EXPANDED VERY LARGE ARRAY

Antenna Monitor & Control System Design Philosophy

Kevin Ryan

May 6, 2003

NATIONAL RADIO ASTRONOMY OBSERVATORY
P.O. Box 0, Socorro, New Mexico 87801

Operated by Associated Universities, Inc.
Under Contract with the National Science Foundation

Abstract:

The design of the EVLA Antenna Monitor and Control System (AMCS) is different than that of previous systems. The design was originally presented at the Preliminary Design Reviews (PDRs) in December 2001, and May 2002. During the year since, some implementation details have emerged that have strayed from the philosophy behind the design.

The purpose of this paper is to present the design again along with new insights gained in the year since the PDRs. This document is meant not to be a formal specification of the design but more of a guide to the philosophy behind it, with the belief that an understanding of this philosophy will help to keep subsequent development on course.

1 The Philosophy

1.1 Introduction

The design of the EVLA Antenna Monitor & Control (AMCS) is different because the EVLA is different and, because of new technology, the design can be different. When the VLA was built, it would have been impractical to implement a design that called for some 900 processors each with capabilities approaching or exceeding the original single central computer used.

During development that has occurred in the year since the PDRs, some implementation details have emerged that do not fit the design philosophy. Some examples of this are:

- The idea that a switch should check to see if it is being used in an observation before letting a technician operate it.
- 'Attention' fields in data streams.
- The idea that the high-level observe layer should have access to low-level raw monitor data.

It is the intent of this first section of the document to explain why these examples and others like them simply do not make sense in the EVLA AMCS.

If the examples still seem valid after reading this document, than it has failed in its purpose.

1.2 The EVLA has Special Requirements

Much of the design of the EVLA AMCS was shaped by two special requirements.

- The system must accommodate different antennas and subcomponent hardware.
- The system uses high level IP over Ethernet for data communication at all levels from the control building down to the 'field bus' within the antennas.

Consequences of the Requirements

1.2.1 Complexity to the Control System

The current VLA consists of 27 functionally identical antennas. To set the elevation angle on any one antenna, Multiplexer (MUX) address 302 of Data Set 0 is set to a value of [Degrees * 3600 / 1.236] where 1.236 is the weight of the servo register's LSB in seconds of arc because that register is 20-bits wide.

The VLA control system might set all antennas to an elevation of 47.0º with the following code segment:

```
for x = 1 to 27
  set(Ant = x, DataSet = 0, MUX = 302, Val = 0x216C1)
```

The EVLA Antennas will not use Data Sets so the control system will have to handle them differently.

```
for x = 1 to 27
  if ant == TYPE_VLA
        set(ant=x, DataSet=0, MUX=302, Val=0x216C1)
  else if ant == TYPE_EVLA
        set (ant=x, MIB=35, SpiAddr=0x60, Val=0x3381776)
              ...
```

A single control system handling both antenna types will suffer increased complexity.

1.2.2 Complexity to the Delivery of Control and Monitor Data Throughout the System

Traditionally, microcontrollers have been used in systems to provide a simple interface between hardware and software. Set a memory location in the micro to a value and almost instantly that value is transferred via a hardwired field bus directly to the hardware.

Communications in the EVLA will be via the high-level Internet Protocol (IP) over Ethernet. This has ramifications to the EVLA hardware and to the software design.

1.2.2.1 Data will move through network devices that are not under our control.

It will not be possible to rely on the IP protocol for real-time delivery of monitor and control data.

1.2.2.2 Higher-level microprocessors will be needed for the higher-level protocol.

The micros used in the EVLA must be ‘smarter’ than those of tradition because they must be able to implement the complex IP. This is not so much a problem as it is a benefit. Because of technology, the micros can be more capable even while being less expensive than in the past. The AMCS design takes advantage of this.

1.3 Solving the Problem

What we have:

- A system of different types of equipment that must be operated synchronously in a somewhat timely manner.

What we have to work with:

- Many fairly 'smart' processors distributed over a sophisticated network that has poor real-time capability.

1.3.1 Take Advantage of Smart Micros - Spread the Work out

The VLA uses one processor to control the whole array of antennas. The VLBA uses one processor per single antenna. Some 30 or more microcontrollers within a single EVLA antenna will be utilized for its control.

The EVLA will employ its micros to perform functions that when combined become a single radio-telescope antenna. This is called *distributive processing*. Instead of one processor providing all the functionality, it is now divided into many.

The big, complex system is broken into bite-size chunks.

It is important to understand how this is different than the role of the micros of yesteryear. The old micros simply passed data to their hardware, they did not really do much in terms of being responsible for the *behavior* of their hardware.

1.3.1.1 So Who Does What?

Breaking a system into manageable pieces is making it a modular system. Making a system be modular is not difficult if certain rules are used.

1.3.1.1.1 Tight is a Good

In the physical world, modularization is done by Nature. A switch is a switch; it turns on, it turns off. Software could be this easy too but usually the software engineer feels obliged to add functionality … because he can. The switch, now in software, must not let a technician operate it if it sees that its antenna is in the middle of an observation.

The onus of changing its behavior based on the context of its use would not be put on a switch in the 'real world'. A physical switch does not know what observing means nor even what an antenna is – its job is to simply be a switch. Usage policy is the job of the user.

There is a term used for this in computer science; it is *cohesion*.

Cohesion measures the 'single-mindedness' of a module.

The goal is to maximize or to exhibit *tight* cohesion within a module. This means a switch can only be a switch – even in software. If its user does not want someone else operating the switch then that user should hang a lock on it like he would in the physical world.

1.3.1.1.2 Loose is Good

Back to the Elevation Servo. To set it to 47.0º the central processor calculates the raw valued needed for the particular model of servo in use. To send this value, it might use a byte to represent the DataSet number, another byte to represent the MUX address and an int or long depending on the hardware for the value.

The central computer has to know a lot about how the antenna wants its data in order that it is formatted properly and sent to the correct place in the antenna.

Computer science also has a term for measuring the complexity of communications between two processes. It is called *coupling*.

Coupling is a measure of a module's interaction with other modules and/or data.

The goal is to minimize or have *loose* coupling.

Since the AMCS micros are smart, they can calculate their own raw values and know how to address their own hardware – no sense making the main control system have to know about these details.

Now setting the elevation of all antennas to 47.0º becomes:

```
for x = 1 to 27
  ant[x].setElevation(47.0)
```

A single value is sent to each antenna, no DataSets, MUX's, bytes, ints or longs just a value in everyday degrees. It won't matter what type of antenna it is, or even if it's similar antenna-types that have different models of servos because they all now take a high-level, generic, value.

The coupling between the control building and the antenna just became less.

1.3.1.1.3 Less Data is Good Data

Proper modularization minimizes data flow throughout a system. We can apply the servo command example to monitor data also. In the VLA the main control system fetches MUX 003 of Data Set 0, multiplies it by 0.005 to convert to an analog voltage from the A/D, and then multiplies that by 3.2 to scale it to the proper range. He just obtained the Elevation Motor #1 Current; again, with the overhead of a lot of details that he really shouldn't have to know. Most likely, the value will be checked to see if it is in range and then tossed out.

The micro can do these trivial details. Let him fetch the value and do the conversions. He can even do the range checking and send a simple go/no-go if that is all the main control system cares about. Even less coupling.

1.3.1.1.4 Keeping Secrets

Besides reducing traffic, doing the implementation details at the lower levels has another tremendous advantage; the details are compartmentalized.

If our elevation servo is upgraded to a model with 24-bit resolution, the weight of the LSB would become $360/2^{24}$ = 0.7725 seconds. Since our system sends the value generically (in degrees) this change is localized to the module. In a non-modular system, all software at all levels that had anything to do with the servo would have to be changed and/or regression tested.

Computer science's term for information hiding is *encapsulation*.

Encapsulation reduces the spread of change-affects throughout a system.

This is extremely good for maintenance and scalability.

1.3.1.1.5 Standing on Their Own

Passing only high-level, generic command and monitor data is a good start but we can do more to optimize the big three – tight cohesion, loose coupling and detail encapsulation.

In our improved system of generic data passing, the micro reports to the central process that both his +5V power supplies are operating within range – it is not reporting what the actual values are, just that they are in range. In fact, if the system is truly modular, *the central process would not have to know that a module even had two +5V power supplies*. The number of power-supplies in a module has no bearing on its operation with regard to the rest of the system.

In order to be modular to the max, a module must be independent of the rest of the system. If it can be fully operated by itself on a test-bench then it is independent.

This implies that a module must provide all of its own functionality. It means our servo must accept a position value, move itself there, provide indications of where it is, and check on its own power-supplies.

A module must monitor and control itself

1.3.1.2 Guidelines for Modularity

Applying the above concepts, the guidelines for the design of the AMCS can be discovered.

1.3.1.2.1 Modules must be stand-alone.

No module must rely on another for operation. This includes the software that operates them. A standalone module can be operated on the test-bench. A module that is normally operated in the system by its parent module will not require the parent module's software in order to be operated out of the system.

To be standalone means that the module must provide its own monitor and control, accepting generic, high-level commands, and providing generic high-level 'front-panel' status indications.

1.3.1.2.2 A module knows itself intimately.

It is the only module in the system to know about its own implementation specific details.

Since this is the case, it makes no sense to send low-level, raw monitor point, data to other modules – they will not know what it means nor what to do with it.

1.3.1.2.3 A module is aware of its children.

An antenna is aware that it has an elevation servo and it knows how to operate it. It does not know however, what is inside the servo nor how it works. The antenna does not care if the servo has one power-supply or two.

The antenna does care if the servo is working properly though. The module monitors itself in great detail but provides general (front-panel) indications of its health and operational status. The parent must monitor its children's front panels. If the child module fails, it is the job of the parent to report it since a failed module may not be able to report itself.

1.3.1.2.4 A module is not aware of its parent.

A switch used in a radio-telescope does not know about radio-telescopes; its area of expertise is the making or breaking of an electrical path. It knows how to that no matter where it is installed.

1.3.1.2.5 A module is not aware of its peers.

In the same regard, a switch does not know about servos, nor do antennas know about correlators.

1.3.1.2.6 Commands are generic.

Commands tell a module *what* to do, not *how* to do it. If the high-level process tells a device how to go about configuring itself, then this would imply that the high-level process would have to know low-level details about the device it is controlling.

1.3.1.2.7 Commands flow from high level to low.

It makes no sense for a servo to command an antenna or LO if it does not know what they are.

1.3.1.2.8 Raw monitor data will not flow across module boundaries.

Considering that - because of encapsulation - the receiving module cannot know what the data means anyway, it makes no sense for it to have it.

This means raw monitor data will not be broadcast except as noted here:

The Archiving System. Every monitor point will periodically be sent to this system to provide a system ‘snapshot’. This does not break modularity rules because the Archive System does not *do* anything with the data other than log it. Adding, deleting or changing monitor points will not break the archiving system since it simply logs anything that it receives. In this regard, encapsulation is not lost.

1.3.1.2.9 Monitor data flows from low level to high.

Since a module knows nothing of its peers or parents, it makes no sense for a child to monitor its parent or peer.

1.3.1.2.10 Front-Panel control and monitor data may cross layer boundaries.

Since higher layers control and monitor lower layers via their front-panel, the front-panel data must be communicated across layers.

How front-panel data is communicated has not yet been implemented.

It is possible that front-panel controls can be broadcast down through the system and that front-panel indicators can be broadcast up through the system without breaking rules of modularity.

1.3.2 Plan Ahead

Since we cannot rely on our commands arriving immediately at their destinations we are left with little choice but to have them already be in place for when they are needed. This means sending them early and implies a queing mechanism in each device.

This only applies to commands traveling between modules because control within a module will not have to be done over the network.

1.3.2.1 Special Consideration for High-Speed Data Monitoring.

This is another area affected by the chosen network that must be considered. Data monitoring such as total power and even round-trip phase might have to be given special consideration as they will present efficiency problems. (Imagine one total-power value being wrapped in a UDP packet and sent over the network every 2-milliseconds.) Total power can be buffered, but round-trip phase readings 10 times a second probably can’t.

1.4 Putting it Together

1.4.1 Modules have Two Modes of Operation

To facilitate interaction with a module in both normal operation and maintenance modes, each module will support two interfaces.

1.4.1.1 Normal Operation is Via a Module's Front-Panel

The software front-panel is like a physical front-panel in that it provides all the information needed to operate the module in a normal manner.

Even though AMCS Devices will be standalone, there will still be a need to see information from them during normal operation. For instance, the operators may want to see the current position of the Elevation Servo or the phase lock indicator of a PLL.

The front-panel must provide all the monitor information and control functionality needed to operate the module fully during normal operation.

1.4.1.2 The Front-Panel can be Removed for Maintenance

A module must be able to be opened for close examination by the technician for maintena nce. This means exposure to all raw monitor and control points and other implementation specific details.

Every control/monitor-point shall be accessible when the front panel is removed.

Very detail-specific GUIs shall be allowed. Everything we learned about encapsulation, coupling and cohesion can be discarded. It is okay for a GUI to be made showing details of a particular model of a device; it will be understood that if the physical device changes that GUI will have to change also.

Scripts or programs that manipulate raw CP's and MP's from a host computer shall be allowed to do so with the front-panel removed.

1.4.2 No Real-Time Operations Take Place Over the Network.

All inter-module commands that must be completed at a particular time must be sent in advance of that time.

No hard data as to how far in advance is available yet.

2 The Design

The design of the AMCS was predicated on the following assumption:

The higher the level of detailed implementation, the higher the quantity and complexity of data that must flow between layers in a system, resulting in its increased complexity.

Distributive processing puts 'the intelligence near the action' resulting in less data flow across layers and a less complex system.

2.1 The System as Modules

In the EVLA AMCS almost everything is a module. Each is naturally defined by the physical devices of the system such as Antennas, Weather Stations, the Atmospheric Phase Interferometer (API), the elevation servo, LO's, switches and etc..

The generic name given to every software device module in the AMCS is *Device*.

All Devices will be hosted on processors. EVLA Antenna sub-component Devices will be hosted on the Module Interface Board (MIB), VLA Antennas and associated sub-components will all be hosted on a single processor called the Interim Control and Monitor Processor (CMP).

EVLA sub-arrays will be hosted on desktop machines or card-cage based processors in the Control Building.

It is legal for a single processor to host more than one Device.

2.2 The System as a Hierarchy

The AMCS is designed in a specific hierarchy of layers. At the top is the high-level operation of the system known as the observing layer or just operations. This top layer controls the whole system including the Correlator Monitor & Control System (CMCS), the Correlator Backend System and provides access to the system from outside entities such as e2e. The next layer is the composite device layer that contains the antennas Devices, (sub)arrays, weather station, API and etc. . The lowest layer contains individual component Devices.

The observe layer controls antennas (or sub-arrays) through their front-panels. The observe layer does not control antenna subcomponents. *Antennas control their own subcomponents via the subcomponent's front-panel*. Control from the highest layer to the lowest is simply a parent module telling a child module what to do and so on until the desired action is accomplished.

To draw contrast to the current system, the VLA uses a non-hierarchal approach where all components of the system are at one layer functionally. The lowest control point is set directly by the operations layer.

2.2.1 Observing Layer

The observing layer is responsible for specifying what the system must do in order to perform a desired task such as an observation.

It is important that this first statement is understood. It implies that the observing layer does not directly control the system but only specifies *what* it wants the system to do. The reason for this is simple:

> *The observer should not have to know how the instrument works in order to use it – only what it is capable of doing.*

An observe script should only be required to specify *what* is to be done and when; *how* the system is postured to satisfy that request is the jobs of the control systems - the AMCS and CMCS.

2.2.1.1 Observe Scripts

First attempts at defining observe scripts did not do so in the context of specifying 'what's' as opposed to 'how's'. Consequently, the scripts contained actual programming code (scripting code) and made reference to software objects

Those scripts mimicked much of what was being done in the system itself in terms of control functionality. This is dangerous because whoever or whatever generates those control scripts would have to know details about how the system works.

Observe scripts should simply specify <what><when>. This implies that programming code is unnecessary; indeed, simple lines of text-based keyword/value pairs are preferred. Using industry standard 'markup' languages such as XML to specify these pairs is even better. Section three of this document discusses this further.

2.2.2 Composite Component Layer

This layer consists of high-level hardware such as the antennas, weather station and API. These are all represented in the AMCS by software Devices. These Devices break their composite, high-level commands into those needed for their individual subcomponents.

An example might be setting the frequency band on which an antenna is to observe. The observe layer tells the antenna what band it wants it to be on. The antenna then tells the necessary subcomponents (switches, frequency converters, etc.) what they must do so that the whole antenna is postured to receive on the desired band.

2.2.3 Component Layer

The lowest layer of the system. It represents the individual component Devices such as our trusty elevation servo Device. This layer is where most of the software meets the hardware. In the EVLA, component layer software will reside in the MIB that is built onto the hardware device itself.

In the band example, the switch is told what position it must be in. The switch has no idea that he helped set the whole antenna to a particular band because it has no idea of what a band is or what an antenna is.

2.3 The System as Software Objects

Object Oriented (OO) software design lends itself very well to the purpose of representing the physical system modules in software. To ensure that no confusion over terms exists, a few explanations of how they are used in this document are given:

- **Class** – The abstract definition of a software module. Classes themselves are not executable.
- **Base Class** – When several classes share common methods and attributes, it is sometimes better to put the common things into a single class that can be *inherited*. Using the venerable Shape example: a Square, a Triangle, and a Circle are all geometric shapes. All shapes have some things in common such as area. The common items can be grouped into a base class called Shape. The Square, Triangle and Circle classes can now all *extend* (or inherit from) Shape and automatically acquire the common attributes and methods. This is a built-in form of software re-use.
- **Subclass** – A class that inherits from a Base Class. It has all the methods and attributes of the base class plus its own.
- **Object** – The instance of a class. At run-time, a class is *instantiated* into an object that is executable.
- **Attributes** – The data (variables, constants, etc.) of a class.
- **Methods** – The functions or subroutines of a class.
- **Interface** – A set of methods and attributes that a class must implement if it implements the interface. The Interface guarantees that users of a class can expect that the class will 'do all of the things' specified in that Interface. The AMCS uses Interfaces as a kind of contract between client applications and system servers so that the clients can know what services are guaranteed to be provided.

2.3.1 The Foundation Classes and Interfaces of the AMCS

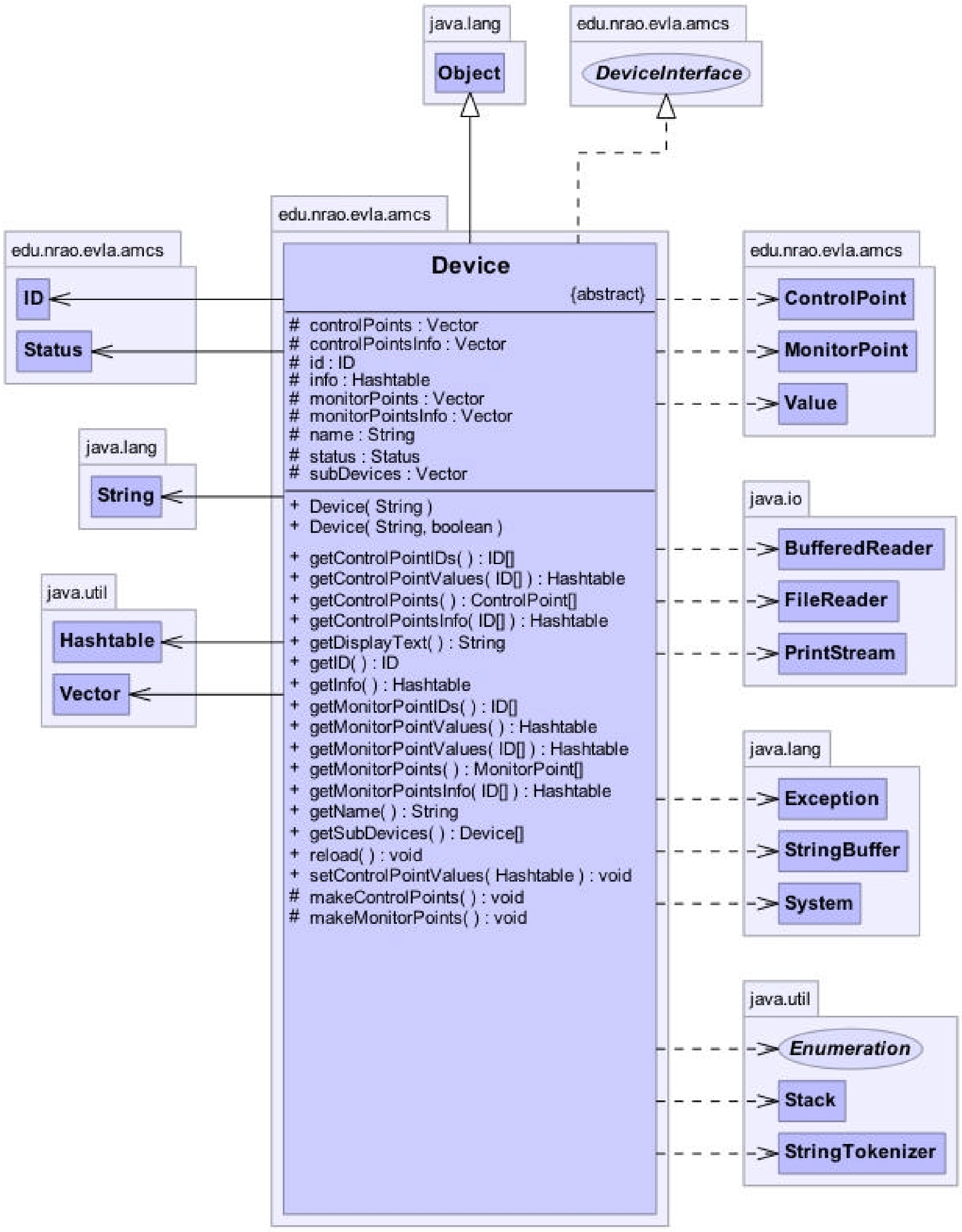
The following paragraphs describe the fundamental classes that make up the ‘infrastructure’ of the AMCS. They lay the foundation upon which the specific components classes will be built.

It is possible that this foundation (or parts of it) might be used in the EVLA Correlator Monitor & Control System (CMCS).

Omitted from this discussion is the ‘middleware’ layer that is responsible for the communications *between* modules.

2.3.1.1 Device

The Device is the fundamental software representation of all the major components in the physical system; these include antennas, servos, synthesizers, etc., the weather station and possibly subarrays.


java.lang
Object
edu.nrao.evla.amcs
DeviceInterface
edu.nrao.evla.amcs
Device
{abstract}
# controlPoints : Vector
# controlPointsInfo : Vector
# id : ID
# info : Hashtable
# monitorPoints : Vector
# monitorPointsInfo : Vector
# name : String
# status : Status
# subDevices : Vector
+ Device(String)
+ Device(String, boolean)
+ getControlPointIDs( ) : ID[]
+ getControlPointValues(ID[]) : Hashtable
+ getControlPoints( ) : ControlPoint[]
+ getControlPointsInfo(ID[]) : Hashtable
+ getDisplayText( ) : String
+ getID( ) : ID
+ getInfo( ) : Hashtable
+ getMonitorPointIDs( ) : ID[]
+ getMonitorPointValues( ) : Hashtable
+ getMonitorPointValues(ID[]) : Hashtable
+ getMonitorPoints( ) : MonitorPoint[]
+ getMonitorPointsInfo(ID[]) : Hashtable
+ getName( ) : String
+ getSubDevices( ) : Device[]
+ reload( ) : void
+ setControlPointValues(Hashtable) : void
# makeControlPoints( ) : void
# makeMonitorPoints( ) : void
edu.nrao.evla.amcs
ID
Status
java.lang
String
java.util
Hashtable
Vector
edu.nrao.evla.amcs
ControlPoint
MonitorPoint
Value
java.io
BufferedReader
FileReader
PrintStream
java.lang
Exception
StringBuffer
System
java.util
Enumeration
Stack
StringTokenizer

The Device base class must be extended by each of the component classes (i.e. the ElevationServo class). This makes the ElevationServo everything that a Device is but with additional characteristics specific to that of an Elevation Servo.

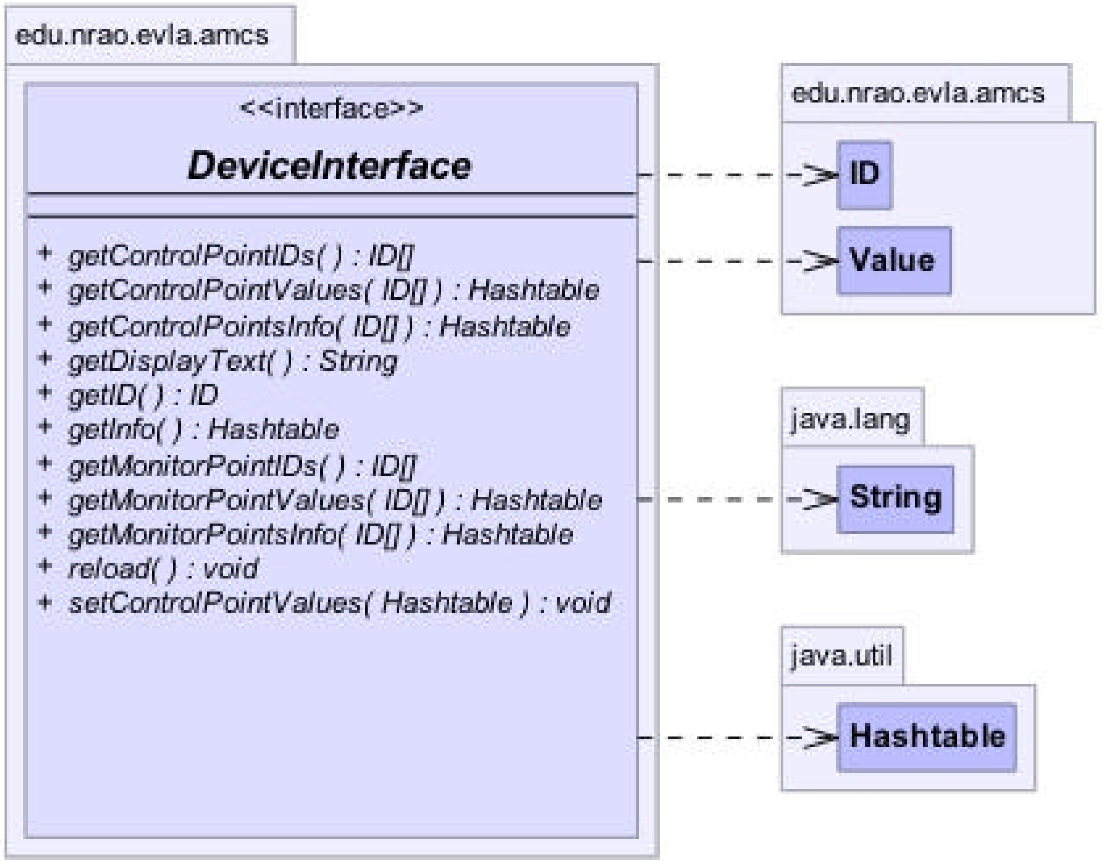
All Devices support a DeviceInterface for access to all of its internals for maintenance purposes, and a FrontPanelInterface for operational access. These are discussed later.

All Devices have the ability to parse a text-based description file to fill in some Device parameters (name or ID for example) and to create most, if not all of its, Monitor and Control points.

2.3.1.2 DeviceInterface

The DeviceInterface provides unlimited access to specific details within a Device.

This is a maintenance interface and as such, it *must* provide the details. Everything about a device including its raw monitor and control data points can be obtained through this interface. This breaks the rules of encapsulation in the normal sense, but when considered in the context of its use, it is perfectly natural since the client (a diagnostic routine, for example.) *must be* intimately familiar with the data it is using.


edu.nrao.evla.amcs
<<interface>>
DeviceInterface
+ getControlPointIDs( ) : ID[]
+ getControlPointValues(ID[]) : Hashtable
+ getControlPointsInfo(ID[]) : Hashtable
+ getDisplayText( ) : String
+ getID( ) : ID
+ getInfo( ) : Hashtable
+ getMonitorPointIDs( ) : ID[]
+ getMonitorPointValues(ID[]) : Hashtable
+ getMonitorPointsInfo(ID[]) : Hashtable
+ reload( ) : void
+ setControlPointValues(Hashtable) : void
edu.nrao.evla.amcs
ID
Value
java.lang
String
java.util
Hashtable

The DeviceInterface will not be used for normal operation of the system because it would mean that raw monitor and control data would be flowing across the system to the operations program. Since operations must be maintained on a generic level this will not work.

For maintenance purposes however, the DeviceInerface will provide several methods of Device and MP/CP control as described here.

It is intended that the DeviceInterface provide enough access to the Device that it will not need to be extended in most cases by specific Devices in order to provide basic maintenance activities on the component.

2.3.1.2.1 The DeviceInterface Provides Various ways to Access Monitor and Control Points

Two modes of access are foreseen at this time; raw, low-level, monitor data broadcasting is not one of them.

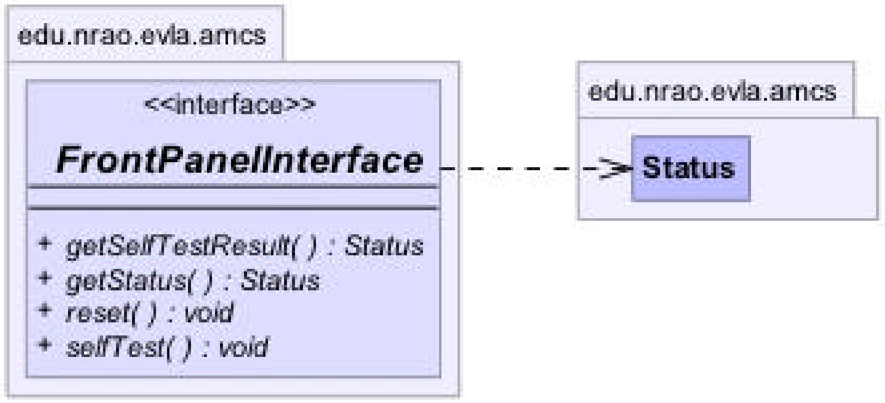
- Peek/Poke mode. MP's and CP's can be requested and manipulated remotely. This implies that programs such as diagnostic routines can poll individual MP's or lists of several MP's.
- Spigot Mode. One or more MP's can be 'turned on' to be repeatedly sent to a requesting client at a requested rate without the client having to ask for them each time. When the requesting client's socket is broken, flow stops.

The DeviceInterface provides various ways for maintenance access.

- Generic (Discovery-Based) Device Browser. The Discovery-based Device 'browser' was first demonstrated at the PDR in May, 2002. It is a client application that can connect to any Device or DeviceHost and display all the available information about it and its MP's and CP's. This will probably be the first high-level client developed for the AMCS as it will be of great utility value during the system development phase.
- Device Specific GUIs. These GUI's will most likely be specific to a particular Device knowing beforehand the MP's and CP's that it will be using. If the hardware device changes, than this type of GUI will have to change with it.
- Non-GUI Based. This encompasses programs and scripts that access raw data for analysis. They may run in the background as CRON jobs and perform functions such as higher speed (than normal archiving) data logging.

2.3.1.3 The FrontPanelInterface

The FrontPanelInterface presents the Device for normal operation. It is this interface that allows a Device to be operated without having to know how it works on the inside. The observe system and everyday-use operator screens will use this interface. This is also the interface used by parent modules to operate their children.



The FrontPanelInterface does not do much by itself except specify a minimal set of methods that all Devices must provide. It is the intent that Specific Devices must extend the FrontPanelInterface to provide the functionality needed to fully operate them.

The FrontPanelInterface to a Device is what make generic operation of it possible. All ElevationServo Devices will have similar front panels even though they may be physically different. This will allow the user to command an elevation position without having to know things such as the weight of the LSB of a particular servo's position register.

The FrontPanelInterface might consist of method calls, an XML schema describing the available functions that are available via XML or a combination of both.

2.3.1.4 DeviceServer

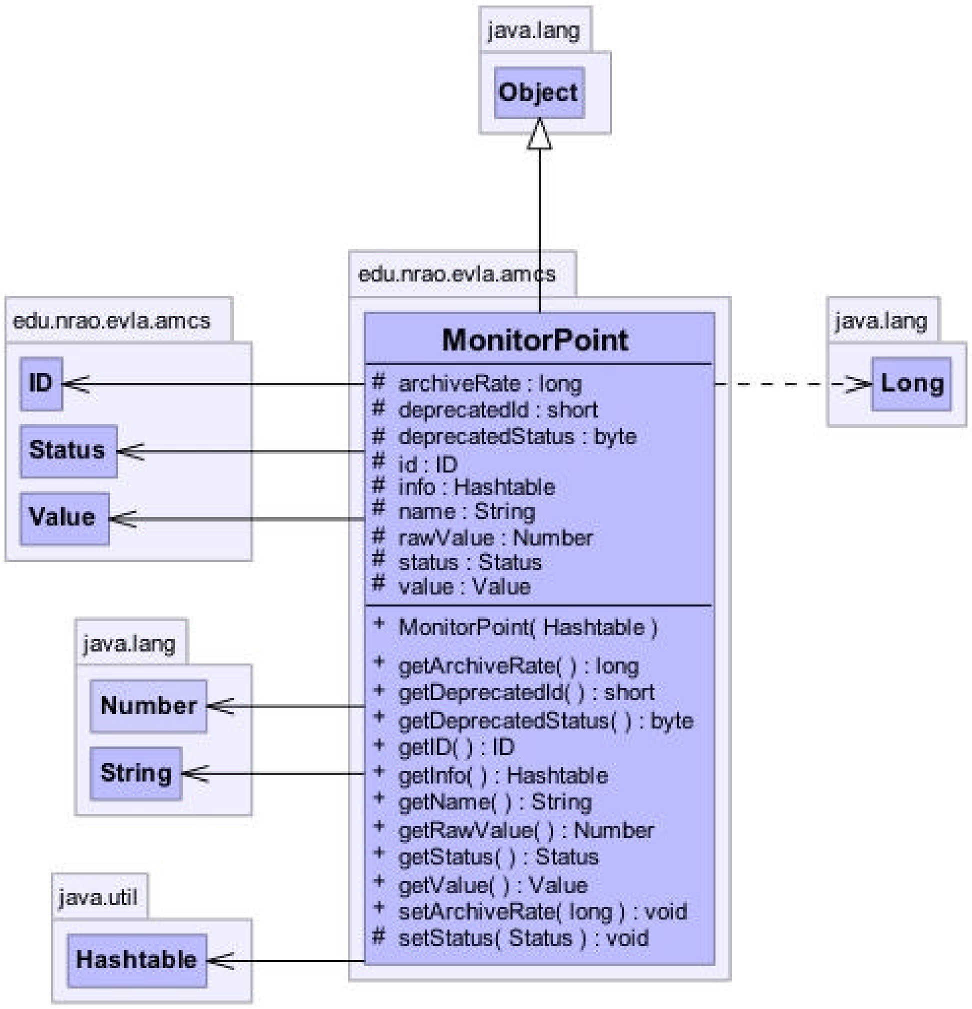
A DeviceServer is a Device that represents the processor that houses component Devices. In other words the MIBs, Antenna Computers, etc.. The DeviceServer that resides in the MIB would be what some have called the 'MIB Object'. DeviceServers have the following characteristics.

- Exactly one DeviceServer per processor.
- Serves zero to many Devices.
- C*ontains* the component Device objects that it serves so it may manipulate them directly without having to 'broker' them as would be case over a network.
- Provides a single network interface shared by all of its Devices.

- Provides an operationally transparent interface to all onboard Devices. The DeviceServer must not be 'seen' by the rest of the system in terms of operation. A block diagram of an antenna depicts the subcomponents of the antenna; things such as servos, synthesizers, switches, etc.. The operational software must see these same items in the same way; this means that it should not view them hanging from a server object. The DevicesServer must present the Devices to the system in a way that the system 'thinks' it is communicating with the Device directly.

2.3.1.5 MonitorPoints and ControlPoints

These classes represent the 'data' of the AMCS. They are monitored and set by the Devices that contain them. They are not normally sent outside of their Device except for maintenance purposes like when the Device is undergoing troubleshooting.

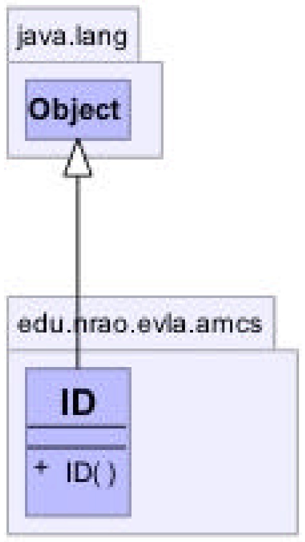


Most MonitorPoints will be updated by their Device, either periodically or asynchronously when the MonitorPoint is requested. It is possible, however, that some MPs and CPs could contain their own Thread. In those cases, they will be compiled into the Device and not instantiated at run-time as described next.

MonitorPoints and ControlPoints are classes that are can be instantiated at run-time from a text-based description file. The ability to instantiate MP's and CP's at runtime makes it possible to change attributes such as the archiveRate or alarm ranges and even to add or delete MP/CP's without having to recompile.

On processors that support file systems the descriptor files will exist as files and be read from the file system. In the case of the MIBs, they will still exist as files, but will uploaded into the MIB's non-volatile memory.

2.3.1.6 ID

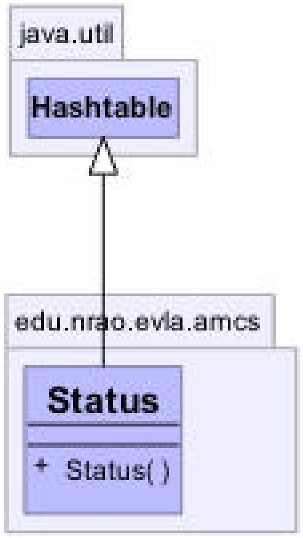


The ID class represents an identifier for every Device, MonitorPoint and ControlPoint in the system. ID was made a class instead of a string or integer to allow more flexibility.

ID is new so much is not known about it yet but the intent is to provide a way to identify elements in ways that make both the human and machine happy (i.e. a readable format for the human and cryptic unique code for the computer).

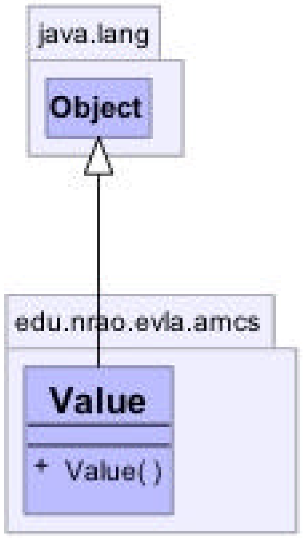
ID might also contain non-unique ID information such as 'this thing is of the VlaAntenna type'.

2.3.1.7 Status



Status is a class that is used to represent any kind of status from a simple Boolean go/no-go to complex structures containing state along with possible amplifying information such as fault analysis or corrective maintenance advise.

2.3.1.8 Value



Value is a class that provides a means to communicate data across module boundaries without having to use specific data types.

Although Value is also new it will most likely be akin to Java's Number class but with the ability to also handle strings.

2.3.2 Building on the Foundation

2.3.2.1 Component Devices

The specific Devices (i.e. Elevation Servos) that extend and use the foundation classes are what will provide the real functionality of the AMCS. Each specific Device will be responsible for the overall 'modularness' of the system. It must provide the functionality needed to make the represented physical device a stand-alone entity to the point that it can operate independently of the rest of the system. This is natural when one considers that a servo does not have to be in a radio-telescope in order to be a servo.

This functionality includes self-monitor and control. AMCS Devices will perform 'checker' services on themselves. This is what distributed-processing is about. Some advantages of this are:

- No other system component knows more about a device than the device himself.
- A central checker system will be a cause for undue amounts of raw data flowing across module boundaries. Self-checking will only send high-level (and meaningful) messages when appropriate.
- A Device will be able to accept commands of a generic nature and know exactly how to break it down into the format needed by its own specific control-points.

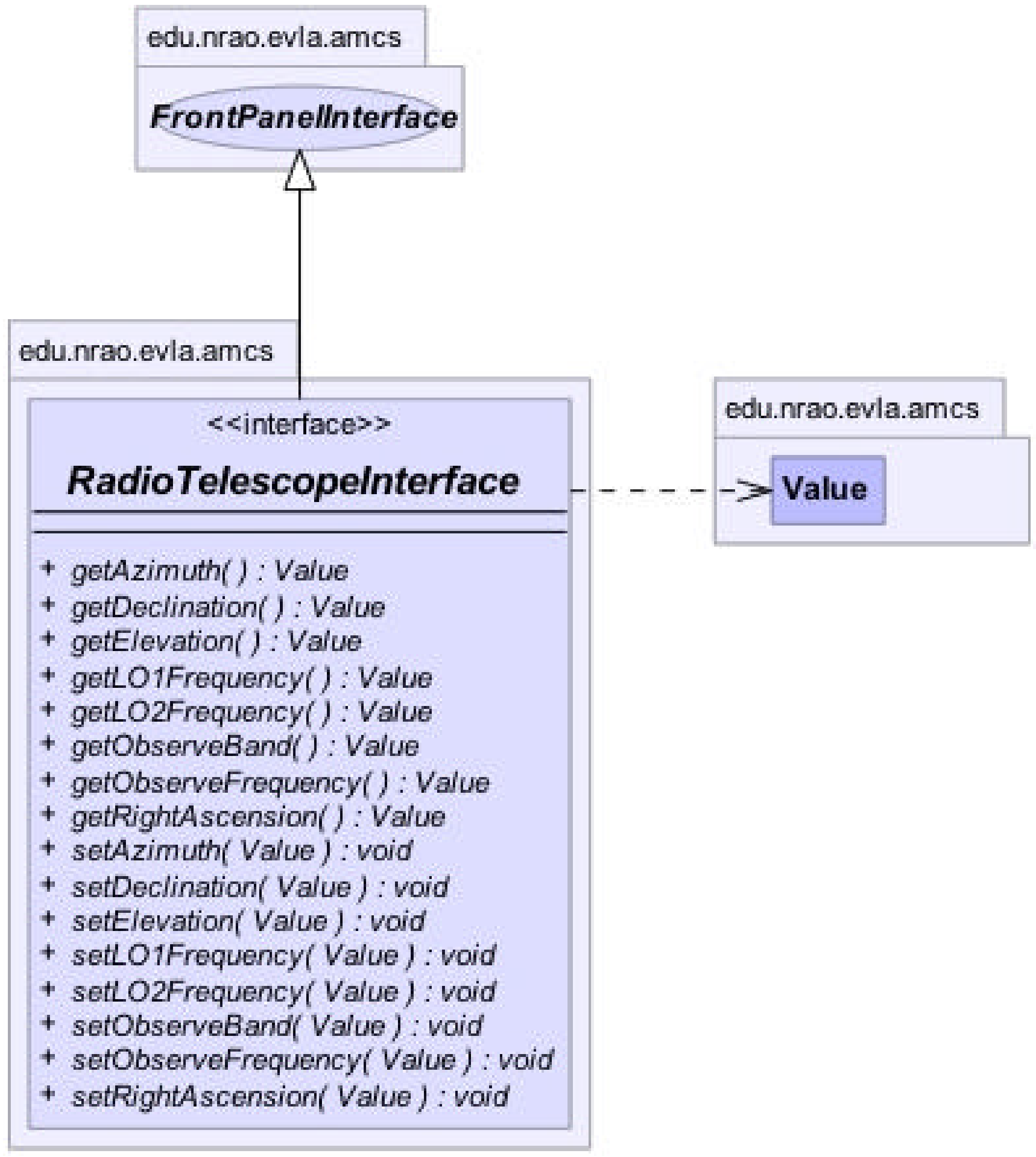
The software that provides the functionality of specific Devices will have to be developed in close coordination with the hardware engineers that created the physical device. In fact, a significant portion of software may be low-level routines written in low-level languages by the hardware engineers themselves.

2.3.2.2 Front Panels

Since 'front-panels' are the sole interface for normal operation of the physical devices, they must also be designed with the aid of the devices' designers and expert users

Software engineers are currently meeting with individual hardware device's designer in order to be better familiarized with the requirements for the control and monitor of each. Even so, the requirements of every front-panel should be formally specified.

The first front-panel has been started; it is the RadioTelescopeInterface.


edu.nrao.evla.amcs
FrontPanelInterface
edu.nrao.evla.amcs
<<interface>>
RadioTelescopeInterface
+ getAzimuth( ) : Value
+ getDeclination( ) : Value
+ getElevation( ) : Value
+ getLO1Frequency( ) : Value
+ getLO2Frequency( ) : Value
+ getObserveBand( ) : Value
+ getObserveFrequency( ) : Value
+ getRightAscension( ) : Value
+ setAzimuth(Value) : void
+ setDeclination(Value) : void
+ setElevation(Value) : void
+ setLO1Frequency(Value) : void
+ setLO2Frequency(Value) : void
+ setObserveBand(Value) : void
+ setObserveFrequency(Value) : void
+ setRightAscension(Value) : void
edu.nrao.evla.amcs
Value

As one can see, this interface is severely lacking input from expert users, let alone a formal specification.

2.3.2.3 Inter-Module Communications

The TC111-IB based MIB limits the communications 'middleware' available for use. If in-house proprietary communications software is required, it must be designed with scalability in mind.

The Value, Status and ID classes were created to help reduce the tendency to over-use specific data types in the protocol.

3 Operation

3.1 Via Front-Panel (Normal Operation).

It was stated that the Observe Script simply needs to tell the system what is to be done and when to do it. This is possible because the front-panel interface of Devices such as the sub-array and antennas provides this generic control type capability. The observe script could simply be XML text of the following (overly-simplified) format for example. The example is for an observation consisting of two scans; one at 15:08:09 the other at 15:14:27 VLA LST on L-Band.

> ***Note:*** *Since the following example was first contrived, it has been learned that e2e may convert RA/DEC, supplying the AMCS with AZ/EL; the antenna then would only be responsible for applying its own pointing error corrections.*

The following 'script' is received from e2e:

```
<configuration time=2002y09m05d15:08:09:LST>
   <subarray>1..27</subarray>
   <source>
      <ra>14:00:28.6526</ra>
      <dec>+62:10:38.526</dec>
   </source>
   <band>L</band>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <subarray>1..27</subarray>
   <source>
      <ra>15:49:17.4686</ra>
      <dec>+50:38:05.788</dec>
   </source>
   <band>L</band>
</configuration>
```

This would tell the system that all 27 antennas are to be used to view the 1st specified source on L-Band starting at 15:08:09 and continuing until it is time to start moving so as to be in position for the next configuration at 15:14:27.

Assuming the Subarray Device receives the script from e2e, it will parse from it the information needed to tell its antenna Devices what to do.

Two methods of inter-module communication are being worked on by the AMCS engineers; one method is a form of remote method invocation the other is via XML. Both are illustrated below by continuing with our example script.

3.1.1 Using Remote Method Calls

After parsing the example observe script, the Subarray would issue the following method calls on each of its antennas.

```
for x = 1 to 27:
   Antenna[x].setBand(“L”, “2002y09m05d15:08:09:LST”)
   Antenna[x].setBand(“L”, “2002y09m05d15:14:27:LST”)
   Antenna[x].setRA(“14:00:28.6526”, “2002y09m05d15:08:09:LST”)
   Antenna[x].setRA(“15:49:17.4686”, “2002y09m05d15:14:27:LST”)
   Antenna[x].setDEC(“62:10:38.526”, “2002y09m05d15:08:09:LST”)
   Antenna[x].setDEC(“50:38:05.788”, “2002y09m05d15:14:27:LST”)
```

Each Antenna Device would convert RA & DEC into AZ & EL and command its subcomponents as follows:

```
AzimuthServo.setPosition(344.270, “2002y09m05d15:08:09:LST”)
AzimuthServo.setPosition(18.3817, “2002y09m05d15:14:27:LST”)
ElevationServo.setPosition(52.9291, “2002y09m05d15:08:09:LST”)
ElevationServo.setPosition(72.2661, “2002y09m05d15:14:27:LST”)
XBandConverterSwitch.setPosition(“L”, “2002y09m05d15:08:09:LST”)
XBandConverterSwitch.setPosition(“L”, “2002y09m05d15:14:27:LST”)
XBandIFAmplifierSwitch.setPosition(“X”, “2002y09m05d15:08:09:LST”)
XBandIFAmplifierSwitch.setPosition(“X”, “2002y09m05d15:14:27:LST”)
FirstLOSwitch.setPosition(“LCS”, “2002y09m05d15:08:09:LST”)
FirstLOSwitch.setPosition(“LCS”, “2002y09m05d15:14:27:LST”)
```

This method implies that the parent must be able to implement remote object method calls on its child Devices.

The advantage of this is that the parents would have a well-defined ‘contract’ via the FrontPanelInterface of what each of his child Devices is capable of.

The disadvantages are that remote method invocation can be complicated and systems using ‘hard-coded’ interfaces such as those that define specific method calls can break if the interface changes on one end without changing on the other.

3.1.2 Using XML

Another method of inter-module communications is currently under discussion and, at first glance, appears favorable. It involves using XML between all the layers of the system – not just between e2e and the AMCS. Since simple strings are passed between layers, remote object calls are unnecessary.

Like the previous example, the subarray Device receives the same observe script from e2e. It would glean the pertinent antenna-specific information and re-issue refined XML to each of its antennas:

```
<configuration time=2002y09m05d15:08:09:LST>
   <source>
      <ra>14:00:28.6526</ra>
      <dec>+62:10:38.526</dec>
   </source>
   <band>L</band>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <source>
      <ra>15:49:17.4686</ra>
      <dec>+50:38:05.788</dec>
   </source>
   <band>L</band>
</configuration>
```

Each antenna converts RA and DEC to AZ and EL and sends the following to its servos:

To AzimuthServo:

```
configuration time=2002y09m05d15:08:09:LST>
   <position>344.270</position>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <position>18.3817</position>
</configuration>
```

To ElevationServo:

```
<configuration time=2002y09m05d15:08:09:LST>
   <position>59.9291</position>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <position>72.2661</position>
</configuration>
```

Switches are positioned to receive on L-Band as follows:

To XBandConverterSwitch:

```
<configuration time=2002y09m05d15:08:09:LST>
   <position>L</position>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <position>L</position>
</configuration>
```

To XBandIFAmplifierSwitch:

```
<configuration time=2002y09m05d15:08:09:LST>
   <position>X</position>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <position>X</position>
</configuration>
```

To FirstLOSwitch:

```
<configuration time=2002y09m05d15:08:09:LST>
   <position>LCS</position>
</configuration>
<configuration time=2002y09m05d15:14:27:LST>
   <position>LCS</position>
</configuration>
```

This example implies that parent Devices must be capable of reading and writing XML and child Devices must be capable of reading it.

Unlike remote object brokering however, XML parsing is relatively simple and it should be possible to find an industry parser that even the MIB will support.

3.1.2.1 XML Method Inter-Module

Perhaps the biggest advantage that the XML model affords is that inter-module communications are greatly simplified. Modules would simply have to know how to pass strings.

Just as a parent device tells its children what to do, so too can a human via a very simple GUI or even command shell if all that is needed to operate the Devices are text-based command/value pairs.

A technician could telnet into a servo on his workbench and type: “<position><123.45>”.

String based protocols are being used extensively in industry. They make extremely 'thin' clients such as web browsers possible. Because of this, it is possible that any device supporting a web browser might be able to fully operate an AMCS Device.

3.1.2.2 XML Method Intra-Module

If the Device understands XML for communications across the network, it can use this same understanding for operations within itself.

3.1.2.2.1 Device Initialization

XML can be used to define the state or mode that the Device should configure itself too on initialization. This XML block would be a part of the same descriptor file that defines a Device's monitor and control points and other parameters.

Since it is text based and file-resident the contents of the descriptor file can be changed and the changes made effective without code recompilation.

3.1.2.2.2 During Operation

XML provides a persistent, ready-made (it was already used for the communications interface) 'list' of the states that a Device is or will be in. This list can be used by the Device as a queue of 'what to do, when' and by their parent Devices (or even human users) for validation purposes.

3.2 With the Front-Panel Removed (Maintenance Mode)

As mentioned in the previous section, the maintenance technician would be able to operate any Device via a simple generic device browser (perhaps even a web browser) via the Device's front-panel. When he needs to go further however, he must 'remove' the front-panel to have access to everything within.

It is envisioned that direct access to a Device (via the DeviceInterface) should be done with the client communicating directly with the Device. In other words, the tech's diagnostic screen would open a direct connection to the Device's IP address and not go through a parent Device.