Astronomical Data Analysis Software and Systems VII
ASP Conference Series, Vol. 145, 1998
R. Albrecht, R. N. Hook and H. A. Bushouse, eds.

Object-Oriented Experiences with GBT Monitor and Control

J. R. Fisher

National Radio Astronomy Observatory, Green Bank, WV 24944, USA

Abstract. The Green Bank Telescope Monitor and Control software group adopted object-oriented design techniques as implemented in C++. The OO approach has led to a fairly coherent software system and a fair amount of module (class) reuse. Many devices (front-ends, spectrometers, LO's, etc.) share the same software structure, and implementing new devices in the latter part of the project has been relatively easy, as is to be hoped with an OO design. One disadvantage of a long design phase is that it is hard to evaluate progress and to have much sense for how the design satisfies the real user needs. The OO process is only as good at the requirement specifications, and the process has had to deal with continually emerging requirements all though the analysis, design, and implementation phases. Large and medium scale tests of the system in the midst of the implementation phase have required quite a bit of time and coordination effort. This has tended to inhibit progress evaluations.

1. Introduction

I should make clear from the outset that the Green Bank Telescope (GBT) monitor and control system is the work of a team of software designers of which I am not a principal member. This paper contains the thoughts and impressions of a part-time contributor, local consultant, and design philosopher to the project. Members of the group may not necessarily agree with all of my assessments, so the only justification for my giving this presentation at all is to offer a somewhat detached perspective. Any reference to this work should be to the primary documentation (Clark, Brandt, & Ford, http://www.gb.nrao.edu), and please see the acknowledgments section for the names of the designers.

Figure 1 shows the basic structure of the GBT monitor and control system. The object-oriented design referred to in this talk mainly concerns the parts of the software below the dotted line in this figure. Other software shown here does use OO design, particularly AIPS++, but this is not part of monitor and control. The operating systems to which this software has been ported are Solaris and VxWorks, and a port is planned to Windows 95/NT.

The design method used for the GBT monitor and control is the "Object Model" in Rumbaugh et al. (1991).

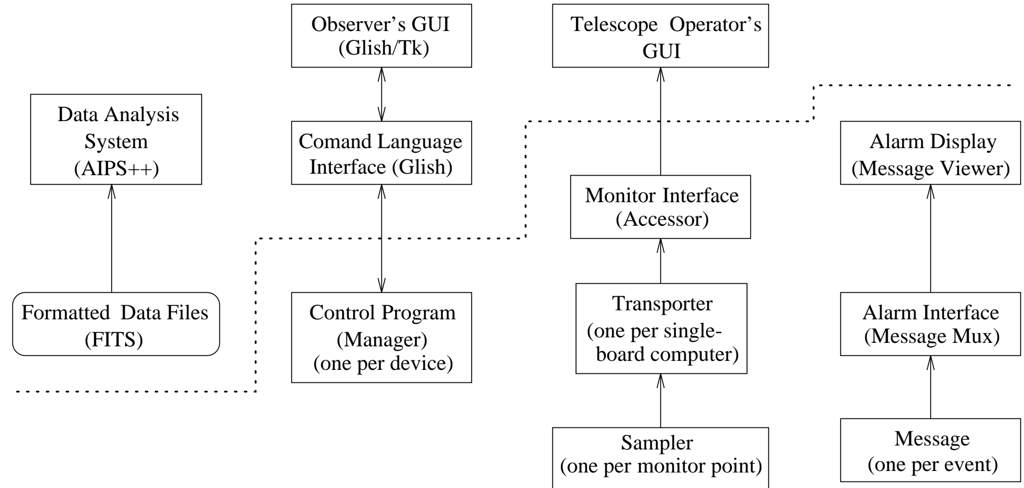

Figure 1. The basic software components of GBT monitor and control. The parts below the dotted line are the main subjects of this presentation.

2. Long Analysis Phase

The most evident feature of the object-oriented approach has been the long analysis phase. In a rigorous design this is as it should be, but it demands that the external requirements for the software be defined early. This has been a weakness in the GBT project, so the designers have been forced to proceed with rough requirements and modify and add to them as the project has progressed. The advantage of early requirements is a more coherent design, and considerable effort has been expended trying to reap this benefit.

Comprehensive requirements are very hard for future users to write. Formal requirements demand a certain amount of abstraction that is quite different from how software is used. It seems fair to say that object-oriented design adds another layer of abstraction and, hence, another language barrier to the communication between user and designer. One of the roles that I have played in this project is translator between the user and software domains. Still, most of the effort of ferreting out requirements has fallen on the designers.

A common method for converging on users' requirements is by building prototypes that the user can comment on and then iterate a number of times until the prototypes are acceptable. On the GBT project this hasn't been very effective because most of the design has remained in software abstractions. It is not clear to me whether this is an inherent practical drawback of the object-oriented approach or whether it is just a weakness in our implementation. It is likely a bit of both. The tension between getting something working and doing a careful design has existed throughout the project. In retrospect, the project might have been divided into smaller units with tangible products at early and mid stages of the project.

Another aspect of the GBT project that has worked against an early design phase is that the software and hardware were built concurrently. In principle,

the designs could evolve together, but the difference in design languages was again a barrier. The parts of the GBT that were experimental for most of the construction were quite difficult to fold into the software design. To a certain extent, one could only hope that the design that fitted the known part of the hardware would apply without too much modification to the experimental parts when they became production items.

None of this is to say that we would not use an object-oriented design again. Many of the benefits that come with this approach were, in fact, realized.

3. Modularity

Software modularity was rule number one in the GBT design. This comes from bitter experience with system interdependencies that have been extremely hard to debug in earlier telescope control implementations. The attraction of object-oriented modules is that a few generic modules can be developed and thoroughly debugged, and this robustness can be carried through to specific instantiations of these modules. This has had the added benefit that different designers of various software/hardware modules have been required to use common code libraries (in our case C++ classes) which encourage much similarity between the devices. Certainly, this similarity can be circumvented, but it is generally easier to adopt the common theme.

The adherence to modularity and code reuse has had an interesting effect on the operational aspects of the GBT system. Large closed loops are contrary to this design philosophy because they impose module interdependence. A typical example is having the start time of the data acquisition modules depend on the position of the telescope. Functionally, this make perfect sense, and it does not really introduce a great deal of complexity, but a combination of these interdependencies may have quite serious consequences.

The goal of the GBT design has been that the failure or removal of any subsystem, even the antenna, will not affect the continued operation of the rest of the system. It may not make sense from the user's point of view to continue, but the intention is system robustness. How well this turns out to work remains to be seen.

This independence goal has dictated that all servo loops be closed at the lowest level possible and almost always within the same CPU. All synchronization of operation is done with a common clock which each CPU reads to the accuracy necessary for its function. Each module is responsible for executing its commands as given, and any negotiation of actions is done with a common coordinator module.

Action sequences are grouped by "scans" where every action of each module is fully specified at the beginning of the scan. This could be as simple as one integration on a fixed sky position or as complex as a fast raster map of a moderately large area around a radio source. Doppler tracking, for example, is done on the basis of commanded antenna pointing, not actual pointing. Failure of a module to live up to expectations can be anything from a flagged condition to a fault error, but it affects independent modules only if a common action is decided upon at the top control level.

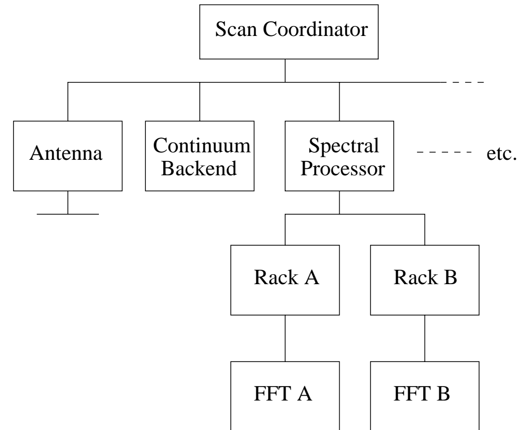

Figure 2. The hierarchy of managers in the GBT system. Each box is a manager which shares common interface, parameter control, and scan sequencing mechanisms with all others.

There is a module hierarchy in the system, but communication is strictly between parent and child.

4. Common Interfaces to Hardware Devices and Systems

A very important aspect of a common software module design is that all interfaces to modules derived from this design are identical. This is not unique to object-oriented design, but it is a valuable byproduct of strong adherence to the method.

The most ubiquitous code unit in the GBT is something called a "manager." In its use at the lowest levels of the system there is a one-to-one correspondence between a manager and a piece of hardware, a spectrometer, for example. The manager contains the mechanisms for receiving and computing parameter setups and for sequencing a scan on the basis of clock time. Managers can be controlled by another manager all the way up to the highest level where the scan coordinator resides. A piece of this hierarchy is shown in Figure 2

The only differences between specific instances of managers are the parameters they contain and, in the case of the lowest level managers, the hardware drivers. Each manager contains a state machine shown in Figure 3.

Upon receiving an activate command the managers that are connected directly to hardware automatically sequence themselves through the Ready, Activating, Committed, Running, Stopping, and Ready states. Parent managers echo the most advanced state of any of their children.

At the beginning of a scan the scan coordinator queries each of its children to find out who has the longest delay to the beginning of the next scan, given the specified scan parameters. The antenna is usually the slowest. If the user has said, "Start as soon as possible." the scan coordinator then issues the same earliest feasible start time to all sub-managers and then leaves them to go about their business until all have returned to the ready state. If the slew time to

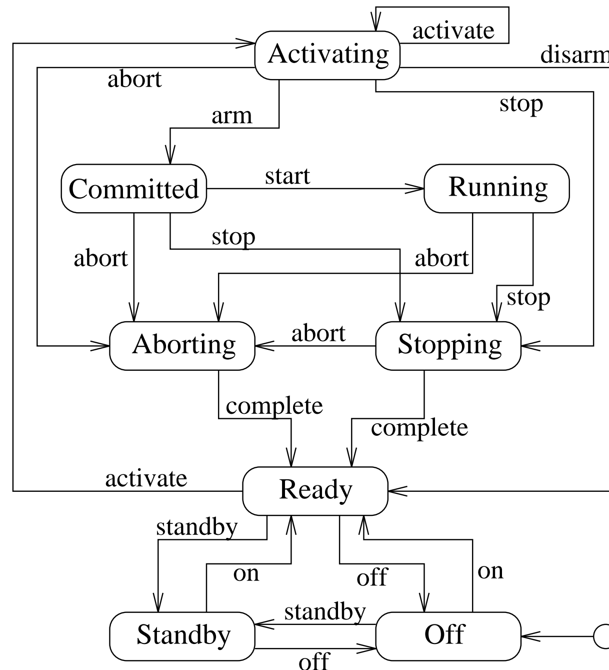

Figure 3. The manager state machine. Each state is shown in a rounded box, and the actions that cause transitions between states are the arrow labels. A typical scan sequence is Ready, Activating, Committed, Running, Stopping, Ready. Data are acquired during the Running state.

the next object is quite long, the scan coordinator might query the antenna for an update on the time-to-source estimate and issue an amended start time to everyone.

In the more complex hardware devices there is a complicated mapping of setup parameters that make sense to the user to values that are required by the hardware. In many cases the order in which the input parameters are used in the calculation of hardware values is important. The generic manager's parameter control mechanism was specifically designed to enforce the correct order of dependent parameter calculations. In most cases the user changes only a few inputs, but the current values of all the rest are still crucial. Keeping the calculation order straight while avoiding a complete recalculation of all unaffected hardware values is a very difficult coding problem. This was solved by a mechanism similar to the compiling *make* rules, plus a few coding rules largely enforced by the parameter class.

All managers inherit a common set of control commands such as 'set a parameter', 'start', and 'abort'. This makes building a user interface to them a very consistent bit of code construction, and it makes connecting one manager to another quite straightforward.

5. Code Reuse

Object-oriented design extends code reuse beyond the sharing of common functions and header parameters. In small software projects this advantage is small, but in a large project, such as the GBT, the benefits can be considerable. However, a lot of early design analysis is required to uncover the common properties of apparently disparate parts of a system. It doesn't happen automatically with the adoption of an object-oriented language.

I have already mentioned a few areas of code reuse, or more importantly design reuse. Others include the error reporting message system, interprocess communication (RPC++ class), and the hardware sampling and monitor system.

6. Generality vs Efficiency

There are two areas where our attempt at extensive software generality, as encouraged by object-oriented design and modularity, might come into conflict with efficiency. However, neither appears to be a major problem. One is the distribution of GBT processes across many workstations and single-board computers. The really time-critical functions are isolated in hardware and closely connected processors. The whole system is tied together by the scan coordinator which communicates with other device managers over a dedicated Ethernet. The Ethernet protocol does not guarantee a maximum transmission time so we need to be conservative about the time latency that can be tolerated by the start-of-scan coordination process.

The other area where efficiency and generality are at odds is in monitoring a large number of hardware diagnostic points. The sampler software nearest the hardware points transfers data continuously into ring buffers at the highest rate expected to be requested by the user of the monitor system for each hardware test point. This allows considerable flexibility for the user to connect to and disconnect from any test point without affecting the hardware and low level software configuration. It does mean that, at any given moment, most data are either ignored or greatly decimated in time before being looked at. Some tuning of the system will be required to avoid a significant processor load at the hardware connections.

7. A Virtual Telescope

We have all heard discussions about reusing software effort from one telescope to the next, but this has worked only in the very few cases where the goals, time scales, and communication proximity of two or more groups have been well matched. Too many factors that deter from system reuse are at work to assign much of the blame to the not-invented-here syndrome. We all have visions of the "virtual telescope" to which all software is designed and every user can use with only one learning curve. Let me finish up with a few thoughts on this from our design experience.

Every astronomical telescope is different for very good reasons. Attempts to hide these differences will usually hide the strengths of an instrument and

make it more difficult for the observer to understand the telescope system. A good user interface should be informative, not an abstraction to some universal instrument.

We are pleased with the design and code structure that has emerged from the GBT effort. Our sense of a virtual telescope is one where a common connection between a user interface and the hardware control software can be defined (the dotted line in Figure 1). This connection is defined in terms of message definitions (start, set_parameter, get_parameter_value, etc.) that are independent of the specifics of the instrument. Within the GBT effort this has made building user interfaces to individual parts of the system much easier because the connections are all the same. Only the setup parameters are different.

From the programmer's point of view, the common functional elements, like scan sequencing, parameter control, and message handling, could be reused from one design to the next. To the extent that the operation of these embedded features is correct, it makes sense to carry them to other telescope implementations. However, this certainly will not happen unless the code system is documented to the extent that a knowledgeable programmer can pick up a manual like the one for, say, Tcl/Tk and start using the code system. This is an extremely tall order for any design group faced with the immediate demands from their home institution. The only alternative would be to carry a design from one project to the next in the heads of key personnel. Even then it is not clear that any of us would ever program a system the same way twice. There always seems to be a better way.

Acknowledgments. Full credit for the analysis and design of the GBT monitor and control system must go to the software engineers, particularly Mark Clark (project leader), Joe Brandt, John Ford, and Aron Benett of the NRAO in Green Bank, WV.

References

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. 1991, Object-Oriented Modeling and Design, (Prentice Hall)