Reuse revisited

Using and reworking software is not new; it has been done since the very beginnings of our industry in the early 1950s. Reuse means using an entity in a different context from that in which it initially had been used. This is often called "black-box" reuse. When an entity is modified before it is used in the new setting, it is called "rework" or "white-box" reuse.

We have more to gain from reusing parts of complex systems than from reusing parts of simple systems because the intellectual and development effort that goes into designing and implementing complex systems is much higher for complex systems. But complexity is relative. Some might think that optimizing compilers, distributed operating systems, entity-attribute-relationship database management systems, and real-time transactions systems are complex while others might think that window managers, report generators, spreadsheets, sort packages, and syntax-directed editors are complex.

Multilevel refinement process. Large, complex systems are all developed through a multilevel refinement process encompassing high-level requirements, specifications, designs, programs, tests, and documentation. A good deal of the work on reuse has focused on low-level procedures (subroutines and the like). Very little work has been done on reuse at the high level.

One can argue that bringing an operating system such as Unix up on a large variety of machines or porting high-level language compilers to many machines are very important forms of reusability, and I would not disagree. However, these are cases of "all or nothing" reuse of large systems. We do not have viable technology that allows us to use substantive portions of such existing systems in the newer and different systems we are creating. The system as a whole may be portable (reusable), but its parts are typically too system-specific to be reused easily.

What does it mean to reuse the input or output objects of any of the interrelated levels of refinement in software development? What objects or object fragments are candidates for reuse? How should such objects be stored? How do we go about locating them? Once located, what must we do to incorporate them into our design, specification, program, test, or documentation?

Software repositories. On-line repositories are integral to software reuse. Yet few standardized software libraries, except for mathematical, input/output, and string manipulation libraries, are in widespread use. Subroutine libraries usually contain small, language-specific, poorly documented programs with bound data structures, date representations, and fixed, inflexible parameter lists.

Even in sophisticated program development environments, libraries cannot be searched in a convenient manner. A programmer often has to exit the programming environment, use a limited set of tools to search for reuse candidates, then reenter the environment, often breaking an important line of thought in solving the original problem at hand. The integration of libraries into our design and programming environments is thus central to reuse technology.

The lead article in this issue, "Classifying Software for Reusability" by Ruben Prieto-Diaz and Peter Freeman, offers a partial solution to the problem of storing software for reuse.

There are other important techniques that can be brought to bear on the problem of software reuse, in the small and in the large. Version and configuration control, structured active repositories, phase-integrated development environments, program transformation and generation techniques, and methods to achieve language and operating system independence are all under investigation.

IEEE Software will publish a special issue on reuse in July that will cover some of these additional issues. We look forward to receiving your contributions to it.

January 1987

Bruce D. Shriver, Editor-in-Chief