

Towards A Tolerable Software QA Program for LIGO Data Analysis

Goal

Our overriding QA goal is:

Avoid any serious or embarrassing errors in public statements of LIGO results.

To this end our collaboration level software checking should be aimed at finding potentially serious errors beyond those readily anticipated in testing by individual developers and their close collaborators.

A secondary goal is:

Detect errors early in the cycle to avoid delaying publications by finding errors at the final stages.

Probably most important for this goal is to avoid misunderstandings by establishing agreement as early as possible in a review process as to what the software is intended to do.

Background, Our Environment, and other Preliminary Postures and Considerations

1. Perhaps the most important established fact about our environment is that excessive unnecessary required standards, documentation, and process will be ignored. And they get in the way of the requirements necessary to our QA goal, so these useful requirements also then get ignored. Ours is not an environment where lengthy standards, requirements and processes will be followed. A simple review process and simple high level software standards common to all projects and working groups are what our colleagues will remember and can be encouraged to follow. Neither the process nor the standards should take more than one page to write down.
2. We are not IBM and we are not Linux and we cannot count on either of their approaches to QA. IBM and the rest of the commercial world do extensive software testing, typically upwards of 50% of the development cost, but they do not do as much as you might think (see below). We simply cannot approach their level of testing and should not put any significant reliance on testing (other than what the individual developer and collaborators intrinsically carry out). In the Linux community thousands examine even obscure code, and problems are quickly found. Working group colleagues take this role for much of LIGO software, but the breadth of the scrutiny comes nowhere near approaching that of Linux, and cannot be depended on.
3. There are two truisms worth repeating about software engineering (referring here to the discipline of reducing errors in software, not to the act of programming). One is that nothing much new of great significance has been discovered or written since the 1970s, despite numerous publications, professional societies and journals, and attention paid to the subject. (A good modern introduction may be found in a recent issue of the IBM Systems Journal, Vol. 41, No. 1, 2002, devoted entirely to this subject. In particular, see the first article, Hailpern and Santhanam, *Software debugging, testing, and verification*, p. 4, available on the web. The early classic papers by Brooks, Dijkstra, Myers, et al. are referenced here.)
4. The second truism is “Program testing can be used to show the presence of bugs, but never to show their absence.” (Dijkstra 1972).
5. There is a huge literature devoted to proving software (“verification”). Verification describes requirements, rigorously, in a specification language document typically longer than code that will be written based on it. Even in the commercial world formal verification of the functional requirements of a program is extremely rare, reserved only for hardware, certain crucial embedded systems (e.g., involving life safety), and protocol verification. (See Hailpern and Santhanam.)

6. Even a complex example of electronic hardware is orders of magnitude less complex, in terms of the number of possible internal states, than a few hundred lines of software. I believe the largest code that has ever been formally proven is now in the range of 1000, maybe 10000 lines, and this is heroic. A famous example by Myers in 1979 of how hard it is to really do software testing, in the form of a test you can give yourself, is in the Appendix.
7. There is relief. Physics analysis software is amenable to extensive physics-based consistency and reasonableness checking. We do this all the time, intrinsically, and our processes must be designed to encourage even more of it. Mock data challenges (MDCs), where signals are injected in the hardware and/or software are an excellent example of this. Real data with signals injected are an essential part of our process. This is not the “silver bullet”, because the test injections cannot cover even a small fraction of the phase space for which the software must be correct. (Brooks, *No silver bullet: essence and accidents of software engineering*, **Computer 20:9** (1987))
8. An individual working alone is error prone. We do not see our own errors when alone. Working with, or in front of others, reduces this problem enormously. We all have experienced how when we have to defend our work to others, errors we missed previously seem to jump out of the page.
9. In our context we can draw two conclusions from this. First, review committees should be wary of isolated individually developed code and analyses. A priori, if a working group or subgroup has *really* been involved, the code is intrinsically much more reliable. Review committees must establish to what extent an activity has been a lone effort.
10. Second, at the appropriate time in the life cycle a walk-through is one of our most valuable tools. In a walk-through (also called a code review) the developer stands in front of a group of colleagues who, at least, appear to be awake, and explains the code, line by line. The developer will find most of the bugs in a code review. The physical presence of the reviewers is essential.

QA Process for Our Two Phases of Development

For QA purposes I believe we can divide our software development into component development and analysis software activities. The QA program for each will include: a) A requirements statement available at the review defining completely but concisely what the intent is for the software; b) A minimalist set of coding and documentation standards aimed at making the code readily understandable by reviewers; c) a review group and schedule plan for reviews and MDC tests; d) a coordinator who signs off on the software (and who also may have other roles depending on the phase and context such as librarian, project QA coordinator, chair of analysis review committee, etc.).

Some details of the process for each phase:

Components

The component phase involves the development of well-defined elements of the larger library, potentially reusable for other analyses beyond that originally motivating the developer. The components will be incorporated in the (appropriate) CVS library and they will need to be integrated into larger production packages. When a component is reasonably mature and ready for use by others or for integration, its QA process kicks in. This includes:

- a) A one-page requirements statement of what the component is supposed to do, including inputs and outputs and specific (version number) reference to algorithms and other components it uses.
- b) Approval that it *reasonably* meets the minimalist LIGO software guidelines and standards and the requirements of the relevant CVS library. The latter includes a testing package, with input data and expected results, so that the librarian can determine that the component is nominally working when integrated with components already in the library into a larger package.

- c) A code review (walk through) as described above with 2 or 3 reviewers (including a chair) physically present.
- d) Sign-off by the coordinator that the component at this major version level corresponds to its requirements statement, follows the spirit of the standards, keeps the code librarian happy, and has suffered a walk through.

Since we do not want to repeat numerous QA reviews (particularly the walk-through) for a component, it will be an important judgment call by the QA coordinator and developer as to what constitutes “reasonably mature”, since components are often under continuous development for some time. A QA signoff will have to occur in advance of the beginning of any QA process on analysis software using this component. How much code development can be permitted before a new major version level is declared and a new walk through of changes (at least) is required, will be a subject of continuing concern.

Analysis Software

An analysis software activity is a combination of software runs with defined data sets, calibration constants, trigger and veto lists, etc., whose results are to be included in a published LIGO talk or paper. This is the most important QA process of all:

- a) At the time of the review there must be a requirements statement detailing what software components and packages (versions) are being used, the pipeline script, data sets, calibration constants, etc. This statement is a complete list with attached references where appropriate (such as the component requirements, the script, etc.) of *everything* that could change the published result.
- b) The review chair (coordinator) should be satisfied that the component documentation is adequate for the review. This should have been resolved in previous phases and should not be an issue here.
- c) The analysis software review group should include at least two persons capable of following the pipeline script and two persons (may be the same) who are familiar with the coding environment used in the components. It may include overlap with the corresponding physics analysis review committee or it may be independent. The committee and reviewee(s) should first go over the requirement statement in detail and establish agreement on its content and its meaning. The committee should review the QA signoffs on all components and determine whether new reviews are required because of development work following the earlier component review. A walk-through of the script is essential; a sub-committee may conduct it separately from the main committee, but it must examine each line of the script and particularly examine data, calibration, trigger, veto, etc., input files scrupulously. The committee should examine results of a MDC run and/or the detection sensitivity of injected signals in the software or hardware data path and establish they are consistent with what is expected.
- d) When the committee is satisfied, the chair (coordinator) signs off on the analysis software activity and informs the corresponding physics analysis review committee.

Appendix

(From Glenford J. Myers, *The Art of Software Testing* (Wiley, 1979))

“Self-Assessment”

Statement of Requirements:

The program reads three integer values from a card.

The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

Your Exercise Requirements:

On a sheet of paper, write a set of test cases (i.e., specific sets of data) that you feel would adequately test this program.

The next step is an evaluation of the effectiveness of your testing. It turns out that this program is more difficult to write than it first appears.

As a result, different versions of this program have been studied, and a list of common errors has been compiled. Evaluate your set of test cases by using it to answer the following questions. Give yourself one point for each “yes”.

Answers on next page.

1. Do you have a test case that represents a valid scalene triangle? (Note that test cases such as 1,2,3 and 2,5,10 do not warrant a “yes” answer, because there does not exist a triangle having such sides.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (A test case specifying 2,2,4 would not be counted.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (e.g., 3,3,4; 3,4,3; and 4,3,3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said 1,2,3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the other two sides (e.g., 1,2,3; 1,3,2; and 3,1,2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (e.g., 1,2,4 or 12,15,30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1,2,4; 1,4,2 and 4,1,2)?
11. Do you have a test case in which all sides are 0 (i.e., 0,0,0)?
12. Do you have at least one test case specifying non-integer values?
13. Do you have at least one test case specifying the wrong number of values (e.g., two, rather than three, integers)?
14. For each test case, did you specify the expected output from the program in addition to the input?

Of course, a set of test cases that satisfies the above conditions does not guarantee that all possible errors would be found, but since questions 1-13 represent errors that have actually occurred in different versions of this program, an adequate test of this program should expose these errors.

If you are typical, you have done poorly on this test. As a point of reference, highly experienced professional programmers score, on the average, only 7.8 out of a possible 14.

The point of the exercise is to illustrate that the testing of even a trivial program such as this is not an easy task. And if this is true, consider the difficulty of testing a 100,000-statement air-traffic-control system, a compiler, or even a mundane payroll program.