13

Complexity: As Simple As Possible, but No Simpler

> Everything should be made as simple as possible, but no simpler.
>
> —Albert Einstein

At the end of Chapter 1, we summarized the Unix philosophy as "Keep It Simple, Stupid!". Throughout the Design section, one of the continuing themes has been the importance of keeping designs and implementations as simple as possible. But what *is* "as simple as possible"? How do you tell?

We've held off on addressing this question until now because understanding simplicity is complicated. It needs some of the ideas we developed earlier in the Design section, especially in Chapter 4 and Chapter 11, as background.

The large questions in this chapter are central preoccupations of the Unix tradition, some of them motivating holy wars that have simmered for decades. This chapter starts from established Unix practice and vocabulary, then goes a bit further beyond it than we do in the rest of the book. We don't try to develop simple answers to these questions, because there aren't any—but we can hope that you will walk away with better conceptual tools for developing your own answers.

13.1 Speaking of Complexity

As with previous issues about modularity and interface design, Unix programmers react to a set of distinctions they have often learned from experience without knowing how to articulate. Therefore we'll need to start by developing some terminology.

We will start by defining what software complexity is. We will make some horizontal distinctions between different flavors of complexity, which sometimes have to be traded off against each other. We will finish by making some even more important vertical distinctions, between the kinds of complexity we must live with and the kinds we have the option to eliminate.

13.1.1 The Three Sources of Complexity

Questions about simplicity, complexity, and the right size of software arouse a lot of passion in the Unix world. Unix programmers have learned a view of the world in which simplicity is beauty is elegance is good, and in which complexity is ugliness is grotesquery is evil.

Underlying the Unix programmer's passion for simplicity is a pragmatic fact: complexity costs. Complex software is harder to think about, harder to test, harder to debug, and harder to maintain—and above all, harder to learn and use. The costs of complexity, rough as they are during development, bite hardest after deployment. Complexity creates places for bugs to nest, from which they will emerge to trouble the world through the entire lifetime of their software.

All kinds of pressures tend to drag programmers into a swamp of complexity nevertheless. We've examined a rogue's gallery of these in earlier chapters; feature creep and premature optimization are the two most notorious. Traditionally, Unix programmers push back against these tendencies by proclaiming with religious fervor a rhetoric that condemns all complexity as bad.

So what exactly do we mean by 'complexity'? This point is worth pinning down, because it varies by observer.

Unix programmers (like other programmers) tend to focus on *implementation complexity*—basically, the degree of difficulty a programmer will experience in attempting to understand a program so he or she can mentally model or debug it.

Customers and users, on the other hand, tend to see complexity in terms of the program's *interface complexity*. In Chapter 11 we discussed the quality of ease and its inverse, mnemonic load. To a user, complexity correlates closely with mnemonic load. Poor expressiveness and concision can matter too, if a weak interface forces the user to perform lots of error-prone or merely tedious low-level operations rather than a few high-level ones.

Driven by both of these is a third measure that is much simpler: the total number of lines of code in the system, its *codebase size*. In terms of life-cycle costs, this is usually the most important measure. The reasons go back to perhaps the most important empirical result in software engineering, one we've cited before: the defect density of code, bugs per hundred lines, tends to be a constant independent of implementation language. More lines of code means more bugs, and debugging is the most expensive and time-consuming part of development.

Codebase size, interface complexity and implementation complexity may all rise together. That is the usual result of feature creep, and why programmers especially dread it. Premature optimization doesn't tend to raise interface complexity, but it has bad effects (often severely bad) on implementation complexity and codebase size. But those sorts of arguments against complexity are relatively easy to win; the difficult ones begin when these three measures have to be traded off against each other.

We've already mentioned one situation in which two measures vary in opposite directions: a user interface that has been designed primarily to preserve implementation simplicity, or keep codebase size down, may simply dump low-level tasks on the user. (A crude example of this, barely imaginable to a Unix programmer but all too common elsewhere, might be an editor that lacked a global-replace feature.) Though this sort of design failure is all too common, it does not traditionally have a name. We'll call it a *manularity trap*.

Pressure to keep the codebase size down by using extremely dense and complicated implementation techniques can cause a cascade of implementation complexity in the system, leading to an un-debuggable mess. This used to happen frequently when fitting programs onto very small systems demanded assembler programming or tricks like self-modifying code; nowadays it is uncommon except in embedded systems, and rapidly becoming rare even there. This kind of design failure doesn't have a traditional name, but one might call it a *blivet trap*, after an old Army term for the results of attempting to stuff ten pounds of horse manure into a five-pound bag.

The blivet trap won't appear in our case studies, but we've defined it for contrast with its opposite. It can happen that the designers of a project are so wary of implementation complexity that they reject a complex but unified way to solve a whole class of problems in favor of lots of duplicative, ad-hoc code that solves each individual one in turn. The result is bloat in the size of the codebase, and maintainability problems more severe than if the unified method had been accepted. For example, a Web project that really needs a centralized relational database behind its pages might instead spawn several different keyed data files containing information that has to be re-integrated at page generation time. This sort of failure is all too common. It doesn't have a traditional name; we'll call it an *adhocity trap*.

These are the three faces of complexity, and some of the traps designers fall into in attempts to avoid them.[1] We'll see more examples when we get to the case studies later in the chapter.

13.1.2 Tradeoffs between Interface and Implementation Complexity

One of the most perceptive observations ever made about the Unix tradition by someone standing outside it was contained in Richard Gabriel's paper called *Lisp: Good News, Bad News, and How to Win Big* [Gabriel]. Gabriel is a long-time leader of the Lisp community, and the paper was primarily an argument for a particular style of Lisp design, but the author himself acknowledges that it is now remembered primarily for the section called 'The Rise of *Worse Is Better*'.

The paper argued that Unix and C have the characteristics of viruses, and that in the evolutionary struggle among software designs traits like implementation simplicity and portability which lead to rapid propagation (infectiousness) are more effective than correctness and completeness of the design. Gabriel came so close to anticipating the 'many-eyeballs' effect on open-source software that the open-source community retrospectively adopted him as one of its theorists after 1997.

Less remembered is that the Gabriel's central argument was about a very specific tradeoff between implementation and interface complexity, one which rather exactly fits the categories we have examined in this chapter. Gabriel contrasts an 'MIT' philosophy most valuing interface simplicity with a 'New Jersey' philosophy most valuing implementation simplicity. He then proposes that although the MIT philosophy leads to software that is better in the abstract, the (worse) New Jersey model has better propagation characteristics. Over time, people pay more attention to software written in the New Jersey style, so it improves faster. Worse becomes better.

In fact, the MIT and New Jersey philosophies have analogs as conflicting tendencies within the Unix design tradition itself. One strain of Unix thinking emphasizes small sharp tools, starting designs from zero, and interfaces that are simple and consistent. This point of view has been most famously championed by Doug McIlroy. Another strain emphasizes doing simple implementations that work, and that ship quickly, even if the methods are brute-force and some edge cases have to be punted. Ken Thompson's code and his maxims about programming have often seemed to lean in this direction.

The tension between these approaches arises precisely because one can sometimes get a simpler interface if one is willing to pay implementation complexity for it, or

1. The terms we have invented for these design traps, unlikely as they may sound, come from established hacker jargon described in [Raymond96].

vice versa. Gabriel's original example, about how system calls that do long operations handle interrupts they cannot hold or mask, is still one of the best. Under the MIT philosophy, the right thing to do would be to back out of the system call and automatically resume it once the interrupt has been handled; this is harder to implement but leads to a simpler interface. Under the New Jersey philosophy, the system call would return an error indicating that it has been interrupted and the user must re-execute; this can be implemented far more simply, but leads to a programming interface that is more difficult to use.

Both approaches have been tried. Old Unix hands will instantly think of System-V-style vs. BSD-style handling of software signals; the latter follows the MIT philosophy, while the former hails from New Jersey. Underlying the choice between them is a pressing question that has nothing directly to do with the software's infectiousness: if your goal is to hold down total global complexity, where are you most willing to pay to do that? Where *should* you be most willing to pay?

One epochal example not mentioned in Gabriel's paper is from distributed hypertext systems. Early distributed-hypertext projects such as NLS and Xanadu were severely constrained by the MIT-philosophy assumption that dangling links were an unacceptable breakdown in the user interface; this constrained the systems to either browsing only a controlled, closed set of documents (such as on a single CD-ROM) or implementing various increasingly elaborate replication, caching, and indexing methods in an attempt to prevent documents from randomly disappearing. Tim Berners-Lee cut through this Gordian knot by punting the problem in classic New Jersey style. The simplicity of implementation he bought by allowing "404: Not Found" as a response was what made the World Wide Web lightweight enough to propagate and succeed.

Gabriel himself, while sticking with the observation that 'worse' is more infectious and tends to win in the end, has publicly changed his mind several times about the underlying complexity-related question of whether or not this is actually a good thing. His uncertainty mirrors a lot of ongoing design debates within the Unix community.

We cannot offer a one-size-fits-all answer. As with most of the large questions in this chapter, good taste and engineering judgement will demand different answers in different situations. The important thing is to develop the habit of thinking carefully about this issue on each and every one of your designs. As we have observed before in discussing software modularity, complexity is a cost you must budget very carefully.

13.1.3 Essential, Optional, and Accidental Complexity

In an ideal world, Unix programmers would craft only small, perfect gems of software, each minimal, each elegant, each perfect. But one of the unfortunate things about re-

ality is that it often poses complex problems that demand complex solutions. You can't control a jetliner with an elegant ten-line procedure. There are too many pieces of equipment, too many channels and interfaces, too many different processors—too many different subsystems defined by independently operating human beings who often don't agree even on fundamental conventions. Even if you are successful at making all the individual software parts of an avionics system elegant, integration is likely to produce a large, complex, and grubby body of code with (one hopes) the single virtue that it will actually *work*.

Jetliners have *essential complexity*. There is a rather sharp point past which it's not possible to trade away features for simplicity, because the plane has to stay in the air. Because of that very fact, avionics control systems do not tend to spawn religious wars about complexity—and Unix programmers tend to stay away from them.

Jetliners are certainly not immune from system failures due to overcomplexity. But the design issues are easier to discern and think about in software for which the requirements are more flexible, in which it is easy to trade off between anticipated features and complexity. (Here, and in the rest of this chapter, we will use 'feature' in a very general sense that includes things like performance gains or overall degree of interface polish.)

To sharpen our vision, we need to begin by noticing a difference between *accidental complexity* and *optional complexity*.[2] Accidental complexity happens because someone didn't find the simplest way to implement a specified set of features. Accidental complexity can be eliminated by good design, or good redesign. Optional complexity, on the other hand, is tied to some desirable feature. Optional complexity can be eliminated only by changing the project's objectives.

When we fail to distinguish between optional and accidental complexity, design debates become seriously confused. Questions about what a project's objectives are get confused with questions about the aesthetics of simplicity, and whether people have been sufficiently clever.

13.1.4 Mapping Complexity

So far, we've developed two different scales for thinking about complexity. These scales are actually orthogonal to each other. Figure 13.1 may help clarify the relationships. Each of the nine boxes of the figure lists a common source of a particular kind of complexity.

2. The distinction between accidental and optional complexity means that the categories we're discussing here are *not* the same as essence and accident in Fred Brooks's essay *No Silver Bullet* [Brooks], but they have common ancestry in philosophy.

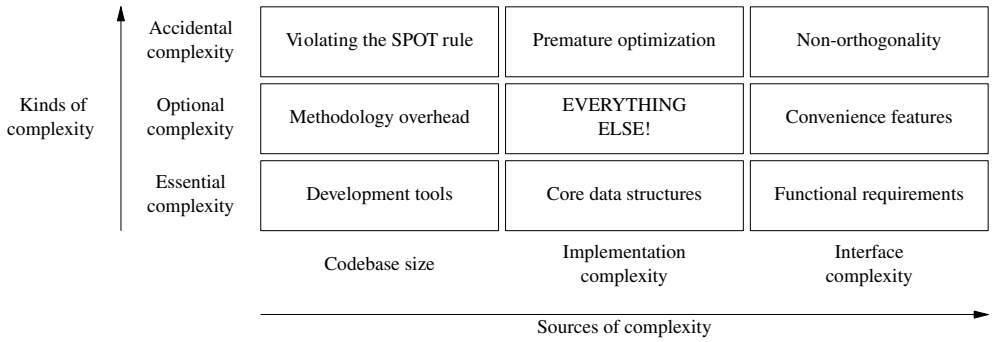

Figure 13.1: Sources and kinds of complexity.

We've touched on some of these varieties of complexity earlier in this book, especially the accidental ones. In Chapter 4 we saw that accidental interface complexity often comes from non-orthogonality in the interface design—that is, failing to carefully factor the interface operations so that each does exactly one thing. Accidental code complexity (making code more complicated than it needs to be to get the job done) often results from premature optimization. Accidental codebase bloat often results from violating the SPOT rule, duplicating code or organizing it poorly so that opportunities for reuse aren't recognized.

Essential interface complexity usually can't be cut without trimming the basic functional requirements for the software (a theme we'll develop further in this chapter's case studies). Essential codebase size is related to choice of development tools because, if the feature list is held constant, the most important factor in codebase size is probably the choice of implementation language (as we implied in Chapter 8).

Sources of optional complexity are the most difficult to make useful generalizations about, because they so often depend on delicate judgments about which features it is worth paying the complexity cost for. Optional interface complexity often comes from adding convenience features that make life easier for users but aren't essential to the function of the program. Optional increases in codebase size (supposing the user-visible features and the algorithms used are held constant) can often come from various sorts of practices intended to make it more maintainable—adding mode comments, using long variable names, and so forth. Optional implementation complexity tends to be driven by *everything* that touches a project.

The sources of complexity have to be grappled with in different ways. Codebase size can be attacked with better tools. Implementation complexity can be addressed with better choice of algorithms. Interface complexity has to be addressed with better

interaction design, a skill involving considerations of ergonomics and user psychology. This skill is less common (and possibly more difficult) than writing code.

Attacking the kinds of complexity, on the other hand, has to be done more with insight than with methods. You cut accidental complexity by noticing that there is a simpler way to do things. You cut optional complexity by making context-dependent judgments about what features are worthwhile. You can only cut essential complexity by having an epiphany, fundamentally redefining the problem you are addressing.

13.1.5 When Simplicity Is Not Enough

The failure mode that goes with the Unix tradition's insistence on simplicity is that Unix programmers often talk (and sometimes even behave) as though all optional complexity is accidental. More than this, there is a strong bias in the Unix tradition toward removing features rather than accepting optional complexity.

The case for this attitude is easy to make (indeed, we spend much of this book making it). Clean minimalism makes us feel virtuous on many levels, and designing for it is a valuable counter to the natural tendency of software systems to develop ever-more-elaborate encrustations of ill-considered features. But computing resources and human thinking time, like wealth, find their justification not in being hoarded but in being spent. As with other forms of asceticism, one has to ask when design minimalism stops being a valuable form of self-discipline and starts being a mere hair shirt—a way to indulge those feelings of virtue at the expense of actually *using* that wealth to get work done.

This is a perilous question, all too easily turned into an argument for abandoning good design discipline altogether. Unix old hands often shy away from it, fearing that failing to hold the hardest possible line against complexity and bloat will lead us inexorably to damnation. But it's also a *necessary* question. We'll tackle it directly when analyzing this chapter's case studies.

13.2 A Tale of Five Editors

Now we're going to use five different Unix editors as case studies. It will be helpful to bear in mind a set of benchmark tasks as we examine these designs:

- *Plain-text editing.* Manipulating plain ASCII (or, in this internationalized age, perhaps Unicode) files with no structure known to the editor above byte level, or perhaps line level.

- *Rich-text editing.* Editing of text with attributes; these might include font changes, color, or other sorts of properties of text spans (such as being a hyperlink). Editors

that can do this have to be able to translate between some presentation of the attributes in the user interface and some on-disk representation of the data (such as HTML, XML, or other rich-text formats.)

- *Syntax awareness.* An editor that is syntax-aware knows that input events have a grammar, and does things like automatically changing the indent level when it recognizes the beginning or end of a block scope in a programming language. Editors that are syntax-aware also commonly highlight syntax with colors or distinguished fonts.

- *Output parsing* of batch command output. The commonest case of this in the Unix world is running a C compilation from inside the editor, trapping the error messages, and then being able to step through the error locations without leaving the editor.

- *Interaction* with helper subprocesses that persist and maintain state between editor commands. This capability, when present, has powerful consequences:

    - It's possible to drive a version-control system from inside the editor, performing file checkins and checkouts without dropping out to a shell window or separate utility.

    - It's possible to front-end a symbolic debugger inside the editor, such that (for example) when the run stops on a breakpoint the appropriate file and line is automatically visited.

    - It's possible to edit remote files within the editor, by having it recognize when a filename refers to another host (recognizing some syntax like `/user@host:/path/to-file`). Provided you have the right access, such an editor can automatically run a utility like *scp*(1) or *ftp*(1) to fetch a local copy, then automatically copy the edited version back to the remote location at file-save time.

All our case studies can edit plain text. (The reader should not take this capability for granted—there are many things called editors, such as 'word processors' that are too specialized to do this!) We begin seeing variable degrees of optional complexity in how they handle the more complex tasks.

13.2.1 *ed*

ed(1) is the truly Unix-minimalist way of plain-text editing. It dates from the days of teletypes.[3] It has a simple, austere CLI, and there is no screen display. In the following listing, computer output is *emphasized*.

```
ed sample.txt
sample.txt: No such file or directory
# This is a comment line, not a command.
# The message above warns that the sample.txt file is newly created.
a
the quick brown fox
jumped over the lazy dog
.
# That was an append command, which added text to the file.
# The dot on a line by itself terminated the append.
1s/f[a-z]x/dragon/
# On line 1, replace the first substring matching an f followed by a
# lowercase alphabetic followed by x with 'dragon'.  The
# substitute command accepts basic regular expressions.
1,$p
the quick brown dragon
jumped over the lazy dog
# Print all lines from 1 to the last.
w
51
# That wrote the file to disk. The 'q' command ends the
# editing session.
q
```

Unbelievable as it may seem to a modern reader, most of Unix's original code was written with this editor. The reader with DOS experience may recognize here the original on which *EDLIN* was (crudely) modeled.

If one defines the job of an editor simply as enabling the user to create and modify plain text files, *ed*(1) is entirely sufficient for the job. Importantly to the Unix view of design correctness, it does nothing else. Many old-school Unix programmers half-seriously maintain that all editors with more features than ed has are simply bloated—and a few still who seriously believe this.

Appropriately, *ed* was Ken Thompson's deliberate simplification of the earlier *qed*[RitchieQED] editor—which was very similar (and the first editor to use regular expressions in the characteristic Unix way) but had multiple-buffer capability that Ken deliberately discarded. He judged it not worth the additional complexity.

3. Younger readers may not be aware that terminals used to print. On paper. Very slowly.

A notable characteristic of *ed*(1) and all its descendants is the object-operation format of its commands (the session example shows an explicit range on the 'p' command). There is a relatively powerful syntax for specifying line ranges, either numerically, or by regular-expression pattern match, or by special shorthands for the current and last line. Most editor operations can be applied to any range. This is a good example of orthogonality.

Nowadays, *ed*(1) is primarily used as a program-driven editing tool in scripts—a role to which editors with more elaborate modes of interactivity are unsuited. There is a close variant called *ex*(1) which adds a few useful interactivity features such as command prompts; it is occasionally useful in rare cases when editing must be done over a slow serial line, or in certain unusual crash-recovery situations where the library support needed to run other editors is not accessible. For these reasons, every Unix includes an *ed* implementation and most include *ex* as well.

The *sed*(1) stream editor mentioned in Chapter 9 is also closely related to ed; many of the basic commands are the same, though designed to be invoked through command-line switches rather than from standard input.

Almost all Unix programmers have strayed from the path of austerity and minimalist virtue enough to normally use editors that at least present a roguelike, screen-oriented interface. However, the fact that the religion of ed persists[4] says a great deal that is worth noting about the Unix mindset.

13.2.2 vi

The original *vi*(1) editor was the first attempt to bolt a visual, roguelike interface onto the command set of *ed*(1). Like ed, its commands are generally single keystrokes, and it is particularly well suited to use by touch-typists.

The original vi didn't have mouse support, editing menus, macros, assignable key bindings, or any form of user customization. In line with the religion of ed, vi's partisans considered the lack of these features a virtue. On this view, one of vi's most important virtues is that you can start editing immediately on a new Unix system without having to carry along your customizations or worrying that the default command bindings will be dangerously different from what you're used to.

One characteristic of vi that beginners tend to find frustrating is a result of its terse single-keystroke commands. It has a *moded* interface—you are either in command mode or in text-insertion mode. In text-insertion mode, the only commands that work

4. The religion of ed is exemplified by a famous Usenet posting which the reader may be able to find with a Web search for "Ed is the standard editor". While it is clearly intended as parody, it is by no means clear that the author was entirely joking. Most Unix hackers would read it as an example of "Ha ha, only serious."

are the ESC key for mode exit and (on newer versions) the cursor-movement keys. In command mode, typing text will be interpreted as commands and do odd (and probably destructive) things to your content.

On the other hand, one property of the command set that vi fans particularly tout is the object-operation format it inherited from ed. Most of the extended commands also operate in a natural way on any line range.

Over the years, vi has bulked up considerably. Modern versions add mouse support, editing menus, unlimited undo (the original vi could only undo the last command), multiple files in separate buffers, and customization with a run-control file. However, the use of run-control files is still unusual, and in contrast to Emacs, the use of embedded general-purpose scripting has never caught on. Instead, vi implementations have grown individual capabilities to do things, like syntax awareness of C code and output parsing of C compiler error messages, by adding C code to vi itself. Subprocess interaction is not supported.

13.2.3 *Sam*

The *Sam* editor[5] was written by Rob Pike at Bell Labs in the mid-1980s. Sam was designed for the Plan 9 operating system, which we'll survey in Chapter 20. While the Sam editor is not widely known outside the Labs, it's favored by many of the original Unix developers who went on to work on Plan 9, including Ken Thompson himself.

Sam is a fairly straightforward descendant of ed, remaining much closer to its parent than vi. Sam incorporates only two new concepts: a curses-style text display and text selection with the mouse.

Each Sam session has exactly one command window, and one or more text windows. Text windows edit text, and command windows accept ed-style editing commands. The mouse is used to move between windows, and to select text regions within text windows. This is a clean, orthogonal, modeless design that discards most of the interface complexity of vi.

Most commands operate by default on a select region that can be painted with a mouse drag operation. The select region for a command can also be set by specifying a line range in the fashion of ed, but Sam gains considerable power from the fact that the user can select at finer granularity than a line range. Because the mouse is available to do selections and rapidly change focus between buffers (including the command buffer), Sam needs no equivalent of the default (command) mode of vi. The hundreds of extended vi commands are unnecessary and, therefore, omitted. Overall, Sam adds

5. `http://plan9.bell-labs.com/sys/doc/sam/sam.html`

only about a dozen commands to the seventeen or so in the ed set, for a total of about thirty.

Four of the new commands in Sam join two inherited from *ed*(1) and *vi*(1), as ways to apply regular expressions to the task of selecting files and file regions to operate on. These provide limited but effective loop and conditional facilities to the command language. There is, however, no way to name or parameterize command-language procedures. Nor can the language do interactive control of a subprocess.

An interesting feature of Sam is that it's split into two parts. separating a back end that manipulates files and does searches from a front end that handles the screen interface. This instance of the "separated engine and interface" chapter has the immediate practical benefit that, though the program has a GUI, it can run easily over a low-bandwidth connection to edit files on a remote server. Also, the front and back ends can be retargeted relatively easily.

Sam, like recent versions of vi, has infinite undo. By design, it supports neither rich-text editing, nor output parsing, nor subprocess interaction.

13.2.4 Emacs

Emacs is undoubtedly the most powerful programmer's editor in existence. It's a big, feature-laden program with a great deal of flexibility and customizability. As we observed in the Chapter 14 section on Emacs Lisp, Emacs has an entire programming language inside it that can be used to write arbitrarily powerful editor functions.

Unlike vi, Emacs doesn't have interface modes; instead, commands are normally control characters or prefixed with an ESC. However, in Emacs it is possible to bind just about any key sequence to any command, and commands can be stock or customized Lisp programs.

Emacs can edit multiple files, each in a separate buffer, and supports moving text among the buffers. Versions running under X have native mouse support.

The Lisp programs bound to Emacs keystrokes can perform arbitrary text transformations on a buffer. This capability is heavily used, among other things to define syntax-aware and rich-text editing modes for dozens of different languages and markup formats (beginning with support and color highlighting of C code as in vi, but going way beyond that). Each mode is simply a library file of Lisp code that is loaded on demand.

Emacs Lisp programs can also interactively control arbitrary subprocesses. Some notable consequences of this capability were listed earlier, including the ability to serve as a front end for version-control systems, debuggers, and the like.

The designers of Emacs[6] built a programmable editor that could have task-related intelligence customized into it for hundreds of different specialized editing jobs. They then gave it the ability to drive other tools. As a result, Emacs supports dealing with all things textual in one shared context—files, mail, news, debugger symbols. It can serve as a customizable front end to any command with an interactive textual interface.

It is a common joke, both among fans and detractors of Emacs, to describe it as an operating system masquerading as an editor. That overstates the case, but Emacs certainly does fulfill the role occupied by integrated development environments (IDEs) under non-Unix operating systems (a theme to which we shall return in Chapter 15).

This power comes at a price in complexity. To use a customized Emacs you have to carry around the Lisp files that define your personal Emacs preferences. Learning how to customize Emacs is an entire art in itself. Emacs is correspondingly harder to learn than vi.

13.2.5 Wily

The *wily* editor[7] is a clone of the Plan 9 editor *acme*.[8] It shares some facilities with Sam, but is intended to provide a fundamentally different user experience. Although Wily probably sees the least widespread use of any of these editors, it is interesting because it illustrates a different and arguably more Unixy way of implementing an Emacs-like programmable editor.

Wily could be described as a minimalist IDE, an implementation of Emacs-style extensibility without the decades of accompanying cruft. In Wily, even global search and replace, that *sine qua non* of Unix editors, is supplied by an external program. The built-in commands relate almost exclusively to windowing operations. Wily is designed from the ground up to use the mouse as much, and as well, as possible.

Wily attempts to replace not only conventional editors but conventional terminal windows such as *xterm*(1) as well. In Wily, any piece of text within the main window (which contains multiple non-overlapping Wily windows) can be an action or a search expression. The left mouse button is used to select text, the middle button to execute

6. The designers of Emacs were Richard M. Stallman, Bernie Greenberg, and Richard M. Stallman. The original Emacs was Stallman's invention, the first version with an embedded Lisp was Greenberg's, and the now-definitive version is Stallman's derived from Greenberg's. No complete account of the design history has been written in 2003, but Greenberg's *Multics Emacs: The History, Design, and Implementation* is illuminating and readily discoverable via keyword search on the Web.

7. `http://www.cs.yorku.ca/~oz/wily`

8. `http://plan9.bell-labs.com/sys/doc/acme/acme.html`

text as a command (either built-in or external), and the right button to search either Wily's buffers or the file system for text. No permanent or popup menus are required.

In Wily, the keyboard is used *only* to enter text. Shortcuts are achieved not by special use of the keyboard, but by holding down more than one mouse button at the same time. These shortcuts are always equivalent to using the middle button on some built-in command.

Wily can also be used as the front end for C, Python, or Perl programs, reporting to them whenever a window is changed or an execute or search command is performed with the mouse. These plugins function analogously to Emacs modes, but don't run in the same address space with Wily; instead, they communicate with it via a very simple set of remote procedure calls. Wily comes packaged with an *xterm* analog and a mail tool which uses it as the editing front end.

Because Wily depends on the mouse so heavily, it cannot be used on a character-cell-only console display; nor can it be used over a remote link without X forwarding. As an editor, Wily is designed for editing plain text; it has only two fonts (one proportional and one fixed-width) and has no mechanism that could support rich-text editing or syntax awareness.

13.3 The Right Size for an Editor

Now let us examine our case studies using the complexity categories we developed at the beginning of this chapter.

13.3.1 Identifying the Complexity Problems

Every text editor has a certain amount of essential complexity. At minimum, it has to maintain an internal buffer copy of the file or files the user is editing. Functions to import and export file data are a minimum requirement (usually from and to disk, though the stream editor *sed*(1) is an interesting exception). Some way to modify the buffer must be supported, though we cannot specify what way without describing specific features that are optional. Our four examples show widely varying levels of optional and accidental complexity beyond this.

Of all of these, *ed*(1) has the least complexity. Almost the only non-orthogonal feature in its command set is the fact that many of its commands can take a 'p' or 'l' suffix to print or list command results. Even after three decades of feature additions there are fewer than thirty editing commands, and the normal working set for most users will be less than a dozen. There is not much in the way of optional complexity that could be removed here, and it's hard to identify any accidental complexity at all. The user interface of ed is strictly compact.

On the flip side, the ed interface is not really suitable for editing tasks even as basic as rapidly flipping through a text file. One has to limit one's objectives pretty sharply for ed to become an acceptable solution for interactive editing.

Suppose, then, that we add "support visual browsing and editing of multiple files" as an objective? Then Sam seems not very far from being the minimal ed extension that could achieve this. The fact that the designers did not change the semantics of the inherited ed commands is notable; they kept an existing, orthogonal set and added a relatively small set of capabilities that are themselves orthogonal.

One large increase in optional (implementation) complexity is Sam's infinite-undo capability. Another significant one is the new regular-expression-based loop and iteration facility in the command language. These, and the fact that the mouse can be used as a selection device, are about all that distinguish Sam from a hypothetical ed with a mouse-and-windows interface.

Without a thorough code audit it's difficult to be sure, but at the design level it's hard to identify any accidental complexity in Sam. The interface is at least semicompact and arguably strictly compact. This editor lives up to the very highest standards of Unix design—unsurprisingly, given its provenance.

By contrast, vi looks rather bloated and flabby. There are hundreds of commands, many of them duplicative. These are at best optional complexity, and perhaps accidental. At a guess, most users don't know more than 5% of the command set. With the example of Sam before us, it's fair to wonder why the interface complexity of vi is so high.

In Chapter 11 we described the effect of the absence of standard arrow keys on early roguelike programs; vi was one of these. When vi was built, its author knew that many of his users would need to be able to use the cursor motion keys traditional on Unix glass teletypes. This made a modal interface inevitable. Once the hjkl keys had mode-dependent meanings in an edit buffer, it was all too easy to fall into the habit of adding new commands in an ad-hoc way.

Sam, designed as it is to depend on a bitmapped display with both arrow keys and a mouse, can be much cleaner. And it is.

But the clutter of vi commands is a relatively superficial problem. It's interface complexity, yes, but of a kind most users can and do ignore (the interface is semicompact in the sense we developed in Chapter 4). The deeper problem is an adhocity trap. Over the years, vi has had progressively more and more special-purpose C code bolted onto it to perform tasks that Sam refuses to do and that Emacs would attack with Lisp code modules and subprocess control. The extensions are not, as in Emacs, libraries loaded as needed; users pay the overhead for the resulting code bloat all the time. As a result, the size difference between a modern vi and a modern Emacs is not nearly as great as one might expect; in mid-2003 on an Intel-architecture machine, it's 1500KB for GNU Emacs versus 900KB for vim. There is a whole lot of both optional and accidental complexity in that 900KB.

For vi partisans, not having an embedded scripting language—not beingEmacs—has become an identity issue, a central part of the shared myth that vi is a lightweight editor. While vi fans like to talk about filtering buffers with external programs and scripts to do what Emacs's embedded scripting does, the reality is that vi's "!" command cannot filter regions of an edit buffer selected at finer granularity than a range of lines (Sam and Wily, though they have no more subprocess management than vi does, can at least filter arbitrary text ranges, not just line ranges). All knowledge of file formats and syntaxes that vary at a finer granularity (and most do) has to be built in to C code if vi is going to have it available at all. There is thus little prospect that the codebase-size ratio between Emacs and vi will improve in favor of vi; indeed, it seems likely to get worse.

Emacs is sufficiently large, and has a sufficiently tangled history, to make separating its optional from its accidental complexity quite a challenge. We can at least begin by trying to separate the dispensable accidents of the Emacs design from its indispensable essentials.

Perhaps the most conspicuously dispensable part of the Emacs design is Emacs Lisp. It is essential to what Emacs does that it feature what we nowadays call an embedded scripting language, but Emacs would be little different in capability if that language had been Python or Java or Perl. At the time Emacs was designed in the 1970s, however, Lisp was about the only language that had the characteristics (including unlimited-extent types and garbage collection) to fit it to the job.

Much in the particulars of the way *emacs* handles event processing and drives a bitmapped display (including the support for internationalization) is accidental as well. The one great schism in its history (the GNU Emacs/XEmacs fork) was over these issues, and demonstrates that nothing in the rest of the design prefers or requires any one event model.

On the other hand, the ability to bind arbitrary event sequences to arbitrary built-in or user-defined functions is indispensable. The scripting language could change and the event model could change, but without the anything-goes polymorphism in the way they are connected, the Emacs design would be both unrecognizable and crippled. Extension modes would have to fight each other for ownership of a limited event set, and activating multiple cooperating modes on the same buffer would be difficult or impossible.

The huge library of extension modes shipped with Emacs is accidental as well. The *ability* to construct such extensions may be essential, but the particular set we have is a product of history and chance. They could all be different or replaced; the result would still, recognizably, be Emacs.

But subprocess interaction is indispensable. Without it, Emacs modes could not perform the expected IDE-like integration and front-ending of many different tools.

Experience with small editors that clone the default keybindings and appearance of Emacs without emulating its extensibility is instructive. There have been several

such clones, of which the best known are probably *MicroEmacs* and *pico*, but none have ever acquired significant mindshare.

Having identified accident and essence in the Emacs design helps us get a handle on which of its complexity is optional and which accidental. But, more importantly, they help us see past the superficial differences between Emacs and the previous three editors we have considered, to the really critical difference: the fact that the objectives of the Emacs design are far more broad. Emacs wants to be a unified interface to all tools that operate on text.

Wily makes an interesting contrast with Emacs. As with Sam, the amount of optional complexity is low; the Wily user interface can be succinctly but effectively described in a single page.

But this elegance comes with a price; it is not possible to bind functions to any keystrokes or input gestures other than a restricted set of mouse chords. Instead, every editor function other than very basic text insertion and deletion has to be implemented with a program outboard of the editor, either a standalone script or a specialized symbiont process listening to Wily input events. (The former technique relies on outboard program startups being fast enough not to produce noticeable interface lag, something which was emphatically not the case in either Emacs's natal environment or under the Unixes it was first ported to.)

Optional complexity which Emacs would implement in Lisp extension modes is instead distributed through specialized symbionts; each has to know the special Wily messaging interface. An advantage of this approach is that such symbionts can be written in any language the user chooses. In addition, the symbionts (because they run outboard) cannot adversely affect each other or the Wily core (which is not true of Emacs modes). A disadvantage is that Wily itself cannot directly do subprocess interaction with ordinary Unix tools at all.

In this and other ways, *wily's* distributed scripting is not as powerful as the embedded scripting of Emacs. The scope of Wily's objectives is correspondingly narrower; the authors disclaim any interest in syntax-aware editing, or rich text, for example, and neither Wily nor its Plan 9 ancestor *acme* can do these things.

This brings us to another, and sharper way of posing the central question of this chapter: When do large objectives justify a large program?

13.3.2 Compromise Doesn't Work

The comparison between Sam and vi suggests strongly that, at least where editors are concerned, attempts to compromise between the minimalism of ed and the all-singing-all-dancing comprehensiveness of Emacs don't work very well; vi attempts this, and ends up with neither virtue. Instead, it falls into an adhocity trap. Wily avoids the adhocity trap, but cannot match the power of Emacs and must demand a custom

process interface from each of its interactive symbionts in order to come anywhere close.

Evidently something about editors tends to push them in the direction of increasing complexity. In the case of vi, that something is not hard to identify; it's the desire for convenience. While ed may be theoretically adequate, very few people (other than perhaps Ken Thompson himself) would forgo screen-oriented editing to make a statement about software bloat.

More generally, programs that mediate between the user and the rest of the universe notoriously attract features. This includes not just editors but Web browsers, mail and newsgroup readers, and other communications programs. All tend to evolve in accordance with the Law of Software Envelopment, aka Zawinski's Law: "Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can."

Jamie Zawinski, inventor of the Law (and one of the principal authors of the Netscape and Mozilla Web browsers), maintains more generally that all really useful programs tend to turn into Swiss Army knives. The commercial success of large, integrated application suites outside the Unix world tends to confirm this, and directly challenges the Unix philosophy of minimalism.

To the extent Zawinski's Law is correct, it suggests that some things want to be small and some want to be large, but the middle ground is unstable. The superficial problems with vi can be put down to history, but the deeper ones trace back to the combination of steady pressure to add features with refusal to embed the scripting and subprocess-control features that vi partisans associate with excessive size. On a different level, accepting that there would be two modes in the interface (insertion versus character-motion) opened a can of worms—it became far too easy to add new commands without thinking about their complexity impact on the overall design.

The examples of Emacs and Wily further suggest *why* some things want to be large: so that several related tasks can share context. Editing and version control (or editing and mail, editing and symbolic debugging, etc.) are separate tasks from the point of view of the implementers—but users would often prefer to have one big environment that lets them point at pieces of text, rather than spend time and attention ping-ponging between several programs that each have to have the same filename or the contents of some cut buffer handed to them.

More generally, let's suppose we view the entire Unix environment as a single work of design by community. Then the religion of "small, sharp tools", the pressure to keep interface complexity and codebase size down, may lead right to a manularity trap—the user has to maintain all the shared context himself, because the tools won't do it for him.

Returning to the specific context of editors, Sam shows us that vi is the wrong thing. Wily is a valiant effort to avoid the vastness of Emacs that falls short because it can't be syntax-aware. But Wily, or some realization of the Emacs design ideas

cleaned up and stripped of historical baggage, might be the right thing. The value of optional complexity depends on the objectives you choose, and the ability to share context among all the text-oriented tools related to a task is valuable.

13.3.3 Is Emacs an Argument against the Unix Tradition?

The traditional Unix view of the world, however, is so attached to minimalism that it isn't very good at distinguishing between the adhocity-trap problems of vi and the optional complexity of Emacs.

> The reason that vi and emacs never caught on among old-school Unix programmers is that they are *ugly*. This complaint may be "old Unix" speaking, but had it not been for the singular taste of old Unix, "new Unix" would not exist.
>
> —Doug McIlroy

Attacks on Emacs by vi users—along with attacks on vi by the hard-core old-school types still attached to ed—are episodes in a larger argument, a contest between the exuberance of wealth and the virtues of austerity. This argument correlates with the tension between the old-school and new-school styles of Unix.

The "singular taste of old Unix" was partly a consequence of poverty in exactly the same way that Japanese minimalism was—one learns to do more with less most effectively when having more is not an option. But Emacs (and new-school Unix, reinvented on powerful PCs and fast networks) is a child of wealth.

> As, in a different way, was old-school Unix. Bell Labs had enough resources so that Ken was not confined by demands to have a product yesterday. Recall Pascal's apology for writing a long letter because he didn't have enough time to write a short one.
>
> —Doug McIlroy

Ever since, Unix programmers have maintained a tradition that exalts the elegant over the excessive.

The vastness of Emacs, on the other hand, did not originate under Unix, but was invented by Richard M. Stallman within a very different culture that flourished at the MIT Artificial Intelligence Lab in the 1970s. The MIT AI lab was one of the wealthiest corners of computer-science academia; people learned to treat computing resources as cheap, anticipating an attitude that would not be viable elsewhere until fifteen years

later. Stallman was unconcerned with minimalism; he sought the maximum power and scope for his code.

The central tension in the Unix tradition has always been between doing more with less and doing more with more. It recurs in a lot of different contexts, often as a struggle between designs that have the quality of clean minimalism and others that choose expressive range and power even at the cost of high complexity. For both sides, the arguments for or against Emacs have exemplified this tension since it was first ported to Unix in the early 1980s.

Programs that are both as useful and as large as Emacs make Unix programmers uncomfortable precisely because they force us to face the tension. They suggest that old-school Unix minimalism is valuable as a discipline, but that we may have fallen into the error of dogmatism.

There are two ways Unix programmers can address this problem. One is to deny that large is actually large. The other is to develop a way of thinking about complexity that is not a dogma.

Our thought experiment with replacing Lisp and the extension libraries gives us a new perspective on the oft-heard charge that Emacs is bloated because its extension library is so large. Perhaps this is as unfair as charging that */bin/sh* is bloated because the collection of all shellscripts on a system is large. Emacs could be considered a virtual machine or framework around a collection of small, sharp tools (the modes) that happen to be written in Lisp.

On this view, the main difference between the shell and Emacs is that Unix distributors don't ship all the world's shellscripts along with the shell. Objecting to Emacs because having a general-purpose language in it feels like bloat is approximately as silly as refusing to use shellscripts because shell has conditionals and for loops. Just as one doesn't have to learn shell to use shellscripts, one doesn't have to learn Lisp to use Emacs. If Emacs has a design problem, it's not so much the Lisp interpreter (the framework part) as the fact that the mode library is an untidy heap of historical accretions—but that's a source of complexity users can ignore, because they won't be affected by what they don't use.

This mode of argument is very comforting. It can be applied to other tool-integration frameworks, such as the (uncomfortably large) GNOME and KDE desktop projects. There is some force to it. And yet, we should be suspicious of any 'perspective' that offers to resolve all our doubts so neatly; it might be a rationalization, not a rationale.

Therefore, let's avoid the possibility of falling into denial and accept that Emacs is both useful and large—that it *is* an argument against Unix minimalism. What does our analysis of the kinds of complexity in it, and the motives for it, suggest beyond that? And is there reason to believe that those lessons generalize?

13.4 The Right Size of Software

There is a hidden dual of the Unix gospel of small, sharp tools; a background so implicit that many Unix practitioners do not notice it, any more than fish notice the water they swim in. It is the presence of frameworks.

Small, sharp tools in the Unix style have trouble sharing data, unless they live inside a framework that makes communication among them easy. Emacs is such a framework, and *unified management of shared context* is what the optional complexity of Emacs is buying. The practical impact of unified management of shared context is that the user is not burdened with low-level naming and resource-management issues.

In old-school Unix, the only framework was pipelines, redirection, and the shell; the integration was done with scripts, and the shared context was (essentially) the file system itself. But that was not the end of evolution.

Emacs unifies the file system with a world of text buffers and helper processes, largely leaving the shell framework behind. Wily is also about buffers and helpers, but incorporates the shell framework into itself. Modern desktop environments provide a communication framework for GUIs, also leaving the shell framework behind. Each framework has strengths and weaknesses of its own. Frameworks become homes to ecologies of tools—the shell to shellscripts, Emacs to Lisp modes, and desktop environments to flocks of GUIs communicating both via drag and drop and by more esoteric means such as object brokers.

This suggests a Rule of Minimality: *Choose the shared context you want to manage, and build your programs as small as those boundaries will allow.* This is "as simple as possible, but no simpler", but it focuses attention on the choice of shared context. It applies not just to frameworks, but to applications and program systems.

It is, however, all too easy to get sloppy about how large your shared context needs to be. The pressure behind Zawinski's Law is the tendency of applications to want to share context for convenience. It's easy to end up carrying around too much weight, too many assumptions, and to write programs that are over-complex, bloated, and huge. The paradigmatic example in the 1990s was the way that the `mailto:` URL induced the growth of huge mail clients embedded in Web browsers.

The corrective to this tendency comes straight from the old-school Unix hymnbook. It is the Rule of Parsimony: *Write a big program only when it is clear by demonstration that nothing else will do*—that is, when attempts to partition the problem have been made and failed. This maxim implies an astringent skepticism about large programs, and a strategy for avoiding them: look for the small-program solution first. If a single small program won't do the job, try building a toolkit of cooperating small programs within an existing framework to attack it. Only if both approaches fail are you free (in the Unix tradition) to build a large program (or a new framework) without feeling you have failed the design challenge.

When you do write a framework, remember the Rule of Separation. Frameworks should be mechanism, and have as little policy as possible. In most cases, that is no policy at all. Factor as much behavior as possible into modules that use the framework. One of the benefits of writing or re-using a framework is that it can help you separate what would otherwise be big lumps of policy into separate modules, modes, or tools—pieces that can be usefully recombined with others.

These rules are valuable heuristics, but the tension at the heart of the Unix tradition does not resolve neatly into a set of *a-priori* prescriptions for optimal size of any given project. Circumstances alter cases, and exercising good judgment and good taste is what software designers are for. As in Soto Zen, the journey *is* the destination; enlightenment has to be re-discovered in every day of practice.