# Scientific Software Development at a Research Facility

**Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson,** *Daresbury Laboratory*

Software engineers at Daresbury Laboratory develop experiment control and data acquisition software to support scientific research. Here, they review their experiences and learning over the years.

Developing software in a scientific environment has a particular set of challenges but is particularly fulfilling, if at times frustrating.

Daresbury Laboratory is an internationally renowned scientific research laboratory and is the home of the Synchrotron Radiation Source (SRS) research facility, (www.srs.ac.uk/srs). At Daresbury the Synchrotron Radiation Computing Group (SRCG) develops control and data acquisition software for the SRS's experimental stations. Many of the group's members have higher degrees in scientific disciplines, often with some software development experience as part of that degree. Others have computer science degrees. Each member has roughly 20 years' experience working closely with scientists in developing software for scientific research environments.

Here, we review some of that experience, especially with reference to the Generic Data Acquisition (GDA) project, an object-oriented system for data acquisition and experiment control. Areas of particular interest include our experience with and use of development methodology and group management, including agile programming and Extreme Programming (XP); IDEs and tools for version control, issue tracking, document storage, testing, continuous build, and scripting; and internal and external collaborative software development.

## Organizational structure

The Science and Technology Facilities Council (STFC, www.scitech.ac.uk) is one of seven UK national research councils that fall under the Department for Innovation, Universities, and Skills. Its aim is to enable a broad range of scientists to do high-quality research through a variety of central facilities, expert assistance, and services. It manages several major UK facility sites including the Daresbury Laboratory in Cheshire, the Rutherford Appleton Laboratory (RAL) in Oxfordshire, Edinburgh's Astronomy Technology Centre, and the UK observatories in La Palma and Hawaii. It's the main funding source for physics and astronomy research.

### Daresbury Laboratory and the SRS

The Daresbury Laboratory has been home to the world's first dedicated SRS since 1979. Synchrotron sources deliver intense light beams with wavelengths extending from the infrared to high-energy x-rays. They offer a range of experimental techniques, including x-ray diffraction and spectroscopy for scientific and engineering research. These light beams are delivered via beam lines of components that alter the light delivered to experimental stations. An example component would be physical slits that alter the beam's shape. The experimental stations host equipment that lets scientists probe a sample and measure some of its diffraction or emission.

0740-7459/08/$25.00 © 2008 IEEE

Having collected and analyzed the experimental data, researchers can determine some of the sample's properties, such as a protein's 3D structure.

After a decade of successful operation, the SRS was becoming outdated and, in some scientific areas, it was becoming increasingly difficult to meet experimental requirements. In 1993, an independent panel chaired by Michael Woolfson (Engineering and Physical Sciences Research Council) commissioned a report to assess the academic community's needs. This report identified the need for a new and improved UK-based synchrotron to supersede the SRS. Synchrotrons are expensive facilities, so funding for only one was viable. A feasibility study was undertaken, and in 2000, the government decided to site the new synchrotron, the Diamond Light Source (DLS), at RAL rather than at Daresbury.

## Diamond Light Source

The DLS (www.diamond.ac.uk) is sited at RAL and is a joint venture between the STFC and the Wellcome Trust (the UK's largest charity and one that funds innovative biomedical research). The DLS began operating in January 2007, and from December 2008, when the Daresbury SRS ceases to operate, it will be the UK's only synchrotron source.

## Data acquisition software development

The Daresbury Data Acquisition Group was first established in the late 1970s in preparation for the SRS. The group developed data acquisition software in a project-specific way using a small set of generic, low-level libraries. At the time, scientists, managers and developers perceived little overlap in software requirements for different experimental techniques.

In the 1980s, under pressure to provide dedicated support to the laboratory research scientists, each developer became part of a science team, supporting a specific scientific technique. Consequently, these techniques began to employ differing hardware and software, making each software developer a single point of failure and resulting in similar requirements being met in different ways. Developers collaborated informally, sharing advice, support, and some libraries and drivers.

Increasing informal collaboration in software development, thus maximizing knowledge transfer, and specific projects led to the SRCG's formation in 2002. The SRCG's focus was now on generic data acquisition software spanning many techniques and a broad range of hardware. The group developed the Generic Data Acquisition (GDA) software to meet these requirements. Unlike previous applications, the GDA software development was project-managed, and developers used agile development methods. This project minimized the duplication of effort and aimed to meet the requirements of any SRS experiment through a multilayered, configurable set of software building blocks. Researchers could, however, use and adapt GDA for other data acquisition and control environments such as laser facilities, small laboratory light sources, and automated data acquisition systems in general.

In the same year, the DLS set up its own data acquisition group. DLS management decided to use GDA for data acquisition at the new synchrotron, thus requiring substantial knowledge transfer from Daresbury to the DLS. Their collaboration was legally complicated and needed a formal license agreement between the two organizations. The Daresbury SRCG and the DLS data acquisition group agreed on and documented collaborative practices and development strategies, which had a positive effect on the GDA project—management improved, releases became formalized, and collaborative efforts increased internally as well as between Daresbury and the DLS.

## Acronyms

| | |
|---|---|
| BIOXHIT | Biocrystallography (X) on a Highly Integrated Technology Platform |
| CCT | Change Control Team |
| DL | Daresbury Laboratory |
| DLS | Diamond Light Source |
| EPICS | Experimental Physics and Industrial Control System |
| EPSRC | Engineering and Physical Sciences Research Council |
| GDA | Generic Data Acquisition |
| RAL | Rutherford Appleton Laboratory |
| SRCG | Synchrotron Radiation Computing Group |
| SRS | Synchrotron Radiation Source |
| STFC | Science and Technology Facilities Council |

## GDA: Conception, origin, and purpose

The concept of the GDA software suite arose from an object-oriented (OO) analysis and design course and the formation of the SRCG. Following this course, the SRCG formed an internal working group to design software for the automatic beamline alignment project and a new general-purpose data acquisition system.

To help with the new working process, we hired an external consultant to act as a mentor. The consultant suggested an incremental and iterative development process, which we adopted, moving away from the waterfall model. This approach helped motivate the team because it allowed real

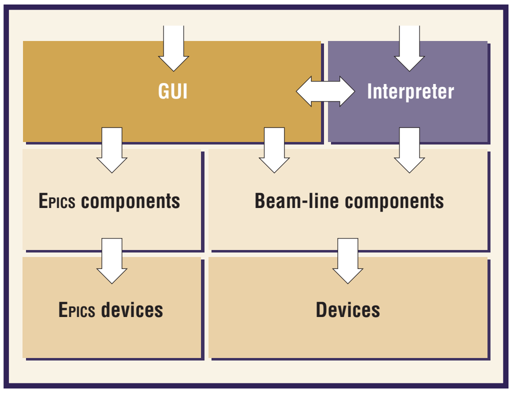


**Figure 1. The core Generic Data Acquisition software's three-layer architecture. The architecture shows how the components connect and build to form a complete system.**

deliverables in parallel with analysis and design. Early attempts at OO code development had been slow owing to analysis paralysis.

Another major achievement of these projects has been that we work better as a team and have common goals. We pooled software into a common repository and encouraged code reuse. We had a large investment in existing code, and refactoring this to fit into the new scheme proved successful—for example, wrapping that code behind Java Native Interface or communicating with remote daemons via TCP/IP.

In developing a software design for the new projects, we used an OO approach and adopted Java as the programming language. Using this, we designed a layered "pluggable" mechanism to create a flexible solution that would be independent of both the operating system and underlying hardware.

The GDA has a three-layer core (see Figure 1) comprising low-level devices, beamline components, and a user interface. It provides a single software framework for all beam-lines and is flexible, adaptable, and configurable. A common look and feel reduces the learning curve for users. A Corba backbone allows distribution across multiple systems and the separation of control and interface. We've implemented configurable support for many hardware types (motors, detectors, temperature controllers, and so on) and can interface using several mechanisms, including serial data communication, Ethernet, and EPICS (Experimental Physics and Industrial Control System). Anyone responsible for maintaining an installation can easily configure GUIs and other components (such as hardware devices) using XML. This gives us an open framework that's easy to extend and a system that's relatively simple to maintain.

## Development methodology and group management

The reformed SRCG group needed to adopt a design and programming methodology that would suit the requirements of scientific data acquisition. Computing for scientific research often lacks firm specifications and is subject to rapid, unexpected requirement changes. Agile programming, with its small design phases and regular small deliverables, seemed ideal for our circumstances. At the time, XP was the accepted methodology.

To enhance group cohesion, we arranged for everyone to attend two courses, which covered advanced Java, unit testing, and XP's main aspects. We believed that we'd benefit most if everyone attended the same courses at the same time. This meant that we'd all receive the same basic knowledge of and exposure to the new methods. Learning together facilitated our team-building process, and we recommend this coordinated training approach to others.

A number of us have attempted to adopt the basic XP practices. The overall difficulty we found is that a lack of coordinating control interferes with adopting group-wide change. We've always tried to operate by consensus; with a group of people trained for, and used to, making their own decisions, this can be difficult. Here, we summarize common XP practices and explain how we've tailored them for our own purposes:

- *Planning game.* We've always had to interpret user requirements ourselves and have had little success getting users to formalize them. We prefer requirements organized as a project plan, which might require breaking down into smaller tasks (stories). Developers estimate requirements, and users define priorities. We've achieved success by grouping requirements into regular releases with dates to match priorities. Commissioning pressures makes it difficult to stick to these releases.
- *Small releases.* Evolutionary development fits the scientific research environment. Regular iterations and deliverables let users influence development and form requirements during the development process.
- *Simple design.* We strive to do no more than is necessary to fulfill requirements. Code that goes beyond requirements has, in some cases, been found to be difficult to understand and maintain. Some has ended up getting deleted and replaced by code that simply fulfills the requirements.
- *Testing.* This practice has seen much talk and insufficient action, to our detriment. Developers have introduced bugs that automated testing could have detected before the code went into production. We've recently begun to ad-

dress this. The agile community strongly advocates test-first development. Insofar as we've tried this, we agree, and in the future, we plan to implement this more. We should use regular, automated project build, testing, and test-first development from the start of a project.

- *Refactoring.* We've found that code refactoring can be a powerful tool for improving and maintaining a code base. Obviously, we must maintain a balance between this and adding new features. The automatic refactorings of IDEs such as Eclipse are very helpful.
- *Pair programming.* Some group members operate a loose pair-programming system and find it very useful. This technique requires buy-in from management because extra resources are required during the development phase to reap benefits later on. Limited resources and differing short-term goals have sometimes prevented pair programming. These reasons, together with traditional suspicions about inefficiency and personal preferences, have made it impractical to enforce pair programming across the group.
- *Collective ownership.* Because we believe anyone should have the right to modify the code, we've removed all signs of individual ownership, including author names in source file headers. However, some remnants of our old working methods, customers' reluctance to change, and the need sometimes to react quickly to problems have all led to individual developers taking control of some areas of the project code.
- *Continuous integration.* As a group, we check new code into our repository regularly. This keeps code consistent and has helped detect bugs early. However, all developers must ensure that modifications are properly tested if there is insufficient automated testing. We've found problems when this doesn't occur. However beware the "blame culture," which can be very destructive.
- *Coding standards.* We adopted standards early on that have helped readability and understanding. We configured the Eclipse IDE to format code to a particular standard, and this was very helpful.
- *Five-minute daily meeting.* This proved unsatisfactory, most likely because we didn't use the full XP approach. We've found weekly meetings more profitable.

The group dynamics proved interesting. People who'd been used to autonomy were initially keen to be involved in the collaboration. However, some members were reluctant to take on the responsibilities associated with working this way and the constraints of adopting group standards for coding style and working practices. Conflict sometimes occurred between group members with different views of how to do things, and some were reluctant to compromise. With no strong overall control to enforce compromises, the group neglected some areas of conflict, even though those areas were important to the project.

We also had to change our relationship with our customers—that is, the scientific community. With the move to greater collaboration, the scientists had to adjust to viewing us as a team. We've had difficulty selling a generic product to some of our users. The perceived interference of wider issues with users' own needs has on occasion been seen as a hindrance to localized progress. Despite these problems, we've developed an appreciation that working as a group can generate greater momentum in the project.

When the DLS data acquisition group joined our project, the need to collaborate over two remote sites meant we had to introduce stricter project management. We needed an issue-tracking system for formally recording bugs and new requirements, and we instituted a release schedule to prioritize and target bugs and new developments for particular releases. We appointed a change control team (CCT) to allocate bugs to developers and keep the storyboard up to date. The storyboard is an arrangement of wall panels describing the next few releases; the CCT place a Post-It note representing each entry in the issue-tracking system on the board according to the release to which it will be assigned, together with the developer working on it. As different developers work on different parts of the project, the storyboard gives us a single point for information on the project's status and is a visual representation of "where we are now." Although this was controversial, it's been a success.

To bring the two groups together, we arranged residential workshops. The first aimed to bring the new collaborators up to speed on the project. Subsequent workshops covered specific parts of the project that were causing problems or required considerable development.

**The perceived interference of wider issues with users' own needs has on occasion been seen as a hindrance.**

## Tools

We've used many tools in the GDA collaboration; Figure 2 shows these tools' categories. Our policy is, whenever possible, to use open source and other free software that runs on both Windows and Unix. Our choices are influenced by case studies of successful tool use and personal recommendations.

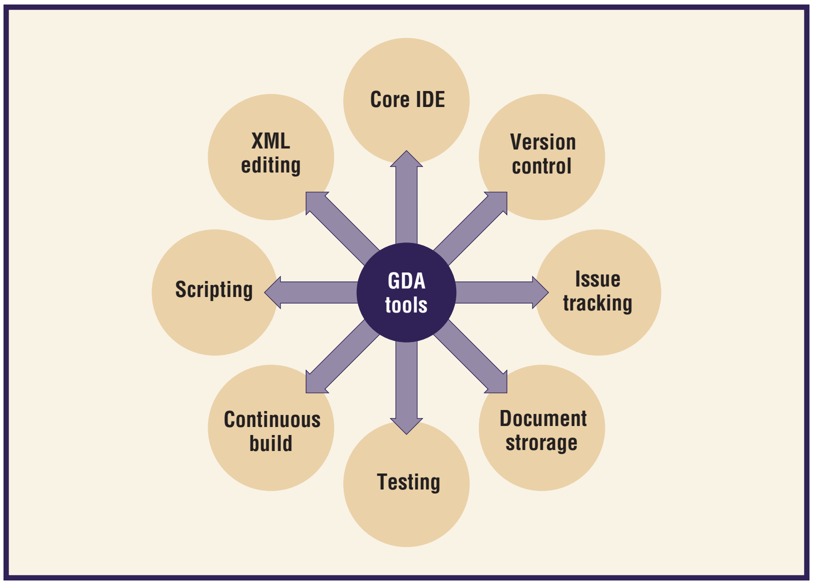


**Figure 2. The various categories of tools used in the Generic Data Acquisition project**

### Core IDE tools

The core IDE provides a language-sensitive editor, compiler, symbolic debugger, and runtime environment. Initially we used Java and Javac from the command line, with Emacs as our editor. By mid-2004, IBM's Eclipse had gained general approval within the SRCG, so we formally adopted it. The DLS group had performed its own evaluation and chosen Borland JBuilder because of a possible need for future database connectivity. The collaboration continued to use both IDEs for a while, but the DLS group eventually adopted Eclipse as well. The move to one IDE simplified communication between the sites and led to greater overall code consistency.

### Version control tools

Any group needs a version control system to safely store files being produced by several developers. A few of us had used tools such as the Revision Control System (RCS) or Concurrent Version System (CVS) in our own projects. A concurrent development process was the model that best fitted our working practices. A revision system that required that files be locked while checked out (such as RCS) was inappropriate. The numerous open source projects utilizing CVS persuaded us to adopt it in 2002, and the adoption of Eclipse provided built-in support. Early on, we formulated a revision-control policy that we thought would combine core source code protection with flexibility for developers.

### Revision control policy

Our revision control policy had several characteristics:

- Some developers were designated administrators.
- Administrators maintained a lock on GDA classes that affected everybody—developers modifying these classes would ask for them to be unlocked for check-in.
- The CCT provided decision making and allocated features and bugs to developers.
- Developers could allocate urgent bugs on target systems individually, without submitting them to CCT review.
- An exclusive Web-based "token" ensured sole repository access on checking in.
- Developers normally committed to the trunk between planned releases.
- Releases had suitably named branches, letting developers commit bug fixes to their tips.
- Major refactoring or features that impact other programmers were given a temporary branch that could be merged with the trunk later.

The revision control policy for the repository structure (see Figure 3) has stood the test of time; we still adhere to it today. Less successful were CCT procedures and the locking of the GDA core. This was perceived to impose restrictions on developers. Some developers still felt ownership of parts of the code. Time constraints imposed by the locking procedure caused bottlenecks and led to frustration. Code locking has since been discontinued, administrative time was reduced, and the system is working well. We migrated to Subversion (SVN) and began using the Eclipse subclipse plugin.

### Issue-tracking tools

A project involving several developers needs good communication, and much of this must be made permanent. Issue-tracking systems provide "tickets" for bugs and new features in a project and create an audit trail of activity that ties in with the version control system's contents. We required a lightweight, open source solution and chose Bugzilla because Daresbury was already using it successfully in various collaborative science projects.

### Document storage tools

As a reference for project developers and other involved parties, static documents must be stored and managed by a document storage system. We've researched availability and tested several document-storage tools, including storage on plain file servers, in revision control repositories and using various Web-based systems. At various times during this process, we've suffered from these problems:

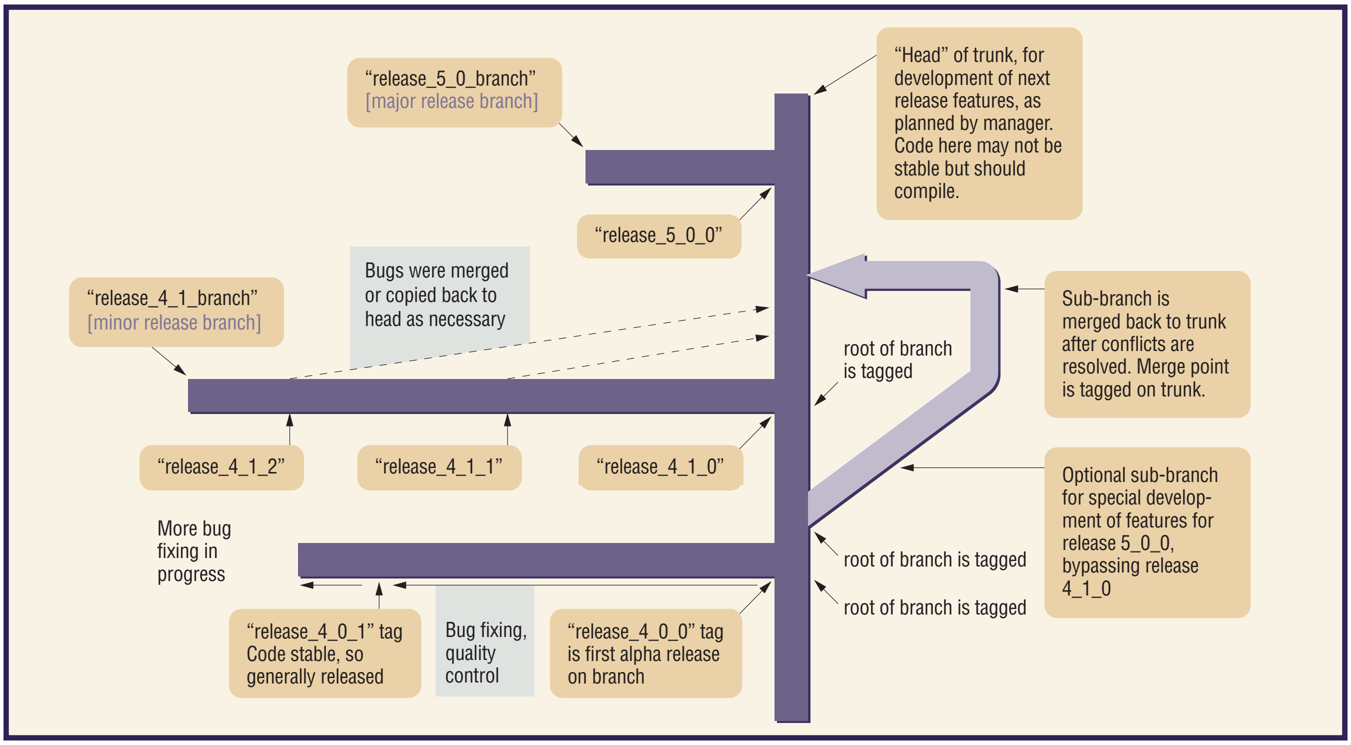

"Head" of trunk, for development of next release features, as planned by manager. Code here may not be stable but should compile.
"release_5_0_branch" [major release branch]
"release_5_0_0"
Bugs were merged or copied back to head as necessary
"release_4_1_branch" [minor release branch]
root of branch is tagged
Sub-branch is merged back to trunk after conflicts are resolved. Merge point is tagged on trunk.
"release_4_1_2"
"release_4_1_1"
"release_4_1_0"
Optional sub-branch for special development of features for release 5_0_0, bypassing release 4_1_0
More bug fixing in progress
root of branch is tagged
root of branch is tagged
"release_4_0_1" tag Code stable, so generally released
Bug fixing, quality control
"release_4_0_0" tag is first alpha release on branch

**Figure 3. A schematic of the Generic Data Acquisition repository structure and release policy. This shows main developments on the trunk and each major and minor release as a separate supportable branch, giving rise to point releases at the branch tips.**

- unclear procedure and location of tools due to instability in policy;
- site-specific procedure and policy;
- duplicated documents in different tools;
- incompatible versioning across different tools;
- difficulty in using or understanding tools;
- difficulty in "selling" new tools to busy developers;
- problems keeping documents updated and compatible with other documents and software releases;
- problems tracking and updating Web pages required as part of a large organization (public, corporate, departmental, and private Web sites);
- lack of deployment and clarity of responsibility for documentation areas;
- lack of resources to deal with documentation; and
- poorly defined purposes for online documents, tools, and Web pages.

Having resolved most of these issues, we now follow a simple policy at Daresbury:

- utilize the "knowledge tree" system for project management and computer systems documents;
- use SVN for code, deployment configurations, license and distribution information, and developer and user documents stored as the source form (such as Word or Excel); an essential set of files for building and running an installation;
- employ a wiki for processed stable documents in pdf or html format (read-only) and open forum and new document areas (writable); and
- keep Web sites to a minimum, storing only necessary, abstract, and medium-to-long-term information such as group purpose and contacts.

## Testing tools

As we mentioned previously, the GDA development process has insufficiently emphasized testing. Until recently, we had only about 50 JUnit3 test classes. Recognizing this as a problem, we proposed a tool set for testing that included TestNG as the testing tool, given its superior feature set and ability to perform higher-level tests as well as unit tests. However, the wider use of JUnit4 and a slightly poorer integration of TestNG in Eclipse led to us selecting JUnit4 as our unit test tool. We held a joint testing workshop in October this year, which successfully helped improve test coverage, and we've started using Emma to measure target-class test coverage.

## Continuous-build tools

We've long seen the need for a tool to continuously check repository modifications and ensure that the code compiles and builds into a jar. We initially chose Maven because it offered many extras, such as code analysis and Web server metric

**Collaborations work well while development speeds are in step across the collaboration.**

displays. However, we found administering it to be complex, and it fell into disuse when the developer responsible for it left Daresbury. We believed CruiseControl offered a lightweight alternative. After evaluation, we began using it to run the JUnit4 tests as part of the build process on dedicated Windows and Linux PCs.

## Scripting tools

Synchrotrons have many expert users who value the ability to tune control loops and data acquisition operations from a command line. This is also highly desirable for core software engineers when programming basic experiment control and data acquisition, because solutions can be delivered and modified quickly and tests easily written. By using a Jython interpreter as a scripting engine from GDA, we've created a scripting tool and given ourselves and our users this functionality. We've used this embedded interpreter for both testing GDA Java objects and with PyUnit as a unit test framework for Python extension commands in GDA.

Jython's early versions lacked system functions present in true Python, so we chose Python as a tool for writing GDA launchers, a launch validator, FTP clients, and backup clients. We set up the launch validator on test machines and now use it to ensure that the GDA simulator configuration starts successfully without errors or warnings.

## Collaboration

Throughout Daresbury's history of data acquisition computing, we've used collaboration to a varying extent as a strategy to mitigate the effort in developing software solutions. Developing scientific software can sometimes be an individual activity, but using common libraries and object packages has reduced our development time. Although developers within collaborations still have individual tasks, collaborating enables idea sharing, broadens the knowledge base, and provides insights into different perspectives. Consequently, the likelihood decreases of having a single point of failure in support. Collaborating has enhanced group members' desire to communicate and perform as a team.

SRCG group members have also been part of specific collaborations funded at a European-wide level. The DNA (www.dna.ac.uk) and BIOXHIT (Biocrystallography (X) on a Highly Integrated Technology Platform, www.bioxhit.org) projects are examples. In these projects, partners have signed up for specific tasks. Sometimes, these tasks have been completely autonomous, requiring only communication at meetings a few times each year, largely to give and receive feedback on progress. We've found these collaborations work well when the interfaces to separate work packages have been well defined. Where requirements and deliverables have been more intertwined, closer communication has proved essential for producing effective results. Making decisions and resolving problems has included a human-management element. We've had to trade ideas and buy-in to solutions. Initially, this was difficult, and progress was slow until we established good rapport with our partners. In the long term, establishing these collaborations has proved beneficial in many ways. Group members have acquired new skills that have proved valuable in the GDA collaboration. Members have received mutual benefit across the project through access to skills and effort that individual institutions couldn't afford. Also, it's given collaborators the opportunity to see different facilities and how they operate.

As we've been involved with the development of the major new facility, DLS, we've witnessed three broad phases of work—initial development, commissioning, and long-term development. During initial development, we put an infrastructure in place, and there was little pressure for deliverables. The mutual benefit between the two sites was high. Daresbury staff had exciting new challenges, and DLS staff gained big rewards from adopting the established code base. This proved to be a very successful period for the GDA collaboration—productivity and the focus on shared goals were high.

As DLS moved into the commissioning phase, the pressure for them to deliver new features and fix bugs on a short timescale increased dramatically. Concurrently, the SRS, having only 18 months left to run, moved into a phase that required maximum stability and reliability. This difference in emphasis between the two sites caused friction within the project because the driving forces behind the collaboration were pulling in different directions. Political pressures within the organizations added to the difficulties. At this point, by mutual consent, the GDA project diverged, and each site is continuing to develop GDA separately. At Daresbury, we've seen that long-term development often involves further cycles of the first two phases, but on a smaller scale.

In our experience, collaborations work well while development speeds are in step across the collaboration and while goals and priorities are well matched. One group's stability easily becomes another's lack of progress; one group's rapid development easily becomes another group's unplanned hacking. How to manage collaborations successfully when development issues differ considerably will be a challenge for the future. One conclusion we can already draw is that this will likely happen at some time, so try to plan for it. Strong leadership to gain buy-in within the team will be important in maintaining a collaboration through a difficult period.

As with many good things in life, collaborations aren't free. They incur overheads in terms of greater organization, more discussion on decisions, and compromises to reach solutions. If these overheads are handled well, the benefits more than outweigh these issues. Progress might seem slow in certain areas, so it is important to highlight the benefits of collaborating to those actively developing and those sponsoring the collaboration, and they must all buy into it.

Several years' experience in software collaborations led us to some general advice on attempting to achieve success. Much of it focuses on communication issues, without being specific, and isn't limited to software projects:

- Collaborations have costs. Do a cost-benefit analysis to ensure you properly account for them. Sell collaborations' benefits to your team and the sponsors on a regular basis.
- Collaborations work best when they're enjoyable, which occurs when those involved feel like part of a team. You can't force this feeling, but don't neglect to give team members opportunities to forge closer links.
- Ensure good communication and don't rely on email, even when an international aspect is present. Face to face is always best.
- Interpersonal conflict can cause problems, so don't let it continue. There are bound to be times when problems surface, but as long as all involved are committed to the project goals, you can resolve these.
- Beware collaborations that turn into talking shops. Idea sharing can be important by itself, but ensure that people commit to long-term actions and follow up.
- When people are working on elements that others will use, ensure that interfaces are well defined in advance and that everyone agrees on them.

Developing software in a scientific environment requires a flexible and pragmatic approach. Scientists in an experimental-research environment must develop requirements as their research proceeds. We've learned and reviewed many ways to help in this endeavor. One major lesson we took away from this collaboration is that the scientific-research environment is driven by highly focused individuals. Success in software projects for this market demands a close relationship between developers and scientists. Attempting to impose a solution is unlikely to be successful. A more satisfactory method is to build mutual respect between the two disciplines. Having a flexible approach to requirements usually leads to good working relationships and a good final product. Scientific research is, of course, fun and creative, and so is developing software in that environment.

## About the Authors

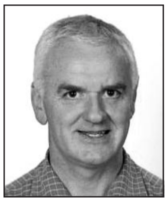
**Karen S. Ackroyd** is a senior software engineer at Daresbury Laboratory. She received her BSc in computational and statistical science from the University of Liverpool. Contact her at k.s.ackroyd@dl.ac.uk.

**Steve H. Kinder** is a senior software engineer at Daresbury Laboratory. He received his PhD in physics from the University of Reading. Contact him at s.h.kinder@dl.ac.uk.

**Geoff R. Mant** is a senior software engineer at Daresbury Laboratory. He received his PhD in photophysics and photochemistry from the University of Southampton. Contact him at g.r.mant@dl.ac.uk.

**Mike C. Miller** is a senior software engineer at Daresbury Laboratory. He received his MSc in applied entomology from the University of Newcastle. He's a member of the British Computer Society with Chartered IT Professional status (MBCS, CITP). Contact him at m.c.miller@dl.ac.uk.

**Christine A. Ramsdale** is a senior software engineer at Daresbury Laboratory. She received her MSc in radiation physics from the University of London, St Bartholomew's Hospital Medical College. Contact her at chris.ramsdale@hotmail.com.


**Paul C. Stephenson** is a senior software engineer at Daresbury Laboratory. He received his PhD in theoretical physics from the University of Bath. Contact him at p.c.stephenson@dl.ac.uk.