Colorado Engineer Magazine
Fall 1996

On the Evolution of Programming

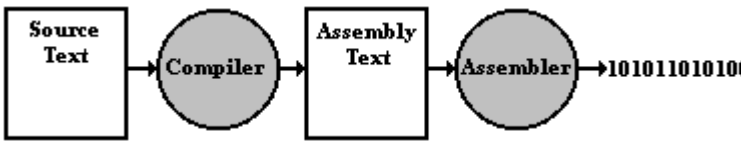
C, C++, Ada, Pascal, Prolog, FORTRAN, Modula3, Lisp, Java, Scheme. This alphabet soup is the secret power of modern software engineering. As *high level* computer programming languages, they provide enormous flexibility and abstraction. Programmers are separated from the physical machine allowing complex problem solutions without fretting with the difficulties of ones and zeros. This idea is clarified by an analogy. If we had to think about every phonetic sound made while speaking, communication of abstract ideas would be next to impossible. Much the same way, programming directly with ones and zeros would focus the designer's attention on trivial hardware details instead of on designing abstract solutions. Considering the historical trend that created high level programming, I believe certain reasonable predictions can be made regarding future advances. It is first necessary to take a simplified look at the evolution of programming.

High level programming is simply the current stage in the growth and change of computer programming. The first computers required their users to do everything. Each binary processor instruction was carefully designed on paper. Then, the program was manually loaded into memory using switches, buttons, and lights. Finally, the programmer pushed a button to run the program. After completion, the process had to be reversed; the program unloaded from memory and the computer reset. As it sounds, this process was extremely difficult, time-consuming, and expensive. The programmer was required to act as the designer, operator, and user of the system.

The second generation saw the invention of the operating system or simply OS. The OS was a permanently loaded program whose job was to automate the trivial details of this process. It would automatically load the program into memory, start it executing, and clean up after it terminated. The job of basic system operations was lifted from the shoulders of the programmer, allowing her to act only as the designer and user. The OS also caused the general public to start viewing the computer as a useful tool instead of a research curiosity.

Assembly language was introduced about the same time as the OS. Suddenly programs no longer had to be represented by ones and zeros; programming could be accomplished with names and symbols. A program called the *assembler* would read through a file of these symbolic names and accurately translate them into the ones and zeros needed by the machine. This new layer improved the programming process by allowing the designer to instruct the computer without worrying about the correctness of the instructions themselves.

Assembly language had the disadvantage that it was specific to one type of computer. An assembly program for an IBM processor simply would not run on an Apple, SUN, Motorola, or any other non-IBM machine. Considering the large number of commercial computer systems, this limitation was a major disadvantage. High level languages solve this portability problem by abstracting the assembly language in the same way assembly abstracted the ones and zeros years ago. Source text written in the high level language is fed into a program called the *compiler* which outputs assembly text. The assembly text is then automatically assembled to the ones and zeros of the machine the same as before.



The benefit of the high level approach to programming is two-fold. First, the same source text can be used with many machines; the compiler verifies that the assembly it generates is correct for the current machine. As a result, it is possible to develop programs without being forced to design for a particular machine. Second, high level languages exploit their distance from the machine by allowing abstract programming techniques that do not seem to mirror machine concepts as much as they do the thought processes of the programmer.

Years ago, the operating system hid computer operations from the programmer. Assemblers then concealed the ones and zeros by creating a level of abstraction. Finally, the high level approach caused the hiding of both assembly and hardware details from the programmer. We have reached a point where users do not need to be aware of the high

level approach or any other programming techniques. They need only know what they must accomplish and buy the correct shrink-wrapped product off the shelf. For example, if Sue needs to create a report on corn production in northern Kansas, she can do so without any knowledge of programming. Sue simply purchases a word processing program and writes her paper. Considering all these developments in computing that now allow such easy use by the general public, it stands to reason that another change is forthcoming.

The ensuing transition will split the users into two categories. I refer to these categories as *designing users* and *traditional users*. Traditional users are the same as before - people like Sue. The important change is the insertion of designing users. Designing users do not program nor do they simply use the computer as a tool. Instead they operate at a level in the middle; both serving other users and being served by the programmers. A designing user can apply a set of "building blocks" written by a programmer (in a high level language) to solve a problem for the public.

Employing the common vernacular, the designing user approach can be likened to the preparation of pancakes. Pancake mixes such as Bisquick or Aunt Jemima are pre-made with the correct proportions of dry ingredients. The anxious diner only has to add water, stir, cook, and eat. However, those with more time can add fresh fruit, syrup, or other condiments. Pancake preparation becomes almost effortless, yet the flavors can still be customized to suit the diner's palate. The connection to programming is apparent. If programmers provide the basic software ingredients, the designing user can freely mix them into the combination that best suits her computing needs.

The Hypertext Markup Language, or HTML, is a perfect example of an emerging computer technology in this paradigm. HTML is a publishing notation for easily building multimedia world wide web documents. A complex software tool called a browser interprets HTML notation and creates a multimedia document for use by many traditional users. Document designers do not need programming experience to author impressive custom documents - just an understanding of HTML notation and their document's content. At the same time, engineers who construct the browser need no knowledge

about the content of the documents it will produce, but can throw the full arsenal of high level language programming tools behind the software.

I believe the designing user approach will snowball in the next few years. We may even see the day where entire computer systems can be configured by their owner instead of at the factory. The programmers building a system will not need to make assumptions about its use. Instead, they create software pieces to support all computing functions and allow the purchaser to connect them like so many Legos. Admittedly, this flexibility is beyond our current technology - programming is far too complex to make effective building blocks - but it is quite an appealing thought.

At each stage in the development of computing, the big winner has always been the general computing public. The OS allowed programs to use the computer more rapidly. Assembly made the development of programs quicker and easier. High level languages further sped development plus made useful software accessible to many users with dissimilar computer systems. It seems the future will be no exception. The designing user philosophy may be the next generation of programming by allowing custom tailoring of computer systems. It has the potential both to increase the efficiency of all users, and to keep ambitious programmers busy for years.