The Craft of Programming

Neville Holmes
University of Tasmania

Building programs should be made enjoyable for more people.

When I looked at the contents of *Computer's* January issue, I was delighted to see the title "Can Programming Be Liberated, Period?" (David Harel, pp. 28-37). As an old-time programmer, I have long been dismayed by the computing establishment's imprisonment and starvation of programming.

However, the article showed that Harel wants programming to be liberated "from the keyboard, from the thankless tension between the what and the how, and from having to partition the dynamics along the lines of the structure [of the system]." The aim, if I understand it correctly, is to get software to do as much of the work of building a program as possible. The splendid work Harel describes strongly suggests that such liberation is feasible, but it's not the kind of liberation I would most like to see. Let me explain.

WHAT I'VE DONE

My programming experience started nearly 50 years ago and, though I've done less and less as time has gone by, I have always thoroughly enjoyed the work of building a program. Most people I worked with enjoyed it as much as I did. The liberation I would like to see for programming would spread such enjoyment much wider.

Programmers as operators

My early training as a systems engineer came from a few years in a data-processing service bureau using a variety of unit record machinery mostly programmed by plugging a panel. Typically, the user brought a trolley of punched cards and a program panel to the next machine in operating sequence, put in the program panel, and fed the cards through. Sorters put card files in sequence; collators merged, matched, and split files; calculators extended numeric values in them; and tabulators printed reports from them.

The operators fetched plugged program panels from storage for use in the case of repetitive jobs; they plugged others a little before use. Different machines had different panels, with the various holes marked to name what they provided a connection to. Plug wires were colored according to their length and used to complete circuits for signals to travel along—for example, to take a digit from a column position in a card reading station to an accumulator.

Plugging a program was a lot of fun. The user simply built a new machine every time, a physical act with a physical result. The programs were simple to understand, but not necessarily simple in function, especially for calculators. For example, an IBM 604 calculator in the Melbourne office of an international oil company saw use in linear programming.

Computer manufacturers gradually phased out plugged programs. In an intermediate phase, programs were partly plugged and partly fed onto a magnetic drum from cards. On the IBM 650, the user could program in Fortransit, a Fortran dialect. Ordinary steps (lengthily and lately called instructions) in a machine code program had one address that pointed to where on the drum the computer would fetch the next step from. Branching steps had to choose between two such addresses. But plugged panels formatted all data coming into the program from a card file, or going out.

Such machines, though simply souped up versions of earlier calculators, let programs be much more complex. Developing a program was still enjoyable but lonelier. The user booked the machine for an hour at a time, usually late at night, and with luck this allowed one compile, link, and test run. The user started a program by manually feeding in through the console a short program that read a loader program in from cards. The user's program cards were then read in by the loader program. The user maintained direct control of what went on.

Programmers as problem solvers

When computers without plugged panels came along, they simply replaced the calculators, tabulators, and magnetic drum computers. Master files were then kept on magnetic tape and magnetic disk,

Continued on page 90

Continued from page 92

though transient data was still handled on punched cards or paper tape.

These practices affected programming profoundly. Programs became more complex and so took a lot more effort to develop. This led to employing specialist programmers responsible for both designing and implementing suites of programs. This, in turn, separated the programmer from the computer, and moved typical program coding and testing to a daily cycle. Programmers would pick up their card decks and printouts from the front office in the morning, work during the day preparing the next test run, and late in the day hand in a revised card deck for running overnight.

This approach made the actual programming rather dull compared to the preceding design of the application, though design could usually be overlapped with coding quite a bit. After a short while, responsibility for application development split between systems analysts and programmers, which made programming a thoroughly dull task. Luckily, I moved on to other work at this time.

Users as programmers

In the early 1970s, I began doing research and development work in an institute with a time-sharing system called CP/CMS. This virtual machine system gave each user a computer and operating system with a typewriter terminal as its console. To write and format papers, I used an embedded programming system called Script, and for computation I used an interpreter called APL. With Script, I could use markup to completely control the format of my documents within the capability of the system's line printer. With APL, I could key in computations and get the result straight away through the typewriter.

Although no longer a programmer, I was programming as a large part of my work, and enjoying it greatly. Both Script and APL were a delight to use, and both allowed complex work to be simplified by coding macros or functions.

After a while, I also got to use the IBM 5100 personal computer, which was based on an APL interpreter. I could take this machine to schools for special work, and found it particularly gratifying to see how quickly primary school children learned to use it for simple graphical computation.

WHAT I'VE SEEN

The picture at the industrial and professional levels looked quite different from my early experience with programming at a personal level.

As computing systems became more complex, data processing departments became larger and larger, and moved up the organization charts.

Programmers as line workers

The growing use of stored program computers in the 1960s led to the enlargement of data processing (DP) departments. Their budgets, head counts, and political status grew rapidly. Part of this growth stemmed from management's separation of design by system analysts from coding by programmers. This made the programmers' work rather boring, particularly as they were given specifications often made very complex to highlight the analysts' responsibilities and skills.

Program coding was often drudgery, which didn't improve its quality. Difficulties in interpreting and implementing specifications meant that DP departments often delivered applications late. The applications' increasing complexity often meant that teams did the programming, which didn't in practice help productivity much.

As computing systems became more complex, DP departments became larger and larger, and moved up the organization charts. With their increasing political power, their projects more often reflected what the DP department wanted to do rather than what the organization needed as a whole. DP favored big projects.

In the early stages of this evolution, mainframe computers did the work by running batch jobs. The DP department was a kind of island within the organization, with users kept at bay across the causeway of input data preparation and report delivery. Programmers and machine operators were isolated within the department.

This changed somewhat when time-sharing systems became popular in the business world. DP clients liked the timeliness of putting data in through terminals and being able to use terminals to make inquiries and get simple reports. The DP department liked time-sharing systems because they increased the department's budgets and required more personnel to design and implement the more complex applications.

Eventually, management asked the programmers to interface their programs with users beyond the DP department. The politics, and early limited capabilities of the time-sharing mainframes, meant that the programmers had to write interactive interfaces that put the user through a specific and fixed procedure. In effect, the programmers coded programs to drive the user as a peripheral device.

Many users didn't like this much. They felt dictated to by their DP departments. When personal computers became available, a struggle broke out between users who wanted to control their own work and DPers who saw the adoption of PCs as threatening their political status. Through all this, the programmers sat in their cubicles and coded in social isolation.

Programmers as profiteers

The popular adoption of PCs in business featured most prominently

in the use of program packages to do basic work. They were made particularly effective by the use of full-screen displays in word processing and spreadsheet applications some time before the mainframe world adopted the approach—and look what happened to the typewriter industry as a result.

These and other simple applications were first written by clever programmers, without the formal support of system analysts, and sold at a profit largely independent of the development effort, depending instead on the volume of sales. Programmers created new versions of the originally simple programs and in turn made a profit for the successful companies.

However, the substance of programs doesn't wear out. So continuing profit required continuing development—meaning enlargement—of PC programs to be sold in a stream of new versions. In a way, the rise of software companies mirrored the rise of the DP departments. Programs became larger and more complex, with a continuing search for marketable features. The programmers wrote the programs that drove the users.

WHAT I WANT

The problem is that software users are seen as automatons to be driven along the predestinate grooves of menus, options, and features to achieve what is often a very simple task. Users must learn the ins and outs of the software to do anything complex. Inexperienced users face a complex learning task to find the options and sequences that get the software to do what they really need, and often simply give in to the program. Users are not in control.

The programming I have always done gave me control of the computer, long past the time when I was formally a programmer. These days, I use LaTeX to develop documents whenever I can because it lets me specify just what I want done in simple and extensible terms. I code my presentations in vanilla HTML because I can code specifically what I want shown. I key in my computations using tacit J (a dialect of APL) because I can directly specify just what I want computed and get the result straight away.

What I want is to have ordinary computer users gain direct control of what they do on a computer, to be liberated from the shackles of what professional programmers working for software companies are prepared to let them do. This could be done in many ways. Perhaps the popular adoption of Linux will be a start.

Skilled programmers will long be needed for industrial work, at least until Harel's liberation idea takes place. But ordinary people should be able to enjoy programming as a craft. After all, many people still enjoy crafts like gardening and carpentry, even though technology has made industries of farming and woodworking.

The importance of giving users control, and thus the opportunity for taking initiatives, developing skills, and making discoveries, is not just for the personal use of PCs. It also applies to employees using their organization's computing system, and additionally lets them have responsibility for the quality of their work. It particularly applies to students as it will also give them the opportunity to develop individuality.

Neville Holmes is an honorary research associate at the University of Tasmania's School of Computing and Information Systems. Contact him at neville.holmes@utas.edu.au.