Overview of the Persistent Object Manager

by Dan Miner and Wojtek Skulski

Department of Physics and Astronomy

University of Rochester

Summer Research Experience For Undergraduates 2003

Info: skulski@pas.rochester.edu

Summary: The Persistent Object Manager (POM) is a tool to organize several objects such as histograms into a tree structure, which is presented on-screen to the user in the form of a tree window similar to the Windows Explorer. The user can open graphical views of collected data by double-clicking on the object icons. The entire tree can be saved to disk as a binary file with an extension ".pom". The data file can be read back into the POM at a later time. Arbitrary objects can be managed by POM, but only so-called "persistent objects" can be written to disk and read back. Examples of such persistent objects include the Gr histograms which are provided by the Gr subsystem. The interface of the POM to the Gr, as well as a demo program, are both provided. Other types of objects can be managed as well, provided that a very simple interface code is written by the user. POM is optimized for the ease of use by the application programmer. The projected application of POM are data acquisition and data analysis, where several dozens histograms and other graphical objects will be used to collect data and present the results to the user.

1. The main POM panel (the GUI).

In the following the POM features will be presented based on the demo code and a demo panel. Please click the following two links to open both the GUI and the demo code which uses the POM to organize the results of the computations. (Several other demo applications are also available. Click on links in POM-Sys-Map to open those.)

For the demo GUI click this link:	POM FFT demo GUI
For the FFT demo code click this link:	POM FFT demo application

Example "experiments" are in the directory **POMexamples**

In order for the GUI to be functional, both the PomPOM and the Gr Toolbox have to be installed and compiled. If either of the two packages is not properly installed, the GUI panel can be opened, but it will not function as described below.

In the GUI panel there are two large rectangular windows. One is white and the other is grey. The white one has the "mouse focus", i.e., it responds to the mouse. The grey one is not responsive. You can switch the focus at any time by toggling one of the two radio buttons below the windows. When the GUI is opened for the first time, both windows are empty.

The left winow is named "main". It is the place where the data acquisition is happenning. In the demo

application you can fill the left window by clicking the large command button labeled Calc all. This button will call the procedure *MakeDemoTree* of the POM demo application. The procedure will allocate a bunch of histograms, fill them with Fourier Transform data, and add them to the *mainTree* which is displayed in the left window. From now on the tree image is visible, and if the main window is toggled to white, the tree can be navigated and you can display the pictures. The calculations can be repeated by clicking the either of the two buttons, the Calculate or Calc all button. The parameter *delta* governs the density of the sine wave which is to be Fourier transformed.

The content of the left *main window* can be saved to disk in the binary .pom format. The subdirectory **POMexamples** contains a few "experiments" performed with a range of *delta* values. You can read these "experiments" from disk into the *aux window* on the right, where the results can be examined by navigating the tree. There is no button provided to read the data back into the *main window* in order to prevent the on-going data acquisition from being overwritten. Likewise, there is no button to save the data from the *aux window*, because by definition the data have already been saved. In other words, the main window is for data acquisition, and the aux window is for data browsing. You can browse the data when the acquisition is in progress. You can read different experiments into the aux window at any time without disrupting the acquisition. (Note: the word *acquisition* refers to the projected applications of POM. In the present demo the acquistion is simulated by filling the histograms with Fourier transform data.)

There may be a few auxillary buttons in the POM GUI. The buttons *ExamineMain* and *ExamineAux* will print some diagnostic information into the system log. The button *Update Gr Views* does not function yet in the release 0.9 of POM because omnicast handling has not been implemented in the current Gr release 1.0a.

2. How to use POM?

POM is meant to be used by the application programmer as a service utility. The target audience are physicists performing data acquisition or modeling. Imagine you want to perform some numerical study of, for example, the spectral power density as a function of the frequency, as governed by the parameter *delta*. Please have a look at the POM FFT demo application. All you need to do is the following:

1. Create your own module similar to the provided demo, or just copy the demo. When copying it is best to change the module prefix to something different from Gr or Pom, in order to keep the examples intact for future reference.

2. Declare a global exported variable *delta*, see the demo. *You do not need to write any code to change the value of delta*. Just declare the variable, this is all. It has to be exported with the "*" mark.

3. Declare the Gr histograms to hold the numerical results. The histograms can either be global if they are meant to survive between procedure calls, or local to some procedure if all is done "in one shot". Global histograms are probably more convenient. The histograms do not need to be exported. Please see the demo.

4. Initialize the histograms and other variables just like in the demo.

5. Write the numerical code using *delta*, assuming that the needed value of *delta* will appear magically.

6. Write the exported procedure similar to the PomPOMDemo.MakeDemoTree. The procedure should add the histograms to the *mainTree* of the POM. Do not use the *auxTree* to hold your data, because the *auxTree* will be overwritten without warning when you read some data from disk. The *mainTree* will not be overwritten.

7. Compile your code. You can keep the name PomPOMDemo to keep the GUI working with minimal changes, but this is not recommended. It is better to use another name, and to replace the GUI links using Layout->Replace. This command is provided by the BlackBox system. In such a way the demo is kept intact for future reference.

8. Modify the demo GUI panel by adding any interactor variables which you defined in step 2. The variable *delta* is already linked, but you will need to link your other interactor variables, if any. In case you do not know how to link the interactors, the automatic GUI builder can do it for you. Please read the Forms user manual how to use the automatic GUI builder.

9. Save the modfied GUI panel under some other name and prefix, then open it in the "user mode" clicking a commander similar to the one below.

"StdCmds.OpenAuxDialog('Pom/Rsrc/FFTDemoPanel', 'Persistent Object Manager FFT Demo')"

10. Perform your numerical simulations by changing *delta* and clicking buttons, either Calculate or Calc all. Note that the value of *delta* has magically propagated to your calculations, without you ever writing any code to actually obtain the value from the screen.

11. You can save the entire "tree of histograms" to disk at any time, such that it can be read back into POM at the later date.

12. The Gr Toolbox provides many commands to deal with the displayed data. In particular, you can convert the Gr histogram to Ascii with just one mouse click. The Ascii data can be saved to disk as a plain .txt file, which can be read by any other program which can read Ascii. Alternatively, the Ascii numbers can be copied-and-pasted to such spreadsheet programs as Excel. The Gr User Guide will tell you how to use the Gr graphics.

These steps look like a fair amount of work, but actually they are not. Basically, most of work is to write the numerical part of the code, which needs to be done anyway. The idea of POM is that the histograms are registered with the POM just once, and from then on the programmer does not bother with the display.

3. How does POM help with developing data acquisition experiments?

We work with data acquisition hardware ourselves. We performed a successful experiment last year, where a number of CAMAC digitizers were operated using the Gr Toolbox. We also read out the digital signal processing boards using BlackBox and Gr. The POM software is being developed as an add-on to the Gr Toolbox to manage a large number of histograms and other data objects, which we will need to perform advanced experiments.

Having said that, interfacing with hardware is an advanced topic. One has to understand the hardware one is using, as well as the low-level Windows drivers which talk to it over the interface of some sort. Managing graphical objects is the tip of the iceberg, while the actual hardware is at the bottom. The low-level DLLs, as well as the data processing, fill the entire middle. POM will bring a significant simplification of the GUI, which in our experience is always labor-intensive to develop. This will help by providing more time to do all the rest.

4. How are data objects organized into POM folders?

You might have noticed in the demo that histograms are inserted into the POM trees more than once, under different branches. This is intentional. For example, the demo histogram *Real Frequency* belongs to the "Output", "Real", and "Cartesian" categories. The motivation to allow for such a repetition stems from the experimental practice. During experiment it is often convenient to file the same data object under different categories. Being able to do just that was one of important design goals for POM.

5. How can a new type of data objects be added to POM folders?

POM is extensible. Any type of data, persistent or not, can be added to POM by writing a dedicated wrapper object. The prototype "abstract" POM wrapper is defined in the PomPOM under the name *ViewableObject*. The example of a concrete GrHistogram wrapper is provided in the module PomPOMGr under the name *ViewableGrObject*. The ViewableObject (henceforth abbreviated VO) provides two kind of services: (1) it can make a non-persistent object into being persistent, and (2) it can make a non-graphical object into being graphical. These two tasks are of course almost trivial in the case of the GrHistograms, which are both persistent and graphical just by themselves. However, it is also entirely possible to turn a very simple data object, such as a POINTER TO ARRAY OF REAL, into being a persistent and graphical object.

In order to develop a new class of wrappers named *MyVO* the user has to implement both the persistency and the graphical display. Concerning persistency, the *MyVO* must implement the mandatory methods *Externalize, Internalize,* and *CopyFrom*. Please consult the documentation of Stores why these methods are necessary. The ones already implemented in PomPOM may actually be sufficient, provided that the object being wrapped is persistent and no new data fields are added to the *MyVO*. This was the case with GrHistograms. The Gr extension POMGr does not reimplement the three methods because the default ones defined in the PomPOM turned out sufficient to wrap the GrHistograms.

Concerning the graphical display, the prototype wrapper defines an empty method *ShowObject*, which is called upon double-clicking on the icon representing the object in the POM tree. The actual implementation can be quite elaborate, please read the POMGr to see the example. The GrHistograms are half-way between being pure data objects and graphical objects, in the sense that they provide some graphical support, but are not self-contained views which can be popped up to screen. The Gr-specific method *ShowObject* therefore has to close the gap by instantiating a GrView object, which is a full-fledged "canvas plotter". The GrView is created, its range adjusted to show the entire histogram, and then the plotter is submitted to the display queue to be displayed.

Depending on the exact nature of the *MyObject* which is wrapped into the *MyVO*, the *MyVO.ShowObject* may range from almost trivial to quite elaborate. It will be almost trivial if the *MyObject* is a full-fledged view capable of plotting itself on screen. Conversely, in case the *MyObject* is something as simple as an array of

numbers, the corresponding *ShowObject* may end up fairly complicated. In either case, the Gr-specific solution will provide a good starting point to develop another variant of *ShowObject*.

6. The content of the PomPOM package.

PomPOM consists of several modules, panels, and documentation. The map of the PomPOM subsystem is available by clicking on the following link: POM-Sys-Map.

7. Copyright, terms and conditions.