Lessons I Learned from SDSS

Robert Lupton

Tucson, July 2002

These are in the order that they came to mind, so I suppose that they're roughly sorted. I haven't included the technical lessons concerning such matters as how to calibrate TDI data.

It hardly needs to be emphasized that these are my opinions which, for all I know, may not be those of any other person, living or dead.

Some of these are obvious, but were honoured in the breach by the SDSS project.

1. You need a strong and impartial project manager. SDSS is a collaboration of a large number of institutions and we have never managed to take technical decisions unimpeded by politics.

2. Don't go into a project that isn't fully funded. Our problems (see 1) have been compounded by our continuing need to placate universities that might put in money. (I'm not sure that I'm going to follow the Rule if I get involved in LSST; but failing to follow the rule makes a good project manager even more important).

3. Neither Science nor Software can be run as a democracy. Not all participants are equal, and it's folly to pretend that they are. This is not to say that the most senior (or smartest) individual should simply lay down the law.

4. When, What, and How to Review

If some component is failing, admit it even if this involves embarrassing people and institutions. You should not, of course, make this any more public than necessary.

Put resources where they're needed, not where it's politically convenient to put them.

Saying that broken code is good enough isn't a conservative approach, even if it saves resources in the short term.

Only hold reviews if you really want to learn from the review board.

5. Standard software practices are necessary. This includes:

Source code control at the level of files (e.g. cvs)

Code control at the level of releases that can be reconstructed, along with their dependent products (but probably not at the level of being able to recover old versions of the O/S, compilers, etc.)

Enforced rules about tests associated with each new feature

Tools, preferably integrated with the code/release control system, to track problems and feature requests (we use Gnats).

An insistence on adhering to standards; e.g. coding to ISO C89 and Posix 1003.1.

An insistence that all code should compile with no warnings on all platforms, with all warning turned on.

6. Distribute Data and Information Freely

Be as open as possible, and make all information and discussion open to the entire collaboration as soon as practical. We have a system of mailing lists that we archive on the web, and to which anyone in the collaboration may subscribe. Our software problem report system (5d) is also available on the web, as is a listing of all papers that are being prepared for publication.

Make data available to the collaboration (or the world) as soon as possible, in a form as close to the final format as possible. In SDSS it's very hard to find recent data, and it was months (years?) before it started to appear in the science database. It is still true that newly taken and reduced data disappears into a (different, rather inaccessible) database and only reappears significantly later and generally incompletely (e.g. the cutouts around all detected objects are not always available).

7. Avoid single points of failure. OK, so this is totally obvious, but there are subtler aspects. If one person is allowed to become essential it implies that it's proved impossible to find someone else who could fill their role. In consequence, if they are on the critical path, and problems arise, it's hard to add resources to solve the problem. Another problem is that if someone with essential knowledge isn't very good, then an essential component of your system isn't going to work very well.

8. Find some way to reward people working on the project. In SDSS we did this by promising them early access to the data via a proprietary period. Not only is this impossible for publically funded projects, but it doesn't really work very well. One problem is that the promise of data in the distant future doesn't help a post-doc much; another is that the community (at least in the US) doesn't value work on the technical aspects of a large project. I don't think that the solution 'Hire Professional Programmers' is viable (although hiring a significant number of *competent* software professionals is a good idea. My experience has been that we cannot afford to hire good programmers).

`<hobbyhorse>` My personal belief is that the only long term way out of this is to integrate instrumentation (hardware and software) into the astronomy career path, much the way that the high-energy physicists appear to have done (at least from the outside). `</hobbyhorse>`

9. Strive to ensure that the software takes full advantage of the hardware, even at the beginning of a project. This is partly a matter of principle, and partly because if you don't push to the instrumental limit you don't really know if things are working as well as they should. There is some tension in achieving this, and you need someone to keep the schedule.

10. Don't generate an inverted management structure . Work to ensure that everyone in a position of authority has a clear view of their own strengths and weaknesses, and try to ensure that decisions are taken for technical reasons wherever possible.

If you are forced into a situation where the software effort is large, divide and conquer — manage the project as a tree with a branching ratio of less than 10.