Lessons we should have learned from SDSS, HSC, and LSST

Robert Lupton

LSST Pipeline Scientist

2016-10-18

SDSS and Hipparcos

This talk started out with a conversation with Michael Perryman (then P.I. of GAIA) about Hipparcos and SDSS.

Lessons I Learnt from the SDSS

- Lesson 1: You need a Project Manager
- Lesson 2: Don't join under-funded projects
- Lesson 3: Don't generate an inverted management structure
- Lesson 4: Learn when, what, and how to review
- Lesson 5: Practice good software engineering
- Lesson 5b: Don't Write Your Main Program in C
- Lesson 6: Distribute Data and Information Freely
- Lesson 7: Strive to ensure that the software takes full advantage of the hardware, even at the beginning of a project
- Lesson 8: Treat neither science nor software as a democracy
- Lesson 9: Avoid single points of failure
- Lesson 10: Find some way to reward people working on the project

I first wrote these lessons down in 2002; what have we learned in the last 14 years? Some of these are obvious, but were nevertheless ignored by the SDSS project. Many are being ignored by HSC and LSST too.

Disclaimer

The advice in this talk comes from a PCA analysis of my involvement in

- SDSS
- PanSTARRS
- ACT
- HSC
- LSST
- Euclid
- PFS
- WFIRST

Any resemblance to actual projects, living or dead, or actual events is largely coincidental.

No animals were harmed in the writing of this talk.

First the good news

Some of us have learned a lot about software engineering:

- C++
- `python`
- `git`
- `jUnit`
- Jenkins, travis, ...
- JIRA / `github` issues
- docker, conda, ...

Actually SDSS didn't do too badly. We used `C`, `TCL` and `cvs`, with `GNATS` as a bugtracker.
Some people who boycott ADASS still haven't seen the light and hack things together with no thought for tomorrow.

Douglas Adams (The Hitchhiker's Guide to the Galaxy. 1978)

Orbiting this at a distance of roughly ninety-two million miles is an utterly insignificant little blue green planet whose ape-descended life forms are so amazingly primitive that they still think fortran is a pretty neat idea.

First the goodish news

Some of us have learned a lot about tools for software engineering. What have we learned about software design and practice?
Quite a lot:

- testing
- interfaces
- reusing packages
- community standards
- documentation tools
- code review

Unfortunately our new-found wisdom sometimes makes the job of managing software groups harder; some people love the journey more than the destination, others are Luddites who don't care about technical debt.

More good news

Lesson 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 wrote a mailing lists manager that archives on the web, and to which anyone in the collaboration may subscribe. A listing of all papers that are being prepared for publication is also on the web.

- Make data available to the collaboration (or the world) as soon as possible

The first item's a bit dated, isn't it? But it turns out that just making web-enabled mailing lists doesn't mean that people use them, and wikis are often write-only, so this is still good advice. Web bug-trackers took off, though. A newer piece of technology that's proved very useful is a live chat tool such as `slack`.
I should have entitled this lesson:

Lesson 6: Distribute Data, Code, and Information Freely

The LSST code is distributed under the GPL; it's at `https://github.com/orgs/lsst`.

How should you manage projects?

Hardware

You design everything including every detail; then you build it. Building the system is slow and hard.

Software

You design everything including every detail; then you're done. Designing the system is slow and hard.

How should we manage them?

Hardware

Little of our hardware pushes the state of the art, and once the design is done an external expert can see if it makes sense. Requirements + reviews work well.

Software

Software is different. At the time you need to pass your FDR/CD-3 the codes are not written. You usually don't know enough to write tight requirements.

Project Managers

Robert Lupton "Lesson 1: You need a Project Manager"

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.

H. H. Munro (Reginald at the Theatre. 1904)

"When I was younger, boys of your age used to be nice and innocent."
"Now we are only nice. One must specialise these days"

I no longer innocently believe that all we need to do is to hire a project manager, as they mostly come from quite another world. We *do* need them, of course, but we need to choose them carefully.

The problem[s] with project managers

Many project managers come from hardware backgrounds. They are comfortable writing contracts with requirements for stiffness and mass and knowing that the sub-contractor will deliver (and that the milestones mean something).
Strategies that I have witnessed for managing Data Management (DM):

- Ignore the problem
- Make the scientists responsible for DM
- Treat DM as a subcontract; believe what you are told
- Appoint a DM Project Manager; believe what you are told

So we have a recursive problem. How do we manage the DM Project Manager? Or, better, how should we find managers who don't need managing?

This is a fine topic for ADASS, and the problem's not insuperable. Just hard.

People

Tools don't write programs, people do. Some of our problems are engineering (`How should we handle multiprocessing?'), some are algorithmic (`Please write me a weak-lensing-quality deblender'), and some are computational (`I'll give you 10ms per object to fit a galaxy model'), but most are about working together, making the best use of our varied skills.

Robert Lupton "Lesson 8: Treat neither science nor software as a democracy"

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.

A piece of good news:

Janel Garvin, Dr. Dobbs Journal (2015-10-01)

So, all told, developers are not the lonely, antisocial nerds that they are portrayed to be, nor are they free-wheeling socialites.

Roberts' Paradox

Unfortunately I'm naming it not for me, but for Eric Roberts at Stanford who in 2000 wrote a report for the US National Academy with the blessing of the ACM. The paradox is that:

- There are unemployed software engineers
- There is a shortage of software engineers

The resolution is that the shortage is of the best engineers, not the median:

> *If the best software developer can do the work of 10, 20, or even 100 run-of-the-mill employees, a single-person company that attracts such a superstar can compete effectively against a much larger enterprise*
> *[…]*
> *In some cases, software developers who fall at the low end of the productivity curve may be essentially nonproductive or even counterproductive*

Single point failures

Robert Lupton "Lesson 9: 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 rôle
  In consequence, if they are on the critical path, and problems arise, it's hard to add resources to solve the problem.
- If someone with an essential job isn't very good, then an essential component of your system isn't going to work very well.

My only update would be:
Hire as many people as you can who have with the ability to become single points of failure; then try to manage the project so that they don't.

Robert's Corollary

Roberts' paradox implies that we'll almost always have single points of failure

What is the problem managing software?

- The fluidity which is the blessing and curse of software
- The large dynamic range in programmers' talents
- The large diversity in programmers' talents
- The wide range of topics included in `Data Management':
  - Framework design
  - Pipelines
  - Algorithm development
  - Conops
  - Running data centres
  - User support

  I'm not sure that that's a larger diversity of topics than in building a camera
- Programmers have opinions about *everything*

> *Developers enjoy arguing about style issues almost as much as they enjoy arguing about which is the One True Editor. (As if there's any doubt. It's Emacs.) Scott Meyers, "Effective C++ "*

Timescales

When I get a problem report

- Things are broken due to `ndarray`'s change from `int` to `size_t`

I know what to do, and roughly how long it'll take.
When I get a request

- Implement storage agnostic data access (with a list of requirements that define the needs)

I can split the problem into its parts and estimate the work involved.
When I am asked to

- estimate colours of faint blended galaxies with errors of less than 5%.

I don't know what to do. I can ask for better (actionable?) requirements, but that doesn't help all that much. I don't know how long it'll take until I have some idea of how I'll do it -- at best I'll have some idea of who to give it to.
In fact, I was asked

- reduce the data from LSST. The requirements are at `http://ls.st/srd`; the desired data products are at `http://ls.st/dpdd`

How should I plan the work?

Why Agile Development is a Good Idea

These problems with software development (especially in the absence of crystal-clear requirements and use-cases) are not new; it's why Agile methods are becoming popular in scientific programming.
Agile encourages us to:

- Work with the customer on their requirements
- Always have something working

and (in practice)

- Estimate the work involved at the point of implementation

I personally don't think that we have any choice but to adopt and tame Agile.

Why Agile Development isn't a Panacea

At a fine-grained level I believe in Agile.
But agility brings problems for the project manager:

- How to manage Agile development in a world governed by Earned Value Management with a 6-month reporting cycle? I can use Agile with a 1-month cycle, but I need to plan 6 of those cycles up front. The top-level requirements must be met in a few years time; the flow-down to intermediate requirements isn't very Agile.
- One reason why Agile works well is because it gives the teams power over how they implement solutions. This can cause problems; it assumes that each team is strong enough to come up with good solutions. In practice astronomical software groups are often run as sets of individuals and this makes things worse; now *everyone* has to have good taste.
  You might argue that this is because we misuse (or only use in name) Agile development. I might not disagree.
  These problems are all exacerbated for new, geographically-distributed, teams.
- Who is the customer? There are many levels and layers of development, serving different masters *i.e.* `Hierarchical Agile'.
  Different masters mean different customers.

Who is the Customer?

Customer (OED definition 2a)

A purchaser of goods or services

So that means the funding agencies, acting on behalf of the science community. Who does that mean in practice for a given piece of functionality?

- The DM scientist?
- The Pipeline scientist?
- The System Architect?
- The Integration Manager?
- Or a developer/scientist at a different university?

And who decides on the relative importance of infrastructure, algorithms, and stability? And who prevents feature-creep hidden as `it's scientifically essential that we do XXX'?

Re-enter the Project Manager.

The Project Manager.

The project manager's *primary* job is to find a way to manage a large, diverse, project that they themselves don't necessarily understand at the beginning.
For big projects this will probably involve

- hiring the right subordinates
- learning about critical subjects

If we can't or won't find a good PM we're back to needing a Superhero Programmer, but now one who loves Gantt charts as well as CRTP and weak lensing.
And has the trust of the overall project manager.

Good Luck!

The End

Hiring (Bonus slides for submitted version)

Robert Lupton "Lesson 10: Find some way to reward people working on the project"

In SDSS we did this by promising them proprietary data access. Not only is this impossible for publicly 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 would be a good idea).
<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>

I don't think that much has changed, except that I'm less pessimistic about hiring good programmers.
However, the problem of luring brilliant scientists to work on building the software remains, although it isn't hard to get them to join the project to do science.

Training Our Successors

EU Initial Training Networks (ITN)

Embarking on a research career is not always easy. And yet today's young researchers are vital to Europe's future. At Marie Curie Actions, we are well aware of that [...]

Our [ITNs] offer early-stage researchers the opportunity to improve their research skills, join established research teams and enhance their career prospects.

I've been involved in a couple of ITNs, and I think that we can learn from the idea if not the details.

- Teach a series of {Summer,Winter,Spring,Autumn,Candlemas,May Day,Lammas,All Hallows} schools to an evolving set of students and post-docs
- Concentrate on techniques (instruments, software, statistics) tied to the science
- Get the experts from on and off project involved

After a couple of years, we should have a knowledgeable younger generation, ready and itching to do science with the next generation of telescopes. My current attempt is the LSST Data Science Fellowship Program; the first workshop was late this summer.

The End