Programming Bottom-Up

(This essay is from the introduction to On Lisp.)  
It's a long-standing principle of programming style that the functional  
elements of a program should not be too large. If some component of a  
program grows beyond the stage where it's readily comprehensible,  
it becomes a mass of complexity which conceals errors as easily  
as a big city conceals fugitives. Such software will be  
hard to read, hard to test, and hard to debug.In accordance with this principle, a large program must be divided  
into pieces, and the larger the program, the more it must be divided.  
How do you divide a program? The traditional approach is  
called top-down design: you say "the purpose of the  
program is to do these seven things, so I divide it into seven major  
subroutines. The first subroutine has to do these four things, so  
it in turn will have four of its own subroutines," and so on.  
This process continues until the whole program has the right level  
of granularity-- each part large enough to do something substantial,  
but small enough to be understood as a single unit.Experienced Lisp programmers divide up their programs differently.  
As well as top-down design, they follow a principle which  
could be called bottom-up design-- changing the language  
to suit the problem.  
In Lisp, you don't just write your program down toward the language,  
you also build the language up toward your program. As you're  
writing a program you may think "I wish Lisp had such-and-such an  
operator." So you go and write it. Afterward  
you realize that using the new operator would simplify the design   
of another part of the program, and so on.  
Language and program evolve together.  
Like the border between two warring states,  
the boundary between language and program is drawn and redrawn,  
until eventually it comes to rest along the mountains and rivers,  
the natural frontiers of your problem.  
In the end your program will look as if the language had been  
designed for it.  
And when language and  
program fit one another well, you end up with code which is  
clear, small, and efficient.  
It's worth emphasizing that bottom-up design doesn't mean  
just writing the same program in a different order. When you  
work bottom-up, you usually end up with a different program.  
Instead of a single, monolithic program,  
you will get a larger language with more abstract operators,   
and a smaller program written in it. Instead of a lintel,  
you'll get an arch.  
In typical code, once you abstract out the parts which are  
merely bookkeeping, what's left is much shorter;  
the higher you build up the language, the less distance you  
will have to travel from the top down to it.  
This brings several advantages:  
 By making the language do more of the work, bottom-up design  
yields programs which are smaller and more agile. A shorter  
program doesn't have to be divided into so many components, and  
fewer components means programs which are easier to read or  
modify. Fewer components also means fewer connections between   
components, and thus less chance for errors there. As  
industrial designers strive to reduce the number of moving parts  
in a machine, experienced Lisp programmers use bottom-up design  
to reduce the size and complexity of their programs. Bottom-up design promotes code re-use.  
When you write two  
or more programs, many of the utilities you wrote for the first  
program will also be useful in the succeeding ones. Once you've   
acquired a large substrate of utilities, writing a new program can  
take only a fraction of the effort it would require if you had to   
start with raw Lisp. Bottom-up design makes programs easier to read.  
  
An instance of this type  
of abstraction asks the reader to understand a general-purpose operator;  
an instance of functional abstraction asks the reader to understand  
a special-purpose subroutine. [1] Because it causes you always to be on the lookout for patterns  
in your code, working bottom-up helps to clarify your ideas about  
the design of your program. If two distant components of a program  
are similar in form, you'll be led to notice the similarity and  
perhaps to redesign the program in a simpler way.  
  
  
Bottom-up design is possible to a certain degree in languages  
other than Lisp. Whenever you see library functions,  
bottom-up design is happening. However, Lisp gives you much broader  
powers in this department, and augmenting the language plays a  
proportionately larger role in Lisp style-- so much so that  
Lisp is not just a different language, but a whole different way  
of programming.It's true that this style of development is better suited to  
programs which can be written by small groups. However, at the  
same time, it extends the limits of what can be done by a small  
group. In The Mythical Man-Month,  
Frederick Brooks  
proposed that the productivity of a group of programmers  
does not grow linearly with its size. As the size of the  
group increases, the productivity of individual programmers  
goes down. The experience of Lisp programming   
suggests a more cheerful way  
to phrase this law: as the size of the group decreases, the  
productivity of individual programmers goes up.  
A small group wins, relatively speaking, simply because it's  
smaller. When a small group also takes advantage of the  
techniques that Lisp makes possible, it can   
win outright.New: Download On Lisp for Free.  
[1] "But no one can read  
the program without understanding all your new utilities."  
To see why such statements are usually mistaken,  
see Section 4.8.  
  
Bottom-up design is possible to a certain degree in languages  
other than Lisp. Whenever you see library functions,  
bottom-up design is happening. However, Lisp gives you much broader  
powers in this department, and augmenting the language plays a  
proportionately larger role in Lisp style-- so much so that  
Lisp is not just a different language, but a whole different way  
of programming.It's true that this style of development is better suited to  
programs which can be written by small groups. However, at the  
same time, it extends the limits of what can be done by a small  
group. In The Mythical Man-Month,  
Frederick Brooks  
proposed that the productivity of a group of programmers  
does not grow linearly with its size. As the size of the  
group increases, the productivity of individual programmers  
goes down. The experience of Lisp programming   
suggests a more cheerful way  
to phrase this law: as the size of the group decreases, the  
productivity of individual programmers goes up.  
A small group wins, relatively speaking, simply because it's  
smaller. When a small group also takes advantage of the  
techniques that Lisp makes possible, it can   
win outright.New: Download On Lisp for Free.  
[1] "But no one can read  
the program without understanding all your new utilities."  
To see why such statements are usually mistaken,  
see Section 4.8.