We are all of us in our daily lives heavily dependent on scientific software. For example, as we take our medicine, listen to the weather forecast, anticipate flying to warmer climes, turn up the thermostat on our central heating, we should stop to acknowledge the underlying software:

- The databases of protein structures which inform the function of proteins and hence the design of drugs
- The models of (for example) wind patterns which enable the simulations to inform the weather forecasts
- The models which inform the design of aircraft parts so that they can withstand the conditions of flying at 35,000 feet
- The geological models which inform the siting of oil wells.

Professional software developers don’t usually have the necessary deep understanding of structural biology, meteorology, aerodynamics or geology: it is thus necessary for scientists in the appropriate disciplines to be heavily involved in developing such software. Indeed, the fore-runner of the software now in use on an everyday basis is very probably software developed in the lab by scientific end-user developers.

End-user developed software is generally regarded with some degree of caution by professional software developers (McBride & Wood-Harper, 2002). It is frequently buggy. For example, Raymond Panko, with reference to spread sheets, talks not about whether a given spread sheet contains a bug but rather the number and type of bugs it might contain, see http://panko.shidler.hawaii.edu/My%20Publications/Whatknow.htm. End-user developed software is also frequently difficult to maintain and difficult to use with other software. In order to address problems such as these, it might seem a good idea for professional software developers and scientific end-user developers to work together so as to exploit the software development expertise of the former and the scientific expertise of the latter. However, such a situation is potentially fraught with communication and collaboration difficulties over and above those normally experienced between developers and end-users due to the clashes between two cultures, professional software development versus scientific end-user development (Segal, 2009).

The aim of this special issue is to begin exploring the concerns of scientific software development from the perspective of the scientist, both as end-user and end-user developer. Perhaps the two most fundamental questions to ask are:
• How can software engineers best support scientific end-user developers in providing suitable tools, methods and technologies? The emphasis here is on the word ‘suitable’. Many authors point out that what is useful in mainstream software developments is totally inappropriate in scientific software development (see, for example, Basili et al., 2009).

• How can software engineers and scientists work together to develop software so as to exploit the expertise of both optimally?

The current situation is that we are very far from providing full answers to either of these questions, (e.g., Kelly, 2007) but it is hoped that this special issue will, at the very least, raise awareness of these concerns.

The papers chosen for this special issue are as follows.

Carver points to an important factor that scientific end-users often face: the bottle-neck in “time-to-solution” (that is, the time needed to affect a software solution to a scientific problem) is not the execution time of the software. Instead, it’s the time required to prepare representations of the scientific model that are suitable for computational solution. Often, this can take months of careful consideration and trial-and-error. Carver describes the development of a specialized tool intended for use with three-dimensional models for such studies as fluid flow or complex design structures. The tool helps the end-users develop and explore their three-dimensional models before submitting them for computational studies. As is true of all scientific software, this tool must return output that is correct, but for its success in adoption, it must also be usable. Usability is heavily influenced by such factors as user experience, the right level of user support, and the handling of special cases. This software development is addressed by an approach that may appear “ad hoc” to software engineers, but is actually driven by a set of goals important to the scientific end-user developers.

Chilana et al. discuss some of the problems in developing software to support multidisciplinary clinical translational medicine. ‘Clinical translational medicine’ aims to bring biomedical sciences and clinical practice closer together by enabling rapid translation of basic research results into meaningful health outcomes. The basic research agenda and, to a certain extent, its execution (for example, the choice of subjects for trials) are both informed by clinical data from diverse sources. The challenge here is to build software to support the deployment of these various data sets by scientists of various disciplines. This involves actively engaging the scientists in the development.

Hochstein et al. conducted a case study of a computational engineering organization to assess the feasibility of performing computational engineering in the cloud. The motivation for this study was that smaller engineering organizations that may not have the resources to build their own cluster are currently unable to reap the benefits HPC machines provide in terms of performance increase. The study showed that while computational engineering in the cloud shows a lot of promise in reducing computation time, there are some issues that may prevent its widespread use at this time. These issues were all related to the cloud infrastructure rather than to the actual engineering software itself. First, the low bandwidth to and from the cloud causes long delays during data transfer. In the case study, this delay overwhelmed the benefits of reduced processing time provided by the cloud. Second, the organization had to deal with licensing and VPN issues that were caused by the fact that potentially sensitive data would be processed and reside on servers that were outside the direct control of the engineering organization.

Segal’s and Morris’s case study describes the development of a laboratory information management system that supports a community of biologists. In this particular development, scientific end-user developers (biologists) worked with professional software developers. Differences between the two communities in established work practices, difficulties in transferring tacit knowledge, mismatched
value systems and priorities, and lack of trust provided challenges in developing a successful system. One factor in the eventual success was the role played by the scientific end-user developers in bridging the communities of scientific end-users and the software developers.

Although superficially quite different, all these papers share a common focus in considering how scientific software development differs from a typical commercial development, and how the problems arising from these differences might be alleviated. A special issue such as this one can only scratch the surface of these concerns, but we feel that investigating them more fully is essential, not only to support scientists in their research but also to support all of us in our everyday lives.

Kelly analyses how well-suited plan-driven and agile development methodologies are to scientific software development. Two characteristics of scientific software help illustrate some of the issues. First, scientific software development needs to explore the solution space rather than the problem space. A substantial portion of the code must be completed before this exploration can occur. Some aspects of plan-driven methods and some aspects of agile methods fit this situation. A second important factor that affects the applicability of different development methodologies is the Iron Triangle Model (scope, schedule and cost) with Beck’s addition of quality as a fourth attribute. In scientific software the scope and quality are typically fixed while schedule and cost can vary. In contrast, for typical software development cost and schedule are fixed while quality and scope are allowed to vary. To provide a framework for the analysis, Kelly uses two models upon which to base this analysis. She uses the five factors in Boehm and Turner’s model (Boehm & Turner, 2004) and the twelve practices of XP as described in Beck, 2000. Using both of these models, Kelly concludes that in some respects agile methodologies are a good fit, while in other respects they are poor.

REFERENCES


Diane Kelly is an associate professor at the Royal Military College of Canada (RMC). She has over twenty years of industrial experience in scientific software development and has spent the past ten years combining academic research with what she's learned from her industrial experience. Dr. Kelly has a BSc in pure mathematics and BEd in mathematics and computer science, both from the University of Toronto. Her MEng and PhD are in software engineering, both from RMC. She is a member of ACM and a senior member of IEEE.

Jeffrey Carver received the PhD degree in Computer Science from the University of Maryland. He is an assistant professor in the Department of Computer Science at the University of Alabama. His main research interests include software engineering for computational science and engineering, empirical software engineering, software quality, software architecture, human factors in software engineering and software process improvement. He is a Senior Member of the IEEE Computer Society and the ACM.