A Software Catastrophe in Scientific Computing?

By Chris Anderson

A few years back, articles predicting a "software catastrophe" began to appear in the popular press. Computer programs were growing so large, in both size and complexity, that they were becoming unmanageable. According to the articles, these programs would cease to function as problems occurred and went uncorrected. As with an asteroid hitting the earth and terminating life, anything that depended on this software would come to a screeching halt. While the subject of these articles was commercial software, the concerns cited had a ring of truth for those involved in scientific computing. Programs for scientific computing were also growing larger, more complex, and more difficult to manage. Would scientific research and development that depends on scientific computing come to a screeching halt as well? The possibility had researchers thinking.

"Natural" Software Management

But the catastrophe in commercial software hasn't happened. While we may be frustrated with PC software crashing our machines now and then, incredibly complex software systems are functioning well and the software components on which they're built are effectively managed. How was the catastrophe averted?

One of the big reasons for the "non-catastrophe" was the creation of languages that enable object oriented programming, e.g., Smalltalk, Object Pascal, C++, Ada, and Java. These languages have constructs that can be used to create distinct software components, or "parts," so that the process of creating and maintaining a large software system is much like that of creating and maintaining a physical system built of physical components.

The ability to manage software in this "natural" way has had a tremendous impact on the way we create and maintain complex software systems—it has, in fact, staved off the software crisis. User interface programming is a good example. Creating and managing a graphical user interface used to be a difficult task that could be accomplished only by hard-core programmers. Because of its complex nature, the software often became unmanageable. Now, using suitable collections of software parts (i.e., class libraries or class packages), the average programmer can create and easily maintain applications that have sophisticated user interfaces. Indeed, a large number of the popular audience "do it yourself" books on C++ and Java use the construction of a graphical user interface as a means of teaching object oriented computing. Beginners are doing now what only experts could do a few years ago.

Wouldn't it be nice if creating and maintaining applications for scientific computing were as easy as creating a user interface or a data base manipulation program?

Wouldn't it be nice if creating and maintaining applications for scientific computing were as easy as creating a user interface or a data base manipulation program? Can we avoid a software crisis in scientific computing by using object oriented computing techniques?

Class Libraries For Scientific Computing

To take advantage of object oriented techniques, of course, requires the identification and construction of the software class libraries that make sense for scientific computing. While it is rather easy to identify the components needed for graphical user interfaces (buttons, menus, sliders), identifying and creating the requisite software parts for scientific computing isn't so easy.

Participants in an interdisciplinary SIAM workshop, Object Oriented Methods for Interoperable Scientific and Engineering Computing,* held at IBM in October 1998, clearly identified the challenges of creating good class libraries.

Obviously, for a given set of problems, the class library has to contain the components needed to create applications for solving those problems. The software components should be computationally efficient, and, if possible, different libraries of components should be compatible. Moreover, once the software components have been constructed, it should be possible to use them without detailed knowledge of their actual implementation.

The difficulty in creating class libraries that meet these objectives lies in the fact that scientific computing codes can be decomposed in several ways; each decomposition has to be considered. If a code works with functions defined on grids, should separate components be created for the grid and the function defined on it? Or should the grid and the function be intimately linked in a single component? If we have matrix components, how do we allow for banded, sparse, and dense matrices? How should components be constructed to run well on both serial and parallel machines? What can be done to make components usable across a network?

There are lots of questions to be considered, and much work to be done, but the SIAM workshop made it clear that in the area of scientific computing, the creation of software components is being actively pursued. Class libraries for scientific computing are being produced, and as their use becomes widespread, we can expect large scientific computing codes to become more manageable—there isn't going to be a crisis. And while tools for combining scientific computing components may not have reached the level of tools for working with graphical user interface components, more advanced tools are certain to be coming soon. Just as average programmers can now create and manage complicated user interfaces, it's reasonable to expect that the average researcher will soon be able to create and manage complex scientific computing codes. Perhaps this will be done through a visual programming interface, perhaps it will involve talking to the computer; it's exciting to think about what might be possible.

Chris Anderson is a professor of mathematics at the University of California, Los Angeles.