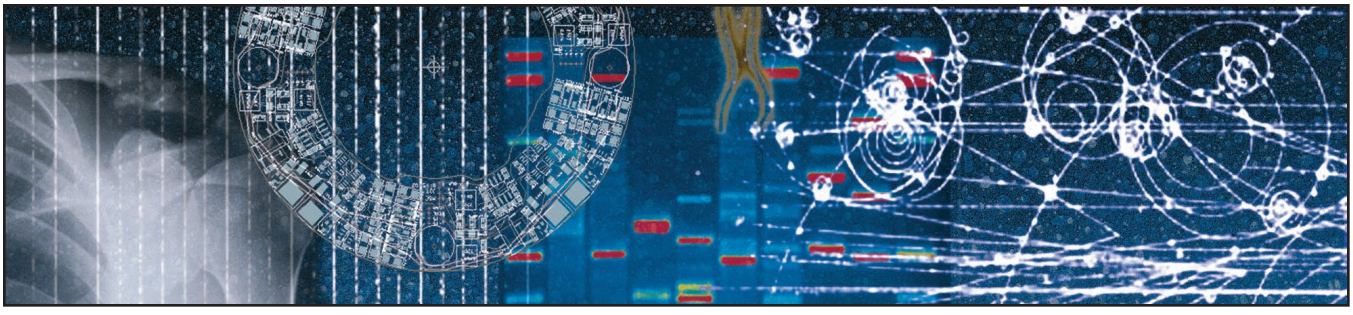
# Developing Scientific Software

**Judith Segal,** *Open University*

**Chris Morris,** *Science and Technology Facilities Council*

How can scientific-software development be improved? Exploring this question requires investigations, solidly grounded in practice, into both the particular characteristics of scientific-software development and potentially relevant software engineering techniques, methods, and tools. That is our goal in this special issue.



Important research in this area, conducted under the aegis of the DARPA High Productivity Computer Systems program (www.highproductivity.org), has already appeared in the literature.[1] However, not all scientific computing is high-performance computing (HPC)—the variety of scientific software is huge. Such software might indeed be complex simulation software developed and running on a high-performance computer, but it might also be software developed on a PC for embedding into instruments; for manipulating, analyzing, or visualizing data; or for orchestrating workflows. We hope this special issue provides some flavor of that variety.

## What makes scientific software development different

Developing scientific software is fundamentally different from developing commercial software. Most software developers have some idea of what a human-resources or accounting package should do, and they feel they can understand (perhaps with some effort) such packages' requirements. But do you understand, for example, how genomic DNA gets transformed into protein crystals? Do you comprehend the intricacies of fluid dynamics? Or how to solve 20 simultaneous partial differential equations? A scientist (domain expert) *must* be heavily involved in scientific-software development—the average developer just doesn't understand the application domain. For this reason, the scientist often *is* the developer.

Another difference has to do with requirements. HR people and accountants broadly know what they want: they might well change their mind as development progresses, but they basically understand their domain. For scientists, this might not be the case. The software's purpose is often to improve domain understanding—for example, by running simulations. Full up-front requirement specifications are impossible: requirements emerge as the software and the concomitant understanding of the domain progress. Related to the users' incomplete understanding of the domain is the additional problem of validating scientific software. Scientists often lack "test oracles"—real data against which they can compare their software's output. Simulation software is a case in point: the science is too complex, too large, too small, too dangerous, or too expensive to explore in the real world.

Field studies of scientists developing their own software have revealed the model of software development shown in Figure 1.[2,3] No software engineering course would teach this model, but it's surpris-


0740-7459/08/$25.00 © 2008 IEEE

ingly prevalent in scientific-software development.

Other field studies have demonstrated that efforts to impose software engineering techniques on scientists are beset with problems.[2,4] Figure 2 illustrates a clash between the software engineers, who expect an up-front specification of requirements, and the scientists, who expect requirements to emerge.[3]

## Three themes

We were told that we received 20 percent more submissions than an average *IEEE Software* special issue. We were happy to have the problem of too many good articles to fit into a single issue. We solved it by organizing the accepted articles into three themes: those in two themes appear in this issue; those in the third will appear early next year.

### Characterizations

The first theme concerns the characterization of scientific software and scientific-software developers. Rebecca Sanders and Diane Kelly describe a qualitative study to explore scientists' perceptions of risk and the management of risk in the software they develop. Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz describe the HPC community's characteristics as identified in their case studies and discuss which established software engineering techniques and tools might benefit this community. Finally, David Woollard, Nenad Medvidovic, Yolanda Gil, and Chris A. Mattmann classify workflow systems according to their focus: discovery, production, or distribution.

### War stories

The second theme might be called "war stories." We received many case studies of actual scientific-software development projects as told by the scientist developers (not regular contributors to *IEEE Software*, we think). Alas, we rejected nearly all of these. Some described software, often exciting, that the author had developed. However, we rejected these articles because we were interested in the *process*, not the *product*, of scientific-software development. Other submissions, more problematical to us as editors, were thoughtful reflections on a particular project but made little or no attempt to discuss how relevant these reflections might be to other projects. We thus faced situations in which one reviewer working in the same scientific area as the submission's authors said, "This is brilliant," whereas other reviewers working in different areas said, "How is this relevant to me?"

Unsurprisingly, reviewers clashed on other issues,

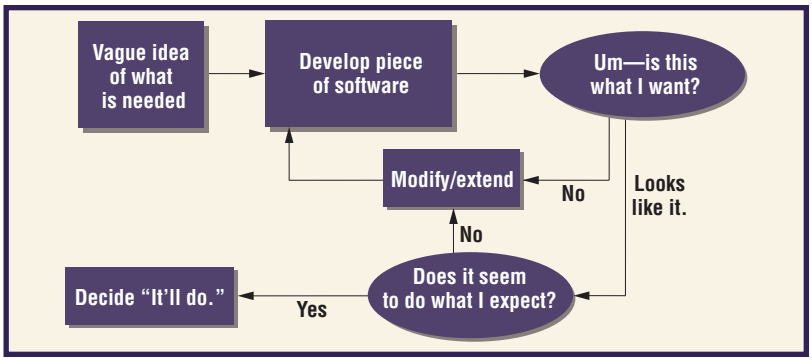



**Figure 1. A model of scientific-software development. Here, requirements are mostly emergent, the emergence of requirements is intertwined with evaluation, and testing is cursory.**

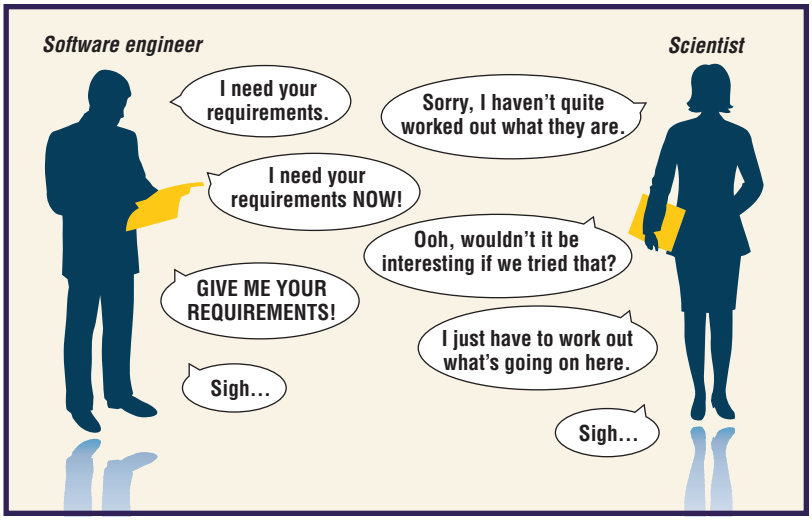



**Figure 2. A clash between software engineers and scientists. The former expect requirements to be specified up front; the latter expect them mostly to emerge.**

too. We assigned each submission at least three reviewers, at least one of whom was a practitioner of scientific-software development and one of whom was a software engineering academic (of course, these categories of practitioner and academic aren't always clear-cut). Software engineering academics sometimes said of a practitioner case study, "The authors don't know the literature." Our sympathies lay toward the authors in such cases. There are interesting issues to explore as to why developers of scientific software, or indeed software in any application domain, don't know the literature that software engineering academics expect them to know. Is this, in fact, the academic community's fault in that it fails to tackle issues that truly concern practitioners? Or do software engineering academics formulate their arguments so as to convince their peers, without concern for how such arguments impact practitioners? This is an important discussion, we feel, but inappropriate to pursue here.

In the end, three case studies made the final cut because we consider them reflective of development practice and of interest to *IEEE Software*'s general

## Further Resources

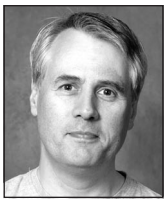
Greg Wilson's Software Carpentry Web site (www.swc.scipy.org) offers instruction in those software engineering techniques and tool uses that he's identified as important to scientific-software developers who don't have a software engineering background.

*Computing in Science & Engineering* magazine (http://cise.aip.org) aims to reach scientists developing software as well as software engineers. The mission of this joint publication of the IEEE Computer Society and the American Institute of Physics is to "support the development of computing tools and methods as well as their effective use in both computational and experimental science and engineering."

## About the Authors


**Judith Segal** is a lecturer in computing at the Open University, working in the Empirical Studies of Software Development group. Her research is grounded in field studies of software development by nonprofessional software developers such as financial mathematicians, earth and space scientists, and molecular biologists. She's also interested in how to bridge the gap between the academic and practitioner communities. Segal received her PhD in algebra from the University of Warwick. Contact her at j.a.segal@open.ac.uk.

**Chris Morris** is a software developer in the **UK government's Daresbury Lab**, where he leads a multidisciplinary team developing a laboratory information management system for molecular biology. Morris received his MA in pure mathematics from the University of Oxford. Contact him at c.morris@stfc.ac.uk.




readership. Karen S. Ackroyd, Steve H. Kinder, Geoff Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson describe 20 years' experience in developing software to handle synchrotron data. Their article is notable in two ways: all the authors are practitioners with no links to the software engineering academic community, and the article describes an attempt to apply an agile method, Extreme Programming. Many practitioner submissions claimed they were following an agile methodology, but often this meant only that they followed the iterative, incremental feedback model in Figure 1. The fact that agile methodologies have their own practices and inherent disciplines seems to have passed these people by. The article by Ackroyd and her colleagues is a noteworthy exception.

Also in this theme, Mark Vigder, Norman G. Vinson, Janice Singer, Darlene Steward, and Keith Mews describe their automation of scientific workflows at the Institute of Ocean Technology in Canada. This article provides insight into how both users and IT support personnel might be involved. Richard Kendall, Jeffrey C. Carver, David Fisher, Dale Henderson, Andrew Mark, Douglass Post, Cliff Rhoades, and Susan Squires discuss the development of weather-forecasting software, and find commonalities with previous case studies of simulation software in different domains. Thus, they derive lessons that might be applied throughout the HPC community.

### Guidelines

We characterize the final theme roughly as guidelines. The three articles in this section discuss in the light of the authors' experiences how requirements, usability, and design might be addressed in the context of scientific-software development. These three will appear in a future issue.

How far has this special issue-and-a-half met our aim to explore how scientific-software development might be improved? We have, we think, made a good start. In fact, we had another aim that we didn't articulate in our call for articles. This was to build a community of people interested in the issues of scientific-software development. The First International Workshop on Software Engineering for Computational Science and Engineering (www.cse.msstate.edu/~SECSE08), recently held at the 30th International Conference on Software Engineering, represents another effort to achieve this aim. The biggest challenge here is to reach the scientists who are developing their own software, often as the sole developers of software in their labs. If you have any ideas on how to meet this challenge, or, indeed, any comments to make on this editorial, we'd be delighted to hear from you. Contact us at j.a.segal@open.ac.uk.

### References


1. J.C. Carver et al., "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," *Proc. Int'l Conf. Software Eng.* (ICSE 07), IEEE CS Press, 2007, pp. 550–559.
2. J. Segal, "Some Problems of Professional End User Developers," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing* (VLHCC 07), IEEE CS Press, 2007, pp. 111–118.
3. J. Segal, "Models of Scientific Software Development," *Proc. 2008 Workshop Software Eng. in Computational Science and Eng.* (SECSE 08), www.cse.msstate.edu/~SECSE08/Papers/Segal.pdf.
4. J. Segal, "When Software Engineers Met Research Scientists: A Case Study," *Empirical Software Eng.*, vol. 10, no. 4, 2005, pp. 517–536.



For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.