GBT MEMO 180

The Control Software Architecture for the Green Bank Telescope

Mark H.Clark[a]

[a]National Radio Astronomy Observatory, P. O. Box 2, Green Bank, WV 24944-0002, USA

ABSTRACT

A difficulty in writing control software for research telescopes is designing sufficient flexibility to handle continually changing requirements. The primary goal of the control software architecture for NRAO's new 100-meter radio telescope in Green Bank, West Virginia is flexibility through the construction of a modular software system.

Using the principles of modular design:

- the number of systems implemented to satisfy the required functionality between users and devices was minimized
- each device was implemented as an autonomous unit
- object-oriented design and C++ were used throughout the system

The purpose of this report is to present the software architecture being used to achieve a modularity in the Green Bank Telescope.

Keywords: telescope control, object-oriented, software architecture, modularity

1. INTRODUCTION

The need for flexibility in software design due to continually emerging requirements is an established axiom in software engineering. This need becomes even more apparent in the design for controlling general-purpose telescopes, such as the Green Bank Telescope* (GBT); the user community by its nature puts these telescopes to innovative uses which will require changes in hardware and software. The key to achieving software flexibility is to make the system as modular as possible. The guiding philosophy in the following discussion of modularity, and indeed in the initial analysis and design of the system, is Bertrand Meyer's views on object-oriented software construction.[2] The principles of modular design (i.e., linguistic modular units, few interfaces, small interfaces, explicit interfaces, and information hiding) are fairly straightforward, but achieving them is a little more challenging in a large system where real-time concerns traditionally have over-ridden desires for modularity.[3–5]

The original challenge for computer programming of telescope control systems was the automatic control of hardware. Initially, computers were used to control only the dynamic aspects of a telescope such as antenna movement or Doppler tracking. The task of software as an interface between a human being and the telescope hardware was inconsequential. In the newest systems, like the GBT, all hardware control and monitoring are through digital interfaces. One is lucky to find an indicator light, let alone a switch on modern pieces of equipment. The role of software has grown greatly; besides handling the automatic aspects of the system, it must configure into a "user-friendly" environment all the complexity generated by an array of sophisticated equipment. A new, complex piece of equipment has become the primary responsibility of the software engineer: the user. On a telescope the user may be an operator, an engineer, an observer, or even the software developer. It is a difficult interface to "get right."

Most of the GBT analysis and design effort has gone into configuring the software that must exist between a telescope device and the user. The problems of digital interfacing, real-time controls, networking, Doppler tracking, astronomical coordinate systems, and other control issues are well-understood and almost always successfully implemented. Flexibility and modularity are important throughout the telescope, but they are critical in the cycle of

Other author information: E-mail: mclark@nrao.edu

*The Green Bank Telescope is a 100 meter, fully steerable radio telescope with an unblocked aperture and is currently under construction in Green Bank, WV.[1]

information that starts and ends with the user. We took several broad steps in our strategy to achieve a modular GBT control system.

Rather than thinking of the telescope as a single instrument, the first step is to try and think of it as a laboratory filled with devices or instruments which have to be coordinated to accomplish observations. We want to minimize, at least in software, the need for the various devices to communicate with each other. We want to isolate real-time dependencies wholly within each piece of device software. We want to see if it is possible to coordinate scans via a "set up" mechanism common to the software for each device. If so, then the scan coordination system can be set up using this common mechanism to build a hierarchy rooted at a single "scan coordination" module. With these goals accomplished, the GBT can then be more aptly described as a laboratory where each device is a fully autonomous system, a laboratory where one may easily run subsets of devices in conjunction with each other.

The next step toward a modular design is to condense the user requirements[6] into a minimum number of functions. We determined that there are four basic functions required of a telescope control system: control, monitor, message/alarm handling, and data production. Control predominately requires information flowing from the user to the telescope; the remaining three functions require only that information flow from the telescope to the user. Having a single mechanism to handle all the information within each function greatly reduces the number of interfaces used in the system and hides almost all of the implementation details from the system's client programs.

Finally, we committed at the beginning of the project to the use of object-oriented design and C++ throughout the system, including real-time code. This decision ensured that our design remained modular from the device level down to the smallest piece of code. Encapsulating code into classes made it possible to understand how changes in one module affect the behavior of other modules. Constraints of the language also helped to keep us on track while under the pressures of schedules and needs for "quick fixes."

2. FUNCTIONS

Figure 1. shows a modified object diagram[7] of the GBT system. Information flows up and down the diagram with the user sitting on top working with the user-interface generated by an `interpreter program`, while in the bottom right corner a `telescope device` is installed underneath the `digital interface`. Boxes representing C++ classes have capitalized names; other boxes represent programs, files, or hardware. The lines between the boxes represent associations where the filled circles indicate multiplicity, e.g., there exists a one-to-many relationship between an `interpreter program` and its `PanelClient`, or a one-to-one relationship between a `Driver` and its `digital interface`. In addition, when the relationship includes the flow of data, it is indicated by an arrow, e.g., between the `Accessor` and a `Monitor`. There are three layers to the software. Starting at the top of the diagram is the `User Programs` layer, next is the `Interface Daemons` layer which provides network access, and last is the `Telescope Systems` layer which interfaces to a `telescope device`. The box representing the telescope device is in the bottom right corner of the diagram.

As stated in the Introduction, the four basic functions of the telescope system handle all of the interactions between a user and a device. Each device is fully encapsulated by a `Manager` object so that all software communications with the device pass through the Manager. Likewise, the user can only interact with the software via a user-interface program, which in the case of the GBT, is an interpreter. The classes implementing the telescope basic functions are represented in the diagram by four sets of intervening boxes between the Manager class and the interpreter program. The control system consists of the `PanelClient`, `RecipientServer`, `PanelServer`, `RecipientClient`, `Manager`, and `Parameter` classes. The monitor system consists of the `Monitor`, `Accessor`, `Archivist`, `Transporter`, and `Sampler` classes. The message/alarm handling system consists of the `MesgMuxIF`, `MessageMux`, and `Message` classes. Data production is represented by the `data file` box. For the writing of data files, all data produced by the telescope devices are written to individual files in Flexible Image Transport System's (FITS) binary tables.[8] This is true whether the data is backend data, scan data associated parameters, or engineering logs.

2.1. Control

For the GBT, we were not able to build a single class that could encapsulate any possible device; however we have built a base class, the `Manager`, which provides a common `control interface` and which implements a core set of functionality required by all devices.

Since the Manager class needs to encapsulate a wide-range of devices (from simple electronic switches to weather stations to large correlators to the antenna itself), a derived class, i.e., a `Device Manager`, is often used to add a

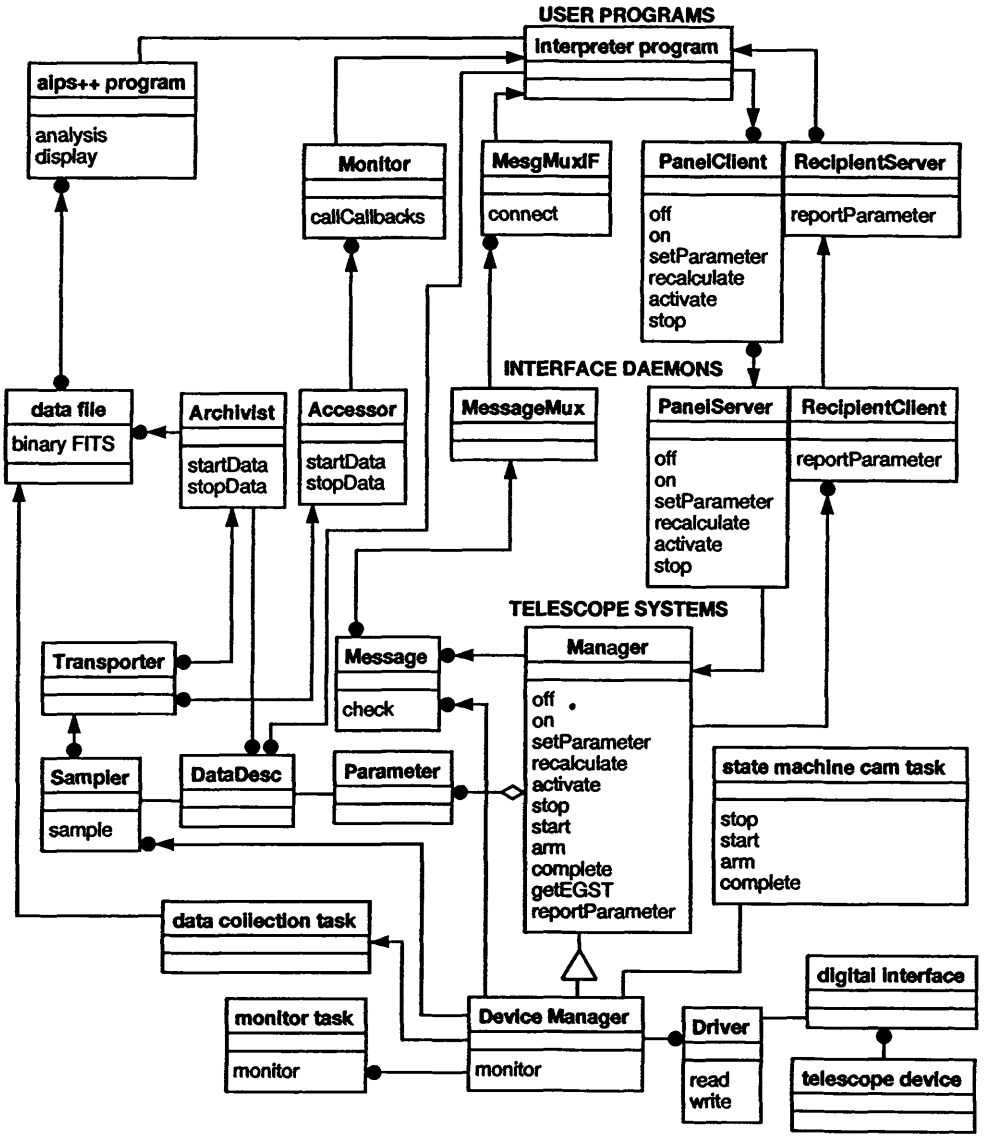

Figure 1. Object diagram of the GBT software architecture.

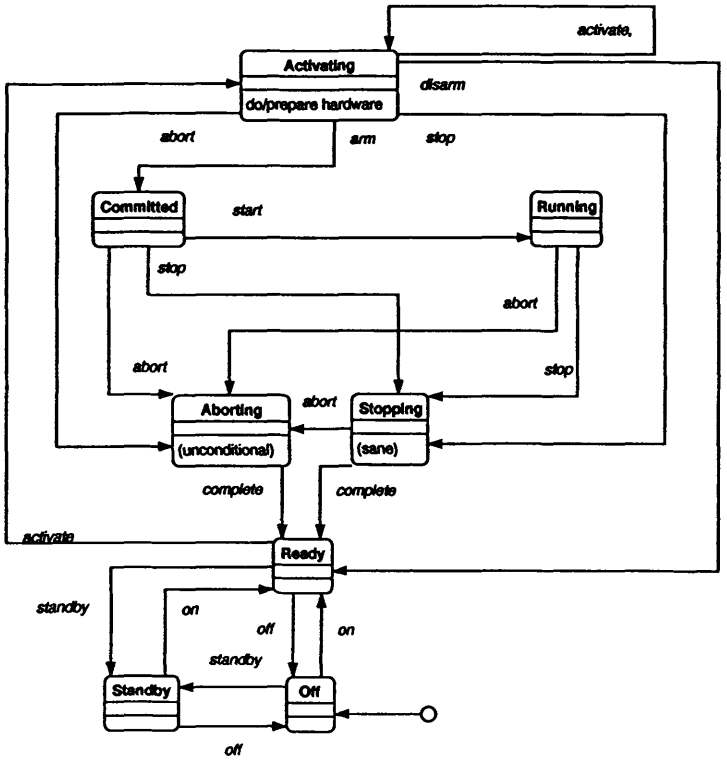

Figure 2. Manager state machine diagram.

variety of methods specific to the needs of its device. However, as far as the control interface is concerned, i.e., defining the operation for the next scan or commanding the scan, the base class guarantees that the interface is the same for all devices. In addition, the Manager has the ability to coordinate other Managers using the control interface. The aggregate of Managers making up the control system is organized into a hierarchy where any single Manager, branch of Managers, or the entire tree is capable of acting as a unit to run scans.

In the Manager, the control or "set-up" variables which define the operation of a device during a scan are each encapsulated in the class `Parameter`. A Parameter may hold primitive data types, arrays, C structures, or simple C++ classes. When given the command `recalculate`, the code in the base Manager class handles the set-up calculations much like a spreadsheet where the cells (Parameters) are computed from other cells via formulas. Recomputation dependencies are handled analogously to recompilation by a Makefile, i.e., only necessary computations are performed. When given the command `activate`, the values of "terminal cells" are loaded into hardware. So basically, what differentiates a receiver Manager from an antenna Manager is the associated sets of Parameters. Note that all of the devices' actions must be fully described by its Manager's Parameters' values prior to the scan.

The Manager is able to maintain one control interface across all possible sets of Parameters because the one method that sets the value of a Parameter, `setParameter`, is capable of setting any Parameter of any type. This is accomplished with the use of C++ templates and "void" storage pointers. The only difference in the control interface from one device Manager to the next is the possible arguments to the method setParameter. Specific Device Managers may also differ in their needs for drivers, monitoring, scanning, data collection (if a backend), and messaging (as can be seen by the Device Manager's relations in the object diagram), but none of these latter device-specific additions alters the control interface in any way.

Other than setting up a scan, a Manager is controlled by changing its state (see the state machine in Figure 2). The events causing state transitions are methods of the Manager class. Once the scan is defined, i.e., its Parameter values are set, then the Manager needs to be commanded through the stages of a scan. Some of these commands

or events come through the control interface and others – for the more complex devices – are driven by a control task labeled `state machine cam task` in the object diagram. The state sequence for a successful scan is Ready, Activating, Committed, Running (the actual scan), Stopping, and back to Ready. A device may remain in the Activating state for as little time as it takes for a digital interface to accept a new setting, or for as long as a physical mechanism needs to complete a movement. One may also command a "prepare," i.e., do everything to prepare for a scan without starting it. The state sequence for a prepare is Ready, Activating, and back to Ready.

A Manager which is coordinating a number of other Managers is able to synchronize the start of a scan by requesting from each of its "children" the Earliest Guaranteed Start Time (EGST). If the Manager is initiating the scan, then it uses the EGST as the commanded start of the scan for its children. If, however, the initiating Manager is further up the hierarchy, then the Manager passes its EGST up to its parent. As can be seen, the process is completely recursive and every Manager only needs to interact with other Managers above or below it in the hierarchy. Notice that even this interaction only occurs just prior to a scan; once the scan begins each Manager is completely independent. We believe this design provides the maximum modularization while still allowing scan synchronization among devices.

The control interface for a Manager, including network access, is encapsulated in the `Panel` classes for control and the `Recipient` classes for control feedback. The feedback is important to user interfaces because the Manager, upon every action, reports back its entire state by sending lists of annotated Parameters. Thus a user program always has a complete picture of the internal workings of the control software including Parameters containing illegal values and problems with activating terminal Parameters. Note that a Manager does not differentiate between whether its parent Manager or a user program is using its control interface. Observing or engineering user-interface programs are able to "plug" into the hierarchy at any point.

2.2. Monitor

The GBT is blanketed with digital interfaces which measure "real-world" values. Examples include ambient temperatures, DC power supply voltages, antenna encoder positions, laser power levels, wind direction, actuator positions, and computer/manual switch positions. These interfaces are read periodically and then the values are time-tagged and stored into a small ring buffer. The stream of values available at the ring buffer may be accessed on-command for logging or display. The philosophy of the monitor system is not for clients to request individual values for a particular point, but rather to open and close spigots which spew out a continuous flow of data values.

The `Sampler` is the monitor class most closely associated with the digital interface. The C++ constructor for a Sampler takes the address of a variable, a data description identifier, and the size of the ring buffer. On a schedule (usually a constant period, though it can also be a commanded periodicity or based on some event), the digital interface is read into the Sampler's associated variable. Then the class method `sample` is called which time-tags the value and places it into the ring buffer. The `Transporter` handles all of the Samplers for a single-board computer. It reads and transfers data from any Sampler's ring buffer to a requesting process over the network.

There are currently two client programs of the Transporter: the `Accessor` for providing monitor streams to in user-interface programs, and the `Archivist` for recording monitor streams that create FITS' binary table files. The clients to the Accessor almost always use the data to update a graphical-user interface widget that displays the data continuously as a digital readout or graph in near real-time. On request from a user the Archivist is able to produce two types of FITS files. The first are engineering logs which store continuously all of the data from a given Sampler. The second are Data Associated Parameter (DAP) files which store a Sampler's values as part of a scan's data set. Each DAP file represents a Sampler's values during one scan. Certain values have been traditionally stored as part of the scan data, e.g., weather values and antenna positions. This design allows the observer to select any Sampler as a DAP source. One might imagine other client programs to the Transporter, for example, one that stores given Sampler data when triggered by specific alarms.

2.3. Message/Alarm Handling

Originally in designing software for the GBT, we thought messages could be treated simply as a subcategory of monitoring values, i.e., the user programs needed the ability to access values (in this case text) from all parts of the telescope. All we needed to do was pipe the text into the Samplers. However, it became clear that the alarm or

messaging system needed more functionality than simply handling text generated from different devices.[†] Messages have requirements beyond those for monitor values.

Most messages need to represent conditions or states rather than transient events. The appearance of a line of text on a terminal is able to describe an event well enough, but more is needed to describe states adequately. For example, the message "Power supply voltage is too low" does not communicate when the voltage became too low or whether the situation still exists. Messages need initial and terminating times associated with them so they can indicate ongoing situations.

Messages also need to have some type of severity level assigned to them. Not all events generating messages have the same impact on telescope operation. Some events cause questionable data to be generated while others may place personnel or equipment in danger.

Messages when displayed need to generate a context to help make the message meaningful. One obstacle that often keeps a user from understanding a message is the loss of context surrounding the message from the time it is generated to when it is displayed. What may be a perfectly clear line of text inside a piece of code can become nebulous when unexpectedly displayed in an operational environment. In the example above think of the questions a user might have: Exactly which power supply is being described? If the power supply itself is bad, would other messages be generated? Is the current scan's data being affected and how? Is there equipment being damaged? Should there be some action taken immediately? A message should have pointers associated with it that are accessible by the user, e.g., device name, computer of origin, relations to other messages, and references to system documentation.

Messages also need some mechanism in place to handle "cascading messages." When some critical element generates a message, but at the same time a number of other elements in the system because they are dependent on the first also begin to fail, each generates its own message, continuing in a domino effect. The resulting flood of text masks the key message that would allow the user to discover the root problem.

To meet these needs, messages have to have a handle so they can be easily manipulated and have additional information attached to them. In object-oriented systems, this is done by making each message an object. In the GBT each message is encapsulated into the class `Message`. The C++ class constructor for Message accepts a unique identifier, a severity level, and a flag indicating whether the Message signals a state change or a transient event. The constructor also attaches additional information to the Message such as device name and computer of origin. The class method `check` tests for trigger conditions, indicating whether the state is asserted or the event has occurred. The check method sends a Message object to the `MessageMux` whenever the method check indicates a change in state or an event occurrence. The MessageMux appends additional information to create an `AnnotatedMessage` that is made available to any program through a `MesgMuxIF` object. The MessageMux also keeps a history of past messages on file. It is necessary for each Message to have a unique identifier in order for a display programs to handle multiple events for the same Message. In the future the identifier will be used to identify Messages in a table which will be read by the MessageMux. The table will include display options, actions to be taken, inhibitions between Messages to control cascading, and inferences between Messages to generate explanatory messages when patterns of Message states/events are detected.

2.4. Data Production

The data describing a scan can be of different types originating from different devices, e.g., digitally processed radio frequency signals from backends, servo system encoders, sensors in a weather station, or antenna drive motors. Most of this information is sampled from continuously changing quantities. To make sense of the data from these various sources, the data must be aligned based on when the samples were taken. For example, when scanning across the sky, a backend's integration periods have to be matched to their associated (or interpolated) antenna positions. Traditionally, this alignment or collating of data has been part of the telescope control system. But by assigning the integration responsibility to the control system, it becomes difficult to make the selection or configuration of data sources flexible. The system is less modular because the software must know about all the possible data sources and their formats. And being part of the control system, the collating code often acquires real-time constraints in order to keep pace with the various data-producing subsystems.

On the GBT, each data source is sampled at a rate sufficient to track its variability, then time-tagged and written independently to separate files in a common format (FITS binary tables). Each subsystem writes its own data

[†]The inadequacies of our original system were pointed out to us by Stephen L. Scott during an external design review, 1993

file with no knowledge of other files or the timing constraints determined by other subsystems. The responsibility for collating and integrating data is that of the off-line analysis system AIPS++.[9] Admittedly, the problem for the analysis system becomes more difficult because it is not dealing with "pre-packaged" scan data. However, the observer now has access to the "raw" data as collected, and the design allows her/him to run multiple backends and to gather as many DAPs as s/he feels are needed to adequately describe the observation.

3. USER PROGRAMS

The object diagram in Figure 1 shows that the software access to all of the GBT is reduced to a few interfaces at the interface daemons' layer: Panel/Recipient, MessageMux, Accessor, and data files.

Putting aside the problem of data analysis as separate from telescope control, the task of creating user interfaces for the system is greatly simplified. The user-interface program only needs to link in the classes `Monitor`, `MesgMuxIF`, `PanelClient`, and `RecipientServer` to gain full access to the telescope. Each of these classes essentially encapsulates the network protocol for interfacing to the telescope programs.

The problem of implementing a friendly, useful user-interface becomes one of how to best present information and choices to the user. The solution requires application expertise (i.e., observing or operation experience with a telescope) and the ability to easily iterate on designs. These two requirements have led us to conclude that the user program should be an interpreter with a scripting language[‡] that has the capability of creating graphical user interfaces.

For the GBT we are using two interpreter programs. One is Glish[10] which has been enhanced to support twelve Tk widgets using the Rivet library.[11] Because Glish is also used by AIPS++, it is being used to implement the observer's interface to the GBT. The observer can define automated interactions between analysis results and telescope control via the Glish interpreter and software bus. Such a configuration will be the basis of the on-line data monitoring system.

The other interpreter program is Tcl/Tk[12] which provides the full range of Tcl capabilities and add-ons for creating user interfaces. The GBT extension of the Tcl/Tk interpreter is called `segeste` and it provides full access to the methods of `Monitor`, `MesgMuxIF`, `PanelClient`, and `RecipientServer`. Also, a graphical-user-interface builder called `Monet`, based on the application development environment `Visual Tcl`, [§] will be available for user-interface designers; this program can create graphical user interfaces using Sampler, Parameter, and Message widgets. Because it is easier to build user interfaces with Tcl/Tk that are more responsive to information flowing from the telescope to the user, Tcl/Tk will be used for the operating and engineering interfaces.

Because Samplers and Parameters have the ability to hold all data types and are read and written through a generic method,[¶] it is necessary for a user program to have access to a data description for correctly handling the data encapsulated by these two classes. Every Parameter and Sampler has a unique identifier which is passed along with its data and an associated `DataDesc` object which contains the unique identifier, the data type, an explanation, and a units identifier. The description is complete enough that a complete FITS description can be generated by the Archivist on command. The DataDescs for both Parameters and Samplers of a Manager are compiled into a dynamically shared library which the client program loads to gain access to the individual descriptions.

4. UTILITY CLASSES

In addition to the classes described above to implement the four basic functions of the system, there are numerous utility classes used throughout the system (See Table 1).

- classes used as Parameter values which describe dynamic aspects of scans such as antenna movement or frequency changes
- classes for handling dynamic loading of libraries

[‡] John Ousterhout defines a scripting language as one designed for gluing and are mostly typeless, e.g., sh, Tcl, Visual Basic, or Perl.
[§] http://www.neuron.com/stewart/vtcl/
[¶] The data held by an AnnotatedMessage all have the same data structure unlike Parameters or Samplers, therefore, individual objects do not require individual descriptions.

Table 1. Table showing the number of lines of compilable code for implementing GBT device drivers, user programs, and libraries. User Programs include only the code added for the GBT. The Astronomical Libraries are used by the antenna and tracking LO and the Receiver Library is used by the receivers.

Library	number
TimeStamp	764
Sequencer	240
Control	4111
FITS	745
MCBInterface	667
RPC++	1913
headers	582
DeviceAccess	190
SocketClass	268
ConfigIO	509
DynamicLoader	60
IF	2114
Message	2104
monitor	1994
util	592
DataDesc	852

Receivers Library	number
Receiver	1623
Astronomical Library	**number**
Coords	539
TimeKeeper	764
Scan	3244
StarLink	6514
Device Driver	**number**
GPIB	811
Mv340	217
Vmi3100	120
Vmi3118	343
z85c30	286
User Program	**number**
Glish	3887
Tcl/Tk	7394

- classes for emulating radio frequency signals from receiver to backend as manipulated and routed by the electronics and cabling of the system
- classes for encapsulating the interface to device drivers
- classes for specifying and computing times
- classes for the storage and retrieval of system configuration parameters stored on disk
- classes for writing FITS data files

5. DISCUSSION

No formal evaluation exists of the importance of achieving modularity in a general-purpose telescope, or of how successful the present design is in achieving that goal; though at least one informal evaluation has been presented.[13] Our main concern early in the construction phase of the project was whether such a highly modular design would even work. Over the last several years tests on an operating telescope[‖] and the use of software on several operational devices have convinced us of the utility of the approach.

We have found during development that integration, especially of additional devices, has been extremely problem free. A software engineer familiar with the classes can completely implement a simple device, i.e., one that does not require a state machine cam task, in a week. There is not a great deal of experience with user-interface development using the interpreter programs and tools, but early indications are encouraging.

Encapsulating common functionality into class libraries, as this design does, one would hope that the majority of the effort required to implement a device would already have been completed in the libraries of core classes. Using

[‖]National Radio Astronomy Observatory's 140 Foot radio telescope in Green Bank, WV.

Table 2. Table showing the number of lines and percentages of compilable code written specifically for a device, i.e., the percentage column shows the lines written solely for the device as a percentage of the total needed to implement the device.

Device	number	per cent
12-18 GHz Receiver	646	2
18-26 GHz Receiver	797	3
4-6 GHz Receiver	449	1
8-10 GHz Receiver	364	1
Prime Focus Receiver	1796	5
IFRack	1565	5
Tracking LO	3140	7
Analog Filter Rack	1102	4
Converter Rack	1284	4
Frequency Coordinator	4115	12
Switching Signal Selector	623	2
Antenna	38554	49
Digital Continuum Backend	3772	11
Spectral Processor Backend	41228	59
Holography Backend	1260	4
Weather Stations	748	2
Accelerometers	275	1
Tipper	894	3

the crude measure of lines of code, this hope seems fairly well borne out. Table 1 shows the number of lines of code for device drivers, user programs, and libraries.** which are common to all or some subset of the devices. Table 2 compares the lines of code specific to a device vs. the lines of code in the libraries.†† Except for the most complex devices, only a small percentage of code is added to fully implement a device. This measure does not address the complexity of the code being written, but we think the most complex code (again, except for very large devices) is in the libraries.

In conclusion, we feel that making modularity the prime design goal was correct and worth the effort. The true test will be how well the software can be configured to respond to requests we did not envision when designing the system. To make this approach truly general, it would be interesting to apply the libraries to a different telescope and measure the effort required. Such an effort would provide a vehicle to partition and configure the software to be telescope independent.

6. ACKNOWLEDGMENTS

My original concepts[15] underwent extensive change and enhancement because of the experiences gained and the close-knit team effort during development. Members of the GBT monitor and control team who have made significant contributions to the architecture are Joesph J. Brandt, J. M. Ford, Stephane Jouteux, Aron Benett, Timothy L. Weadon, and Robert Payne. Other contributors and advisors include Don Wells, Richard Lacasse, Roger Norrod, Ronald J. Maddalena, and Felix J. Lockman. I especially want to thank J. Richard Fisher and Geoff Croes, without whose confidence and invaluable insights, the project would have fallen far short of its goals.

**The StarLink library[14] was only modified so it would compile with a C++ compiler.

††Comments, blank lines, and lines having only a curly bracket were not counted.

The National Radio Astronomy Observatory is a facility of the National Science Foundation operated under cooperative agreement by Associated Universities, Inc.

REFERENCES

1. N. R. A. O., "Request for Proposals, The Green Bank Telescope," tech. rep., Associated Universities, Inc., June 1990.
2. B. Meyer, *Object-oriented Software Construction*, Prentice Hall, New York, 1988.
3. M. H. Clark and J. R. Fisher, "GBT Monitor and Control Design Rationale," GBT Memo Series 89, National Radio Astronomy Observatory, 1992.
4. L. R. D'Addario, "Comments on GBT Memo No 89," GBT Memo Series 90, National Radio Astronomy Observatory, 1992.
5. J. R. Fisher and M. H. Clark, "Reply to GBT Memo 90," GBT Memo Series 92, National Radio Astronomy Observatory, 1992.
6. J. R. Fisher and f. J. Lockman, "Observer Monitor and Control Requirements," GBT Memo Series 91, National Radio Astronomy Observatory, 1992.
7. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, New York, 1991.
8. W. D. Cotton, D. Tody, and W. D. Pence, "Binary table extension to FITS," *Astronomy and Astrophysics Supplement Series* **113**, pp. 159–166, October 1995.
9. B. E. Glendenning, "Creating an object-oriented software system - the aips++ experience," *Astronomical Data Analysis Software and Systems V, A.S.P. Conf. Ser.* **10**, p. 271, 1996.
10. D. Schiebel and V. Paxson, *The Glish 2.6 User Manual*. National Radio Astronomy Observatory, June 1996.
11. B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, New Jersey, 1995.
12. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.
13. J. R. Fisher, "Object-Oriented Experiences with GBT Monitor and Control," *Astronomical Data Analysis Software and Systems VII, A.S.P. Conf. Ser.* , 1997.
14. P. T. Wallace, "SLALIB — Positional Astronomy Library," Tech. Rep. Starlink User Note 67.34, CCLRC / Rutherford Appleton Laboratory, February 1996.
15. M. H. Clark, "GBT M&C requirements analysis model," GBT Memo Series 88, National Radio Astronomy Observatory, 1992.