Throwaway Programs

Peter J. Denning

Razors. Ball point pens. Diapers. Towels. Paper plates. Plastic forks. TV dinner trays. Fast food boxes. Pop bottles. Our world is full of disposable personal objects, transient things soon discarded.

Ten years ago began the debates about structured programming, which sought more correct, more understandable software through restricted program forms. These debates helped bring forth programming languages with strict syntactic structure—e.g., Pascal, Fortran 77, and lately Ada. They stimulated new texts on programming and new sets of rules for students in programming courses. But our students go on to be professional programmers whose software is unreliable and unportable; who often construct new programs from scratch rather than from existing programs; who keep such poor records that they cannot later reuse their programs. We continue to rely heavily on a small set of highly gifted software architects to keep our computer systems running.

Why is this? Why is so much software of poor quality? Not distributable? Why so little progress toward transportable software parts despite so much attention to program structure and project management? The seed of an answer is in a paper by Douglas Comer,[1] who has made the remarkable observation that the most useful public programs started out purely for the personal use of their authors.

Comer's insight is important when viewed in the context of our fondness for the personal and the disposable. As with other things in our daily lives, we have come to think of programs as personal objects. They are poorly documented and not intended for use by others. As a consequence they are difficult to reuse; they become throwaway objects, transient things soon discarded.

Toward Composable Software Parts

I regard the attitude "programs are personal objects" as the fundamental problem. That many programs turn out to be throwaway objects is an unfortunate consequence of this problem. This attitude is encouraged by our operating systems. It is ingrained in our approaches to teaching programming.

Much of our attention in recent years has been directed toward *simplifying the task of writing single programs.* I include structured programming, advanced languages, interactive editors, program synthesizers, and debugging aids in the group of tools that assist in this goal.

We must now direct our attention toward *saving, sharing, reusing, and composing software parts.* Operating systems must become part of the solution of these needs. They are presently part of the problem.

In the 1950s and early 1960s, operating systems were intended as automatic control programs; their objective was to schedule work, to maximize efficiency, and to meter resource use. There was no "user interface." By the late 1960s operating systems began offering services that users could not supply for themselves. These included program libraries, file systems, and backup facilities. It is an unfortunate fact that much of the 1950s approach to the "user interface" survives to the present day. Job control languages are exceedingly difficult to use and often seem devoid of concept. Documentation about system utilities and services can hardly be understood by skilled systems programmers, much less by ordinary users. Many systems

[1] See D.E. Comer, "Principles of program design induced from experience with small public programs," *IEEE Transactions on Software Engineering*, March 1981.

force users to save programs on cards or tapes; only a few have automatic backup and archiving.

In short, to foster a new attitude—that programs are potentially public, sharable, and transportable—we need operating systems that are hospitable toward saving and reusing program parts. I will cite Bell Labs' UNIX operating system to illustrate that the technology is at hand.[2] Most of the concepts in UNIX originated in experimental time sharing systems CTSS and Multics during the 1960s. They have now been refined for use on small machines.

Programming Environments

A programming environment is the set of facilities which permit a programmer to synthesize, connect, and execute program modules. Besides the usual editors, compilers, and debugging aids, a programming environment must also contain long term personal file storage; it must give access to libraries and to programs written by others; and it must provide a command interpreter to facilitate program invocation.

Bell Labs' UNIX is the most widely available example of a system with this type of programming environment. On logging in, a user is assigned a process containing the shell, a command interpreter that listens to his terminal. The shell interprets the first word on each line as the name of a program, which it invokes with the rest of the line as input parameters. Through the shell, the user has access to the directory hierarchy of the system, in which all files are kept. He may construct a personal tree of directories and files and he can specify the degree to which others can share these objects.

Central in the UNIX programming environment is the concept that *every program is a module with a single data-stream input and single data-stream output.* This is true of user programs as well as systems programs—including the shell program. By default, the input and output of a program are connected to the terminal. Simple shell notations permit the user to reconnect a module's input or output to arbitrary files in the system, and also to connect the output of one module to the input of another. For example, typing

filename

invokes the file "filename" with its input being anything typed at the terminal and its output being displayed at the terminal. Typing

filename < X

invokes "filename" with file X as its input, typing

filename > Y

invokes "filename" with its output being stored in file Y, and typing

filename < X > Y

does both. Typing

filename1 | filename2

invokes both "filename1" and "filename2"; the second program's input is the first's output. These simple concepts permit programmers to easily construct and run more complex programs by connecting existing modules and files.

The shell permits users to store sequences of shell commands, called shellscripts, in files for later execution. This facility permits the user to construct more complex modules from simpler ones already available.

UNIX also permits a user to move freely among environments. For example, while editing a file, a user can escape to the shell, invoke the mail program, process his mail, and later return to the editing of his file. Or, a user can escape from processing his mail, use an editor to create a file, then return to the mail program and mail the new file. The user is not confined to any one environment; he can interrupt himself at any time and return later to finish the original task.

The simple set of concepts of UNIX forms a programming environment that encourages the user to employ the *whole system* in the solution of problems. He can connect any programs together at any time. He can save any new modules for later use. He can move freely among environments. He does not fear the operating system because he understands its basic principles. Because sharing is encouraged, he tends not to think of programs as personal or throwaway objects.

Programming Education

Programming education contains strong biases in the wrong direction. Few courses require students to find and use modules in libraries or in public files. (Lesson: Program from scratch.) Collaboration is strongly discouraged; students get no permanent sharable file storage on the computer. (Lesson: Programs are personal.) Once an assignment is handed in and graded, it serves no further purpose. (Lesson: Programs are throwaways.)

To reverse these biases, we need to provide students with programming environments embodying concepts like those in UNIX. But this is not enough. We also need new policies of instruction and grading that encourage students to avoid programming from scratch, to use software tools effectively, to construct modules intended for use by others, and to maintain documentation permitting reuse of their own software.

Why not institute grading policies that assign a higher reward for using existing software parts than for programming from scratch? Why not have projects in which students exchange modules, grades being assigned according to how well others are able to use or modify these modules? Why not require each student to maintain a manual of all software he has developed, and give him adequate file storage for his software during his academic career?

It will not be cheap to upgrade college and university facilities from their present conditions, to provide each student with interactive terminal access whenever he requires it, with sufficient personal file space, and with connection by network to other facilities. But it must be done. Here is an excellent opportunity for cooperation between industry and academia.

There is a danger that the rising interest in personal computers will perpetuate the software problem by encouraging students and other programmers to regard the entire computer as a personal object that seldom interacts with anything else. Most personal computer systems today have very primitive programming environments, little advanced beyond the operating systems of the 1950s. They introduce beginners to throwaway programs—most personal computers and programmable calculators discard programs by default when the power is shut down. Here is an excellent opportunity for cooperation between academia and industry.

[2] See D.M. Ritchie and K.L. Thompson, "The UNIX time sharing system," *Comm. ACM 17*, 7 (July 1974), 365–375.