Rapid Software Development for CLEO III

M. Lohner[1], *C. D. Jones*[1]

Cornell University, USA

Abstract

CLEO III's change from an existing, stable CLEO II software environment to a new software framework and new coding language has been a daunting task. We present here the rapid-development approach we have taken to achieve more in less time.

Keywords: rapid development, software development

1 Introduction

CLEO III's change from an existing, stable CLEO II software environment to a new software framework and new coding language has been a daunting task. We present here the rapid-development approach we have taken to achieve more in less time.

Our software framework allows plug-and-play of code modules, which speeds up the development, debugging, and testing cycle immensely. An embedded scripting language, tcl, eases run-time job configuration. A few words on programming languages, compilers, and platforms are in order. We will address how dynamic loading rather than static linking has cut link-times down from hours or minutes down to minutes or seconds. A system of nightly rebuilds of all our code has provided users with up-to-date, yet stable code. Web-based documentation (including auto-documentation generated from code), and hands-on workshops have provided our users with the knowledge to get up to speed quickly. But, of course, nothing can help users more than an intuitively designed software framework!

2 History of the Project

Around mid-1996, years before the projected CLEO III turn-on date, a small number of CLEO physicists decided that it was time to break with the CLEO-II data access model and develop a new framework for CLEO data access. Several factors helped us at that early time:

- We had no users! Radical changes and redesigns of the system were possible.
- We had grown up in the CLEO-II environment and knew what worked and what did not.

These factors have had a great impact on the development of the new CLEO-III data access model, described in a separate submission to this conference [2]. We will focus here on the software development aspects.

In the fall of 1997 we hosted our first workshop (more on workshops in section 9) and thereafter had our first users to worry about. The first application code was developed around that time. Code development started to ramp up rapidly in 1998 and 1999, and by the end of 1999 about 1.2 million lines of code in 500+ libraries had been written and had to be built on a daily release schedule; further details in section 7.

3 Module Plug-and-Play

The new CLEO III software development environment allows plug and play of software modules in a skeleton executable, called "Suez". All modules share a common type-safe data-bus. Software modules either request data from that bus or supply data to it. Modules contain one or more algorithms, which have hard-coded what type of data they need to do their calculations, and they register these algorithms with the data bus in terms of what data they can supply. This allows easy plug and play of any number of modules supplying the same data and easy configuration of a run-time job based on the particular requirements without the need to recompile any code. For instance, if we have two track-finders, they both provide the same type of data. To the data consumers it makes no difference which track-finder was run, but they may, of course, request a specific one. This greatly simplifies code development: a developer working on track fitting may use a stable simpler track-finder, while another track-finder is under development. This also works great for debugging code: by swapping out one module and replace it by another the problem can be tracked down quickly.

4 Embedded Scripting Languages

Suez uses Tcl as its command language, but care has been taken to only program towards an abstract interpreter, which allows other language interpreters (e.g. perl or python) to be developed in the future. Some higher-level functionality that would have been more time-consuming to develop in the C++ code was implemented in Tcl with no difference to the user.

5 Programming Languages, Compilers, and Platform Support

Supported programming languages are C++ and Fortran, on OSF1 4.x and Solaris 2.x and soon Linux/Intel. In an ideal world C++ features like templates, exceptions, and STL, would all work properly according to the C++ standard. In the real world we handle compiler- and/or platform-specific problems via C pre-processor bug flags, and provide C pre-processor macros for STL containers and iterators. Explicit template instantiation has proven to be more reliable, less error-prone, more efficient for building code, and more usable for dynamic linking.

Some important algorithms (e.g. one of our track-finders) are legacy CLEO II Fortran code, and it would be foolish to rewrite them from scratch in C++. Wrapping them in C++ allows them to be part of the overall C++ framework. Data required by these algorithms have to be translated from C++ objects to Fortran, but common blocks are only allowed for in-algorithm communication, preventing different Fortran algorithms to communicate "behind the scenes", which would circumvent the central type-safe data-bus, discussed in section 3.

6 Dynamic-Loading vs Static Linking

Suez allows static linking, as was supported by the older CLEO II environments. But for rapid code development with fast turn-around time, dynamic linking and loading of software modules into Suez has been invaluable, cutting link-times from hours or minutes down to minutes or seconds. One would imagine that the run-time cost of resolving symbols would make up for any time saved in static linking. But static linking requires searching symbols in a potentially large number of libraries, whereas dynamic linking limits the scope of the symbol search in the link process, and the resulting turn-around time for development is greatly improved.

Dynamic linking of different modules into a running executable requires special care: if two software modules are linked with some of the same symbols, the dynamic linker will resolve these

symbols from the first module loaded and use those symbols for the second module. Symbols therefore should be unique, else different behavior will result based on loading order. Another problem with dynamic loading has to do with interface changes which result in changes in memory layout (e.g. added member data or virtual functions to classes). If one module was compiled and linked with one version of a class, and a second module with a different version, the program will result in run-time errors.

These issues required special care: Most data access system libraries are linked into shared libraries with names reflecting the CVS version tag and a soft link to it from a generic shared library name to simplify linking (e.g. "`libMyTracker.so` → `libMyTracker.so.v01_03_04`"). If a dynamically-loaded module was linked with a different version of a shared library, the loader will complain and refuse to load the module. Versioned shared libraries have made dynamic loading much safer and have dramatically reduced the number of runtime errors. Also dynamic loading is only used in the development phase, as we require static linking for reconstruction jobs, especially since these will most likely require stable executables over long periods of time.

7 Library Submissions and Build Procedure

To allow our developers to rapidly develop code, we wanted a system that:

- allows many users to submit changes at any time
- is stable and usable
- is up-to-date on a daily basis

As you can imagine, these goals compete with each other.

We use the standard gnu tool set of gmake (for building code) and CVS (to keep track of changes). Any person developing software can submit directly to the CVS-based repository. New versions of libraries are required to be tagged with the same CVS tag for all files in the library. Only tagged versions of libraries enter the release cycle. We use part of "SoftRelTools" (see [1]) to manage CVS package checkout and have built some of our own tools on top. SoftRelTools separates the package checkout area from the release areas. The package area actually contains one or more CVS-tagged versions of a library. The release areas soft-link together the various versions of libraries to work together in a release.

A daily "test" release is built every night with all the latest CVS tags; its purpose is to catch problems (compile/link errors, incompatibilities between packages, failed test scripts etc.). Another release, "current", is also built every night, but only with CVS tags of libraries that built and tested successfully in the test release. A certain number of core packages are deemed important enough to halt any "rotation" of tags from test into current in case of problems. If all goes well, a newly released library version appears in current with a lag time of one day.

Each library can have any number of test scripts which are executed at the end of the build, and developers are encouraged to write test libraries whose sole purpose is to test other libraries. Several mock reconstruction and Monte-Carlo-generating scripts test the crux of the library features and are allowed to halt any library rotation in case of problems.

There is a loophole in this procedure that allows some package to build and test successfully, but break some other "unimportant" packages and slip through from the test release into current. These rare cases require rolling back to a previous version of the library. We also have a "production" release that changes once a week to provide stability for people with a less aggressive development schedule, and as a backup in such cases.

With more and more libraries our build times have increased substantially, at times overlapping with the development time of people during the day. Therefore we alternate between two current releases; if the new current rebuild is not finished by 7am, the old current is used.

Surprisingly enough, any motion to move towards a more stable system with a less-aggressive release schedule (e.g. once-a-week and dated) have been voted down. On the other hand, in this aggressive development environment packages can change from day-to-day under people's feet without them realizing that they have to recompile some library in their own area. But with versioned shared libraries, discussed in section 6, this problem has become less severe. Once we enter the reconstruction phase we will release dated releases to provide stability for months at a time.

8 Web Documentation

We employ web-based documentation for all software elements, including design principles, "howto" documents, and detailed descriptions of libraries. Developers are encouraged to document their designs even before writing the code. Overall people have followed the positive examples of the framework developers. Various programs also generate documentation directly from source code (Doxygen, DOC++, LXR; see [3]). The resulting html files can be viewed via web browsers or from the command-line like a man page via a script using lynx (see [4]).

9 Workshops

We found that focussed 1-day workshops were very effective in helping physicists learn about the new CLEO III software framework. Talks were mixed with real hands-on experience gained at terminals programming in the new environment. These workshops were accepted enthusiastically by the participants. New users are strongly encouraged to first work through the workshops, which are available on the web.

10 The Importance of Intuitive Designs

Of course, no workshop or documentation can make up for non-intuitive software. We encourage people to give feed-back on the ease-of-use of the framework, and have made many changes to the system to accommodate requests for simplification. The rule-of-thumb is, "if it's hard to use, let's change it".

11 Summary

CLEO III has been very successful in breaking with the old CLEO II software environment and in developing a new rapid-development approach with fast turn-around time. A core group of developers was needed early in the development process to lead the road without having to support users. Even though the number of developers has increased greatly over the past several years, the number of people to talk to to find answers is still small. What would have happened with 10 times the number of developers?

References

1 BaBar/CDF, "SoftRelTools", a software-release-tool package based on perl scripts on top of CVS. We use a rather early version (dated around mid-1997).

2 C. D. Jones, M. Lohner, "CLEO's User Centric Data Access System", CHEP 2000, Padova, Italy, Spring 2000.

3 See `http://www.stack.nl/~dimitri/doxygen`, `http://zeus.imaginator.com/doc++`, and `http://lxr.linux.no`.

4 See web page `http://lynx.browser.org`.