/~gimp/script-fu/script-fu.html`).

graphic buttons through a CGI interface that passes a generated Scheme program to an instance of GIMP, and returns a finished image.

11.7 Applying Unix Interface-Design Patterns

To facilitate scripting and pipelining (see Chapter 7) it is wise to choose the simplest interface pattern possible—that is, the pattern with the fewest channels to the environment and the least interactivity.

In many of the single-component patterns described above, it is emphasized that the pattern does not require user interaction after startup time. When the 'user' is often expected to be another program (and thus to lack the range and flexibility of a human brain) this is a very valuable feature, maximizing scriptability.

We've seen that different interface design patterns optimize for traits valuable in differing circumstances. In particular, there is a strong and inherent tension between the GUIs and design patterns appropriate for novice and non-technical end-users (on the one hand) and those which serve expert users and maximize scriptability (on the other).

One way around this dilemma is to make programs with modes that exhibit more than one pattern. An excellent example is the web browser *lynx*(1). It normally has a roguelike interface for interactive use, but can be called with a `-dump` option that makes it into a source, formatting a specified web page to text dumped on standard output.

Such dual-mode interfaces, however, are not normally attempted when the program has to have a true GUI. The reasons for this are partly historical, but mostly have to do with controlling global complexity. GUIs tend to require complex startup configurations and large volumes of specialized code; these features coexist uneasily with the simpler patterns. In the worst case, a dual-mode GUI/non-GUI program could require two separate command-interpreter loops, with all that implies in the way of code bloat and potential inconsistencies.

Thus, when "choose the simplest pattern" conflicts with a requirement to produce a GUI, the Unix way is to split the program in two, applying the 'separated engine and interface' design pattern.

In fact, by combining a theme from Chapter 7 with this idea, we can perhaps name a new design pattern emerging under Linux and other modern, open-source Unixes where GUIs are not merely a reluctant add-on but an active focus of lots of development effort.

11.7.1 The Polyvalent-Program Pattern

A polyvalent program has the following traits:

1. The program's application-domain logic lives in a library with a documented API, which can be linked to other programs. The program's interface logic to the rest of the world is a thin layer over the library. Or perhaps there are several layers with different UI styles, any of which the library can be linked to.

2. One UI mode is a cantrip, compiler-like or CLI pattern that executes its interactive commands in batch mode.

3. One UI mode is a GUI, either linked directly to the core library or acting as as a separate process driving the CLI interface.

4. One UI mode is a scripting interface using a modern general-purpose scripting language like Perl, Python, or Tcl.

5. Optional extra: One UI mode is a roguelike interface using *curses*(3).

Notably, the GIMP actually fulfills this pattern.

11.8 The Web Browser As Universal Front End

Separating your CLI back end from a GUI interface has become an even more attractive strategy since the transformation of computing by the World Wide Web in the mid-1990s. For a large class of applications, it make increasing sense not to write a custom GUI front end at all, but rather to press Web browsers into service in that role.

This approach has many advantages. The most obvious is that you don't have to write procedural GUI code—instead, you can describe the GUI you want in languages (HTML and JavaScript) that are specialized for it. This avoids a lot of expensive and complex single-purpose coding and often more than halves the total project effort. Another is that it makes your application instantly Internet-ready; the front end may be on the same host as the back end, or may be a thousand miles away. Yet another is that all the minor presentation details of the application (such as fonts and color) are no longer your back end's problem, and indeed can be customized by users to their own tastes through mechanisms like browser preferences and cascading style sheets. Finally, the uniform elements of the Web interface substantially ease the user's learning task.

There are disadvantages. The two most important are (a) the batch style of interaction that the Web enforces, and (b) the difficulties of managing persistent sessions using a stateless protocol. Though these are not exclusively Unix issues, we'll discuss

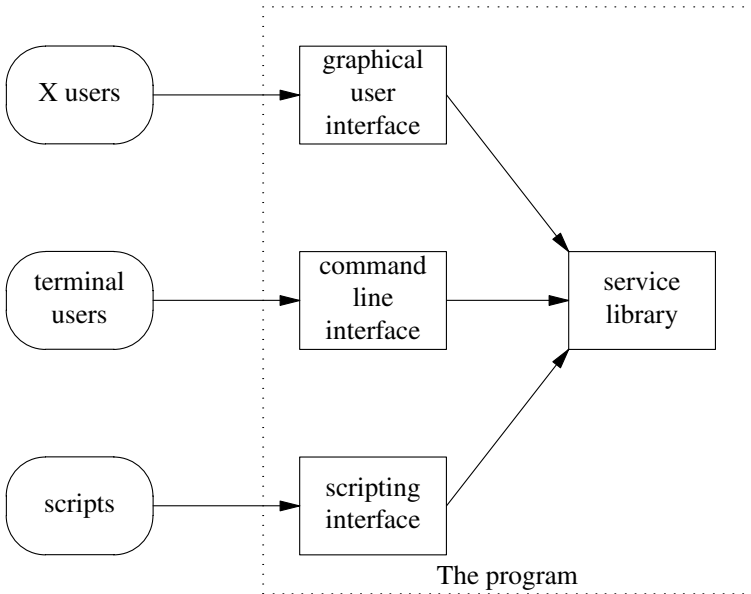

Figure 11.4: Caller/callee relationships in a polyvalent program.

them here—because it's very important to think clearly on the *design* level about when it's worthwhile to accept or work around these constraints.

CGI, the Common Gateway Interface through which a browser can invoke a program on the server host, does not support fine-grained interactivity well. Nor do the templating systems, application servers, and embedded server scripts that are gradually replacing it (in a mild abuse of language, we will use CGI for all of these in this section).

You can't do character-by-character or GUI-gesture-by-GUI-gesture I/O through a CGI gateway; instead, you have to fill out an HTML form and click a submit button that sends the form contents to a CGI script. The CGI script then runs and the server hands you back a page of HTML that it generated (which may itself be another CGI form).

This is essentially a batch style of interaction, not that far removed in concept from dropping punched cards in an input hopper and getting back a printout. It can be made more palatable by using JavaScript to interact with the user, batching up transactions into messages to be shipped to the server.

Java applets can open up their own character-stream connections back to the server to support smoother interactivivity. But Java has technical problems (it can only use a fixed display area on the page, and can't change the portion of the display outside

that rectangle) and much worse political ones (proprietary licensing from Sun has stalled Java deployment and made others reluctant to commit to it; you can't count on every user's browser to support it).

Both Java and JavaScript can run into browser incompatibilities, as well. Microsoft's resistance to implementing JDK 1.2 and Swing on Internet Explorer is a serious problem for Java applets, and differing Javascript version levels can also break your application (though Javascript bugs are easier to fix). Nevertheless, it is frequently less effort to work around these problems than it would be to write and deploy a custom front end. A problem harder to work around is that a growing number of sophisticated users routinely disable Java and even JavaScript in their browsers because of security problems and interface abuses.

As an independent issue, it is tricky to maintain session information across multiple CGI forms. The server doesn't keep any state about client sessions between CGI transactions, so you can't rely on it to connect later form submissions with earlier ones by the same user. There are two standard dodges around this: chained forms and browser cookies.

When you chain forms, you arrange for the CGI for the first form to generate a unique ID in an invisible field of the second form, and for the second and all subsequent forms to pass that ID to their successors. Cookies give a similar effect in a less direct way analogous to environment variables (see any of the hundreds of books on CGI design for details). In either case, your CGI has to use the ID as a session index (or cookies to cache state directly) and to handle multiplexing the sessions explicitly.

It is often possible to live with these restrictions. Many nontrivial applications can fit into a single form and response, evading both problems. Even when this isn't true and the application requires multiple forms, the complexity and cost savings from not having to build and distribute a specialized front end are so large that they can easily pay for the effort required to write CGIs smart enough to do their own session tracking.

The session management problem can be addressed with application servers like Zope or Enhydra which provide a session abstraction, and services like user authentication to programs embedded inside them. The drawback of these programs is identical to their advantage: the fact that they make it easier to keep per-user state on the server. That per-user state can be a problem; it eats resources, and it has to be timed out, because between transactions there is no way to know that the user is still on the other end of the wire.

As usual, the best advice is to choose the simplest pattern possible. Resist the temptation to do a heavyweight design relying on Java or an application server when simple CGIs and cookies will do the job.

One problem with the webserver-as-universal-front-end approach is that CGI back ends aren't readily separable from the browser environment, so it can be hard to script or automate transactions to the back end. The Unix answer is a three-tier architec-

ture—web forms calling CGIs which call commands. The automation interface is the commands.

The way that browsers decouple front and back ends has larger implications. On the Web, locking in consumers to closed, proprietary protocols and APIs has become more difficult and less attractive as this trend has advanced. The economics of software development are therefore tilting toward HTML, XML, and other open, text-based Internet standards. This trend synergizes in interesting ways with the evolution of the open-source development model, which we'll survey in Chapter 19. In the world that the Web is creating, Unix's design tradition—including the approaches to interface design we've surveyed in this chapter—looks more at home than ever before.

11.9 Silence Is Golden

We cannot leave the subject of interactive user interfaces without exploring one of the oldest and most persistent design tropes of Unix, the Rule of Silence. We observed in Chapter 1 that well-designed Unix programs with nothing interesting or surprising to say should shut up, and suggested there are good reasons for this that have long outlasted the slow teletypes on which Unix was born.

Here's one: Programs that babble don't tend to play well with other programs. If your CLI program emits status messages to standard output, then programs that try to interpret that output will be put to the trouble of interpreting or discarding those messages (even if nothing went wrong!). Better to send only real errors to standard error and not to emit unrequested data at all.

Here's another: The user's vertical screen space is precious. Every line of junk your program emits is one less line of context still available on the user's display.

Here's a third: Junk messages are a careless waste of the human user's bandwidth. They're one more source of distracting motion on a screen display that may be mediating for more important foreground tasks, such as communication with other humans.

Go ahead and give your GUIs progress bars for long operations. That's good style—it helps the user time-share his brain efficiently by cuing him that he can go off and read mail or do other things while waiting for completion. But don't clutter GUI interfaces with confirmation popups except when you have to guard operations that might lose or trash data—and even then, hide them when the parent window is minimized, and bury them unless the parent window has focus.[6] Your job as an interface designer is to assist the user, not to gratuitously get in his face.

6. If your windowing system supports translucent popups that intrude less between the user and the application, *use them*.

In general, it's bad style to tell the user things he already knows ("Program <foo> is starting up...", or "Program <foo> is exiting" are two classic offenders). Your interface design as a whole should obey the Rule of Least Surprise, but the content of messages should obey a Rule of *Most* Surprise—be chatty only about things that deviate from what's normally expected.

This rule has even greater force for confirmation prompts. Constantly asking for confirmation where the answer is almost always "yes" conditions the user to press "yes" without thinking about it, a habit that can have very unfortunate consequences. Programs should request confirmation only when there is good reason to suspect that the answer might be "no no no!". A confirmation request that is not a surprise is a strong hint of bad design. Any confirmation prompts at all may be a sign that what your interface really needs is an undo command.

If you want chatty progress messages for debugging purposes, disable them by default with a verbosity switch. Before releasing for production, relegate as many of the normal messages as possible to being displayed only when the verbosity switch is on.