Dealing with Risk in Scientific Software Development

Rebecca Sanders, *Queen's University*

Diane Kelly, *Royal Military College of Canada*

As interviews at two universities demonstrate, scientific software developers use numerous strategies to address risk in their software's underlying theory, its implementation, and its use.

More than 30 years ago, a chasm opened between the computing community and the computational-software community. As a result, there has been little interest in or exchange of ideas relevant to scientific application software. This apparent lack of communication between these two communities[1] could put scientific computational software at risk. Our ultimate purpose is to bridge this chasm. Our immediate purpose is to explore and identify these risks. To this end, we interviewed scientists and engineers at two Canadian universities who develop software in their fields. Our study aimed to identify the characteristics of scientific software development, as well as the diversity of those characteristics and interesting correlations between them.

Although we didn't ask interviewees specifically about risk management, their responses indicated that all scientific applications share three broad risk areas:

- the software's underlying theory,
- the software implementation of that theory, and
- users' operation of the software.

Using the data from our interviews, we looked at how scientists manage those risk areas and with what success. We also wanted to see where the computing community could help through research aimed at the risks scientific software developers face.

Study participants

We interviewed 16 scientists from 10 disciplines: civil, mechanical, electrical, medical, theoretical, and nuclear computing; chemistry; optics; geography; and physics. All were academic researchers at the time of the interviews. Two were delivering a military application, one a medical application, and two others an application for the nuclear industry. Four interviewees had previous industrial experience. Three were primarily users of commercial scientific software and did little development.

We found our interviewees in various ways: we knew them through other projects, were referred to them by department heads and other interviewees, and met them by chance. Others contacted us directly. We sought potential interviewees with as varied experiences as possible. On analyzing our data, we feel we succeeded.

Our interviewees' experience in software development ranged broadly. Some were uncomfortable coding; others had worked on industrial projects

0740-7459/08/$25.00 © 2008 IEEE

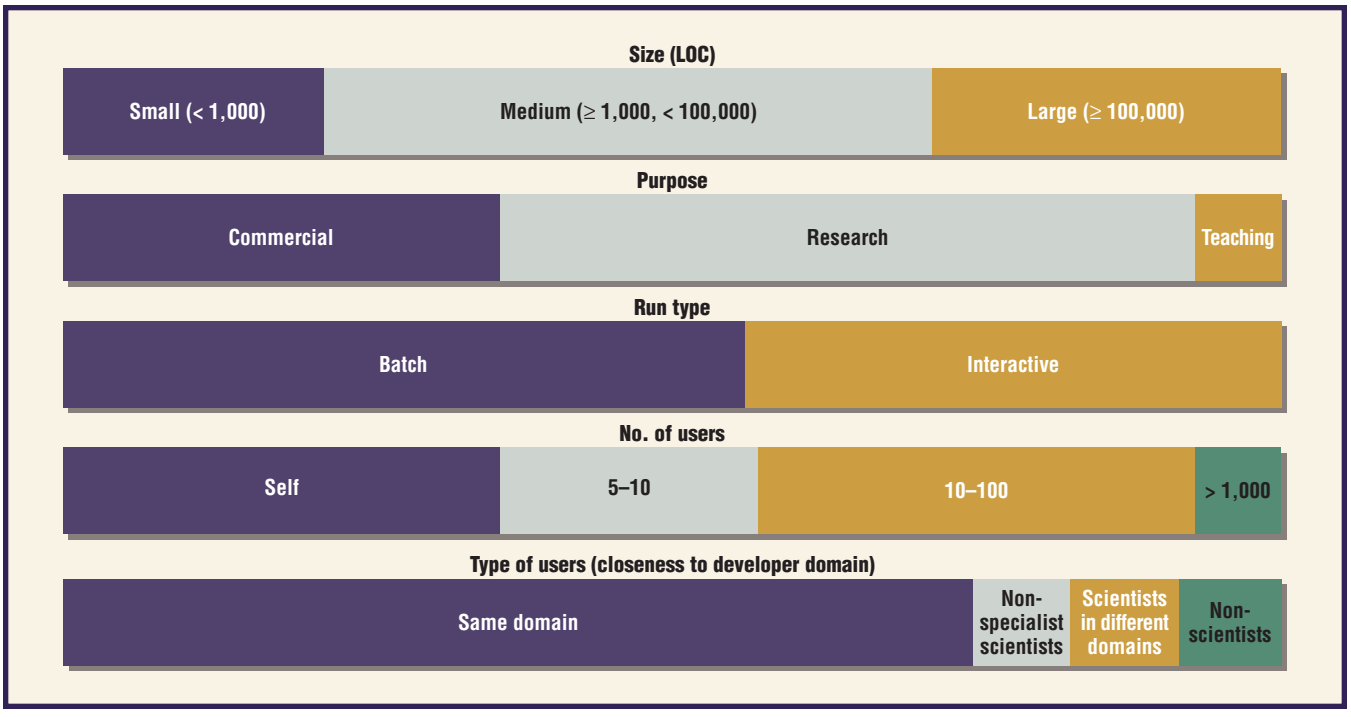


Figure 1. Software characterization. Bar length indicates relative numbers in each category.

and knew about requirements, design, testing, and version control. Their software varied in size from modules of fewer than 1,000 lines of code to programs of more than 100,000 lines of code. Some of the software was interactive, whereas some batch programs took hours or days to run. Figure 1 gives an overview of the software's characteristics. The figure doesn't include the three interviewees who were primarily users of commercial software.

Our study is a broad qualitative exploration,[2] not a statistical confirmation of known facts. Because our goal was to identify a diverse set of characteristics, we didn't use preconceived questions that could bias the interviews. Our interviews, each lasting roughly an hour, were unstructured and open-ended. Our only guidance to interviewees was to talk about their development process, documentation, and users.

To help us classify our data, we asked interviewees to describe their software and backgrounds. We digitally recorded each interview and rendered it into a collection of point-form comments. A comment is a single statement from the interviewee about his or her software experience or environment. We collected more than 1,100 comments.

We developed three sets of categories. The first set (Table 1, top) relates to characteristics of the scientists and engineers interviewed. The second set (Table 1, bottom) relates to the software's static characteristics. The third (Table 2 on page 24) relates to actions and opinions about scientific software development and use.

Purpose

Purpose is fundamental to the software our interviewees described. We found three distinct purposes in our interviews:

- *Research.* Our interviewees develop theories and models to explore their own worlds. They use software as a tool to provide evidence that their theories work. This evidence is often the basis for research publications.
- *Training.* The scientific software tools our interviewees develop are used in the classroom and lab to train the next generation of researchers, scientists, and engineers.
- *External decision support.* Scientific software sometimes moves out of the research lab. Data from this software supports other professionals' decision making. Our interviewees' software has moved to the nuclear industry, the military, and the medical sector.

These purposes aren't mutually exclusive. Software developed for external use is often based on cutting-edge research investigated using software models. These models can in turn be used in the classroom to train the next generation of researchers. The software's purpose is linked to risk but also to the priority the scientist/developer gives to two distinct parts of the software.

One part is the computational engine. This part embodies the scientist's theory and models and comprises the risks related to the underlying theory

and implementation. Scientists focus much of their attention, if not all, on the computational engine. If the engine doesn't work, the software is useless.

The second part is the user interface. The risk here is in the users' operation of the application. The user's knowledge of scientific software often overlaps that of the scientist/developer, but not always. Several interviewees paid little attention to the user interface. Given that their users were scientists whose knowledge overlapped the developer's, usability was not a concern. Others, such as a medical software developer, considered the user interface to be as significant a risk to the software's success as the theory and consequently gave it a lot of attention.

Our interviewees demonstrated their approach to risk management in how they managed the development process, testing, and documentation.

Development process

The software development process is like a feedback control loop (see Figure 2). If the software gives an unexpected output, the scientists/developers might adjust both the theory and the code. They prepare data, run tests, and examine the output again. In short-duration and focused iterations, they change and check the code and the theory side by side.

Even when the theory is mature and well understood, scientists often use an iterative approach. One interviewee explained that as the software proved increasingly useful, he investigated more theoretical models to add features to it. He performed usability testing of the interface while testing his new models, gathered feedback from the tests, and redeveloped the interface and models, continuing with the familiar iterative process.

Winston Royce's "do it twice" development process[3] showed up in two of our interviews.

The civil engineering application's expansion of scope caused one interviewee to reassess his software's structure. In the second year of development, he completely redesigned his software. He spoke with enthusiasm about how the new design made it easier to add new features.

The interviewee developing medical software clearly distinguished between the software's research version and the version put into the hands of the medical personnel. He had developed a two-step process. People in his research group explored their theories and wrote software without any formalized software development process. Once the researchers assessed their software as ready, a professional developer rewrote it.

Design considerations

Most of our interviewees have application domain backgrounds similar to their users. Scientists are probably most comfortable in this situation. They assume that the user interface they develop to suit

Table 1

Categories for classifying scientists and their software

Category	Description
Scientists	
Back	Formal scientific and software development background
Date	Interview date
Devyrs	Years of experience developing software
Field	Research area
Job	Occupation
Software	
Active	If deployed, is the software still in active use?
Datasize	Size of the data used, manipulated, or managed by the software
Datasrc	Source of data used to calculate outputs, including test data
Devsize	Development group size
Hardenv	Hardware environment
Namesoft	Software's name
Numuser	Number of users
Purpose	Software's purpose
Runtime	Typical software runtime
Sedev	Was a computing specialist involved in development?
Softenv	Software environment
Softlang	Languages the software is written in
Softsize	Software size
Stable	Software development group's stability
Status	Development status
Userchar	Characteristics of the software's expected users
Year	Year software was completed, if applicable

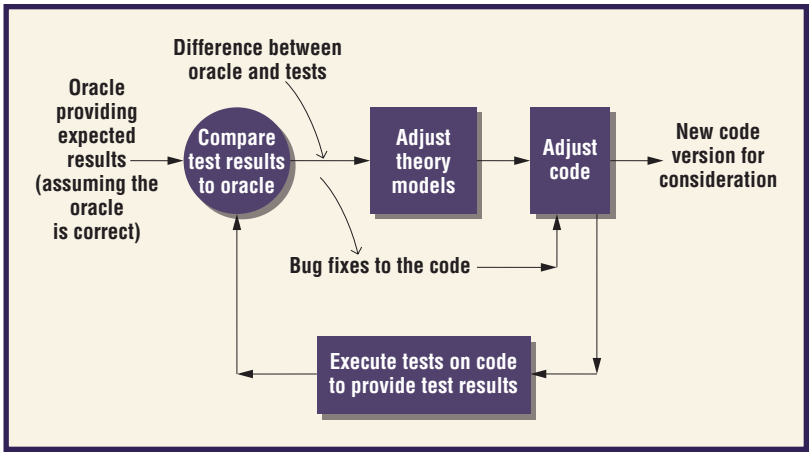


Figure 2. Side-by-side development of theory and code for scientific software.

Table 2

Categories for classifying actions and opinions about scientific software's development and use

Category	Description
Codedoc	Code documentation
Comiss	Issues related to commercially available scientific software used
Dataman	Data management techniques and issues
Design	Scientific software design
Devflow	Software development workflow
Distrib	Software distribution
Domiss	Domain-related characteristics and challenges
Hqp	Education of highly qualified personnel
Inspect	Software inspection
IP	Intellectual property issues
Knowsci	Exchange of knowledge between different scientific domains
Knowsoft	Exchange of knowledge between science and software practitioners
Lang	Commentary on software languages
Maint	Software maintenance
OPSC	Open-source software commentary
Otherdoc	Other forms of documentation affecting software, such as academic papers
Otherflow	Other workflows affecting software development, such as thesis writing
QC	Quality concerns, in related quality factors subcategories
Reg	Regulatory issues/bodies affecting software development and documentation
Req	Requirements gathering and elicitation
Soft	Dynamic software environment concerns
Test	Testing procedures and results, in testing types subcategories
Time	Time spent on development or predevelopment activities
Tools	Tools used to support software development or use
Userdoc	User documentation
Userint	Interaction between developers and users
VC	Version control and configuration management

their use will suit their users. Problems with this approach only arose when the interviewee wasn't representative of the target users.

Sometimes, interviewees made extensive efforts to get the user interface right. Again, they used an iterative approach. This time, however, the iteration wasn't with theory and code, but with user and code. Our interviewees used the following techniques to test and change the user interfaces:

- implementing interactive storyboards in a standard presentation tool,
- observing the software in use in the field multiple times,
- testing the user interface in a lab with students,
- testing the user interface in a controlled environment with selected potential users,
- adding extensive online help screens that mirrored the software's structure, and
- adding extensive diagrams and glossaries.

Interviewees seldom discussed design as a distinct step in software development. Only our two "do it twice" interviewees actually performed design as a separate step. The civil engineering application developer used the object-oriented design philosophy to guide his structuring of the code. The medical software developer used a software architecture from previous medical software projects. For both projects, user risk was high because user knowledge didn't overlap developer knowledge. In addition, both applications were moving to a user outside the research environment. And in both cases, a development team member acted as a software specialist rather than a scientist.

Software design was often under the surface. Some interviewees talked of adding modules to "behemoth" or "monster" programs. These programs were the culmination of many years of work by many different researchers. The scientists didn't consider redesigning some of these monsters a valuable use of their time. Redesign also introduces risk. If the science theory still works in these monsters, it isn't touched. Scientists generally want to do science, not write software, and certainly not introduce risk by changing software that works.

According to some of our interviewees, they only considered redesign when runtime was a critical factor in the software's goals. One interviewee considered moving to a parallel processor. This entailed added code complexity, so the scientist chose a less risky solution. He hired an experienced software developer with a scientific background to improve time performance. The developer achieved enough performance gain through simple and minimal code restructuring and moving the software to a faster single-processor machine.

Programming language decisions

At the implementation level, all our interviewees were willing to discuss software languages. They used both procedural (Fortran, C, and Matlab) and object-oriented (Visual C++, Java, and Visual Basic) languages. When we discussed language preferences, we ran into decided opinions on the different languages' pros and cons.

Some interviewees vehemently defended procedural languages, especially Fortran.

According to one interviewee, "OO doesn't buy me anything." He stated that what he could do in a couple lines of C would take a large amount of C++ code. Another said it was a "dirty little secret" that a lot of scientific software is written in Fortran, even today. This is despite the pressure he claims is applied to scientific developers to switch to object-oriented languages. A common complaint was that graduate students arrive in their programs with no knowledge of Fortran.

Interviewees cited various reasons for the continued use of Fortran in scientific software, including the extent of legacy code written in Fortran and the trust and long experience with the language's scientific computing libraries.

One interviewee considered C the language of choice for "low-level, operational stuff," but another viewed C as "Fortran without some of the more convenient features." Such statements reflect the scientists' risk management approach. If they have no reason to incur the risk of implementation in an unfamiliar language, they don't do it.

Matlab seems to fill a different niche in the scientific software development community. One interviewee said he used it for pre- and postprocessing data; another used it for prototyping. One physicist saw Matlab as self-documenting and "excellent" for scientific software. Performance was the one factor that took the scientists away from Matlab and back to C or Fortran.

Interviewees also made points in favor of object-oriented languages. The scientist working in medical computing chose Visual C++, citing several useful tools available in C++. Both civil engineering interviewees used Visual Basic. One said he switched to Visual Basic because he wanted support in creating his user interface. Another said that object-oriented development fit the problem because objects in the code corresponded to physical objects in his application. The first chose Visual Basic over C++ to reduce distractions in the source code—another risk management choice. He wanted to see his theory in the implementation without the clutter of code language artifacts. The scientist particularly wanted to be able to spot mistakes in the translation of his theory to code.

Some interviewees chose programming languages for other reasons. One chose Java because he learned Java in school. Another chose Visual C++ but was considering C# because he felt pressured to keep up with the latest development languages. Whether it was because of his lack of experience with these languages or because the languages and libraries couldn't support computational applications, his software gave disturbingly different answers when he moved it from one machine to another.

One interviewee who had experience with many development languages commented that the choice between object-oriented and procedural languages comes down to how the developer prefers to think about a problem. If the solution is largely algorithmic, procedural languages are better suited. If, as with the civil engineering applications, the solution is largely object manipulation, object-oriented languages might be a better fit. Writing procedural code with languages such as Fortran should be considered a valid option in a broad tool kit for scientists.

Most scientists produce documentation related to the theory rather than strictly related to the code.

Commercial or open source?

To address risk, some scientists used software already available, such as commercial or open source software. Each has associated risks.

One interviewee used commercial software to design inductors. He found that the user interface was unintuitive and included parameters with no physical interpretation. He commented that a scientist in the application area must be involved in designing a reasonable user interface. Other interviewees talked of steep learning curves and having to trick the software into what they needed it to do. Commercial software is mostly designed to address commonplace tasks. Scientists push the boundaries of the commercial software's capabilities. Some commercial software packages provide hooks for users to add specialized modules, but this can cause problems for scientists. One interviewee told of a commercial software supplier changing implementation languages several times, causing her to rewrite her modules each time.

An added risk in commercial software is unavailability of the source code and sometimes even the underlying theory. One civil engineer stated that such black box software is a problem in his field. Incorrect software could result in scientists making dangerously bad decisions based on its output data. Taking the supplier's word that the calculation is correct is a major risk. Even when the manufacturer provides a theory manual with the software, the scientist can't examine the code to ensure that the software is implementing the theory correctly. Some scientists write their own code rather than use code they can't see. Far from reinventing the wheel, as some would see it, the scientists see this as minimizing risk.

Open source software is another alternative, but interviewees also saw it as risky. One interviewee

If the software's purpose shifts away from just showing the theory's viability, risk shifts to the implementation.

commented, "the problem with open source is that it's free, and you get what you pay for." Interviewees also cited time performance and "shotgun-approach" design as problem areas in open source software. Some scientists share software with colleagues in their field, in a limited open source forum, mostly to reduce the code others have to write. However, our interviewees' use of open source application software was low.

Testing

For scientific software's two parts—the computational engine and the user interface—user interface testing was the more robust, when interviewees performed it at all. Any of our interviewees who were concerned about their user interface and understood that their users' knowledge didn't overlap their own performed extensive usability testing using the techniques we described earlier. Often they found new situations for testing.

Testing the computational engine proves much more problematic. Figure 2 illustrates both the extent and the limitations of typical testing. Scientists use testing to show that their theory is correct, not that the software doesn't work. In the scientist's mind, the code is tightly coupled to the theory; it isn't an entity of its own. So, scientists don't test the code in the computational engine for its own sake.

All our interviewees agreed that testing the computational engine is important. Several admitted that their testing practices were inconsistent, disorganized, and not repeatable. Some were more comfortable with their testing practices, stating that their testing was extensive, systematic, or iterative. Still others, who admitted that their testing was unsystematic, seemed unconcerned by it, possibly because of an overwhelming focus on theory.

Determining a test's success is a particular problem with computational software. Every scientist we interviewed had something to say about this problem. All testing requires an *oracle*—that is, something to compare against the test results. When the oracle and the results from the software don't match, the problem might be with the theory, the theory's implementation, the input data, or the oracle itself.

Interviewees mentioned several sources of oracles. Data might be available from industry or from laboratories in which researchers have taken measurements of some real event, and scientists can compare the code's output to these measurements. Although certainly the most preferable type of oracle, it has some problems. Measurements can be incorrect or incomplete. The circumstances surrounding the measurements can be incompletely understood. Often, obtaining data from real events is expensive, dangerous, and otherwise limited. As one interviewee put it, some software simulates situations that "you don't want to see happen." In such cases, scientists use their professional judgment to determine whether their output looks reasonable. All the scientists we interviewed doggedly pursued causes for their output not matching the oracle, but they focused on the theory, not the code.

Because of the complexity of transforming theory to code, determining a priori where boundaries or singularities lie is often difficult. Having two tests yield acceptable answers doesn't necessarily guarantee that points between or close to the test data will also yield acceptable answers. Notably, none of our interviewees mentioned this problem. This might be because scientists are unaware of the risk involved in the code itself. Another possibility is that the scientists don't have effective and efficient testing techniques to address this risk.

The issue of accuracy arose in a feedback forum we held after completing the interviews. Accuracy issues can affect the results of scientific software to the point at which different implementations of the same algorithm can give different results. Les Hatton and Andy Robert's case study[4] and Hatton's follow-up[5] show how this type of problem can render the software useless. Among our interviewees, handling accuracy relates to risk management. When choosing an approach, some interviewees relied on established solution libraries. Others rewrote software because they didn't trust software from other sources.

Our interviewees also didn't mention negative testing. As opposed to positive testing, which tries to prove that software is doing what it's supposed to do, negative testing aims to determine whether the software is doing things it's not supposed to do. One engineer who was conscientious in his testing said that a tester finds bugs by "doing something stupid … How can you do dumb testing unless you know what the dumb is going to be?"

Documentation

Scientists can use documentation to manage risk. Most scientists produce documentation related to the theory rather than strictly related to the code.

In every case, interviewees produced documentation of the theory underlying the software code,

often through journal or technical papers. In one case, the user documentation included a theory manual.

None of our interviewees created an up-front formal requirements specification. If regulations in their field mandated a requirements document, they wrote it when the software was almost complete. Customers of the scientific software might provide a vision statement, but nothing more. In all cases, customers ceded decisions about the software to the scientists, giving them broad sway in deciding what to include in the software.

None of our interviewees produced user documentation unless their software was intended for use outside their research group. Most users' knowledge overlapped that of the scientist/developer. Producing documentation is time consuming, and domain knowledge is available in other sources, such as textbooks or research papers.

However, one interviewee developed extensive online help files, including detailed glossaries and diagrams, geared toward nonspecialist engineering users. He assumed that the users had at least an engineering background, as he explicitly stated to his user community.

Another interviewee with users in a different field from his took a different approach to user documentation. This interviewee was aware of his users' backgrounds and how they used his software. He stated that he tries to keep the documentation to the length of a videogame instruction manual. "They won't read it anyway," he told us. One of his goals was to make the user interface intuitive so operating the software would require no documentation other than what's on the computer screen.

Another interviewee assumed users would readily understand how to use the software despite not being specialists in the same application domain. When users had questions about operating the software, the interviewee told them to read the theory manual. Obvious friction developed. He knew that the friction signaled risk for his software, but he didn't know how to fix it.

Only one interviewee (one of the "do it twice" projects) demonstrated his design documentation. During the rewrite, he produced extensive hard-copy manuals, primarily aimed at linking the theory to the software. He defined and cross-referenced data structures and variable names from the code to the theory and documented the code's architecture in detail. Two factors distinguish this project.

First, two engineers developed the software. One was an application-domain specialist, the other a self-taught engineer in software and another discipline. The latter spent considerable time learning the specialist's application domain. He became the code keeper while the first was the theory keeper.

Second, the theory was well established so the risk was in the implementation rather than in the theory. The focus was therefore on the code.

Implications

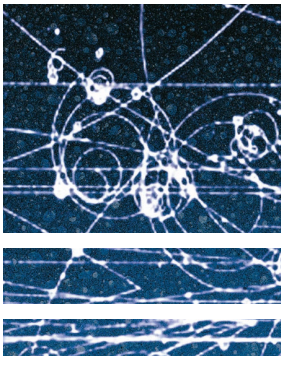
The complexity of developing not only scientific application software but the accompanying theoretical models is high. The risk of failure is also high. Scientific software customers choose developers who can best manage what's usually the highest-priority risk—the theory. Typically, they choose the scientist knowledgeable in the application domain.

Scientists develop software to address the risk in the theory. The software's purpose at this stage is research. Testing aims to show that the theory matches the oracle, whatever that oracle might be. If the test succeeds, the theory is acceptable. The software has achieved its purpose.

If the software's purpose shifts away from just showing the theory's viability, risk shifts to the implementation. At this point, testing must assess the implementation, not the theory. Most scientists miss this shift.

The lack of viable testing methods compounds the problems with testing scientific software. We need research in test-case selection methods that deals realistically with the lack of oracles, computational singularities, long runtimes, large input data sets, extensive output, and software with complex domain content. To be useful, the testing method must provide an appealing ratio of effort to bug-finding capability in the computational engine context. Computing research could provide valuable guidance in this area.

Scientists who take into account risks involved in how well the user understands their software have developed practical ways to drive requirements and perform usability testing. They also judge the necessary extent of different types of documentation to address this risk. These practices provide process alternatives that the computing community should study.



Knowledgeable implementation choices coupled with an iterative feedback development process provide a strong base for scientists to develop code. They perform risk management for the implementation when writing the code. The care they use in developing their

theories transfers to writing the code—that is, choosing conservatively or carefully assessing new tools for advantages.

However, some software systems are growing well past the ability of a small group of people to completely understand the content. Data from these systems are often used for critical decision making. However well designed, however carefully coded, the implementation could be strengthened by better testing methods. Although other issues should also be addressed, testing has the greatest potential for risk management. It's therefore one area in which the computing and scientific communities should be initiating new conversations.

References

1. D. Kelly, "A Software Chasm: Software Engineering and Scientific Computing," *IEEE Software*, vol. 24, no. 6, 2007, pp. 119–120.
2. M.B. Miles and A.M. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook* (2nd ed.), Sage Publications, 1994.
3. W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. 9th Int'l Conf. Software Eng.*, IEEE CS Press, 1987, pp. 328–338.
4. L. Hatton. and A. Roberts, "How Accurate is Scientific Software?" *IEEE Trans. Software Eng.*, vol. 20, no. 10, 1994, pp. 785–797.
5. L. Hatton, "The Chimera of Software Quality," *Computer*, vol. 40, no. 8, 2007, pp. 104–103.

About the Authors

Rebecca Sanders has just completed her MSc in computer science at Queen's University, Kingston. Her research interests include software engineering methods as applicable to the development of scientific application software. Sanders has a BSc in software engineering from McGill University, Montreal. Contact her at smitelf@gmail.com.

Diane Kelly is an assistant professor in the Department of Mathematics and Computer Science at the Royal Military College (RMC) of Canada. Her research interests are specific to scientific application software and include verification and validation, software design, and software evolution. Kelly has a PhD in software engineering from RMC. She's a member of the IEEE Computer Society and the ACM. Contact her at kelly-d@rmc.ca.