Similar processes are at work within the established codebase. Experience writing your own commands makes the code of Emacs itself that much more comprehensible, so when you notice a potential improvement to an existing command, you might bring up the command's source code (linked to from its help text, as mentioned earlier) and try to implement your improvement. You can redefine Emacs Lisp functions on the fly, making it easy to experiment. If your idea works out, you can put your redefinition in your *.emacs* file for your own use, and post a patch for inclusion in the official sources for everyone else.

Of course, truly original ideas are rare. For long-time users it's a common experience to think of an improvement to Emacs, look through the documentation, and find that someone else has already implemented it. Since the Lisp-friendly cohort of Emacs's user base has been adjusting and adapting Emacs for almost 20 years now, it's usually a given that many people have already been wherever you are now, and someone may have done something about it.

But whether Emacs grows by acquiring new packages or by incorporating patches its users contribute, the growth is a grass-roots process, reflecting its users' interests: the features exist, from the obvious to the strange, simply because someone wrote them and others found them useful. The role of Emacs's maintainers, beyond fixing bugs, accepting patches, and adding useful new primitives, is essentially to select the most popular, well-developed packages that the community is already using for inclusion in the official sources.

If this were the whole story, creeping featurism wouldn't be much of a problem. However, it usually has two unpleasant side effects: the program's user interface becomes too complex to understand, and the program itself becomes difficult to maintain. Emacs manages to ameliorate the former with mixed success, but it escapes the latter rather effectively.

Creeping Featurism and User Interface Complexity

There are two dimensions along which one can assess the complexity of an application's user interface: the complexity of the Model being manipulated, and the complexity of the command set that operates on that Model.

How complex is the Model?

How much does the user need to learn before he can be confident that he's put the application's Model in the state he wants? Is there hidden or obscure state that affects the Model's meaning?

Microsoft Word documents have a complex model. For example, Word has the ability to automatically number the sections and subsections of a document, keep the numbering current as pieces come and go, and update references to particular sections in the text. However, making this feature work as expected requires a solid understanding of Word style sheets. It is easy to make a mistake that has no visible effect on the document's contents, but that prevents renumbering from working properly. (For another example, ask a help desk staffer about "automatically updating styles" in the 2003 edition of Word.)