Lessons from the Open Scientist project

G. Barrand[1]

LAL-IN2P3-CNRS,Universite Paris-Sud, Orsay,France

Abstract

Open Scientist is an environment to do data analysis and visualization. This project was born from the rejection of "all-in-one" systems not adapted to a fast evolving technological situation. Since CHEP98 where a draft of the project had been presented, some progress had been done in understanding how to build an open and modular system. We present the main architectal and organizational ideas that sustain the today implementation.

Keywords: data analysis, visualization, OpenSource

1 Introduction

Experiments in high energy and nuclear physic need to be at the edge of computing technologies. This implies that the software must be organized in a way that permits to include or discard some technologies in an easy way. Data analysis, since it touches most of computing domains (storage, visualization, interactivity, networking,...) is probably the place where all the evolutions are received front face. The goal of the Open Scientist project is to extract a software architecture and organisation flexible enough to resist to these evolutions.

In order to apprehend the full structure and features of the today system, one should refer to the web pages [1]. Rather we want here to review the main ideas that emerge from the work done since CHEP98. The presentation order is more or less the historical order in which these ideas and concepts emerged.

2 Methodology

The general strategy adopted was to handle from the beginning multiple solutions for a given domain : multiple storages, multiple scripting languages, multiple GUI systems. This seems to be a natural way to extract "surviving" rules and commonalities.

3 Code organization ; hierarchical-multi-package decomposition

For long now, the Orsay team has adopted a multi-package organisation for handling software. A "package" is a set of code that has its own life ; its own web page, its own distribution, its own CVS repository, its own management system (makefile,...). A package could depend of other packages in a hierarchical structure. For a developer it is simpler to contribute to a small, well defined software unit than to jump in a "all in one system". The distribution of OpenScientist contains eleven packages. Most of the hits on web pages concern subpackages like SoFree (an OpenInventor clone) and Midnight (a C++ version of MINUIT). This validates, without ambiguity, the multi package approach. Good part of the code comes from other sources. Often, like for KUIP and Midnight packages, it is a mere repackaging of some existing code immersed in an all-in-one system.

A difficulty with the multiple package approach is the installation and relationship handling of packages. Some tools had been developed to do that ; CMT [3] developed by C.Arnault and omake developed by the author and used today in OpenScientist. The embedding of software in experiments revealed some constraints over management tools. A package must be ready to be handled by various tools because experiments want often to include a package into their own environment. Such fact reveals the "confinement" rule ; in a given package management features must be confined in a specific directory. Such rule, if followed, permits for example the coexistence of omake and CMT for a given package. We observed that in general this rule is not applied for most of the packages encountered up to now (Makefiles spread over directories,...).

4 The user-adapter-facility model

How does one identify the packages ? What are the relationships between them ? In general a scientist is interested in some data (histogram, detector description, energy deposition, sky picture,...) which he describes with a computing language into some "code" ; let us call this code the "user" code. This same scientist is in general also interested by having his data accessing some "facilities" like storage, network. Software for handling a facility is often provided by a commercial company but it could, if no implementation covers the need, be provided by academic institutions or laboratories ; let us call the associated code the "facility" code. In general to have data able to access a facility, another piece of code is needed to describe the data to the facility : the "adapter" code.

An interesting feature is that this pattern of "user-adapter-facility" code applies for most cases. It is quite obvious for storage and network but in fact it applies also to command interpreter, visualization, user interface,.... An example could be given in the domain of visualization where to have an energy deposition drawn on a computer screen, someone has to find a graphical library, which is then seen as a visualization facility, and has to provide a piece of code to describe the data into a 2D, 3D world by using the primitives of the graphical library. This code is nothing more than the adapter code toward the chosen visualization facility. The same apply if someone wants to manipulate, for example, histograms from a command line interpreter. A piece of code has to be provided in order to let the interpreter know the histogram entity.

This being said, some packages are easily found by putting the user code in one or more packages and each facility at least in one package. It is less clear how to handle the adapter codes. For example, all the adapters of the user classes targeting a facility could be put in one package. A bad solution is to put the adapters into the package handling the user classes (or worst to put the adapter code in the user class !) : if a facility has to be replaced, the user packages would have to be touched !!!

5 Object orientation

Object orientation (OO) helps clearly, through the notions of class, virtuality (or interface in Java), inheritance, namespace, design pattern,... to decompose the code. But OO alone is not sufficient "to identify the classes". Some modeling is needed and the user-adapter-facility model seems to be a very good coarse graining one.

6 Hub packages

Among packages, one category is of special importance : the hub packages. Mainly, a hub package is a place where "links are established" between user code and facilities. A hub package is at the top of

a package hierachy and is presented to users of the software ; for example to our scientist who wants to read data from a file, repesent them on the screen in various way, send part of them to the net,... A general rule is to keep a hub as light as possible and try to displace most of the code into other packages covering user, adapter and facility things.

Today, in general, are found “all-in-one” packages in which no differentiation is done between user, adapter and facility codes. It results of this situation that when a facility has to be replaced due to some technological innovation (a new graphical library, storage system,...), the existing facilities embedded in the all-in-one hub could not be discarded easily. It leads to a sclerotic situation that often results in the global rejection of everything.

7 Coherent set of packages

A hub with its related packages is called a “coherent set” of packages. A coherent set could have its own distribution and web pages that mainly contain references to packages pages. A coherent set is often identified as a “framework”. The distribution of a coherent set permits to install at the same time a hub and all related packages. Without this notion an installation would be done in a more tedious way by installing packages one by one.

8 Hub anatomy, class dressing

Where should we put the adapters ? One solution could be to create one intermediate package per facility. Today in Open Scientist all the adapters are put in the Lab package that is a software hub. Lab is the top of the package hierachy and knows the user packages, like HCL and Lib and the facility packages : Rio, Objectivity, tcl, KUIP, SoFree, HEPVis,...

In the Lab package, the connection of things follows the “hub dressing” logic. A user class like HCL::Histogram has a mirror Lab::Histogram that inherits HCL::Histogram and also does the connection toward facilities with some method : store, visualize,.... The store method instanciates the adapter for a given storage and commits things. The visualize method creates the adapter toward the viewer, etc...

About interactivity, the hub-dressed class is given to the SWIG tool [4] that produces the necessary code (adapters once more) so that the instances could be manipulated for example from tcl or java. Then someone can type in a session program :

```
tcl> Histogram h parameters...
tcl> h visualize parameters...
```

This hub dressing strategy permits to keep as much as possible the user code out of any link toward facilities. Today this is so for HCL, Midnight and some classes of the Lib package. The Lab permits to create various interactive sessions to interact with the dressed classes and facilites through various GUI or command systems.

9 Extensibility, the Lab packages

Extensibility is reached by creating new “user or experiment lab” packages. A user creates its own scientific classes, dresses them to access facilities, declares them to interpreters and creates shared libraries with all this code. Dynamic loading of shared libraries, available now on most operationg system, and done from interpreters permits to hook this user code without relinking a whole application.

Examples of lab packages are : G4Lab, GaudiLab and Dante. G4Lab does the connection between Geant4 and Open Scientist. GaudiLab, developed for LHCb, does the connection between the LHCb Gaudi framework and Open Scientist environment. Dante (= VirgoLab) does the connection between VIRGO data (frames) and OpenScientist. All these packages permit to do visualization and data analysis for their respective environment and the Dante one is already at work.

10 Open Scientist coherent set of packages

The v4 list of packages is :

- Rio, Riot : the file IO system of ROOT put in stand-alone packages.
- Objectivity : a commercial object database.
- Mesa : a free implementation of OpenGL.
- SoFree : a free clone of Open Inventor developed at LAL.
- SGI or TGS Inventor : commercial implementations of Inventor.
- HEPVis : a free collaborative set of classes over Open Inventor.
- Tcl : a scripting language.
- KUIP : the CERN/PAW command language put in a stand alone package.
- HCL : an histogramming package developed at LAL.
- Midnight : a C++ version of MINUIT (code extracted from ROOT).
- Lib : a package to handle random number, cut parser,...
- Lab : the Open Scientist hub.
- omake : a little tool to produce Makefile for UNIX and NMake for NT.

For example someone can build a system by following the free way : Rio, Riot, Mesa, SoFree, HEPVis, Tcl (or KUIP), HCL, Midnight, Lib, Lab. or by following a more commercial way : Objectivity, a propriatery OpenGL, TGS-Inventor, HEPVis, Tcl, HCL, Midnight, Lib, Lab.

11 Conclusions

We have reviewed the main stream organizational ideas of the Open Scientist project dedicated to provide an evolutive environment for doing data analysis. We believe strongly that these ideas form a solid basement to build upon. In particular the approach could ease the establishment of a web distributed development organization. The HEPVis library was one first step in this direction. The existence of Geant4 is another one. Following HEPVis'99, a proposal for such an organization has been done [2]. Some "after HEPVis'99" discussions are underway around "histogram interface". A convergence around this point and other "interfaces" will be a good starting point to break the all-in-one mentality and go toward a web-aware state of mind concerning these questions.

References

1 http://www.lal.in2p3.fr/OpenScientist
2 http://www.lal.in2p3.fr/academic-software
3 http://www.lal.in2p3.fr/technique/si/SI/CMT/CMT.htm
4 http://www.swig.org