Lessons Learned from a Scientific Software Development Project

Chris Morris, Science and Technology Facilities Council, UK

Judith Segal, Open University, UK

In the mid-90s, I was working for a company transitioning to agile software development. We hadn't considered that our biggest problem in this change was that developers would be talking directly to customers. After some small disasters, we realized that somehow we needed to transfer the expertise and experience of our business and marketing teams to developers. When I first read the current Insights article, I immediately recognized many patterns in the list of lessons learned and that what these authors have to say is much broader than "scientific" software -- it's about any kind of software, actually, any kind of development effort. —*Linda Rising, Associate Editor*

WHEN ONE OF us (Chris) accepted the post of project manager/lead developer on a scientific software development project, he was blissfully ignorant of the challenges he faced. Six years later, he emerged bloody but unbowed with a software product that now has a respectable and growing user base. This article is the story of his journey as told to Judith and the lessons he learned and shared with her along the way.

Project Background

The project was to develop a laboratory information management system (LIMS) for biologists working on protein structures. The process by which they search for such structures involves several steps in a laboratory, with each step made up of multiple variables (the concentration of various chemicals, temperatures, and so on). These steps often aren't well understood and frequently lead to failure. Practice has recently changed in many labs so that each step is performed as a set of trials done in parallel, with each trial differing slightly in context—that is, in variable value and type.

The first requirement for the LIMS was to record the context of each of these trials, whether they were successes (so they could then go into a library of known procedures) or failures (so that scientists could be deterred from making the same mistake twice). To develop this LIMS, the chief customers—generally, the heads of the labs involved—provided Chris with a list of very high-level requirements, an implemented model of the science (that is, of the relevant biochemical objects and the relationships between them) along with an automated mechanism for generating the database and the interface, and a development team consisting of people with backgrounds in either software engineering or structural biology. The development plan followed an iterative incremental feedback model with scheduled releases spaced at every few months; users could provide feedback to inform the next release, which would also implement further requirements.

What could possibly go wrong? Chris soon found out.

Lesson 1: Choose Carefully between Soon and Good

It's generally agreed that it's important that iterative incremental feedback development models quickly deliver a piece of software, which, while of limited functionality, offers something of value to the customer and the user. This delivery, it is hoped, will encourage customers and users both to trust the developers and to engage with the development. As to delivering something good, there are many different quality goals that might be met. Such goals vary with each development context: Does the output have to be correct to three decimal places, or does it just have to follow some trend? Does the software have to be easy to install? Is software flexibility more important than absolute correctness? Is portability more important than efficiency?

At the beginning of the development, Chris presented his customers with a list of quality goals that he wanted them to rank. They didn't. Whether this was because they had no time or because they thought it was Chris's responsibility wasn't clear. The articulation and ranking of quality goals emerged gradually, through a deeper understanding of the context of work. However, one quality goal immediately became apparent as being of the upmost importance—deposited data shouldn't be lost or corrupted. Fortunately, Chris thinks he got that right from the beginning. Another fundamental quality goal was that scientists needed to find the LIMS useful and deposit their data in it in the first place. Satisfying this goal turned out to be far more of a problem.

Asking users about their requirements wasn't informative.

Lesson 1a: Customers Don't Always Know User Needs

One impetus for delivering the first version of the LIMS quickly was that the scientists were already producing data that needed managing. It seemed clear to Chris that he should base this first version on the implemented data model and its associated mechanism as supplied to him by the customers. This produced a piece of software that was incredibly complicated to use, with a user interface (UI) driven by the data model rather than user requirements. The users refused to have anything to do with it, so any idea of a feedback model of development seemed doomed. Chris soon realized that he needed to consider the software from the user viewpoint—that is, the bench scientists who entered the data, as well as from that of the customers providing the money.

Lesson 1b: Asking Users Doesn't Establish Requirements

The list of requirements Chris originally received proved to be at too high a level to guide implementation.

His first attempt at more detailed requirements gathering was to ask one member of the development team, a structural biologist, to draw up a set of use cases in collaboration with a user group. This set would have sufficed in a situation such as the development of new software to support the administration of insurance policies—that is, somewhere with well-understood goals, practices, and software embedded in the practice. In this particular situation, however, use cases turned out to be a flop. The developers didn't fully understand the variations in practice between labs or the possible ways to embed data management software within practice. One of the original goals of the LIMS as stated by the customers was to enable data sharing and mining across the whole protein structure community. But a sizeable number of the users were PhD students, and their primary goal was to get their PhD—as far as they were concerned, the wider scientific community could look after itself. More senior scientists were similarly ambivalent about this sharing and mining goal. A very limited pot of public money funds their labs, and they essentially compete with each other for this pot so it's easy to understand why they view the issue of data sharing with a certain degree of ambivalence.

Presumably because of this tension between the goals of individuals and individual labs and the community, asking users directly about their requirements wasn't always very informative. The development team experienced repeated instances of the following cycle: a user group said they couldn't use the software because a particular feature wasn't implemented. The team implemented it. Then the users said they still couldn't use the software because of the absence of some other feature. The team implemented whatever that was. The users then said they still couldn't use the software because it lacked something else. And so on. Chris concluded that users must have had some deep-seated reason for not using the software and were unwilling or unable to communicate it to him.

Another reason why asking users for requirements doesn't always work is because they're experts in (in this case) protein science, not in imagining how software can help them. In particular, they don't know which software features are, or are not, feasible. One way of addressing this challenge is for developers to acquire a deeper understanding of context, but this takes time and resources, and is in direct conflict with the necessity of delivering value quickly. This brings us to lesson 1c.

Lesson 1c: Start from What Users Do Now

You can still deliver software of value to users even when your understanding of the work context is incomplete.

In retrospect, Chris wishes he'd started by asking users how they used their computers and moved forward from there. Then he could have delivered as part of the LIMS a subsystem for, say, keeping stock of and reordering chemicals. While not contributing to the stated primary aim, Chris thinks this would have had the effect of gaining user trust and engagement while buying time to improve his understanding of the work context.

Chris was aware that one big barrier to uptake of early versions of the system was its perceived lack of usability—the UI, in particular, was an inconsistent mess. Chris initially responded to this by asking the developer responsible for UI to write a "design guide," but this turned out to be a write-only document. It wasn't used, and even had it been, it would have only reflected the views of the developer-writer and not of the users. Later, Chris realized that all the users were familiar with the concept of entering data into lab books, so the team redesigned the UI based on that metaphor. This redesign appeared to be very successful.

As the development team's understanding of the work context improved, they were able to construct tools (such as personas) that proved helpful in prioritizing implementation. This increasingly deep understanding also enabled the developers to be creative with the software and introduce innovations (such as providing templates for procedures as opposed to a static database) that the users found effective.

Deep understanding enabled the developers to be creative.

Lesson 2: Good Communication between Developers Can't Be Taken for Granted

All developers, we think, are aware of the need for good communication between developers and users and are prepared to put in the resources to support this. However, many of them take for granted good communication and trust among members of the development team. Chris couldn't do this. His team members comprised nine different people at five different sites with five different line managers (some of whom headed user labs) and five different nationalities. Some members had backgrounds in software engineering, some in structural biology—for others, it was their first job after university. This variety offered plenty of potential for disagreement, for example, when line managers in user labs prioritized the needs of their own institution. Communication was difficult and not just because of cultural norms (people from country x tending to be more forthright than those from country y)—there was also a lack of common knowledge, with some people having no idea what "black-box testing" means and others a bit fuzzy about the term "gene." (We've since discovered that there is considerable discussion among biologists about exactly what a gene is, but we're referring here to everyday use of the term). This mismatch of goals and common knowledge together with general difficulties in communication led initially to what Chris perceived to be mistrust by the development team of him as the project manager.

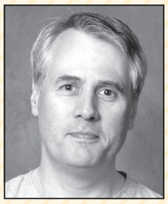
He took positive steps to address this situation by commissioning annual training in soft skills such as conflict management and appreciation of cultural differences. Some developers described this training as very useful; others were more circumspect. Whether by means of this training or by dint of the experience of working together over time to overcome the various difficulties, the team succeeded in gelling approximately half-way through the project.

Lesson 3: Plan for Sustainability from the Beginning

Scientific software developments are often funded by research money that's only available for a certain period of time—in this case, five years. Later, as the technical and scientific environment changes, the software remains useful only if it's maintained. But who is to pay for such maintenance? One way to address this question is to grow the user community so that you gain a critical mass of scientists eager to use the software after the period of research funding ends. Alternatively, it might be possible for the software to be licensed so that it's free for academic use, but commercial users have to pay, with the latter subsidizing the former.

Clearly, you need to grow a user community quickly. In retrospect, Chris wishes that he'd prioritized some development features in light of this need from the beginning.

ABOUT THE AUTHORS



CHRIS MORRIS is a software project manager at STFC, developing data management software for life scientists. Morris has an MA in mathematics from the Queens College, Oxford. Contact him at chris.morris@stfc.ac.uk.

JUDITH SEGAL is a senior lecturer in computing at the Open University, UK. Her research interests include the impact of software engineering on scientific software development, the practice of software development, and interaction design. Segal has PhD in mathematics from the University of Warwick, UK. Contact her at j.a.segal@open.ac.uk.

Lesson 4: If Adopting Software Changes Work Practices, Its Development Has Special Challenges

Compare the use of the LIMS with, for example, the use of software to support the working practices of the insurance industry. Insurance companies have used software to support their record keeping for decades. The same isn't true for lab scientists: keeping records in lab books has been engrained in their work culture for centuries. It's only recently, with the growth of collaboration and data sharing in the biological sciences, that the advantages of electronic record keeping have become apparent. The implication is that developers in this context can't use well-established techniques without a lot of thought. It might also be that the iterative incremental feedback development model, often taken as the one most suitable for scientific software development because it mirrors the essential nature of scientific discovery, should be modified so that it incorporates an extended period at the beginning of the development to understand the work context.

Clearly, we learned a lot in six years, and we would be very pleased to hear about experiences from other scientific software developers.