Workshop by James Spencer on software development in a collaborative environment

Friday 24th July 2009

Format: interspersed slides and discussion from James, and open round-table discussion.

Two kinds of problem (slides 3-7)

Social and technical problems. The former are much harder to solve!

Why should we try to solve them?

- The motivation is to enable us to do science better and to do better science.
- Rigour and reproducibility should be just as important to computational scientists as to experimentalists.
- Code history is important---it should be possible to retrieve exact versions of older code in order to rerun calculations and ensure the code produces reproducible results as it evolves with time.
- People may well think it's a waste of time testing and writing documentation compared to doing science, writing papers: this is often not so, as better science can be done more efficiently if people attend to these things early on and as a matter of course. "Retrofitting" is very painful.
- It's important to have a culture in the research group (etc...) where good software development practices are prioritised/supported/allocated time for.
- It's a good thing for these practices to be passed on. Believe in karma, even when there doesn't seem to be an immediate payoff: a lower entry barrier to your work for new colleagues and collaborators and/or vice versa.

Collaborative environments (slides 8-14)

Getting people to work together as seamlessly as possible on documentation and testing is key. There will always be people who aren't keen to spend time on good development practices, so it's crucial to make the barrier to entry as low as possible. Otherwise, the workload on the people who do participate is unfairly increased.

An illustration: it's usually more efficient to find a bug that causes a test to fail immediately that you become aware of it, rather than putting it off for later when the details have faded from memory and it has caused trouble for other people...

A hierarchy of responsibility is common (slide 13)

Individuals check that their code doesn't break anything else and does what it is supposed to.

Nodes (subgroups in a project) check each other's work.

Project leaders need to be even more careful to check the work being done (mainly in terms of its overall direction)

Communication is key (slide 14)

RSS feeds of commits to a source code repository are rare but have been found very helpful (by JS working on code development during his PhD in a group of around 5 people) for following what other people are doing.

Possible issues: information overload and how to filter out the most important things. The latter is difficult to do generally as the definition of important varies!

An anecdote: in open source projects, source code diffs (after commits to the code repository) are sent round to "everyone" on a project and people actually do respond, to pass either positive or negative feedback! It's a highly collaborative process, and is useful from a software engineering point of view but not so important for just getting the code right for the particular purpose. Results in the highly iterative refinement of patches in open source projects.

Software/tools for code review (i.e. a systematic examination of code, often by peer review, intended to find and fix mistakes):

- reviewboard
- rietveld

both of which provide web-based interfaces for adding comments on lines in diffs.

Also found to be helpful in collaborations:

VNC
: a remote desktop/graphical desktop sharing system, that transmits the keyboard and mouse events from one computer to another, and sends the screen updates in the other direction over a network.

screen
: a terminal version of VNC.

These mean that more than one person can access the work on a particular machine at the same time; often better than emailing to and fro, or asking IT departments for accounts on remote machines (often slow to respond...)

Documentation

Tools:

sphinx
: was used to generate the documentation for Python, and is particularly well suited to documenting python code. Generates HTML, latex and PDF output from a file of source that is meaningfully readable in a terminal. Can handle equations and pictures - uses latex-style formatting for the former, in a way that is easy to use for the basics, but harder for more complex situations.

evince
: document viewer for multiple formats.

Question: can documentation generated from the code be linked to external latex (or other) documents to get an integrated overall documentation? For instance, can a programmers' manual and a users' manual be merged? Via doxygen? not sure. Robodoc should generate latex output which can then by hand be included in other documents.

An example of a problem caused by undocumented code was raised: the gamess-uk dump file for restarting has no documentation describing what data goes where...

It was noted that doxygen can generate something like a call-tree (but had not been coaxed to work for fortran...).

JS has written a short python script to track Fortran module dependencies (http://www.cmth.ph.ic.ac.uk/people/j.spencer/code.php).

gprof2tree (in combination with gprof and a diff program) can be used to compare the call trees from different runs of a program, and hence provide a sort of coverage analysis. However, it was noted as being difficult to use.

A workshop on unix command line tools (e.g. sort, awk) was suggested and welcomed as a generally good idea.

Version control (slide 15)

The main development in recent years in version control systems has been the emergence of distributed version control systems (e.g. git, mercurial, bazaar). These are becoming increasingly popular in the open source community over the older centralized version control systems (e.g. CVS, subversion).

Centralized

there is one repository somewhere; users have access to the history of the code via the particular server; no commits or access to history without network access.

Distributed

individuals have their own local repository; each individual repository has the entire code history; backups are a potential difficulty - individuals are responsible for backing up their own repository; there can be interactions between individuals in a peer-to-peer structure, or back to a central repository - it's flexible. The topology of the repository interactions can follow the topology of the human relationships.

Commits to a local repository (rather than to a branch of a central repository) are a way of flexibly accommodating major changes, e.g. work in progress that's not yet ready for public release, highly experimental code or for making "temporary" commits as part of the development process.

Git was described as a distributed version control system that is very good at merging. Git is capable of talking to CVS and subversion repositories. A demonstration of "git bisect" used in conjunction with a regression testing framework (see http://www.cmth.ph.ic.ac.uk/people/j.spencer/code.php) showed how the revision at which a bug was introduced could be found automatically---very neat!

The problem of bugs arising due to incompatibilities not picked up as conflicts within the version control system still remains, however, and is substantial. Need thorough test suites and schedules, made as automated as possible! (slides 16-18)