Some Lessons Learned Reviewing Scientific Code

Chris Morris
STFC
Computational Science and Engineering Dept
Daresbury Laboratory
+44-(0)1925-603689
C.Morris@stfc.ac.uk

ABSTRACT
This paper recounts the findings of three recent code reviews and discusses the wider applicability of the lessons learned to scientific programming.

Categories and Subject Descriptors
D.2.5 Testing and Debugging – code inspections and walk-throughs.

D.2.0 Software engineering – general.

General Terms
Human Factors, Verification.

Keywords
Review; scientific codes; testing; error handling; depth of inheritance; complexity; quality goals.

1.INTRODUCTION
Some software engineering practices could usefully be adopted by scientific programmers. In order to recognize which these are, and successfully communicate their value, it is useful to understand the context which has given rise to the current practices of scientific programmers.

In Section 2, this paper reports on the results of code reviews of three scientific applications. It argues that the findings of these code reviews show that practices appropriate for developing prototypes or throwaway code were misapplied to code for which the quality goals and necessary practices were different.

Three different scientific applications were reviewed. The first application, referred to as SCHED, is used to schedule the run of other programs, and to make transparent whether they are run on a grid or in other computing environments. The author was asked to review this application. The review report was used to guide the last stage of the development before delivery.

The second, EXP, is used to run experimental stations at a large facility. It controls motors to set up the experiment and acquires the data from the detectors. Six people including the author reviewed EXP. The reviewers were asked to answer the question: "How can we improve the maintainability?"

The last, LIMS, is a web application and database to store laboratory data.The review of this application was not complete at the time of writing. The author was not a reviewer of this code, but part of the development team. The reviewers were asked "How extensible is it, and how can we improve the extensibility?"

The reviews of EXP and LIMS were an exchange: each team spent a week reviewing the other's code.

There is a summary of the applications in table 1.

Some scientific programming is done in a very exploratory way, to answer a single question. The quality criterion is often "Proof of Concept". In particular, defects are important mainly if they cast doubt on the feasibility of the project. If a defect is clearly fixable, then fixing it can reasonably be postponed.

Another form of scientific programming is the development of large numerical codes to be run on High Performance Computing (HPC) facilities. They also have a life cycle measured in decades. Most of the work which has build up the software engineering body of knowledge was done in very different environments to this.

In section 3, this paper will also discuss to what extent the lessons the reviews found are applicable to such codes, and propose some steps that may help the software engineering and scientific coding communities to work together more effectively.

Table 1. The applications reviewed

Name	Age, years	Language	KLOC
SCHED	2	Java	3
EXP	10	Java / C++	50
LIMS	3	Java	32

2.FINDINGS
2.1
The most significant issues found by the reviews are reported here. These applications have many strengths, in their science and their software. This report concentrates on the opportunities for improvement. This report will not use their real names or identifying details. When the review reports are quoted, no reference will be given.

2.2Error Handling
The report on EXP found:

"Reportedly, [...] scientists tend to assume that failures are in [EXP] rather than in hardware. Sometimes the only failure in [EXP] is a failure to detect and report errors that have happened elsewhere. For example, the fact that the constructor of ServerThread ignores IOException is unacceptable."

The consequence of the ignored exception is that a failure to make a network connection will not be reported clearly. Of all the components of a computing system, the network is the least reliable. This failure can be expected to occur many times in the lifetime of the system.

Application SCHED had many catch blocks that print a stack trace and continue. Although some debugging information is created, such code does not guarantee its intended postcondition.

Professional software engineers learn early that “what can go wrong will”, and ensure that even unexpected failures will be reported clearly. This approach is sometimes called “prevent the impossible”. The codes reviewed were clearly not written with this in mind.

2.3Complexity

One method was found to have an NPATH complexity of:

770,943,744,005,163,750,045

The NPATH complexity is often two to the power of the cyclomatic complexity. The number of test cases needed to adequately cover the code lies between the two [1].

All of the applications reviewed had some methods with a complexity too high for testing.

2.4Duplicated Code

One review found:

“An example of a class with a lot of duplicate code is [...], which has lines copied from (or to) five other classes.”

Fifteen per cent of LIMS is lines that have been copied and pasted. EXP has 28 blocks of 100 or more lines that have been copied and pasted. So it is not surprising that the depth of inheritance in these applications is low: 70% of classes have DIT of 0 or 1.

In addition, both LIMS and EXP contain some classes with high instability and low abstractness.

These are all issues that are not important in prototype or throwaway code, but which become important as the life cycle gets longer.

2.5Tangled Code

In EXP and LIMS there were a lot of circular dependencies among packages. These were developed by teams of eight and nine respectively. There were (until recently) no mechanisms to maintain architectural integrity as different developers added code. Again, this is not an important question in prototype code.

2.6Quality Policy

None of these projects had a written and agreed set of quality goals. Appropriate goals were discussed as part of the reviews.

SCHED and EXP are facilities. The code will be used for many years – in the case of EXP parts of it already have been – so failures are inevitable as the environment changes. The inconvenience to users and the cost of maintenance can be reduced by designing for robustness.

The cost of maintenance is the number of defect reports times the average time to fix, which is the time to localize the fault plus the time to correct it. Improvements in robustness will reduce the number of reports. Improvements in error reporting will reduce the time to localize a fault. Improvements in the code structure will reduce the time to correct it.

The main obligation of LIMS is never to lose the users data. It is a database with a web front end. Of the applications mentioned, it most resembles main stream software applications, the core cases from which the ideas of software engineering have been developed.

Table 2. Quality goals and practices

Name	Goal	Coverage of automatic test	Issue tracker?
SCHED	Robust	none	No
EXP	Robust	2%	Yes
LIMS	Reliable	25%	Yes

3.DISCUSSION

Some of the practices that were found in these reviews would be strengths, when working on scientific prototype code. For example an issue tracker is not essential for such work, nor is attention to error reporting.

Scientific programmers often work on prototype or throw-away code as their first programming experience. Commercial programmers by contrast often begin with a maintenance task. (The author found an inch thick listing waiting on his new desk at his first day at work as a programmer.) These two different apprenticeships implicitly teach different lessons about what good programming is, and instill different sets of practices. Each of these sets of practices is fit for some purposes, but not for others.

Another set of scientific applications are large numerical simulations. The main quality goal of these applications is that they can solve some problems which cannot be solved in any other way. In return, the scientists who use them will tolerate defects that would be unacceptable in, say, an email client.

HPC codes are among the most long-lived in existence. HPC developers may have important lessons to teach software enigineers who are working on long-lived applications.

One concern is that scientific prototype code, if successful, segues into applications that are distributed for wider research use. Later it may be adopted for production purposes, sometimes even for safety critical use.

Standard practice for developing these codes includes the existence of system tests, complete runs of the application that are performed before each new release and for which the correct results are often known by experiment. However, the code coverage of these tests is often not known. Unit tests often do not exist.

Scientific programming projects would benefit from discussing and writing a set of quality goals. The result will be a short report answering the question “what would this code be like if it was great?” The answer is often obvious for commercial software development, but may be harder to find in a scientific environment.

The practice of unit testing would benefit scientific programming. In LIMS, the existence of unit tests has helped to coordinate the work of a scattered team.

For some codes, system tests can only be run on an HPC facility, and are expensive to perform. In some cases it will be possible to devise unit tests that are quick to run and have high test coverage.

The fundamental idea is one that is familiar to scientists. There are some hypotheses that are held about the code. A unit test is an economical test which is capable of refuting a significant hypothesis about the code.

If code is written to be testable, it is often easier to understand. Such code is separated into small units, each with a clearly defined purpose [2]. For example there is likely to be a clearer separation between the model and the solution method, along with some tests capable of refuting the hypothesis that the model implemented is the one described in the theory document, and other tests capable of refuting the hypotheses that the solution methods converge and are accurate.

The software engineering literature about unit test contains little about techniques in FORTRAN, or suitable for numerical codes. There is room here for some convergence between the two cultures.

When testing a numerical application, a goal for Def-Use coverage seems especially relevant. A value is “defined” where it is calculated and “used” where it becomes input for another calculation. A test suite has complete Def-Use coverage if every feasible Def-Use from definition to use is exercised. Optimising compilers routinely identify these pairs.

Unfortunately there are few tools which report this coverage measure. The development of tools to measure Def-Use coverage, and report uncovered cases in a user-friendly way, is an open challenge. This is one way that the computer science community can contribute to numerical computing practice.

It is not the case that software engineering practices are the best approach for every coding activity. In the author's experience, claiming that it is weakens the case for SE methods when they are appropriate. Software engineering is a culture too, and its methods are fit for specific purposes, not for all purposes.

The reviewers gained an opportunity to be more deeply reflective about coding practices. For example, they could trace the consequence of having no written quality policy all the way to individual code defects. One reviewer remarked about the review “We wouldn't have had the maturity to do this a year ago”. Everyday coding offers few opportunities to gain this level of reflectivity.

The reviews of LIMS and EXP were done by mutual exchange: each team agreed to spend a week reviewing the others' code. This approach could be used by other scientific computing projects.

ACKNOWLEDGMENTS

Thanks to all those who offered code for review and participated in performing the reviews; for the purpose of this paper they must remain anonymous. Thanks to Judith Segal for a discussion of refutable hypotheses and test coverage. Thanks to Yiannis Kanellopoulos for calculating DIT for EXP and LIMS. Thanks also to the reviewers of this paper.

3.REFERENCES

[1]Nejmeh, B. NPATH: a measure of execution path complexity and its applications. CACM 31,2 (February 1988) 188 - 200 .

[2] Janzen, D and Saiedian, H. 2008. Does Test-Driven Development Really Improve Software Design Quality? IEEE Software March 2008 77-84.