CMS

Software Development Process

DILBERT: ©1989 United Feature Syndicate, Inc.

Giulio Eulisse - FNAL hitman

Before VoIP #FAILs

These slides are much more verbose than the ones I usually make and I apologize: in general I prepare much more entertaining presentations... ;-)

I simply wanted to have my thoughts black-on-white, in case the telephone connection does not work as expected.

Scale of the problem

- ~3M lines of code, subdivided in ~1000 compilable units
- ~100 external software packages
- ~300 occasional contributors, 20-30 hardcore ones
- 3 supported architectures (slc5_amd64_gcc434 and the development ones slc5_amd64_gcc451, osx106_amd64_gcc421)
- Two integration builds per day, per architecture.

Also sprach Zarathustra

The Development Process is what needs to be defined up-front

> *The main point I would like to make is that when talking about software development one has to think about the whole process, from how to plan for new features to deployment, not about the single separate tools and steps.*

The Development Process as code

> *The more of the development process is encoded as a program itself, the better.*

End-to-end solution always working

> *Tools might be "beta" and incomplete, tests might be simple, but the process must be always successfully carried out end-to-end. No shortcuts should be allowed. The tools being incomplete is not an excuse for the process being broken.*

Release planning

Development of experiment software

Integration of external packages

Integration Builds & QA

Release

User installation and grid deployment

Development process

Continuous integration

The process is actually circular (or a helicoidal if you believe in progress) since each release serves as the basis for the subsequent one.

In CMS the whole loop gets carried out automatically twice a day: i.e. twice a day we end up with a new integration build which is potentially as good and as tested as an actual release.This is what is usually referred as continuous integration.

Release planning meeting

The only bit of loop which (thanks $DIVINITY) is not carried out twice a day (but only once a week) is the release planning meeting. However it has a fundamental role as features which will go in a given version must be "declared" before hand in there and will be then tracked for progress. This applies to both the development of the experiment software but also about the external software packages which are used.

Experiment SW Development (2)

Configuration & build tool

Of course developers must be able to build their software. CMS uses SCRAM V1, which is a full rewrite of the old LCG SCRAM (V0). It now has a much lower (1/10) overhead to the actual build and it is much less disk intensive (1/100) than the old version to the point it is simply not an issue anymore. This is thanks to the fact that rather than trying to be a better "make" than "make", it is now basically a thin wrapper around it and exploits most of modern make (v3.81) features, like "$(eval)", to avoid verbosity & recursive Makefiles (which are #SLOW). SCRAM simply generates a makefile which takes care of the compilation / linking (in parallel, of course) of the release and also of CMS specific bits like registering plugins, precompiling python configuration files etc.

Environment management

One of the requirements for the CMS software is to be able to use different releases (possibly using different architectures e.g. 32bit and 64bit, gcc4.3.4 and gcc 4.5.1) in parallel and to be able to switch between them at runtime. This is again taken care by SCRAM which can set and unset the correct runtime environment for a given release.

Developer areas

With roughly 3M lines of code and a few thousands of DSOs (libraries / plugins) it would be unpractical for the casual developer (i.e. the physicist) rebuild the whole release from scratch, especially given the AFS based workflow of CERN. For this reason another requirement on the build system is to have the ability to rebuild only smaller parts of a release, reusing whatever can be reused out of the central installation. Again, in CMS, SCRAM transparently takes care of this, by creating a so called "developer area" which falls-back to the "release area" when needed.

Experiment SW Development

Improving the VCS experience

It's clear in the industry that file based VCS (ala CVS / subversion) are a dead-end since quite a while. Unfortunately CMS has to work around its obsolete design choices in term of VCS using it's own "poor-man version" of a modern VCS. This is what we call the "CMS CVS Tag Collector" which makes sure that developer / release managers only care about "tag sets" (i.e. the set of packages with associated tags) and not files (which are the natural unit for CVS). The transitive-closure of all tag-sets gives a release.

Wide range of modern development and debugging tools

*Besides the build tool, the development environment should be furnished with modern and effective development tools from day 1. Having the latest, greatest version of gdb, **valgrind** and a good profiler like **igprof** very often makes the difference between 1 afternoon and 1 week of headaches.*

*Moreover it is also important to have the usual set of documentation / reference tools. In particular we have a command line version of lxr (**cmsglimpse**) which is extremely useful to look up for code. In general the more the tools are command-line based, the better.*

Usually the way it works for us is that command line tools are used for actual development while the web version is really just for the "social" bit of development (i.e. "please have a look at this url").

Support for multiple architectures

While Mac support is often seen as not particularly useful (you'll never run production on Macs), it was actually very convenient to us to spot real issues with the correctness of our code and in general a good way to reduce the number of assumptions one makes about the environment.

Integration of external tools

Build recipes for all externals

*It's crucial to be able to control as much as possible the external software the experiment software depends on. In CMS we have a **full set of build recipes** for all of them (in the form of RPM spec files). A driver script (**cmsBuild**) can use them to produce user installable RPMs of all the dependencies.*

Dependency aware build tool

Just like when you touch an header file you want all the source files including it to be recompiled, the same should happen when you change the build recipe of one of the externals. Rebuilding the whole software stack, while feasible, it's clearly sub-optimal. cmsBuild automatically takes care of the dependency structure and uses it both for building things in order, exploiting parallelism when possible, and to decide what to rebuild after a recipe file (or sensible parts in cmsBuild itself) change.

User-mode, relocatable & autarchic installations

Given the fact that root (sudo) installation is simply not an option for CMS (for both security reasons and policy of GRID sites), a large fraction of the recipe maintenance effort goes into making sure that our packages can be installed by a user without admin rights, in any directory and depending on the "bare necessities" from the system (in CMS case this means kernel, libc and X11)

How cmsBuild works

md5 sum of the recipe **and** the md5 sum of the dependencies. If the recipe of the dependency changes, the hash changes, and also the dependant packages are rebuilt

FEFE89...

Package A

package A has a dependency on B

AFEF9645...

Package B

```
./configure
make
make install
```

md5 sum of the build recipe and other bits uniquely identifying a given invariant build of a package

Repository with prebuilt package

AFEF9645...

Package B

If a package with the same hash is already present in the binary repository it will be downloaded and installed, rather than rebuilt.

Integration builds & QA

Continuos integration

Given the number of people involved in the project, it is important to make sure their unrelated developments get integrated very ofter, to avoid divergency and conflicts between various changes. For this reason CMS has automatic integration builds, twice a day, which go under the full release validation procedure.

Dynamic QA

Apart from the usual unit tests, we have a bunch of more complex "physics" release validation tests which are carried out every single IB. A few of the most comprehensive of those workflows are also run under ***valgrind*** *and profiled with* ***igprof*** *in an effort to catch non-fatal memory corruption errors, leaks and performance issues, both from speed and memory point of view.*

Static QA

We have started to include in our process some "static" (build time) analysis of our source code. The simplest one is done by the ***include checker,*** *which preprocess a given set of headers, looking for useless include statements or those which could be moved from the interface to the implementation. Recently we have started using a commercial tool called* ***Coverity*** *which does an offline analysis of the source code to spot run-time errors. While we have currently mixed feelings about it, due to the high number of false positives which would require lots of "modelling" of our source-code, it clearly also spots genuine (and quite embarassing) programming mistakes.*

Releases

IBs are first-class releases

> *The difference between IBs and releases is actually mostly artificial. At some point some IB is stable / feature complete enough to be declared a release, but the procedures to obtain a release should not be different from those to obtain an IB.*

Release validation

> *Multiple tests run for each IB (i.e. twice a day):*
>
> - *unit-tests (mostly for low-level bits, e.g. the framework and plugin manager)*
> - *120 separate workflows, testing various physics processes*
> - *valgrind (subset)*
> - *igprof (subset)*

User installation and Grid deployment

Software packaging

cmsBuild produces as final result a set of user installable RPMs and a work area where those packages have been installed. The choice of RPM is actually an historical one and despite the cursing of a poor soul (myself) who had to patch it to work in a non-root, relocatable manner, it served us pretty well.

Software distribution

cmsBuild uploads created packages in an APT repository. Again the choice was an historical one (slc3 came with apt based repositories) but it worked out quite well. YUM was also investigated at the time but it was way behind apt in terms of features (no root-less installations, no ability to install in anything different than /usr) and being python dependent made it difficult to have it working seamlessly both on the grid and in the CMS environment (which has a different kind of python). The uploaded packages are also used to do incremental builds of newer ones, when possible.

Grid deployment

Grid deployment in CMS is done using exactly the same infrastructure as the normal user installation, via apt-get. Depending on the site, sys-admins can install software either by hand or (like it happens on LCG sites) via a pilot job which takes care of the installation

AFS installation

Due convenience, we also install CMS offline software also over AFS, mostly because the large CERN community is habit at working on lxplus, rather than on their desktop. However, just like Grid installations the AFS based ones are "just another laptop" from the point of view of the the tools used.

What I think we did right...

Complete control

Taking full control of our externals was a turning point for CMS back in 2006. It allowed us to reduce turnaround time for cutting a releases from weeks to hours and resolved all the packaging issues before-hand, simply because building IS packaging. Having a whole software stack build reproducible in one single command is king. The integration between the build script and the the binary repository allows for an incredibly fast turnaround, even in the case of large projects like CMS offline software.

Build tools should not be an issue performance wise

If you plan to have extremely small build units in your software (CMS has roughly 1000 of them) you might want to seriously consider SCRAM V1 as it was optimized quite lot for this kind of workflow. If you really want to rewrite it from scratch, make sure you do not base it on recursive makefiles, because it will never work beyond the first 50 packages (this is what the LCG SCRAM was doing and it was a pain). Foresee at least 6 months of an extremely skilled engineer if you plan to start from scratch (and regardless how good he is, he might still fail to deliver, like it happened to the LCG guy).

Services based on simple tools, not tools based around services

Historically the bits which have been working the best were those which could be used by a local user to work locally and which were based on files. Whenever we had more fancy infrastructure to do more complex tasks things felt apart as soon as the guy who set it up changed job. This is particularly due to the lack of "continuos deployment" (see next slide), but also because tools are usually easier to get right / test than services.

Continuos integration

I'll never never be tired of stressing how having two IB per day built centrally is simplifying the whole development process and allows bugs to be fixed early.

...and what could use some improvement *

Use a proper VCS

CMS workflows were really thought in a time when CVS was the only reasonable solution available. We are now 2 generations ahead in that and it might make sense to build yours on top of some modern VCS system. For example Debian / Ubuntu did that (albeit their use cases are simpler, because they don't care about user installations). This also mean that the idea of splitting a project in many smaller units, like CMS does, could actually be replaced by a "patch based workflow" like the one used to develop the linux kernel.

Continuos Deployment

While the whole build is documented and fully reproducible in the form of RPM spec files and SCRAM configuration, the actual system running the whole thing is setup ad-hoc. This is, IMHO, the biggest deficiency of what we currently have in CMS and what I strongly suggest you to look at first of all. There are now many tools to "code" your deployment recipe (capistrano, fabric) and I would strongly suggest to pick one and get the whole infrastructure described in a programmatic manner upfront.

* These are my own views not necessarily those of CMS... But sooner or later they will see the light as well. ;-)

Somewhat useful links

igprof: http://igprof.sf.net

valgrind: http://www.valgrind.org

coverity: http://www.coverity.com/

CMS IB pages: http://cmssdt.cern.ch/SDT/html/showIB.html

CMS igprof-navigator: http://cms-service-sdtweb.web.cern.ch/cms-service-sdtweb/qa/igprof/navigator