Software Organisation

Murrough Landon – 6 September 2001

`http://www.hep.ph.qmw.ac.uk/~landon/talks`

Overview

- Organising Packages
- Directory Structures
- CMT
- Questions

Organising Many Packages

Assumptions

- Our software is organised in a number of separate packages.
- These are stored in one or more CVS repositories.

Requirements

- Should be able to check out and build an individual package – without having to check out and build the packages it depends on.
- Should be able to use header files and libraries from other packages in a simple consistent way.
- Should be able to make a “release” of a consistent set of packages for simple installation elsewhere.

ATLAS Approach

- Each package has its own separate directory structure in CVS for source code and control files (eg Makefiles etc).
- Each package is built and installed into a common area – this includes binary programs, libraries and header files.
- If you are working on a package, your checked out version should take precedence over the installed distribution.
- ATLAS has used tools (formerly SRT, in future CMT) to organise and formalise this process.
- To make this all work, some choices of organisation of the code and directory structure are required.

Directory Structure (1)

Directories within the package

- Top level directory with the package name, eg `modserv` (with CMT an extra version subdirectory is added)
- All source kept with `src` subdirectory (and its subdirectories)
- With CMT, control files kept in `cmt` subdirectory
- Can have other subdirectories (eg for data, docs, etc)
- Header files (which form part of the external interface to the package) kept in *package* subdirectory
(*NB this seems to be the ATLAS convention, not* `src/`*package as I had in SW note 8*).
- Purely internal header files could still be kept in `src` subdirectories along with class file if desired.
- And some scheme for linking external header files from `src` subdirectories into *package* could probably be devised...

Directory Structure (2)

Building and installing a package

- Separate directory tree for installed components of a package.
- Single lib and bin subdirectories for all packages, but separate for machine/compiler selection.
- Separate directory per package for header files.

Accessing other packages

- For header files `#include "`*package/header*`.h"`
- NB this style is also require within a package for any header file forming part of the external interface.
- Single installed `lib` directory makes it easy to link with all required libraries.

Directory Structure (3)

Example

Two checked out packages in your working directory, one traditional, one with CMT:

```
~/someworkdir/l1calo/modserv/
~/someworkdir/l1calo/modserv/Doxyfile
~/someworkdir/l1calo/modserv/Makefile
~/someworkdir/l1calo/modserv/modserv/
~/someworkdir/l1calo/modserv/src/
~/someworkdir/l1calo/modserv/src/cpmclasses/
~/someworkdir/l1calo/modserv/src/otherclasses/

~/someworkdir/l1calo/confdb/v001
~/someworkdir/l1calo/confdb/v001/cmt/
~/someworkdir/l1calo/confdb/v001/confdb/
~/someworkdir/l1calo/confdb/v001/src/
~/someworkdir/l1calo/confdb/v001/src/calibclasses/
~/someworkdir/l1calo/confdb/v001/src/menuclasses/
```

Where these get built and installed locally:

```
~/installdir/i386_linux/bin/
~/installdir/i386_linux/lib/
~/installdir/include/confdb/
~/installdir/include/modserv/
```

Where the default distribution of all packages is installed on your system:

```
/someroot/dist/pro -> release0.3
/someroot/dist/release0.3/i386_linux/bin/
/someroot/dist/release0.3/i386_linux/lib/
/someroot/dist/release0.3/include/confdb/
/someroot/dist/release0.3/include/modserv/
/someroot/dist/release0.3/include/otherpackage/
```

CMT

Overview

- Becoming an ATLAS standard (slowly)
- CMT is single C++ program which uses a `requirements` file in the `src/cmt` directory of each package to build and run a Makefile.
- The Makefile is constructed from a number of fragments supplied with the CMT package.
- CMT interworks with CVS (but doesnt require it).

Features

- The `requirements` file can specify the desired outputs: binary applications, libraries or other documents.
- You can set global options for the package and for each output.
- Packages can `"use"` other packages. In this way they also inherit all their options (but can override them). Global options for all packages can be set this way.
- All packages get version numbers. You can build a distribution by `use`ing the latest versions of all packages (eg in a separate build package) and doing `cmt broadcast make`.

Comments

- CMT has a preferred directory structure which is not the ATLAS one, but it can be customised to be so.
- Currently missing equivalent to `make install`
- Support for generation of code (eg by moc) would need to be added as a new `document` or `language` type.
- Feedback from ATLAS evaluations seems (weakly?) favourable.

Questions

Tools

- Should we try to use standard ATLAS tools (ie CMT) from the start? Or stick to Makefiles?
- Use of other tools, eg code checking, memory leaks, etc
- Documentation: use Doxygen (or other similar product) for source code documentation and eg Together for making diagrams?

Coding, naming, etc

- How strictly should we follow ATLAS coding and naming rules?
- Convention/style for our package names? Prefix everything with `l1calo` or `L1Calo` to avoid any possible clashes with other packages in Atlas?
- Possible problem with two level (eg `l1calo/`*package*) structure in CVS when using CMT?

Namespaces

- Newer compilers will require `std::` prefix for STL. Use of `using namespace std;` is discouraged.
- Use namespaces for our software? Eg single `l1calo` namespace? At least for new packages?

My Suggestions

Directories and Installation

- Adopt directory structure compatible with future use of CMT, ie all source in `src` subdirectory, include files available in *package*
- Convert small or new packages.
- Leave HDMC until we are sure we (and the rest of ATLAS) are happy with CMT.
- But add `make install` to HDMC now.

CMT

- Works for single simple package (`l1calo/rc`).
- Try it on a more complicated one involving code generation and also one with Java (eg for IGUI panels).
- If OK, start using it seriously for coherently building all (new) packages.

Documentation

- Include a doxygen control file (`Doxyfile`) with each package.
- This will at least extract classes, methods etc
- Encourage addition of suitable comments too!

Namespaces and Naming

- Use `l1calo` namespace for all new packages.
- Aesthetic or potential practical problems with any scheme. To avoid any future clashes with Online software (and Offline and other detector online software) prefix all our packages and use 2–8 character suffixes, eg L1CaloModServ. *PS I dont really like this...*